CISQ

CONSORTIUM FOR IT SOFTWARE QUALITY

Advances in IA Standards

“Gaining Assurance”

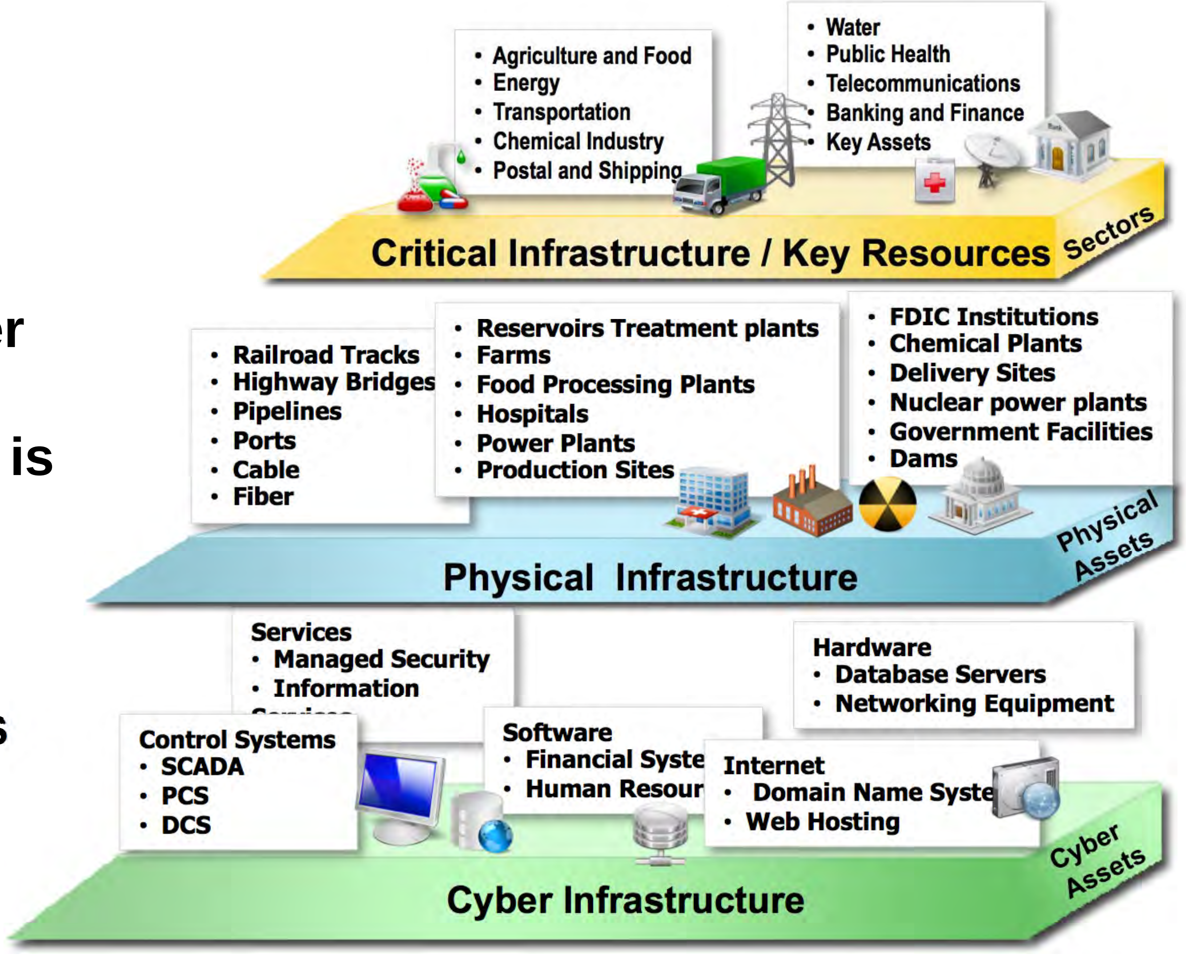
Robert A. Martin MITRE
Emile Monette, GSA
Dr. Paul Black, NIST
Michael Kennedy, ISE DNI
Don Davidson, Office of DoD CIO

Software Engineering Institute | Carnegie Mellon

OMG
OBJECT MANAGEMENT GROUP

Today's Reality – Requires confidence in our software-based cyber technologies

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



- Agriculture and Food
- Energy
- Transportation
- Chemical Industry
- Postal and Shipping

- Water
- Public Health
- Telecommunications
- Banking and Finance
- Key Assets

Critical Infrastructure / Key Resources Sectors

- Railroad Tracks
- Highway Bridges
- Pipelines
- Ports
- Cable
- Fiber

- Reservoirs Treatment plants
- Farms
- Food Processing Plants
- Hospitals
- Power Plants
- Production Sites

- FDIC Institutions
- Chemical Plants
- Delivery Sites
- Nuclear power plants
- Government Facilities
- Dams

Physical Infrastructure Physical Assets

Services
- Managed Security
- Information

Control Systems
- SCADA
- PCS
- DCS

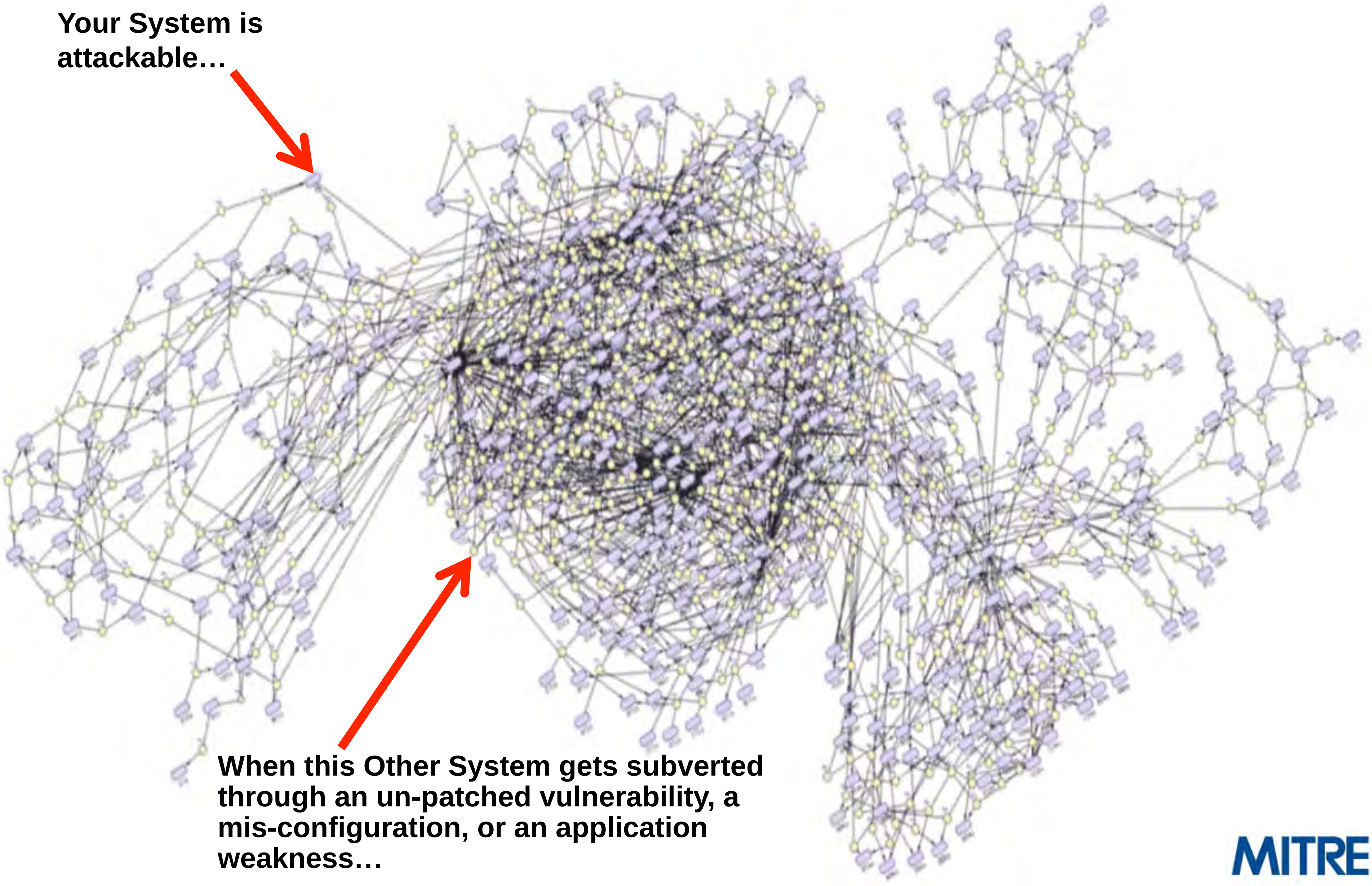
Software
- Financial Syste
- Human Resour

