Advances in IA Standards

“Gaining Assurance”

Robert A. Martin, MITRE
Emile Monette, GSA
Dr. Paul Black, NI ST
Michael Kennedy, I SE DNI
Don Davidson, Office of DoD CIO
Today’s Reality – Requires confidence in our software-based cyber technologies

- Dependencies on technology are greater than ever
- Possibility of disruption is greater because hardware/software is vulnerable
- Loss of confidence alone can lead to stakeholder actions that disrupt critical business activities
Everything’s Connected

When this Other System gets subverted through an un-patched vulnerability, a mis-configuration, or an application weakness…

Your System is attackable…
Assurance: Mitigating Attacks That Impact Operations

Known Threat Actors

Attack Patterns (CAPECs)

Weaknesses (CWEs)

Controls*

System & System Security Engineering Trades

Asset

Function

Impact

Operational Impacts

* Controls include architecture choices, design choices, added security functions, activities & processes, physical decomposition choices, code assessments, design reviews, dynamic testing, and pen testing
Assurance on the Management of Weaknesses

Manage Risk During Development

- Attack Surface Analysis/Threat Modeling
- Operational Mitigation

Assess Deployment Risk

- Block from Attack
- Validate/Verify

Using similar information to Function Point calculations
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Description
The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.

Extended Description
Without sufficient removal or quoting of SQL syntax in user-controllable inputs, the generated SQL query can cause those inputs to be interpreted as SQL instead of ordinary user data. This can be used to alter query logic to bypass security checks, or to insert additional statements that modify the back-end database, possibly including execution of system commands.

SQL injection has become a common issue with database-driven web sites. The flaw is easily detected, and easily exploited, and as such, any site or software package with even a minimal user base is likely to be subject to an attempted attack of this kind. This flaw depends on the fact that SQL makes no real distinction between the control and data planes.

Time of Introduction
- Architecture and Design
- Implementation
- Operation

Applicable Platforms
All

Languages
Database-Server
For DoD Software Assurance is defined by Public Law 113-239 “Section 933 - Software Assurance”

Software Assurance.—The term “software assurance” means the level of confidence that software functions as intended and is free of vulnerabilities, either intentionally or unintentionally designed or inserted as part of the software, throughout the life cycle.

Sect933
DoD Program Protection Plan (PPP) Software Assurance Methods

Development Process
Apply assurance activities to the procedures and structure imposed on software development

Operational System
Implement countermeasures to the design and acquisition of end-item software products and their interfaces

Development Environment
Apply assurance activities to the environment and tools for developing, testing, and integrating software code and interfaces

Additional Guidance in PPP Outline and Guidance
13.7.3. Software Assurance
13.7.3.1. Development Process
13.7.3.1.1 Static Analysis
13.7.3.1.2 Design Inspection
13.7.3.1.3 Code Inspection
13.7.3.1.4. Common Vulnerabilities and Exposures (CVE)
13.7.3.1.5. Common Attack Pattern Enumeration and Classification (CAPEC)
13.7.3.1.6. Common Weakness Enumeration Information (CWE)
13.7.3.1.7. Penetration Test
13.7.3.1.8. Test Coverage
13.7.3.2. Operational System
13.7.3.2.1. Failover Multiple Supplier Redundancy
13.7.3.2.2. Fault Isolation
13.7.3.2.3. Least Privilege
13.7.3.2.4. System Element Isolation
13.7.3.2.5. Input Checking/Validation
13.7.3.3. Development Environment
13.7.3.3.1. Source Code Availability
13.7.3.3.2. Release Testing
13.7.3.3.3. Generated Code Inspection
13.7.3.3.4. Additional Countermeasures
4. VULNERABILITY AND WEAKNESS ASSESSMENT

Purpose and Use

FY 2013
Chief Information Officer
Federal Information Security Management Act
Reporting Metrics

Prepared by:
US Department of Homeland Security
Office of Cybersecurity and Communications
Federal Network Resilience

November 30, 2012

For systems in development and/or maintenance:

Use methods described in Table 9 to identify and fix instances of common weaknesses, prior to placing that version of the code into production. Can the organization find SCAP compliant tools and good SCAP content?

