Software Security Knowledge: Training

Robert A. Martin
Sean Barnum
May 2011
Agenda

8:00-8:45am  Software Security Knowledge about Applications Weaknesses
9:00-9:45am  Software Security Knowledge about Attack Patterns Against Applications

Training in Software Security

10:15-11:00am Software Security Practice
11:15-12:00am Supporting Capabilities Assurance Cases
                Secure Development & Secure Operations
CWE is Meant for People to Use
A complete body of knowledge covering the entire field of software engineering may be years away. However, the body of knowledge needed by professionals to create software free of common and critical security flaws has been developed, vetted widely and kept up to date. That is the foundation for a certification program in software assurance that can gain wide adoption. It was created in late 2008 by a consortium of national experts, sponsored by DHS and NSA, and was updated in late 2009. It contains ranked lists of the most common errors, explanations of why the errors are dangerous, examples of those errors in multiple languages, and ways of eliminating those errors. It can be found at http://cwe.mitre.org/top25.

Any programmer who writes code without being aware of those problems and is not capable of writing code free of those errors is a threat to his or her employers and to others who use computers connected to systems running his or her software.
The Certified Secure Software Lifecycle Professional (CSSLP) Certification Program will show software lifecycle stakeholders not only how to implement security, but how to glean security requirements, design, architect, test and deploy secure software.

An Overview of the Steps:

(ISC)²® 5-day CSSLP CBK® Education Program
Educate yourself and learn security best practices and industry standards for the software lifecycle through the CSSLP Education Program. (ISC)² provides education your way to fit your life and schedule. Completing this course will, not only teach all of the

establish a security plan across your

COMPUTER BASED TESTING NOW AVAILABLE FOR THE CSSLP
MSC00-CPP. Compile cleanly at high warning levels

MSC00-CPP. Compile cleanly at high warning levels

According to CERT [ISO/IEC 9899:1999] Section 5.1.1.3, "Diagnostics"

References

[ISO/IEC 9899:1999] Section 5.1.1.3, "Diagnostics"

MITRE 07] CWE ID 563, "Unused Variable"; CWE ID 570, "Expression is Always False"; CWE ID 571, "Expression is Always True"

[Sutter 05] Item 1

[Seacord 05a] Chapter 8, "Recommended Practices"
Manually review code after security education

Manual code reviews, especially reviews of high-risk code, such as code that faces the Internet or parses data from the Internet, is critical, but only if the people performing the code review know what to look for and how to fix any code vulnerabilities they find. The best way to help understand classes of security bugs and remedies is education, which should minimally include the following areas:

- C and C++ vulnerabilities and remedies, most notably buffer overruns and integer arithmetic issues.
- Web-specific vulnerabilities and remedies, such as cross-site scripting (XSS).
- Database-specific vulnerabilities and remedies, such as SQL injection.
- Common cryptographic vulnerabilities and remedies.

Many vulnerabilities are programming language (C, C++, etc.) or domain-specific (web, database) and others can be categorized by vulnerability type, such as injection (XSS and SQL Injection) or cryptographic (over random number generation and weak secret storage) so specific training in these areas is advised.

Resources


- Common Weakness Enumeration, MITRE: http://cwe.mitre.org/

- Security Code Review


- Security Code Review Guidelines, Adam Shatkin:


CWE

CAPEC

Making Security Measurable

© 2011 MITRE
Hi, Michael.

Every bug is an opportunity to learn, and the security update that fixed the data binding bug that affected Internet Explorer users is no exception.

The Common Vulnerabilities and Exposures (CVE) entry for this bug is CVE-2008-4944.

Before I get started, I want to explain the goals of the SDL and the security work here at Microsoft. The SDL is designed as a multi-layered process to help systematically reduce security vulnerabilities; if one component of the SDL process fails to prevent or catch a bug, then some other component should prevent or catch the bug. The SDL also mandates the use of security defenses whose impact will be reflected in the "mitigations" section of a security bulletin, because we know that no software development process will catch all security bugs. As we have said many times, the goal of the SDL is to "Reduce vulnerabilities, and reduce the severity of what's missed."

In this post, I want to focus on the SDL-required code analysis, code review, fuzzing and compiler and operating system defenses and how they fared.

Background

The bug was an invalid pointer dereference in MSHTML.DLL when the code handles data binding. It's important to point out that there is no heap corruption and there is no heap-based buffer overrun.