Hardware
- Database Servers
- Networking Equipment

Internet
- Domain Name Syste
- Web Hosting

Cyber Infrastructure Cyber Assets

Everything's Connected

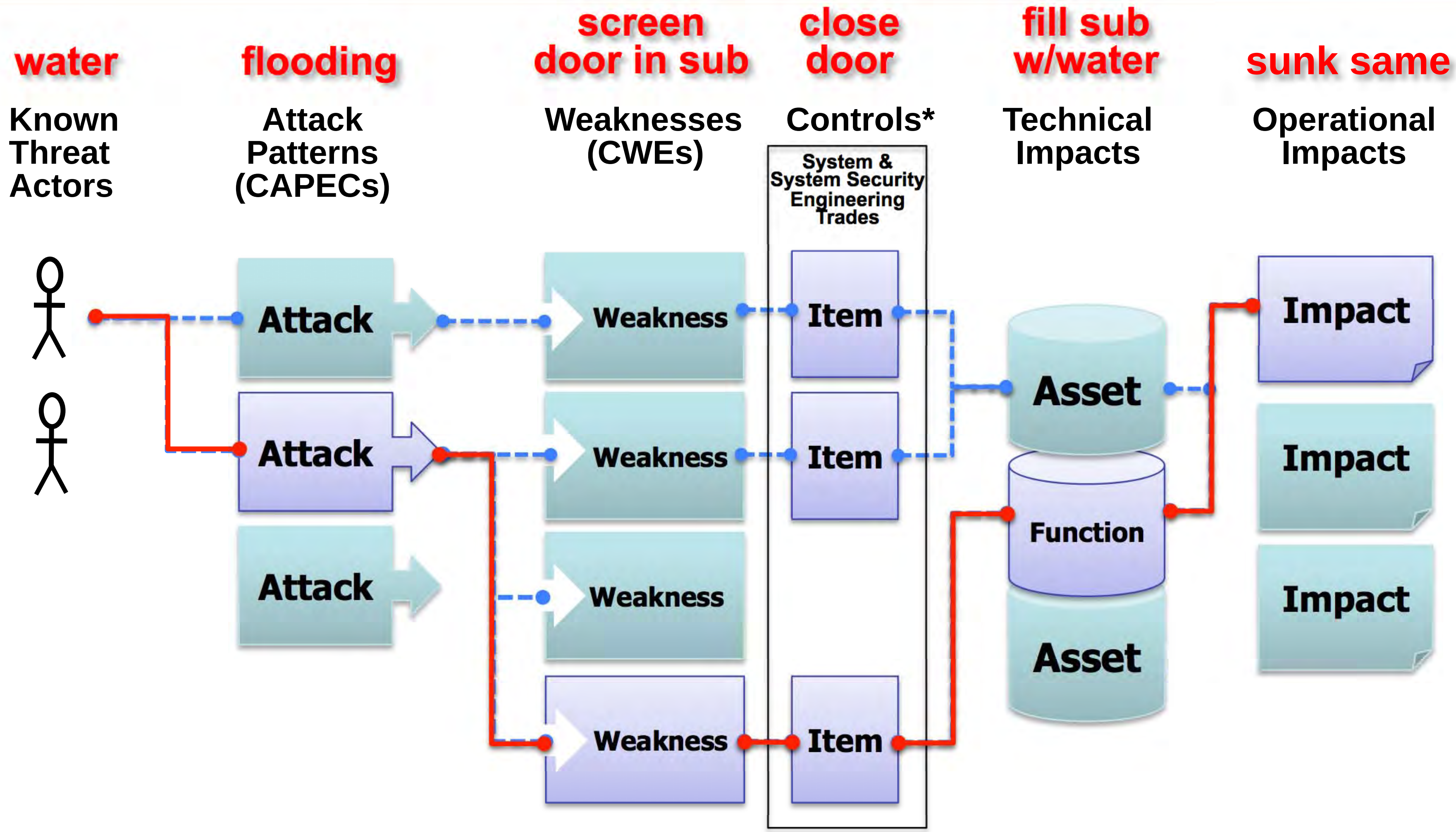


Your System is attackable…

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

CISQ

Assurance: Mitigating Attacks That Impact Operations



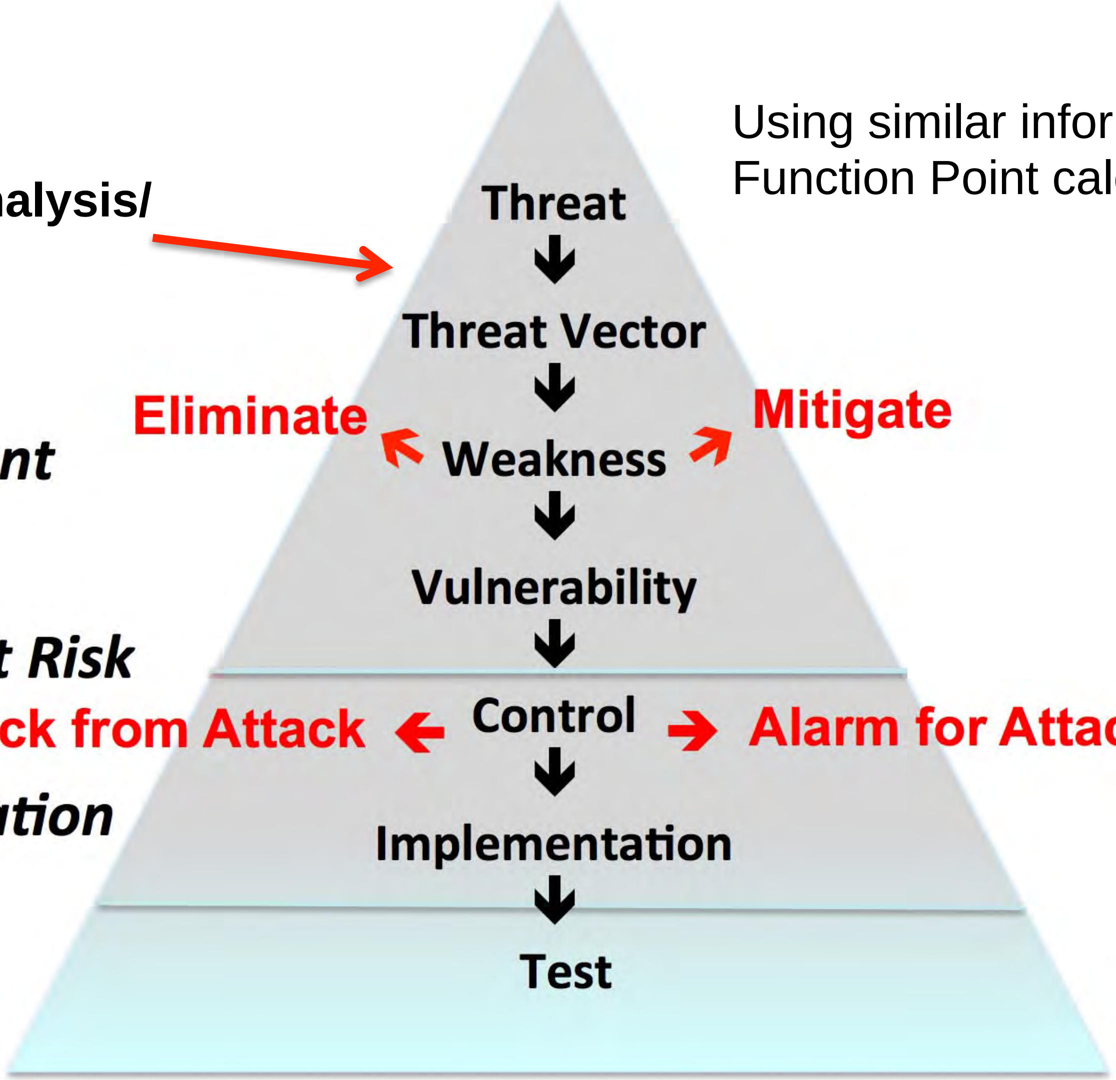
water	flooding	screen door in sub	close door	fill sub w/water	sunk same
Known Threat Actors	Attack Patterns (CAPECs)	Weaknesses (CWEs)	Controls*	Technical Impacts	Operational Impacts

* Controls include architecture choices, design choices, added security functions, activities & processes, physical decomposition choices, code assessments, design reviews, dynamic testing, and pen testing