Impact Level
High
Moderate
Low

Table 8 – Responses to Question 4.3

<table>
<thead>
<tr>
<th>Identify Universe Enumeration</th>
<th>Find Instances Tools and Languages</th>
<th>Assess Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Weakness Enumeration (CWE)</td>
<td>Static Code Analysis tools</td>
<td>Common Weakness Scoring System (CWSS)</td>
</tr>
<tr>
<td>Web scanners for web-based applications</td>
<td>Manual code reviews (especially for weaknesses not covered by the automated tools)</td>
<td></td>
</tr>
<tr>
<td>Common Attack Pattern Enumeration and Classification (CAPEC)</td>
<td>Dynamic Code Analysis tools</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Web scanners for web-based applications</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PEN testing for attack types not covered by the automated tools</td>
<td></td>
</tr>
</tbody>
</table>

See guidance that describes the purpose and use of these tools and how they can be used today in a practical way to improve security of software during development and maintenance.
Industry Uptake

The paper also contains two important, additional sections for each listed practice that will further increase its value to implementers—Common Weakness Enumeration (CWE) references and Verification guidance.

CWE References:
- CWE-89: Improper authentication is an example of weakness that could be exploited by a Spoofing threat.
- CWE-264: Permissions, Privileges, and Access Control is a parent weakness of many Tampering, Repudiation and Elevation of Privilege threats.
- CWE-339: Missing Encryption of Sensitive Data is an example of an Information Disclosure threat.
- CWE-400: Uncontrolled resource consumption is one example of an Unintended Use of Service threat.

Verification:
- The verification plan is a directive of the Threat Model and Threat Model itself will serve as a clear guide for verification, containing enough information to confirm the correctness of the threat model and mitigation can be verified.
- During verification, the Threat Model’s no mitigated threats, as well as the annotated threat diagram, should also be made available to the developer to help identify further steps and verify the verification process.
- A walk-through of the Threat Model and verification results should be made integral part of the activities required to complete the verification.

An example of a portion of a test plan derived from a Threat Model could be:

<table>
<thead>
<tr>
<th>Threat Identified</th>
<th>Design Element(s)</th>
<th>Mitigation</th>
<th>Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spoofing threat</td>
<td>GUI</td>
<td>Ensure random session identifiers of appropriate length</td>
<td>Collect session identifiers over a number of sessions and examine distribution and length</td>
</tr>
<tr>
<td>Process A on server to Process B on client</td>
<td>Live SSL to ensure that data isn’t modified in transit</td>
<td>Assert that communication cannot be established without the use of SSL</td>
<td></td>
</tr>
</tbody>
</table>

![CWE References](image-url)
## Industry Uptake

### Agile

<table>
<thead>
<tr>
<th>No.</th>
<th>Security-focused story</th>
<th>Backlog task(s)</th>
<th>SAFECODE Fundamental Practices</th>
<th>CWE-ID</th>
</tr>
</thead>
</table>
| 1   | As an architect/developer, I want to ensure AND as QA, I want to verify allocation of resources within limits or throttling. | [A] Clearly identify resources. A few examples:  
- Number of simultaneous connections to an application on a web server from same user or from different users  
- File size that can be uploaded  
- Maximum number of files that can be uploaded to a file system folder  
[A/D] Define limits on resource allocation.  
[T] Conduct performance/stress testing to ensure that the numbers chosen are realistic (i.e. backed by data).  
[A/D/T] Define and test system behavior for correctness when limits are exceeded. A few examples:  
- Rejecting new connection requests  
- Preventing simultaneous connection requests from the same user/IP, etc.  
- Preventing users from uploading files greater than a specific size, e.g., 2 MB  
- Archiving data in file upload folder when a specific limit is reached to prevent file system exhaustion. | - Validate Input and Output to Mitigate Common Vulnerabilities  
- Perform Fuzz/Robustness Testing | CWE-770 |
<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></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>
# Injection

## Application Specific

- **Threat Agents**: Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.

- **Attack Vectors**: Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.

- **Prevalence**: Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, XPath, or NoSQL queries, OS commands, XML parsers, SMTP headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.

- **Detectability**: Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.

## Business Specific

- **Impact**: Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?