When data binding is used, IE creates an object which contains an array of data binding objects. In the code in question, when a data binding object is released, the array length is not correctly updated leading to a function call into freed memory.

The vulnerable code looks a little like this (by the way, the real array name is arvPRef, but I figured ArrayOfObjectsFromI is a little more descriptive for people not in the Internet Explorer team.)

```csharp
int MaxIdx = ArrayOfObjectsFromIE.Size();
for (int i = 0; i < MaxIdx; i++) {
    if (!ArrayOfObjectsFromIE[i])
        continue;
    ArrayOfObjectsFromIE[i]->transferRemote();
    ...
}
```

Here's how the vulnerability manifests itself: if there are two data transfers with the same identifier (so MaxIdx is 2), and the first transfer updates the length of the ArrayOfObjectsFromIE array when its work was done and releases its data binding object, the loop count would still be whatever MaxIdx was at the start of the loop.

This is a time-of-check-time-of-use (TOCTOU) bug that led to code calling into a freed memory block. The Common Weakness Enumeration (CWE) classification for this vulnerability is CWE-367.

The fix was to check the maximum iteration count on each loop iteration rather than once before the loop starts. This is the correct fix for a TOCTOU bug - move the check as close as possible to the action because...
Our methodology for the Top 10 2007 was simple: take the MITRE Vulnerability Trends for 2006, and distill the Top 10 web application security issues. The ranked results are as follows:

- Cross-site scripting: 30.00%
- Injection Flaws: 25.00%
- Malicious File Execution: 20.00%
- Insecure Direct Object Reference: 15.00%
- Cross-site Request Forgery (CSRF): 10.00%
- Information Leakage and Inappropriate Access: 5.00%
- Broken Authentication and Confidentiality Management: 3.00%
- Insecure Cryptographic Storage: 2.00%
- Insecure Cryptographic Communications: 1.00%
- Failure to Restrict URL Access: 0.00%

Figure 2: MITRE data on Top 10 web application vulnerabilities for 2006
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 guide 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”
Some High-Level CWEs Are Now Part of the NVD CVE Information

Overview
SQL injection vulnerability in mods/banners/navlist.php in Clansphere 2007.4 allows remote attackers to execute arbitrary SQL commands via the cat_id parameter to index.php in a banners action.

Impact
CVSS Severity (version 2.0):
CVSS v2 Base score: 7.1 (High) (AV:N/AC:L/Au:N/C:P/I:P/A:S) (legend)
Impact Subscore: 6.4
Exploitability Subscore: 10.0

Access Vector: Network exploitable
Access Complexity: Low
Authentication: Not required to exploit

References to Advisories, Solutions, and Tools
External Source: BID (Disclaimer)
Name: 25770
Hyperlink: http://www.securityfocus.com/bid/25770

External Source: MILWORM (disclaimer)
Name: 4443
Hyperlink: http://www.milworm.com/exploits/4443

Vulnerable software and versions
Configuration 1
Clansphere, Clansphere, 2007.4

Technical Details
Vulnerability Type (View All)
SQL Injection (CWE-89)

CVE Standard Vulnerability Entry:
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5061
Common Platform Enumeration:
Welcome to the NIST SAMATE Reference Dataset Project

The purpose of the SAMATE Reference Dataset (SRD) is to provide users, researchers, and software security assurance tool developers with a set of known security flaws. This will allow end users to evaluate tools and tool developers to test their methods. These test cases are designs, source code, binaries, etc., i.e., from all the phases of the software life cycle. The dataset includes "wild" (production), "synthetic" (written to test or generated), and "academic" (from students) test cases. This database will also contain real software application bugs and vulnerabilities. The dataset intends to encompass a wide variety of possible vulnerabilities, languages, platforms, and test cases. The dataset is anticipated to become a large-scale effort, gathering test cases from many contributors. We have more details about the SRD, including goals, structure, test suite selection, etc.

Browse, download, and search the SRD

Anyone can browse or search test cases and download selected cases. Please click here to browse the test case repository, or select or all test cases. To find specific test cases, please click here.