MITRE

CISQ

Assurance on the Management of Weaknesses



Using similar information to Function Point calculations

Attack Surface Analysis/ Threat Modeling →

Threat

↓

Threat Vector

↓

Eliminate ← **Weakness** → **Mitigate**

↓

Vulnerability

↓

Control

Block from Attack ← **Control** → **Alarm for Attack/Exploit**

↓

Implementation

↓

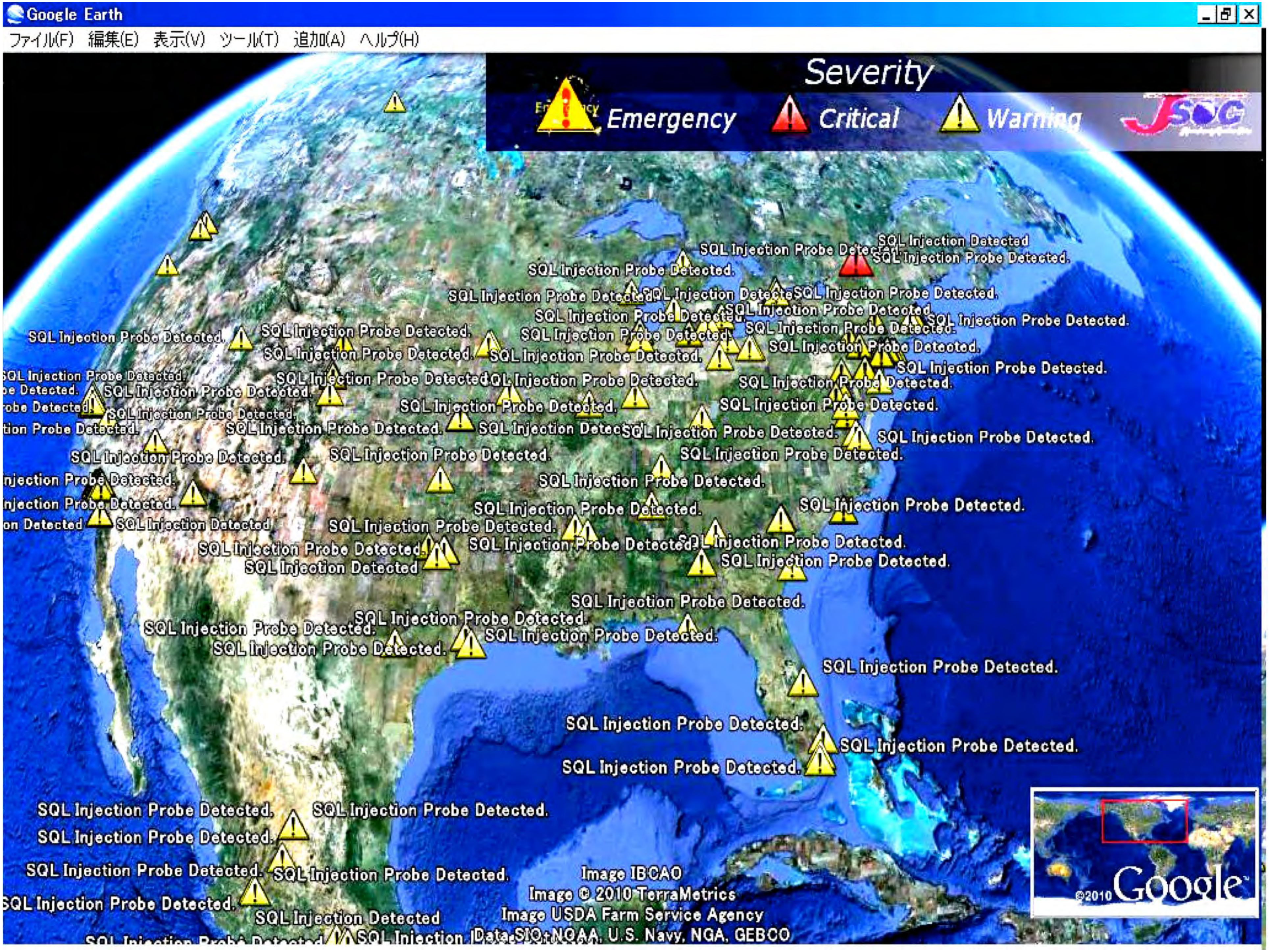
Test

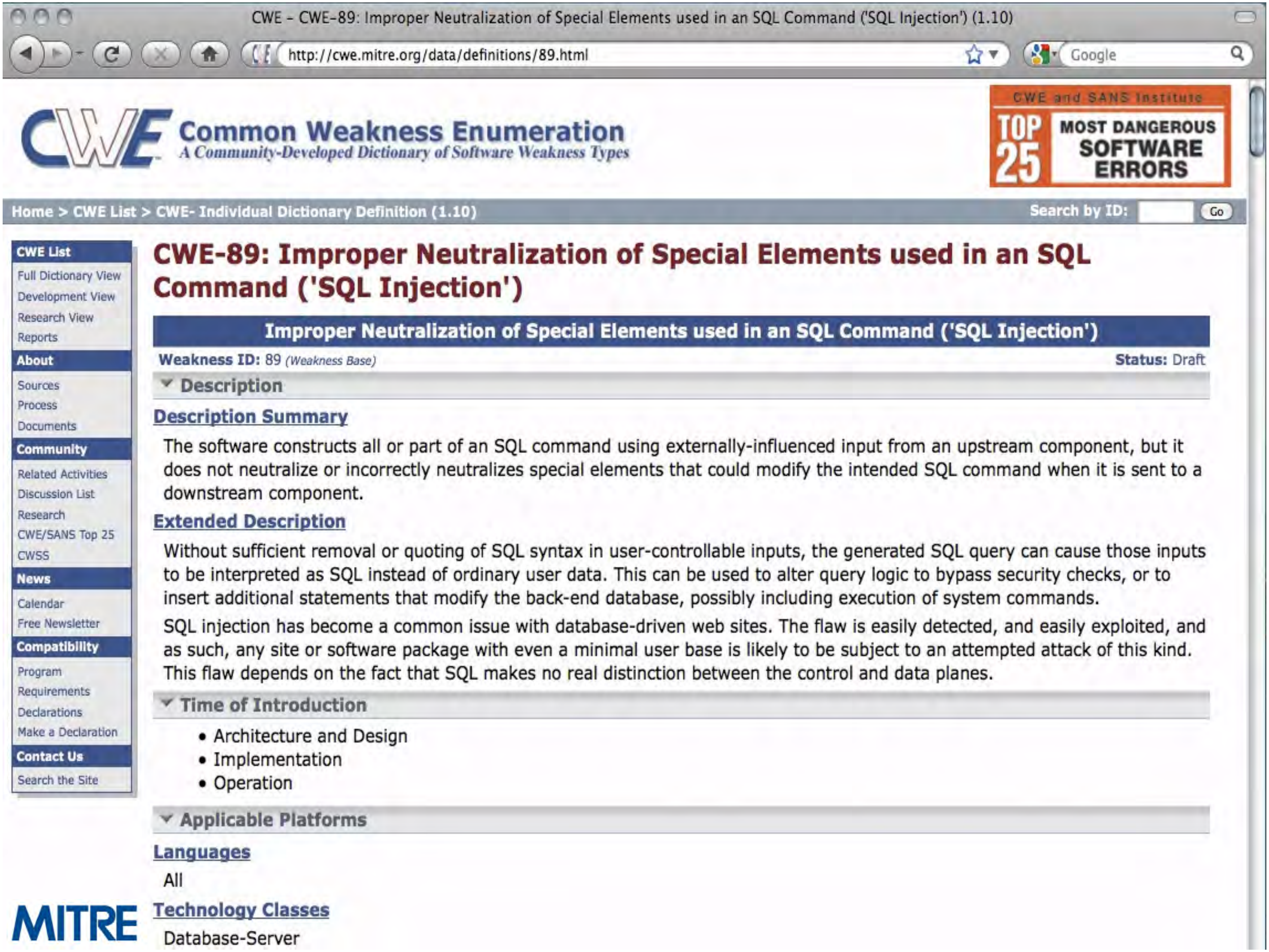
Manage Risk During Development

Assess Deployment Risk

Operational Mitigation

Validate/Verify


Google Earth
ファイル(F) 編集(E) 表示(V) ツール(T) 追加(A) ヘルプ(H)
Severity
Emergency
Critical
Warning
JSOC
SQL Injection Detected
SQL Injection Probe Detected
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected
SQL Injection Detected
SQL Injection Probe Detected.
SQL Injection Probe Detected
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
be Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected
QL Injection Probe Detected.
SQL Injection Probe Detected.
robe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
tion Probe Detected.
SQL Injection Probe Detected.
SQL Injection Detec
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
njection Probe Detected.
SQL Injection Probe Detected.
njection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
on Detected
SQL Injection Detected
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected
SQL Injection Probe Detected.
SQL Injection Detected
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
SQL Injection Probe Detected.
QL Injection Probe Detected.
SQL Injection Detected
SQL Injection Probe Detected
SQL Injection
Image IBCAO
Image © 2010 TerraMetrics
Image USDA Farm Service Agency
Data SIO, NOAA, U.S. Navy, NGA, GEBCO
©2010 Google



CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Weakness ID: 89 *(Weakness Base)* **Status:** Draft