## How Do I Prevent Injection?

- **References**: OWASP SQL Injection Prevention Cheat Sheet, OWASP Query Parameterization Cheat Sheet, OWASP Command Injection Article, OWASP XML eXtension Entity (XXE) Reference Article, ASVS: Output Encoding/Escaping Requirements (V2)

## Example Attack Scenarios

**Scenario #1**: The application uses untrusted data in the construction of the following vulnerable SQL call:

```
String query = "SELECT * FROM accounts WHERE custId = " + request.getParameter("id") + "";
```

**Scenario #2**: Similarly, an application’s blind trust in frameworks may result in queries that are still vulnerable, e.g., Hibernate Query Language (HQL):

```
Query query = session.createQuery("FROM accounts WHERE custId = " + request.getParameter("id") + "";
```

In both cases, the attacker modifies the 'id' parameter value in her browser to send: `1 OR 1 = 1` or `1 = 1`. For example:

```
http://example.com/app/accountView?id=1 OR 1 = 1
```

This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

- **OWASPEC Entry 77 on Command Injection**
- **OWASPEC Entry 89 on SQL Injection**
- **OWASPEC Entry 164 on Hibernate Injection**

---

**Am I Vulnerable To Injection?**

The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries.

Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability.

Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

**Common Consequences**

- **Confidentiality**
  - *Technical Impact*: Read application data
  - Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.

- **Access Control**
  - *Technical Impact*: Bypass protection mechanism
  - If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password.

- **Integrity**
  - *Technical Impact*: Modify application data
  - If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability.
  - Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.
CWE’s all lead to these Technical Impacts

1. Modify data
2. Read data
3. DoS: unreliable execution
4. DoS: resource consumption
5. Execute unauthorized code or commands
6. Gain privileges / assume identity
7. Bypass protection mechanism
8. Hide activities
Which static analysis tools and Pen Testing services find the CWEs I care about?
Scoring Weaknesses Discovered in Code

Steps:
1. Establish weightings for the vignette
2. Run code through analysis tool(s)
3. Tools produce report of CWE’s found in code
4. CWSS scoring engine automatically scores each CWE based on vignette definition
5. Go to step 2 for each piece of code applicable to this vignette

Line 212: CWE-9: 9.9
Line 72: CWE-84: 7.9
Line 23: CWE-109: 5.6
Line 104: CWE-482: 3.1
Line 213: CWE-754: 0.0

Step 1 is only done once – the rest is automatic
Assurance & the Systems Dev. Life-Cycle

Cyber Threat/Attack Analysis

Technology Development and Systems Design
- Attack Analysis against Supply Chain & Application Architecture Security Review

System Development & Demonstration
- Design Readiness Review
- Attack-based Application Design Security Review

Production & Deployment
- Application Security Code Review (developed and purchased), Penetration Testing & Abuse Case Driven Testing

Operations & Support
- Gather All of the Evidence for the Assurance Case and Get It Approved

* Ideally Insert SwA before RFP release in Analysis of Alternatives

Application Security Code Review, Penetration Testing & Abuse Case Driven Testing of Maintenance Updates

Pre-Systems Acquisition

Sustainment
Different assessment methods are effective at finding different types of weaknesses

Some are good at finding the cause and some at finding the effect

<table>
<thead>
<tr>
<th></th>
<th>Static Code Analysis</th>
<th>Penetration Test</th>
<th>Data Security Analysis</th>
<th>Code Review</th>
<th>Architecture Risk Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cross-Site Scripting (XSS)</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>SQL Injection</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Insufficient Authorization Controls</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Broken Authentication and Session Management</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Information Leakage</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Improper Error Handling</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Insecure Use of Cryptography</td>
<td>X</td>
<td></td>
<td></td>
<td>X X</td>
<td></td>
</tr>
<tr>
<td>Cross Site Request Forgery (CSRF)</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Denial of Service</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Poor Coding Practices</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

**Detection Methods**

**Automated Static Analysis**

This weakness can often be detected using automated static analysis tools. Many modern tools use data flow analysis or constraint-based techniques to minimize the number of false positives.

Automated static analysis might not be able to recognize when proper input validation is being performed, leading to false positives - i.e., warnings that do not have any security consequences or do not require any code changes.