How to submit test cases

NIST Special Publications:

SP500-268   CWE
SP500-269   CWE
SP800-53a   CVE, OVAL, CWE
SP800-115   CVE, CCE, CVSS, CWE

NIST Interagency Reports:

NISTIR-7435  CVE, CVSS, CWE
NISTIR-7628  CVE, CWE
Idaho National Labs SCADA Report
<table>
<thead>
<tr>
<th>Weakness Classification</th>
<th>Vulnerability Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE-19: Data Handling</td>
<td>CWE-228: Improper Handling of Syntactically Invalid Structure</td>
</tr>
<tr>
<td></td>
<td>CWE-229: Improper Handling of Values</td>
</tr>
<tr>
<td></td>
<td>CWE-230: Improper Handling of Missing Values</td>
</tr>
<tr>
<td></td>
<td>CWE-20: Improper Input Validation</td>
</tr>
<tr>
<td></td>
<td>CWE-116: Improper Encoding or Escaping of Output</td>
</tr>
<tr>
<td></td>
<td>CWE-195: Signed to Unsigned Conversion Error</td>
</tr>
<tr>
<td></td>
<td>CWE-198: Use of Incorrect Byte Ordering</td>
</tr>
<tr>
<td>CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer</td>
<td>CWE-120: Buffer Copy without Checking Size of Input (“Classic Buffer Overflow”)</td>
</tr>
<tr>
<td></td>
<td>CWE-121: Stack-based Buffer Overflow</td>
</tr>
<tr>
<td></td>
<td>CWE-122: Heap-based Buffer Overflow</td>
</tr>
<tr>
<td></td>
<td>CWE-125: Out-of-bounds Read</td>
</tr>
<tr>
<td></td>
<td>CWE-129: Improper Validation of Array Index</td>
</tr>
<tr>
<td></td>
<td>CWE-131: Incorrect Calculation of Buffer Size</td>
</tr>
<tr>
<td></td>
<td>CWE-170: Improper Null Termination</td>
</tr>
<tr>
<td></td>
<td>CWE-190: Integer Overflow or Wraparound</td>
</tr>
<tr>
<td></td>
<td>CWE-680: Integer Overflow to Buffer Overflow</td>
</tr>
<tr>
<td>CWE-398: Indicator of Poor Code Quality</td>
<td>CWE-454: External Initialization of Trusted Variables or Data Stores</td>
</tr>
<tr>
<td></td>
<td>CWE-456: Missing Initialization</td>
</tr>
<tr>
<td></td>
<td>CWE-457: Use of Uninitialized Variable</td>
</tr>
<tr>
<td></td>
<td>CWE-476: NULL Pointer Dereference</td>
</tr>
<tr>
<td></td>
<td>CWE-400: Uncontrolled Resource Consumption (“Resource Exhaustion”)</td>
</tr>
<tr>
<td></td>
<td>CWE-252: Unchecked Return Value</td>
</tr>
<tr>
<td></td>
<td>CWE-690: Unchecked Return Value to NULL Pointer Dereference</td>
</tr>
<tr>
<td></td>
<td>CWE-772: Missing Release of Resource after Effective Lifetime</td>
</tr>
<tr>
<td>CWE-442: Web Problems</td>
<td>CWE-22: Improper Limitation of a Pathname to a Restricted Directory (“Path Traversal”)</td>
</tr>
<tr>
<td></td>
<td>CWE-79: Failure to Preserve Web Page Structure (“Cross-site Scripting”)</td>
</tr>
<tr>
<td></td>
<td>CWE-89: Failure to Preserve SQL Query Structure (“SQL Injection”)</td>
</tr>
<tr>
<td>CWE-703: Failure to Handle Exceptional Conditions</td>
<td>CWE-431: Missing Handler</td>
</tr>
<tr>
<td></td>
<td>CWE-248: Uncaught Exception</td>
</tr>
<tr>
<td></td>
<td>CWE-755: Improper Handling of Exceptional Conditions</td>
</tr>
<tr>
<td></td>
<td>CWE-390: Detection of Error Condition Without Action</td>
</tr>
</tbody>
</table>
As requested here are the links to all the posts on the Top 25 Most Dangerous Programming Errors. Please let us know if you have any suggestions or comments.