Description

Description Summary

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

Languages

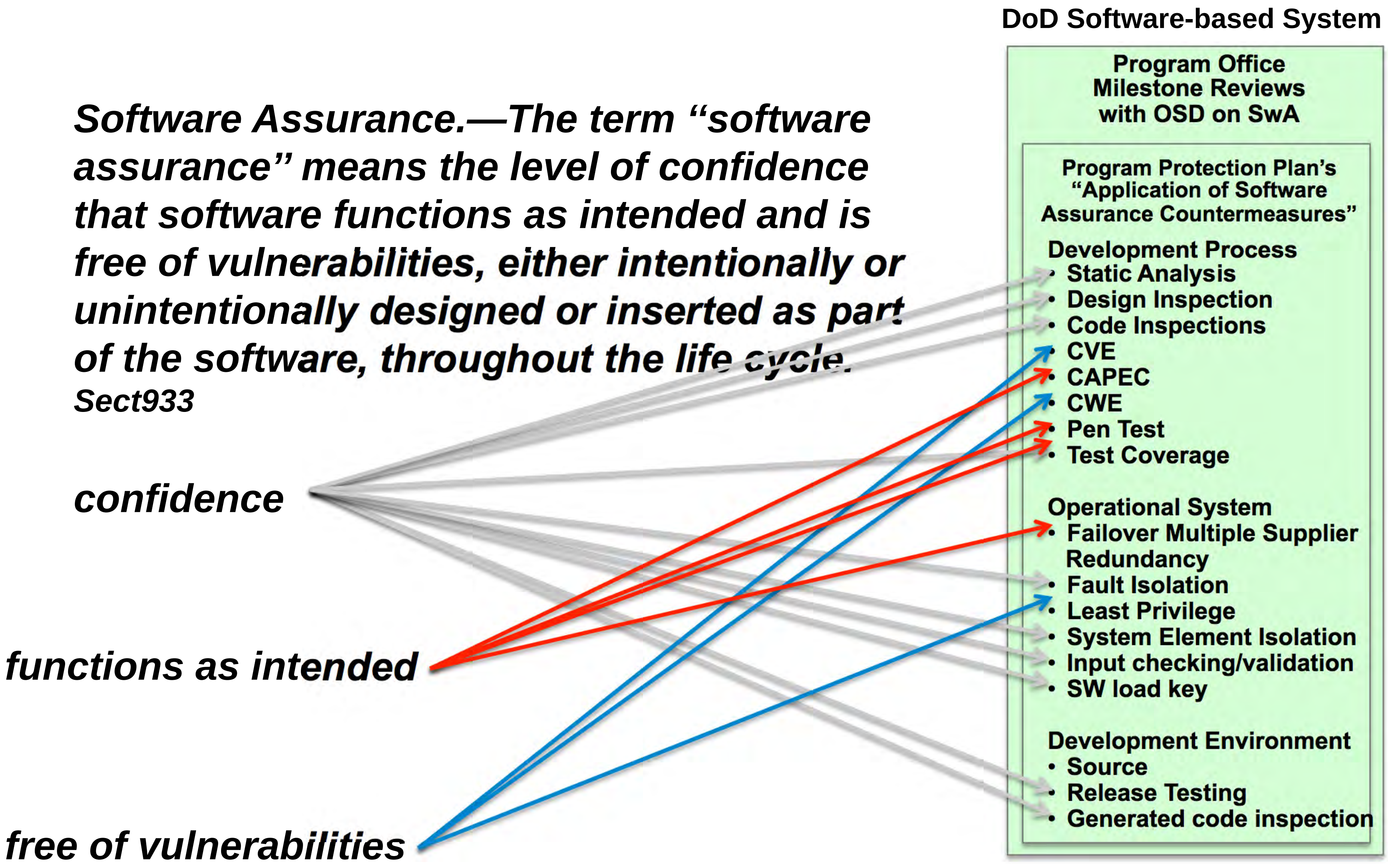
All

Technology Classes

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



confidence

functions as intended

free of vulnerabilities

DoD Software-based System

Program Office
Milestone Reviews
with OSD on SwA

Program Protection Plan's
"Application of Software
Assurance Countermeasures"

Development Process
- Static Analysis
- Design Inspection
- Code Inspections
- CVE
- CAPEC
- CWE
- Pen Test
- Test Coverage

Operational System
- Failover Multiple Supplier Redundancy
- Fault Isolation
- Least Privilege
- System Element Isolation
- Input checking/validation
- SW load key

Development Environment
- Source
- Release Testing
- Generated code inspection

DoD Program Protection Plan (PPP) Software Assurance Methods

Countermeasure Selection

Development Process

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

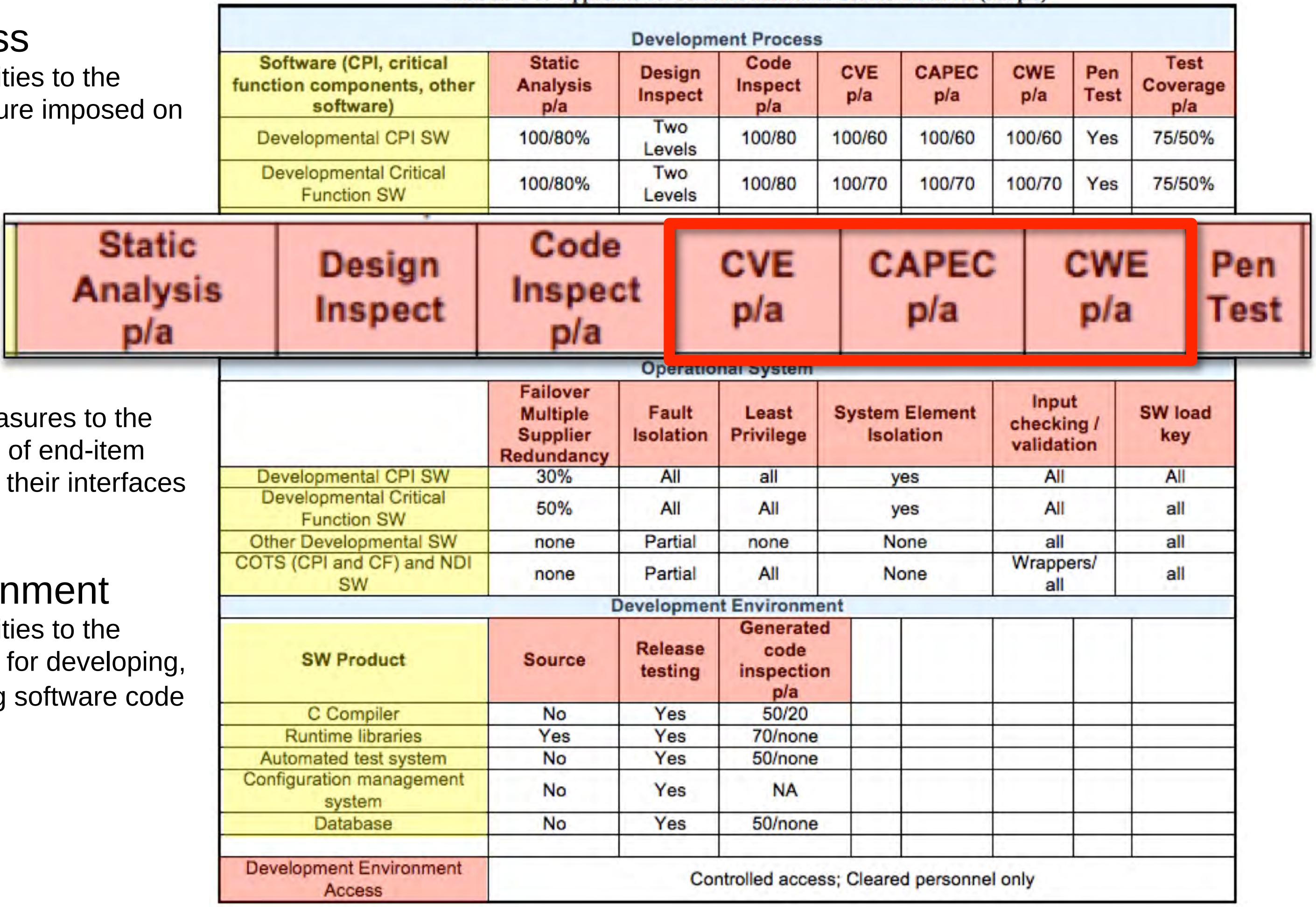
Operational System

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

Table 5.3-5-5: Application of Software Assurance Countermeasures (sample)



Development Process

Software (CPI, critical function components, other software)	Static Analysis p/a	Design Inspect	Code Inspect p/a	CVE p/a	CAPEC p/a	CWE p/a	Pen Test	Test Coverage p/a
Developmental CPI SW	100/80%	Two Levels	100/80	100/60	100/60	100/60	Yes	75/50%
Developmental Critical Function SW	100/80%	Two Levels	100/80	100/70	100/70	100/70	Yes	75/50%

Static Analysis p/a | Design Inspect | Code Inspect p/a | CVE p/a | CAPEC p/a | CWE p/a | Pen Test

Operational System

	Failover Multiple Supplier Redundancy	Fault Isolation	Least Privilege	System Element Isolation	Input checking / validation	SW load key
Developmental CPI SW	30%	All	all	yes	All	All
Developmental Critical Function SW	50%	All	All	yes	All	all
Other Developmental SW	none	Partial	none	None	all	all
COTS (CPI and CF) and NDI SW	none	Partial	All	None	Wrappers/ all	all

Development Environment

SW Product	Source	Release testing	Generated code inspection p/a
C Compiler	No	Yes	50/20
Runtime libraries	Yes	Yes	70/none
Automated test system	No	Yes	50/none
Configuration management system	No	Yes	NA
Database	No	Yes	50/none
Development Environment Access	Controlled access; Cleared personnel only		