Automated static analysis might not be able to detect the usage of custom API functions or third-party libraries that indirectly invoke SQL commands, leading to false negatives - especially if the API/library code is not available for analysis.

This is not a perfect solution, since 100% accuracy and coverage are not feasible.

**Automated Dynamic Analysis**

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

**Manual Analysis**

Manual analysis can be useful for finding this weakness, but it might not achieve desired code coverage within limited time constraints. This becomes difficult for weaknesses that must be considered for all inputs, since the attack surface can be too large.

**Example 1**

In 2008, a large number of web servers were compromised using the same SQL injection attack string. This single...
**Detection Methods**

The "Detection Methods" field within many CWE entries conveys information about what types of assessment activities that weakness can be found by. Increasing numbers of CWE entries will have this field filled in over time. The recent Institute of Defense Analysis (IDA) State of the Art Research report conducted for DoD provides additional information for use across CWE in this area. Labels for the Detection Methods being used within CWE are:

- Automated Analysis
- Automated Dynamic Analysis
- Automated Static Analysis
- Black Box
- Fuzzing
- Manual Analysis
- Manual Dynamic Analysis
- Manual Static Analysis
- White Box

With this type of information (shown in the table below), we can see which of the specific CWEs that can lead to a specific type of technical impact are detectable by dynamic analysis, static analysis, and fuzzing evidence and which ones are not.

This table is incomplete, because many CWE entries do not have a detection method listed.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute unauthorized code or commands</td>
<td>78, 120, 129, 131, 476, 805</td>
<td>78, 79, 98, 120, 129, 131, 134, 426, 798, 805</td>
<td>79, 129, 134, 190, 426, 494, 698, 798</td>
<td>98, 120, 131, 190, 426, 494, 805</td>
<td>476, 798</td>
<td>78, 798</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify data</td>
<td>311, 327</td>
<td>78, 89, 129, 131</td>
<td>78, 89, 129, 131, 190, 352</td>
<td>129, 190, 319</td>
<td>89, 131, 190, 311, 78</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------------</td>
<td>--------------------</td>
<td>---------------------------</td>
<td>--------------------------</td>
<td>-----------</td>
<td>---------</td>
<td>----------------</td>
<td>--------------------------</td>
<td>------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>Execute unauthorized code or commands</td>
<td>78, 120, 129, 131, 476, 805</td>
<td>78, 79, 98, 120, 129, 131, 134, 190, 426, 798</td>
<td>79, 129, 134, 190, 426, 494, 698, 798</td>
<td>79, 129, 134, 190, 426, 494, 698, 798</td>
<td>98, 120, 131, 190, 426, 494, 805</td>
<td>476, 798</td>
<td>78, 798</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Modify data</td>
<td>311, 327</td>
<td>78, 89, 129, 131</td>
<td>78, 89, 129, 131, 190, 352</td>
<td>129, 190, 319</td>
<td>89, 131, 190, 311, 327, 352</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DoS: unreliable execution</td>
<td>78, 120, 129, 131, 400, 476, 665, 805</td>
<td>78, 120, 129, 131, 190, 352, 400, 426, 805</td>
<td>129, 190, 426, 690</td>
<td>129, 190, 426, 690</td>
<td>400</td>
<td>120, 131, 190, 805</td>
<td>476, 665</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>DoS: resource consumption</td>
<td>120, 400, 404, 770, 805</td>
<td>120, 190, 400, 770, 805</td>
<td>190</td>
<td>400, 770</td>
<td>120, 190, 805</td>
<td>404</td>
<td>770</td>
<td>412</td>
<td></td>
</tr>
<tr>
<td>Bypass protection mechanism</td>
<td>89, 400, 601, 665</td>
<td>79, 89, 190, 352, 400, 601, 798</td>
<td>14, 79, 184, 190, 733, 798</td>
<td>14, 79, 184, 190, 733, 798</td>
<td>89, 190, 352</td>
<td>665, 798</td>
<td>601, 798, 807</td>
<td>14, 733</td>
<td></td>
</tr>
<tr>
<td>Hide activities</td>
<td>327</td>
<td>78</td>
<td>78</td>
<td>327</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CWE will leverage the “State of the Art Resource” (SOAR): Software Table of “Verification Methods”