1 – Cross-Site Scripting (XSS)
2 – SQL Injection
3 – Classic Buffer Overflow
4 – Cross-Site Request Forgery (CSRF)
5 – Improper Access Control (Authorization)
6 – Reliance on Untrusted Inputs in a Security Decision
7 – Path Traversal
8 – Unrestricted Upload of Dangerous File Type
9 – OS Command Injection
10 – Missing Encryption of Sensitive Data
11 – Hardcoded Credentials
12 – Buffer Access with Incorrect Length Value
13 – PHP File Inclusion
14 – Improper Validation of Array Index
15 – Improper Check for Unusual or Exceptional Conditions
16 – Information Exposure Through an Error Message
17 – Integer Overflow Or Wraparound
18 – Incorrect Calculation of Buffer Size
19 – Missing Authentication for Critical Function
20 – Download of Code Without Integrity Check
21 – Incorrect Permission Assignment for Critical Response
22 – Allocation of Resources Without Limits or Throttling
23 – Open Redirect
24 – Use of a Broken or Risky Cryptographic Algorithm
25 – Race Conditions
# SDL and the CWE/SANS Top 25

Bryan here. The security community has been buzzing since SANS and MITRE’s joint announcement earlier this month of their list of the [Top 25 Most Dangerous Programming Errors](https://www.sans.org/top25). Now, I don’t want to get into a debate in this blog about whether this new list will become the new de facto standard for analyzing security vulnerabilities (or indeed, whether it already has become the new standard). Instead, I’d like to present an overview of how the Microsoft SDL maps to the CWE/SANS list, just as I did in May.

Michael and I have written extensive coverage of the Top 25 and believe that the results to date show that the Top 25 were developed independently, rooted in the community, and driven by the analyst process and guidance around every major vulnerability in 2010. We even made many of the same mistakes, but in a different way, so it’s worth you to download and review these as part of the SDL.

Below is a summary of how the Top 25 map to the SDL. I hope to see the SDL covers everything and provides the context for all of them (race conditions and security flaws). The below list is a result of multiple SDL requirements (and tools) to prevent or detect weaknesses highlighted in the Top 25.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Improper Input Validation</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>116</td>
<td>Improper Encoding or Escaping of Output</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>89</td>
<td>Failure to Preserve SQL Query Structure (aka SQL Injection)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>79</td>
<td>Failure to Preserve Web Page Structure (aka Cross-Site Scripting)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>78</td>
<td>Failure to Preserve OS Command Structure (aka OS Command Injection)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>319</td>
<td>Cleartext Transmission of Sensitive Information</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>352</td>
<td>Cross-site Request Forgery (aka CSRF)</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>362</td>
<td>Race Condition</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>209</td>
<td>Error Message Information Leak</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>119</td>
<td>Failure to Constrain Memory Operations within the Bounds of a Memory Buffer</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>642</td>
<td>External Control of Critical State Data</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>External Control of File Name or Path</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>426</td>
<td>Untrusted Search Path</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>94</td>
<td>Failure to Control Generation of Code (aka 'Code Injection')</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>494</td>
<td>Download of Code Without Integrity Check</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>404</td>
<td>Improper Resource Shutdown or Release</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>665</td>
<td>Improper Initialization</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>982</td>
<td>Incorrect Calculation</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>285</td>
<td>Improper Access Control (Authorization)</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>327</td>
<td>Use of a Broken or Risky Cryptographic Algorithm</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>259</td>
<td>Hard-Coded Password</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>732</td>
<td>Insecure Permission Assignment for Critical Resource</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>339</td>
<td>Use of Insufficiently Random Values</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>Execution with Unnecessary Privileges</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>602</td>
<td>Client-Side Enforcement of Server-Side Security</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CWE Outreach: A Team Sport

May/June Issue of IEEE Security & Privacy...

Improving Software Security by Eliminating the CWE Top 25 Vulnerabilities

January 2009. METRE and SANS issued the May 2009 CWE/SANS Top 25 Most Dangerous Programming Errors” to help make developers more aware of the bugs that can cause security vulnerabilities.

CWE-79: Use of Insecure Random Values

Identify all the random number generators in your code and verify that, if any, are properly seeded, periodic or unpredictable. Many systems and libraries use the same random number generator, so a single seed can cause a multi-system vulnerability.

CWE-22: Improper Input Validation

Ensure that all user input is properly validated. Be sure to validate any user input that could be manipulated or be used to control an application’s behavior.