Additional Guidance in PPP Outline and Guidance

Defense Acquisition Guidebook

Your Acquisition Policy and Discretionary Best Practice Guide

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

13.7.3.2.6. Software Encryption and Anti-Tamper Techniques (SW load key)

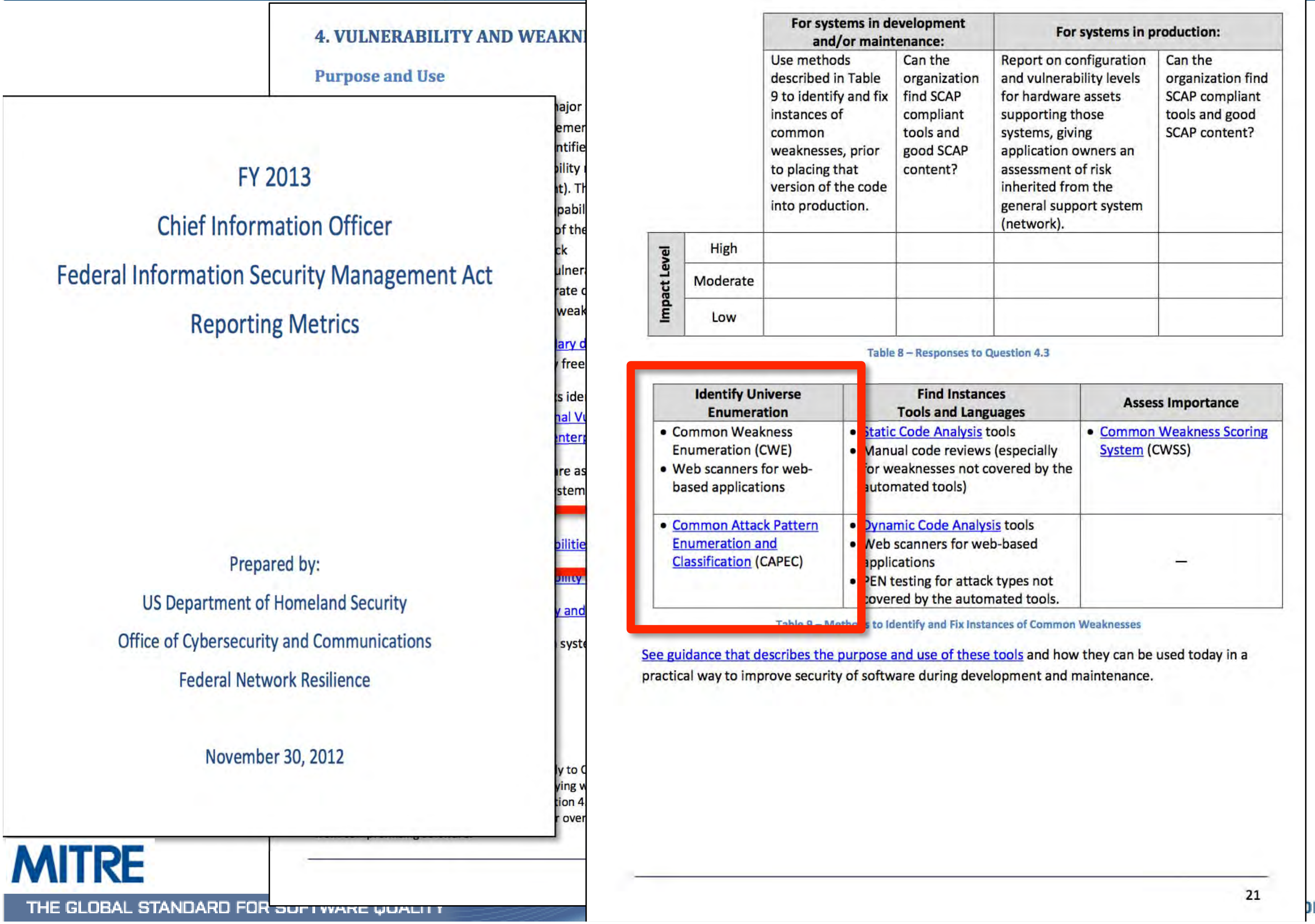
13.7.3.3. Development Environment

13.7.3.3.1 Source Code Availability

13.7.3.3.2. Release Testing

13.7.3.3.3. Generated Code Inspection

13.7.3.3.3. Additional Countermeasures



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

---

4. VULNERABILITY AND WEAKN

Purpose and Use

		For systems in development and/or maintenance:		For systems in production:	
		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?	Report on configuration and vulnerability levels for hardware assets supporting those systems, giving application owners an assessment of risk inherited from the general support system (network).	Can the organization find SCAP compliant tools and good SCAP content?
Impact Level	High				
	Moderate				
	Low				

Table 8 – Responses to Question 4.3

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

Table 9 – Metho s to Identify and Fix Instances of Common Weaknesses

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

Foreword

In 2008, the Software Assurance Forum for Excellence in Code (SAFECode) published the first version of this report in an effort to help others in the industry initiate or improve their own software assurance programs and encourage the industry-wide adoption of what we believe to be the most fundamental secure development methods. This work remains our most in-demand paper and has been downloaded more than 50,000 times since its original release.

However, secure software development is not only a goal, it is also a process. In the nearly two and a half years since we first released this paper, the process of building secure software has continued to evolve and improve alongside innovations and advancements in the information and communications technology industry. Much has been learned not only through increased community collaboration, but also through the ongoing internal efforts of SAFECode's member companies. This 2nd Edition aims to help disseminate that new knowledge.

Just as with the original paper, this paper is not meant to be a comprehensive guide to all possible secure development practices. Rather, it is meant to provide a foundational set of secure development practices that have been effective in improving software security in real-world implementations by SAFECode members across their diverse development environments.

It is important to note that these are the "practiced practices" employed by SAFECode members, which we identified through an ongoing analysis of our members' individual software security efforts. By bringing these methods together and sharing them with the larger community, SAFECode hopes to move the industry beyond defining theoretical best practices to describing sets of software engineering practices that have been shown to improve the security of software and are currently in use at leading software companies. Using this approach
enables SAFECode to encourag
best practices that are proven to
and implementable even when
requirements and development
taken into account.

Though expanded, our key goals
remain—keep it concise, actiona

What's New

This edition of the paper prescri
updated security practices that s
during the Design, Programming
ties of the software developmen
practices have been shown to be
diverse development environme
original also covered Training, Re
Handling and Documentation, th
given detailed treatment in SAF
security engineering training and software integrity
in the global supply chain, and thus we have refined
our focus in this paper to concentrate on the core
areas of design, development and testing.

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

SAFECode
Driving Security and Integrity

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

SAFECode
Software Assurance Forum for Excellence in Code
Driving Security and Integrity

xample.

s are available that support the Threat Model-
rocess with automated analysis of designs and
estions for possible mitigations, issue-tracking
gration and communication related to the
ess. Some practitioners have honed their Threat
eling process to the point where tools are used
utomate as much of it as possible, raising the
atability of the process and providing another
of support with standard diagramming,
ation, integration with a threat database and
cases, and execution of recurring tasks.

CWE References

Much of CWE focuses on implementation issues, and Threat Modeling is a design-time event. There are, however, a number of CWEs that are applicable to the threat modeling process, including:

- CWE-287: Improper authentication is an example of weakness that could be exploited by a Spoofing threat
- CWE-264: Permissions, Privileges, and Access Controls is a parent weakness of many Tampering, Repudiation and Elevation of Privilege threats
- CWE-311: Missing Encryption of Sensitive Data is an example of an Information Disclosure threat
- CWE-400: (uncontrolled resource consumption) is one example of an unmitigated Denial of Service threat

erification plan is a dire
tive of the results of the Threat Model act
Threat Model itself will serve as a clear ro
verification, containing enough informati
each threat and mitigation can be verified

During verification, the Threat Model and
mitigated threats, as well as the annotate
tectural diagrams, should also be made av
to testers in order to help define further t
and refine the verification process. A revie
Threat Model and verification results shou
made an integral part of the activities req
declare code complete.