<table>
<thead>
<tr>
<th>IDA</th>
<th>Description</th>
<th>Verification Method</th>
<th>Security</th>
<th>Policy</th>
<th>Compliance</th>
<th>Monitoring</th>
<th>Testing</th>
<th>Training</th>
<th>Emerging Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Method 1</td>
<td>Method 1</td>
<td>Security</td>
<td>Policy</td>
<td>Compliance</td>
<td>Monitoring</td>
<td>Testing</td>
<td>Training</td>
<td>Emerging Tools</td>
</tr>
<tr>
<td>2</td>
<td>Method 2</td>
<td>Method 2</td>
<td>Security</td>
<td>Policy</td>
<td>Compliance</td>
<td>Monitoring</td>
<td>Testing</td>
<td>Training</td>
<td>Emerging Tools</td>
</tr>
<tr>
<td>3</td>
<td>Method 3</td>
<td>Method 3</td>
<td>Security</td>
<td>Policy</td>
<td>Compliance</td>
<td>Monitoring</td>
<td>Testing</td>
<td>Training</td>
<td>Emerging Tools</td>
</tr>
</tbody>
</table>
Getting Started in Software Assurance (SwA)

Success of the mission should be the focus of software and other assurance activities. Although increasing automation of various capabilities has provided great boons to our organizations, this automation is also at risk for becoming a targeted focus for attackers’ attentions and techniques. Recognizing that your software and supply chain have exploitable weaknesses is a major step to improving the reliability, resilience, and integrity of your software when it faces attacks.

The key to gaining assurance about your software is to make incremental improvements when you develop it, when you buy it, and when others create it for you. No single remedy will absolve or mitigate all of the weaknesses in your software, or the risk. However, by blending several different methods, tools, and change in culture, one can obtain greater confidence that the important functions of the software will be there when they are needed and the worst types of failures and impacts can be avoided.

There is no crystal ball, or magic wand one can use to ensure software is absolutely secure against the unknown. However, there are ways to limit negative impacts and improve confidence in software-based capabilities and their ability to deliver their part to the organization’s mission.

This section of the CWE Web site introduces specific steps you can take to 1) assess your individual software assurance situation and 2) compose a tailored plan to strengthen assurance of integrity, reliability, and resilience of your software and its supply chain. Learn more by following the links below:

- Engineering for Attacks
- Software Quality
- Prioritizing Weaknesses Based Upon Your Organization’s Mission
- Detection Methods
- Manageable Steps
- Software Assurance Pocket Guide Series
- Staying Informed
- Finding More Information about Software Assurance
CISQ Security Measure

Objective
Develop automated source code measures that predict the vulnerability of source code to external attack. Measure based on the Top 25 in the Common Weakness Enumeration.
Measuring Security by Violated Rules

Structure of ISO 25023 Measures

- Software Quality Characteristics
  - Quality Sub-Characteristics
    - Software Quality Attributes

Structure of CISQ Security Measure

- Security
  - Confidentiality, Authenticity, Integrity, Accountability, etc.
  - Quality Measure Elements
    - Quality Rule Violations
      - Cross-site scripting
      - SQL injection
      - Buffer overflow
      - OS command injection
      - Unvalidated array
      - Etc.
<table>
<thead>
<tr>
<th>Issue</th>
<th>Quality Rule</th>
<th>Quality Measure Element</th>
</tr>
</thead>
<tbody>
<tr>
<td>CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')</td>
<td>Rule 1: Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid, such as Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.</td>
<td>Measure 1: # of instances where output is not using library for neutralization</td>
</tr>
<tr>
<td>CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')</td>
<td>Rule 2: Use a vetted library or framework that does not allow SQL injection to occur or provides constructs that make this SQL injection easier to avoid or use persistence layers such as Hibernate or Enterprise Java Beans.</td>
<td>Measure 2: # of instances where data is included in SQL statements that is not passed through the neutralization routines.</td>
</tr>
</tbody>
</table>

CISQ measure aggregates violations of 19 of the CWE Top 25: 79, 89, 22, 434, 78, 798, 706, 129, 754, 131, 327, 456, 672, 834, 681, 667, 772, 119
Non-Malicious Taint: Bad Hygiene and a Danger to the Mission as Malicious Intent