CWE-674: Use of Third-Party Library Without Code Analysis

Use of third-party libraries without code analysis can lead to vulnerabilities. Make sure to properly validate and test any third-party libraries used in your applications.

CWE-119: Insecure Deserialization

Deserialization is a complex process that can be exploited in various ways. Be sure to properly validate and test any deserialization processes used in your applications.

CWE-690: Improper Access Control

Improve access control mechanisms to prevent unauthorized access to sensitive data. Be sure to properly validate and test any access control mechanisms used in your applications.

CWE-75: Improper Authorization

Authorize access to sensitive data based on user roles and permissions. Be sure to properly validate and test any access control mechanisms used in your applications.

CWE-139: Improper Resource Management

Properly manage resources to prevent buffer overflows or memory leaks. Be sure to properly validate and test any resource management processes used in your applications.

CWE-122: Improper Input Validation

Ensure that all user input is properly validated. Be sure to validate any user input that could be manipulated or be used to control an application’s behavior.
[비즈니스 임팩트를 줄여주는 새로운 품질 관리 방법론]

y5를 사용하여, 소프트웨어 결정을 없애는 5가지 스텝은 아래와 같습니다.

1. 스캔 소프트웨어
2. 검출 결함 우선순위
3. 「ビジネ스インパクト를考慮する新しい品質管理」

Coverity5를 사용하여, 소프트웨어 결함을 쉽게 제거하는 5단계는 다음과 같습니다.

1. 스팬 소프트웨어
2. 검출 결함 우선순위
3. 마핑 결함 품질
4. 수정 우선순위의 높은 결함
5. 레포트 결함 수정결과

Korean
Making Security Measurable
Japanese
The Web Application Security Consortium / Threat Classification Taxonomy Cross Reference View

Threat Classification Taxonomy Cross Reference View

last edited by Robert Auger 10 months, 3 weeks ago

Threat Classification 'Taxonomy Cross Reference View'

This view contains a mapping of the WASC Threat Classification's Attacks and Weaknesses with MITRE's Common Weakness Enumeration, MITRE's Common Attack Pattern Enumeration and Classification, OWASP Top Ten 2010 RC1 (original mapping with OWASP Top Ten from Jeremiah Grossman & Bill Corry) and SANS/CWE and OWASP Top Ten 2007 and 2004 (original mapping from Dan Cornell, Denim Group).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WASC-01</td>
<td>Insufficient Authentication</td>
<td>287</td>
<td></td>
<td>642</td>
<td>A3 - Broken Authentication and Session Management, A4 - Insecure Direct Object References</td>
<td>A7 - Broken Authentication and Session Management, A4 - Insecure Direct Object Reference</td>
<td>A3 - Broken Authentication and Session management, A2 - Broken Access Control</td>
</tr>
<tr>
<td>WASC-02</td>
<td>Insufficient Authorization</td>
<td>284</td>
<td></td>
<td>285</td>
<td>A4 - Insecure Direct Object References, A7 - Failure to Restrict URL Access</td>
<td>A10 - Failure to Restrict URL Access, A4 - Insecure Direct Object Reference</td>
<td>A2 - Broken Access Control</td>
</tr>
<tr>
<td>WASC-03</td>
<td>Integer Overflows</td>
<td>199</td>
<td>128</td>
<td>682</td>
<td>A10 - Insufficient Transport Layer Protection</td>
<td>A9 - Insecure Communications</td>
<td></td>
</tr>
<tr>
<td>WASC-04</td>
<td>Insufficient Transport Layer Protection</td>
<td>311 523</td>
<td></td>
<td>319</td>
<td>A10 - Insufficient Transport Layer Protection</td>
<td>A9 - Insecure Communications</td>
<td></td>
</tr>
<tr>
<td>WASC-05</td>
<td>Remote File Inclusion</td>
<td>98</td>
<td>193</td>
<td>253</td>
<td>426</td>
<td>A3 - Malicious File Execution</td>
<td></td>
</tr>
<tr>
<td>WASC-06</td>
<td>Format String</td>
<td>134</td>
<td></td>
<td>67</td>
<td>A2 - Cross-Site Scripting</td>
<td>A1 - Cross Site Scripting (XSS)</td>
<td></td>
</tr>
<tr>
<td>WASC-07</td>
<td>Buffer Overflow</td>
<td>119</td>
<td>120</td>
<td>10 100</td>
<td>119</td>
<td>A5 - Buffer Overflows</td>
<td></td>
</tr>
<tr>
<td>WASC-08</td>
<td>Cross-site Scripting</td>
<td>79</td>
<td>18 19 63</td>
<td>79</td>
<td>A2 - Cross-Site Scripting</td>
<td>A1 - Cross Site Scripting (XSS)</td>
<td></td>
</tr>
<tr>
<td>WASC-09</td>
<td>Cross-site Request Forgery</td>
<td>352</td>
<td>62</td>
<td>352</td>
<td>A5 - Cross-Site Request Forgery</td>
<td>A4 - Cross Site Scripting (XSS)</td>
<td></td>
</tr>
<tr>
<td>WASC-10</td>
<td>Denial of Service</td>
<td>221</td>
<td></td>
<td>404</td>
<td>A7 - Failure to Restrict Access to Information</td>
<td>A10 - Failure to Protect Confidential Information</td>
<td></td>
</tr>
</tbody>
</table>