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

Threat Identified	Design Element(s)	Mitigation	Verification
Session Hijacking	GUI	Ensure random session identifiers of appropriate length	Collect session identifiers over a number of sessions and examine distribution and length
Tampering with data in transit	Process A on server to Process B on client	Use SSL to ensure that data isn't modified in transit	Assert that communication cannot be established without the use of SSL

SAFECode
Software Assurance Forum for Excellence in Code
Driving Security and Integrity

SAFECode
Software Assurance Forum for Excellence in Code
Driving Security and Integrity

Fundamental Practices for
Secure Software Development
2ND EDITION

A Guide to the Most Effective Secure
Development Practices in Use Today

February 8, 2011

EDITOR Stacy Simpson, SAFECode

AUTHORS

Mark Belk, Juniper Networks	Mikko Saario, Nokia
Matt Coles, EMC Corporation	Reeny Sondhi, EMC Corporation
Cassio Goldschmidt, Symantec Corp.	Izar Tarandach, EMC Corporation
Michael Howard, Microsoft Corp.	Antti Vähä-Sipilä, Nokia
Kyle Randolph, Adobe Systems Inc.	Yonko Yonchev, SAP AG

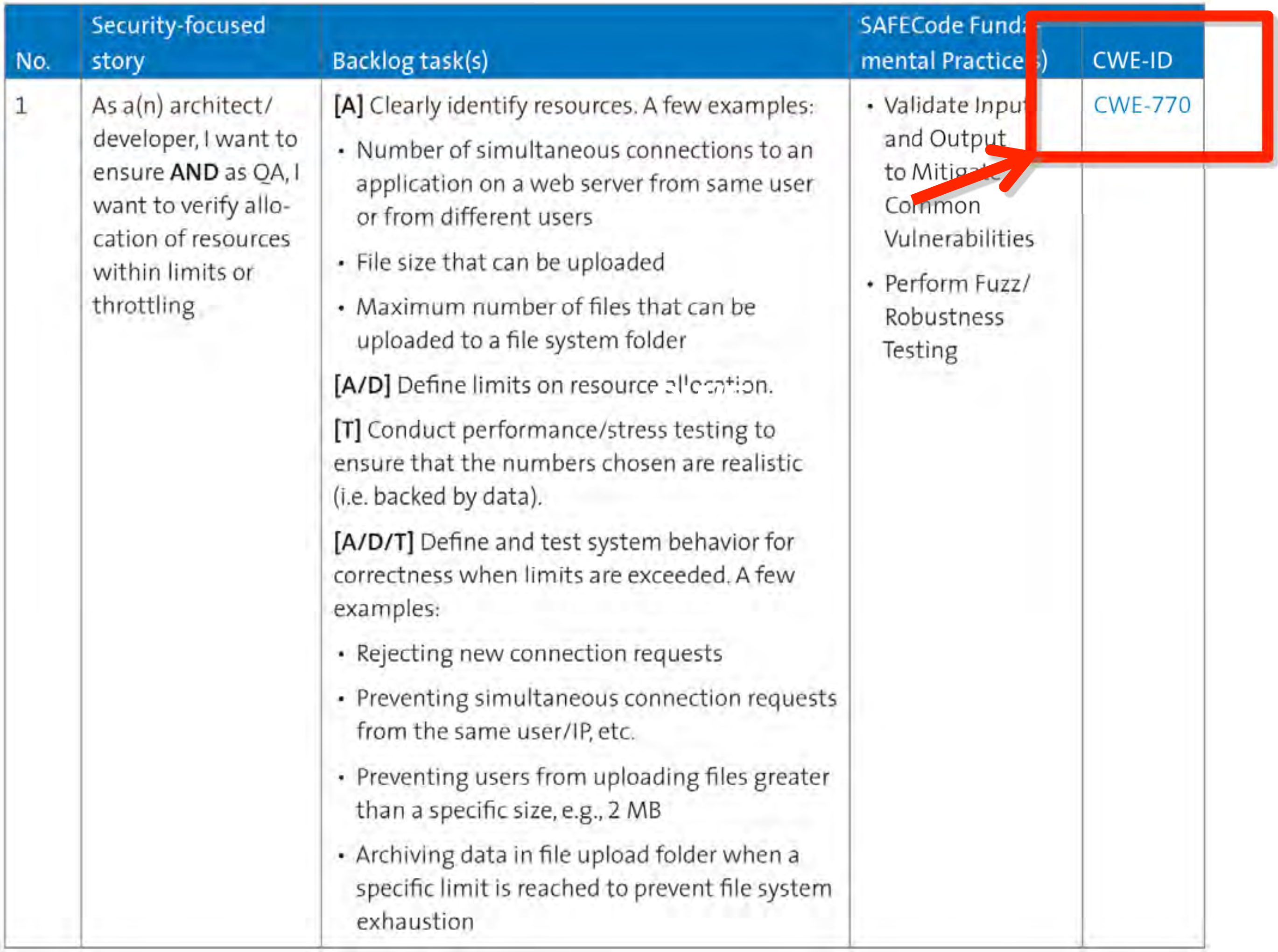
Industry Uptake
Agile

SAFECode
Software Assurance Forum for Excellence in Code
Driving Security and Integrity

Practical Security Stories and Security Tasks for Agile Development Environments

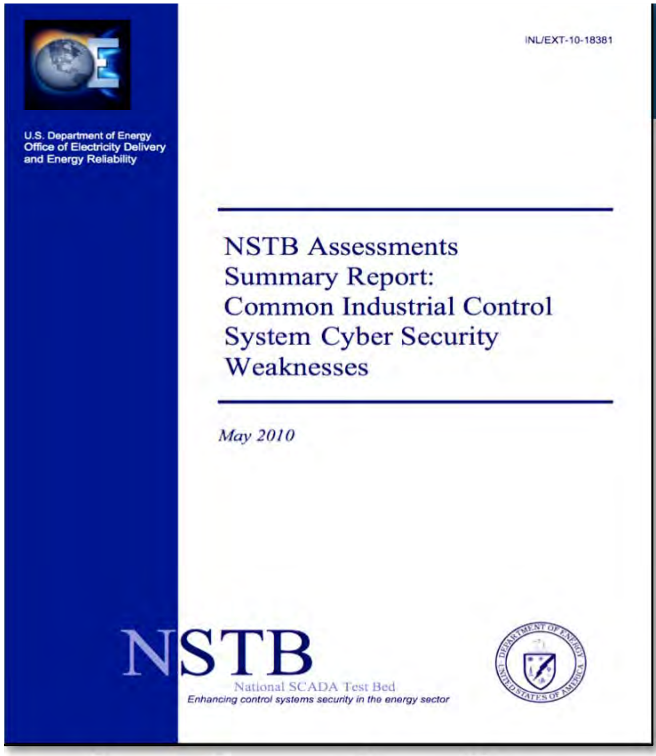
JULY 17, 2012

Table of Contents



No.	Security-focused story	Backlog task(s)	SAFECode Fundamental Practice(s)	CWE-ID
1	As a(n) 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

Idaho National Labs SCADA Report



INL/EXT-10-18381

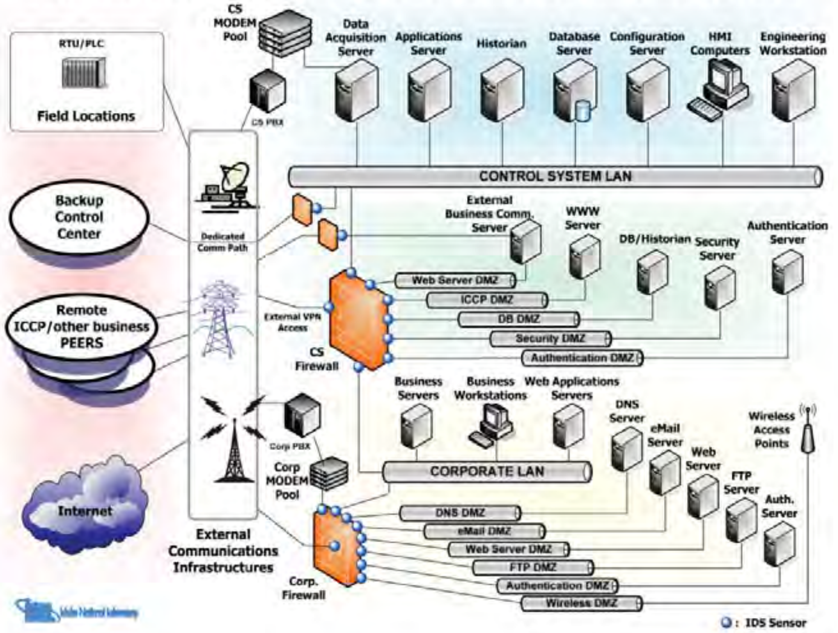
U.S. Department of Energy
Office of Electricity Delivery
and Energy Reliability

NSTB Assessments
Summary Report:
Common Industrial Control
System Cyber Security
Weaknesses

May 2010

NSTB
National SCADA Test Bed
Enhancing control systems security in the energy sector