Robert A. Martin, MITRE Corporation

Abstract. Success of the mission should be the focus of software and supply chain assurance activities regardless of what activity produces the risk. It does not matter if it is a malicious act or the cause. It does not matter if it is malicious logic inserted at the factory or inserted through an update after delivery. It does not matter if it comes from an error in judgment or from a failure to understand how an attacker could exploit a software feature. Issues like software hygiene, like independence code flaws or weak architectural constructs are as serious as the malicious acts. Enormous energies are put into hygiene and quality in the medical and food industries to address any source of tainted. Similar energies need to be applied to software and hardware. Until both malicious and non-malicious aspects of taint can be dealt with in ways that are visible and verifiable, there will be a continued lack of confidence and assurance in delivered capabilities throughout their lifecycle.

Background

Every piece of information and communications technology (ICT) hardware—this includes computers as well as any device that stores, processes, or transmits data—has an internally embedded software component that requires follow-on support and sustainment throughout the equipment’s lifecycle.

The concept of supply chain risk management (SCRM) must be applied both to soft and hardware components within the ICT. Because of the way ICT hardware items are manufactured, the supply chain is often disconnected from the support for the hardware. This disconnect makes software hygiene, like independence code flaws or weak architectural constructs are as serious as the malicious acts. Similar energies need to be applied to software and hardware. Until both malicious and non-malicious aspects of taint can be dealt with in ways that are visible and verifiable, there will be a continued lack of confidence and assurance in delivered capabilities throughout their lifecycle.

Some may want to focus on just “low hanging fruit” like banning suspect products by the country they come from or the ownership of the producer due to their focused nature and ignore more critical issues surrounding the software aspect of ICT like the exploitable vulnerabilities outlined in this article. It is a misdirection that “adding” software assurance to the mix of supply chain concerns and activities will add too much complexity.

The need to address these differences has accelerated as more of the nation’s critical industrial, financial, and military capabilities rely on cyber-space and the need to validate their software assurance as a fundamental aspect of SCRM for ICT equipment is a critical component of delivering mission assurance.

While there is no concrete definition of what “taint” specifically means within the cyber realm, we would be remiss not to look to the general use of the term, as well as synonyms and antonyms. Merriam Webster [3] provides a useful point of departure, as shown in Table 1 below.
Mission Fulfillment

Exploitable Weakness #1 (a vulnerability)

Exploitable Weakness #2 (a vulnerability)

Impact from Weakness #1

Mission

Mission Fulfillment

Impact from Weakness #3

Weakness #3
Many Capabilities Support the Mission

Supply Chain Activities

Capability

People

Capability

Systems

Capability

Software

Capability

Chips

THE GLOBAL STANDARD FOR SOFTWARE QUALITY
CWE-937: OWASP Top Ten 2013 Category A9 - Using Components with Known Vulnerabilities

<table>
<thead>
<tr>
<th>Nature</th>
<th>Type</th>
<th>ID</th>
<th>Name</th>
<th>Description</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemberOf</td>
<td></td>
<td>928</td>
<td>Weaknesses in OWASP Top Ten (2013)</td>
<td>Weaknesses in this category are related to the A9 category in the OWASP Top Ten 2013.</td>
<td>Incomplete</td>
</tr>
</tbody>
</table>

**Relationship Notes**

This is an unusual category. CWE does not cover the limitations of human processes and procedures that cannot be described in terms of a specific technical weakness as resident in the code, architecture, or configuration of the software. Since "known vulnerabilities" can arise from any kind of weakness, it is not possible to map this OWASP category to other CWE entries, since it would effectively require mapping this category to ALL weaknesses.

**References**


**Submissions**

<table>
<thead>
<tr>
<th>Submission Date</th>
<th>Submitter</th>
<th>Organization</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>2013-07-16</td>
<td>MITRE</td>
<td>Internal CWE Team</td>
<td></td>
</tr>
</tbody>
</table>
Assurance on the Management of Weaknesses

- **CWE**
- **CVE**
- **CAPEC**

Eliminate Mitigate

Block from Attack Alarm for Attack/Exploit

Threat ➔ Threat Vector ➔ Weakness ➔ Vulnerability ➔ Control ➔ Implementation ➔ Test