Done
Test and vulnerability assessment

Testing applications for security defects should be an integral and organic part of any software testing process. During security testing, organizations should test to help ensure that the security requirements have been implemented and the product is free of vulnerabilities.

The SEF refers to the MITRE Common Weakness Enumeration (CWE) list and the Common Vulnerability Scoring System (CVSS) 2.0 to be tested. This helps to prioritize the information about vulnerabilities and provides a way to assess vulnerabilities against the model.

Creating a security findings mitigation plan includes:

- Identifying the vulnerabilities
- Prioritizing the vulnerabilities
- Developing a mitigation plan
- Implementing the mitigation plan
- Reassessing the application for any remaining vulnerabilities

Resources available to help organizations protect systems in development

<table>
<thead>
<tr>
<th>Resource</th>
<th>Focus</th>
</tr>
</thead>
<tbody>
<tr>
<td>DoD Information Assurance Certification and Accreditation Process (DIACAP)</td>
<td>The DIACAP defines the minimum standards for application-level security controls, but it includes activities, general tools, and a management activities checklist.</td>
</tr>
<tr>
<td>Defense Information Systems Agency (DISA)</td>
<td>The DISA provides a security technical in-depth review of the security and development that offer more granular information on application-level security controls and vulnerabilities assessment techniques. The checklist is the same one used by DoD auditors.</td>
</tr>
<tr>
<td>The Common Weakness Enumeration project, a community-based program sponsored by the MITRE Corporation, an IBM Business Partner</td>
<td>The MITRE Corporation maintains the online common vulnerabilities and exposures (CVE) knowledge base, which includes current and released vulnerabilities. This knowledge base focuses on packaged software and deals with patches and known vulnerabilities.</td>
</tr>
<tr>
<td>The Open Web Application Security Project (OWASP)</td>
<td>One of the best sources for information on web application security issues, the OWASP Top 10 list of the most dangerous and most commonly found and commonly exploited vulnerabilities how to identify, fix and avoid them.</td>
</tr>
<tr>
<td>Digital Building Security in Maturity Model (BSIMM)</td>
<td>Created by Digital, an IBM Business Partner, the BSIMM is designed to help organizations plan a software security initiative. The focus is on making applications more secure and at later stages in the software life cycle.</td>
</tr>
<tr>
<td>IBM X-Force research and development team</td>
<td>A global team of threat and risk analysts that monitor traffic and attacks around the world. The IBM X-Force team is an excellent resource for trend analysis and information about the vulnerabilities. Attacks are most common, where they are coming from and what organizations can do to mitigate the risks.</td>
</tr>
<tr>
<td>IBM Institute for Advanced Security (IAS)</td>
<td>This companywide cybersecurity initiative applies IBM research, services, software and tools to help governments and other clients improve the security and resiliency of their IT and business operations.</td>
</tr>
</tbody>
</table>

Security in Development: The IBM Secure Engineering Framework

- Investigating common development processes and the IBM Integrated Product Development process
- Emphasizing security awareness and requirements in the software development process
- Discussing test and vulnerability assessment strategies
Making the Business Case for Software Assurance

Nancy R. Moad
Julie H. Allen
W. Arthur Condlin
Antonio Drommi
John Hamilton
Jeff Ingham
James Rainey
Don Sheenemaker
April 2006