SECURE CONTROL SYSTEM/ENTERPRISE ARCHITECTURE



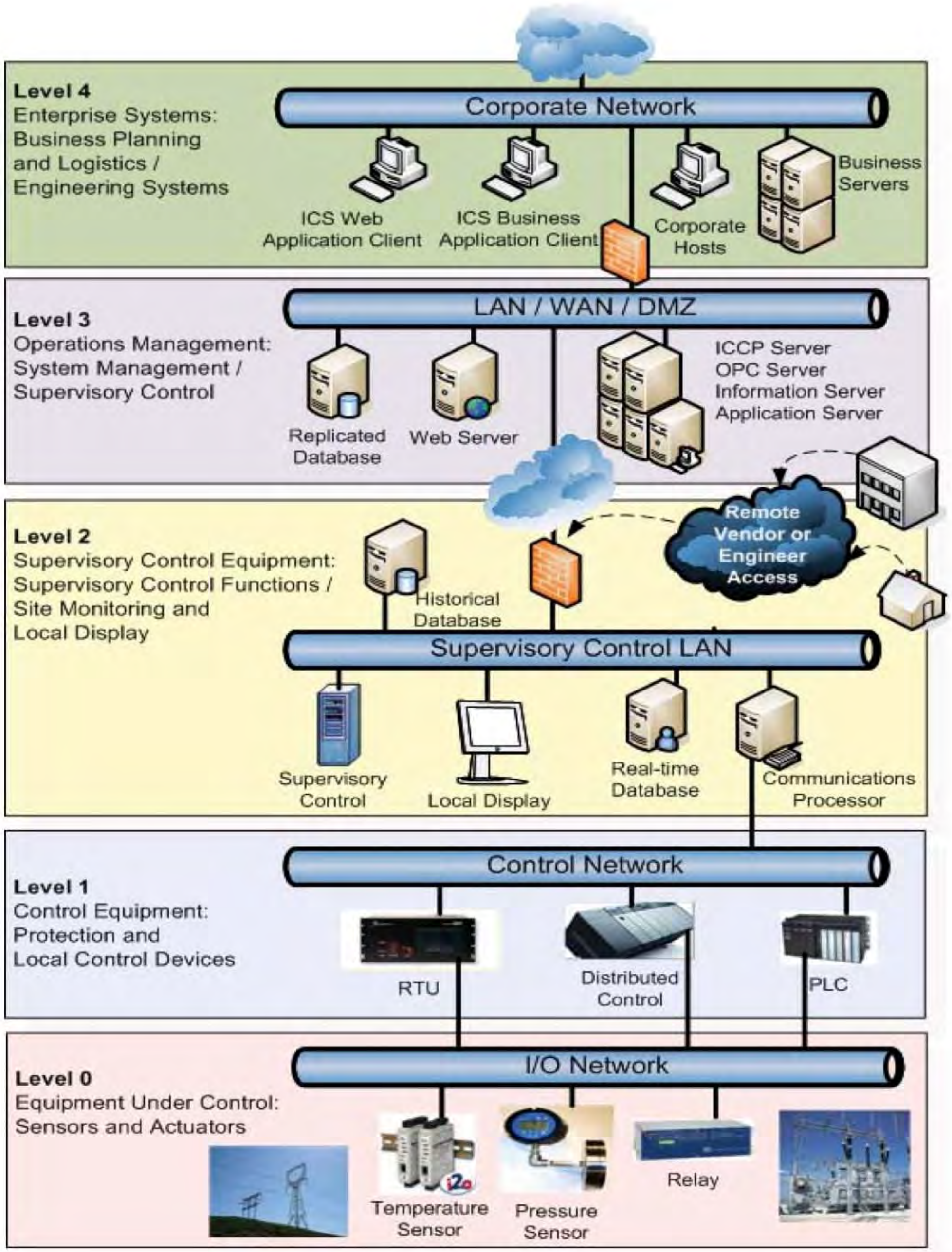


Table 27. Most common programming errors found in ICS code.

Weakness Classification	Vulnerability Type
CWE-19: Data Handling	CWE-228: Improper Handling of Syntactically Invalid Structure
	CWE-229: Improper Handling of Values
	CWE-230: Improper Handling of Missing Values
	CWE-20: Improper Input Validation
	CWE-116: Improper Encoding or Escaping of Output
	CWE-195: Signed to Unsigned Conversion Error
	CWE-198: Use of Incorrect Byte Ordering
CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer	CWE-120: Buffer Copy without Checking Size of Input ("Classic Buffer Overflow")
	CWE-121: Stack-based Buffer Overflow
	CWE-122: Heap-based Buffer Overflow
	CWE-125: Out-of-bounds Read
	CWE-129: Improper Validation of Array Index
	CWE-131: Incorrect Calculation of Buffer Size
	CWE-170: Improper Null Termination
	CWE-190: Integer Overflow or Wraparound
	CWE-680: Integer Overflow to Buffer Overflow
CWE-398: Indicator of Poor Code Quality	CWE-454: External Initialization of Trusted Variables or Data Stores
	CWE-456: Missing Initialization
	CWE-457: Use of Uninitialized Variable
	CWE-476: NULL Pointer Dereference
	CWE-400: Uncontrolled Resource Consumption ("Resource Exhaustion")
	CWE-252: Unchecked Return Value
	CWE-690: Unchecked Return Value to NULL Pointer Dereference
	CWE-772: Missing Release of Resource after Effective Lifetime
CWE-442: Web Problems	CWE-22: Improper Limitation of a Pathname to a Restricted Directory ("Path Traversal")
	CWE-79: Failure to Preserve Web Page Structure ("Cross-site Scripting")
	CWE-89: Failure to Preserve SQL Query Structure ("SQL Injection")
CWE-703: Failure to Handle Exceptional Conditions	CWE-431: Missing Handler
	CWE-248: Uncaught Exception
	CWE-755: Improper Handling of Exceptional Conditions
	CWE-390: Detection of Error Condition Without Action

OWASP

The Open Web Application Security Project

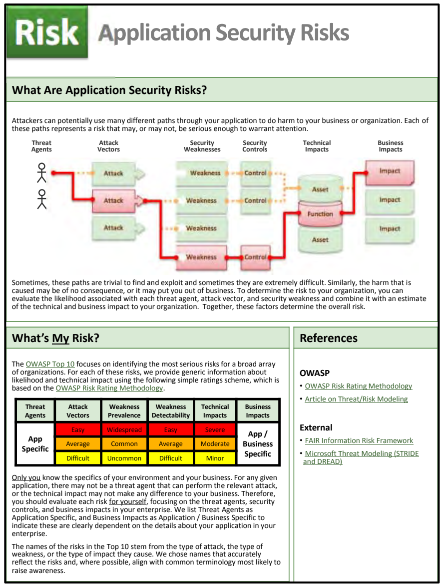
OWASP

The Open Web Application Security Project

OWASP Top 10 - 2013

The Ten Most Critical Web Application Security Risks

release



Risk Application Security Risks

What Are Application Security Risks?

Attackers can potentially use many different paths through your application to do harm to your business or organization. Each of these paths represents a risk that may, or may not, be serious enough to warrant attention.

Sometimes, these paths are trivial to find and exploit and sometimes they are extremely difficult. Similarly, the harm that is caused may be of no consequence, or it may put you out of business. To determine the risk to your organization, you can evaluate the likelihood associated with each threat agent, attack vector, and security weakness and combine it with an estimate of the technical and business impact to your organization. Together, these factors determine the overall risk.

What's My Risk?

The OWASP Top 10 focuses on identifying the most serious risks for a broad array of organizations. For each of these risks, we provide generic information about likelihood and technical impact using the following simple ratings scheme, which is based on the OWASP Risk Rating Methodology.

Threat Agents	Attack Vectors	Weakness Prevalence	Weakness Detectability	Technical Impacts	Business Impacts
App Specific	Easy	Widespread	Easy	Severe	App / Business Specific
	Average	Common	Average	Moderate	
	Difficult	Uncommon	Difficult	Minor	

Only you know the specifics of your environment and your business. For any given application, there may not be a threat agent that can perform the relevant attack, or the technical impact may not make any difference to your business. Therefore, you should evaluate each risk for yourself, focusing on the threat agents, security controls, and business impacts in your enterprise. We list Threat Agents as Application Specific, and Business Impacts as Application / Business Specific to indicate these are clearly dependent on the details about your application in your enterprise.

The names of the risks in the Top 10 stem from the type of attack, the type of weakness, or the type of impact they cause. We chose names that accurately reflect the risks and, where possible, align with common terminology most likely to raise awareness.

References

OWASP

- OWASP Risk Rating Methodology
- Article on Threat/Risk Modeling

External

- FAIR Information Risk Framework
- Microsoft Threat Modeling (STRIDE and DREAD)

CWE-928: Weaknesses in OWASP Top Ten (2013)

Weaknesses in OWASP Top Ten (2013)