SPECIAL REPORT
CMU/SEI-2006-SR-001

CERT Program
Unrestricted distribution subject to the copyright.

http://www.cert.org

OVM: An Ontology for Vulnerability Management
Ji An Wang & Minhe Guo
Southern Polytechnic State University
1100 South Marietta Parkway
Marietta, GA 30060
(01) 678-9515-3718
jwang@spsu.edu

ABSTRACT
In order to reach the goals of the Information Security Automation Program (ISAP) [1], we propose an ontological approach to capturing and utilizing the fundamental concepts in information security and their relationships, and relating vulnerabilities data and resources about the cause and impact of vulnerabilities. Our ontology for vulnerability management (OVM) has been populated with all vulnerabilities in NVD [2] with additional inference rules, knowledge representation, and data-relating mechanisms. With the seamless integration of common vulnerabilities and their related concepts such as attacks and countermeasures, OVM provides a promising pathway to making ISAP successful.

Categories and Subject Descriptors

General Terms
Ontology, Security, Vulnerability Analysis and Management

Keywords
Security vulnerability, Semantic technology, Ontology, Vulnerability analysis

I. INTRODUCTION
The Information Security Automation Program (ISAP) is a U.S. government multi-agency initiative to enable automation and standardization of technical security operations [1]. Its high-level goals include establishing standards and procedures for vulnerability checking and remediation as well as automation of technical compliance activities. As in high-level objectives include enabling standards based communication of vulnerability data, removing and managing configuration baselines for various IT products, accessing information systems and reporting compliance status, using standard metrics to weight and aggregate potential vulnerabilities, and implementing identified vulnerabilities [1].

Secure computer systems ensure that confidentiality, integrity, and availability are maintained for users, data, and other information assets. Over the past few decades, a significant amount of knowledge has been accumulated in the area of information security. However, a lot of concepts in information security are vague and sometimes they have different semantics in different contexts, causing misunderstanding among stakeholders due to the language ambiguity. On the other hand, the standardization, design and development of security tools [1-3] require a systematic classification and definition of security concepts and techniques. It is important to have a clearly defined vocabulary and standardized language as means to accurately communicate system vulnerability information and their countermeasures among all the people involved. We believe that semantic technology in general, and ontology in particular, could be a useful tool for system security. Our research work has turned our initial beliefs into data, pages, and reports, more of our work is to come.

An ontology is a specification of concepts and their relationships. Ontology represents knowledge in a formal and structured form. Therefore, ontology provides a better tool for communication, description, and organization of knowledge. Ontology is a knowledge representation (KR) system based on Description Logics (DL) [6], which is an umbrella name for a family of KR formalisms representing knowledge in various domains. DL formalisms specify a knowledge domain as the “world” by first defining the relevant concepts of the domain, and then it uses these concepts to specify properties of objects and individuals occurring in the domain [10-12]. Semantic technologies not only provide a tool for communication, but also a foundation for high-level reasoning and decision-making. Ontology, in particular, provides the potential of formal logic inference based on well-defined data and knowledge bases. Ontology captures the relationships between collected data and use the explicit knowledge of concepts and relationships to deduce the implicit and inherent knowledge. As a matter of fact, a heavy-weight ontology could be defined as a formal logic system, as it includes facts and rules, concepts, concept taxonomies, relationships, properties, axioms, and constraints.

A vulnerability is a security flaw, which arises from computer system design, implementation, maintenance, and operation. Research in the area of vulnerability analysis focuses on discovery of previously unknown vulnerabilities and quantification of the security of systems according to some metrics. Researchers at MITRE have provided a standard format for naming a security vulnerability, called Common Vulnerabilities and Exposures (CVE) [14], which assigns each vulnerability a unique identification number. We have designed a vulnerability ontology OVM (ontology for vulnerability management) populated with all existing vulnerabilities in NVD [2]. It supports research on measuring and quantification of vulnerabilities and their impact on computing systems. Vendors and users can use our ontology in support of vulnerability analysis, tool development and vulnerability management.

The rest of this paper is organized as follows. Section 2 presents the architecture of our OVM. Section 3 discusses how to populate the OVM with vulnerability instances from NVD and other
A Policy-Based Vulnerability Analysis Framework

By
SOPHIE JEAN ENGLE
B.S. (University of Nebraska at Omaha) 2002

DISSERTATION
Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY
in
Computer Science
in the
OFFICE OF GRADUATE STUDIES
of the
UNIVERSITY OF CALIFORNIA

APPVED

Professor Matt Bishop (Chair)
Professor S. Felix Wu
Professor Karl Levitt
Professor Sean Peisert
Committee in Charge
2010

Analysis-Based Verification: A Programmer-Oriented Approach to the Assurance of
Mechanical Program Properties

T. J. Halloran
May 27, 2010
CMU-ISR-10-112

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Thesis Committee:
William E. Scherlis (adviser)
James D. Herbold
Mary Shaw
Joshua J. Bloch, Google, Inc.

Copyright © 2010 T. J. Halloran

This material is based upon work supported by the following agencies: NASA, DARPA-1189 and NSA/DOE/NSSA, Lockheed Martin, SERVSITE, ARCS Foundation, and IBM. The views expressed in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the sponsor, the U.S. Government, or Carnegie Mellon University.
Linkage with Fundamental Changes in Enterprise Security Initiatives

Twenty Critical Controls for Effective Cyber Defense: Consensus Audit Guidelines

What the 20 CSC Critics say...

20 Critical Security Controls - Version 2.0
- 20 Critical Security Controls - Introduction (Version 2.0)
- Critical Control 1: Inventory of Authorized and Unauthorized Elements
- Critical Control 2: Inventory of Authorized and Unauthorized Elements
- Critical Control 3: Secure Configurations for Hardware and Software
- Critical Control 4: Secure Configurations for Network Devices
- Critical Control 5: Boundary Defense
- Critical Control 6: Maintenance, Monitoring, and Analysis of System Components
- Critical Control 7: Application Software Security
- Critical Control 8: Controlled Use of Administrative Privileges
- Critical Control 9: Controlled Access Based on Need to Know
- Critical Control 10: Operations Security
- Critical Control 11: Vulnerability Assessment and Management
- Critical Control 12: Incident Response
- Critical Control 13: Computer Incident Analysis
- Critical Control 14: Third-Party Support Management
- Critical Control 15: Information Systems Planning
- Critical Control 16: Access Control Management
- Critical Control 17: Client-Driven Applications
- Critical Control 18: Electronic Data Management
- Critical Control 19: Cybersecurity Management
- Critical Control 20: Executive Management

CWE and CAPEC included in Control 7 of the “Twenty Critical Controls for Effective Cyber Defense: Consensus Audit Guidelines”

How do attackers exploit the lack of this control?
Attacks against vulnerabilities in web-based and other application software have been a top priority for criminal organizations in recent years. Application software that does not properly check the size of user input, fails to sanitize user input by filtering out unneeded but potentially malicious character sequences, or does not initialize and clear variables properly could be vulnerable to remote compromise. Attackers can inject specific exploits, including buffer overflows, SQL injection attacks, and cross-site scripting code to gain control over vulnerable machines. In one attack in 2008, more than 1 million web servers were exploited and turned into infection engines for visitors to those sites using SQL injection. During that attack, trusted websites from state governments and other organizations compromised by attackers were used to infect hundreds of thousands of additional systems.

Procedures and tools for implementing this control...
Source code testing tools, web application security scanning tools, and object code testing tools have proven useful in securing application software, along with manual application security penetration testing by testers who have extensive programming knowledge as well as application penetration testing expertise. The Common Weakness Enumeration (CWE) initiative is utilized by many such tools to identify the weaknesses that they find. Organizations can also use CWE to determine which types of weaknesses they are most interested in addressing and removing. A broad community effort to identify the “Top 25 Most Dangerous Programming Errors” is also available as a minimum set of important issues to investigate and address during the application development process. When evaluating the effectiveness of testing for these weaknesses, the Common Attack Pattern Enumeration and Classification (CAPEC) can be used to organize and record the breadth of the testing for the CWEs as well as a way for testers to think like attackers in their development of test cases.
Linkage with Fundamental Changes in Enterprise Security Initiatives

Enabling Distributed Security in Cyberspace

Building a Healthy and Resilient Cyber Ecosystem with Automated Collective Action

CWE and CAPEC included in “Enabling Distributed Security in Cyberspace: Building a Healthy and Resilient Cyber Ecosystem with Automated Collective Action”
Questions?

ramartin@mitre.org