View ID: 928 *(View: Graph)* Status: Incomplete

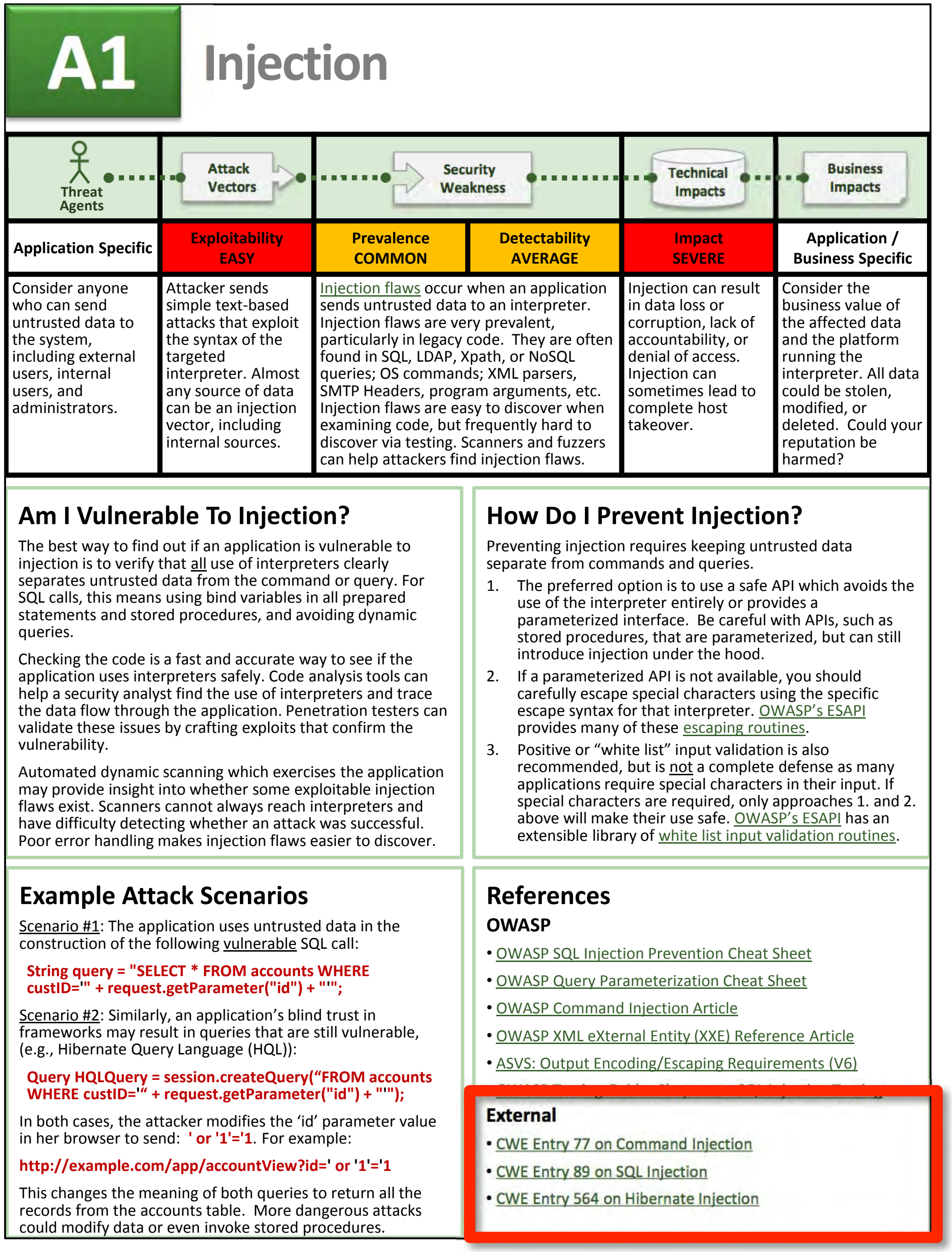
View Objective

CWE nodes in this view (graph) are associated with the OWASP Top Ten, as released in 2013.

	CWEs in this view		Total CWEs
Total	27	out of	940

928 - Weaknesses in OWASP Top Ten (2013)

- OWASP Top Ten 2013 Category A1 - Injection - *(929)*
- OWASP Top Ten 2013 Category A10 - Unvalidated Redirects and Forwards - *(938)*
- OWASP Top Ten 2013 Category A2 - Broken Authentication and Session Management - *(930)*
- OWASP Top Ten 2013 Category A3 - Cross-Site Scripting (XSS) - *(931)*
- OWASP Top Ten 2013 Category A4 - Insecure Direct Object References - *(932)*
- OWASP Top Ten 2013 Category A5 - Security Misconfiguration - *(933)*
- OWASP Top Ten 2013 Category A6 - Sensitive Data Exposure - *(934)*
- OWASP Top Ten 2013 Category A7 - Missing Function Level Access Control - *(935)*
- OWASP Top Ten 2013 Category A8 - Cross-Site Request Forgery (CSRF) - *(936)*
- OWASP Top Ten 2013 Category A9 - Using Components with Known Vulnerabilities - *(937)*



A1 Injection

Threat Agents	Attack Vectors	Security Weakness		Technical Impacts	Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	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.	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.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	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?

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.

How Do I Prevent Injection?

Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI provides many of these escaping routines.
3. Positive or "white list" input validation is also recommended, but is not a complete defense as many applications require special characters in their input. If special characters are required, only approaches 1. and 2. above will make their use safe. OWASP's ESAPI has an extensible library of white list input validation routines.

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 HQLQuery = session.createQuery("FROM accounts
WHERE custID='" + request.getParameter("id") + "'");
```

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

http://example.com/app/accountView?id=' 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.

References

OWASP

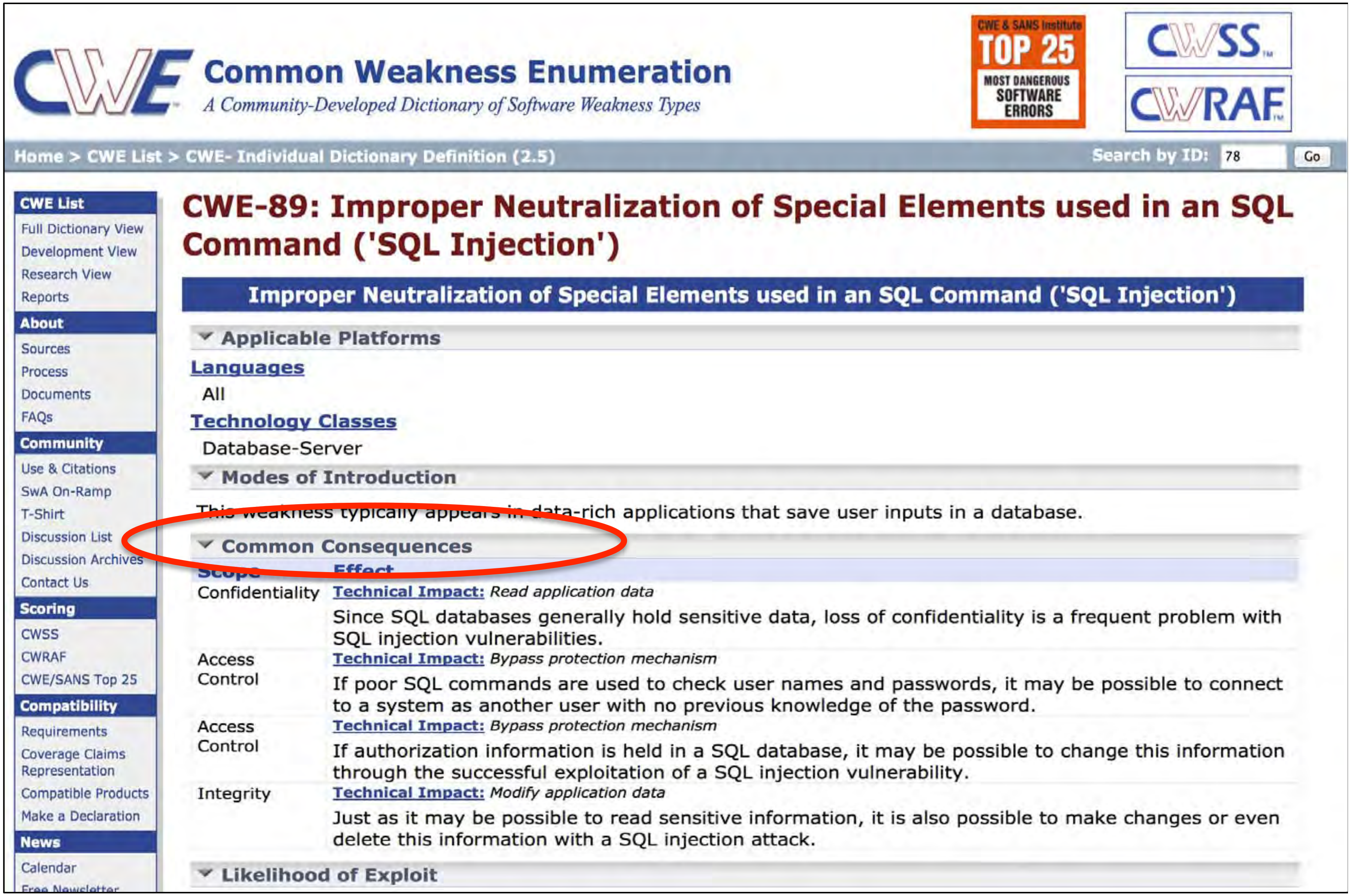
- OWASP SQL Injection Prevention Cheat Sheet
- OWASP Query Parameterization Cheat Sheet
- OWASP Command Injection Article
- OWASP XML eXternal Entity (XXE) Reference Article
- ASVS: Output Encoding/Escaping Requirements (V6)

External

- CWE Entry 77 on Command Injection
- CWE Entry 89 on SQL Injection
- CWE Entry 564 on Hibernate Injection

CISQ

Prioritizing by Technical Impacts: CWE's Common Consequences



CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Applicable Platforms

Languages

All

Technology Classes

Database-Server

Modes of Introduction

This weakness typically appears in data-rich applications that save user inputs in a database.

Common Consequences

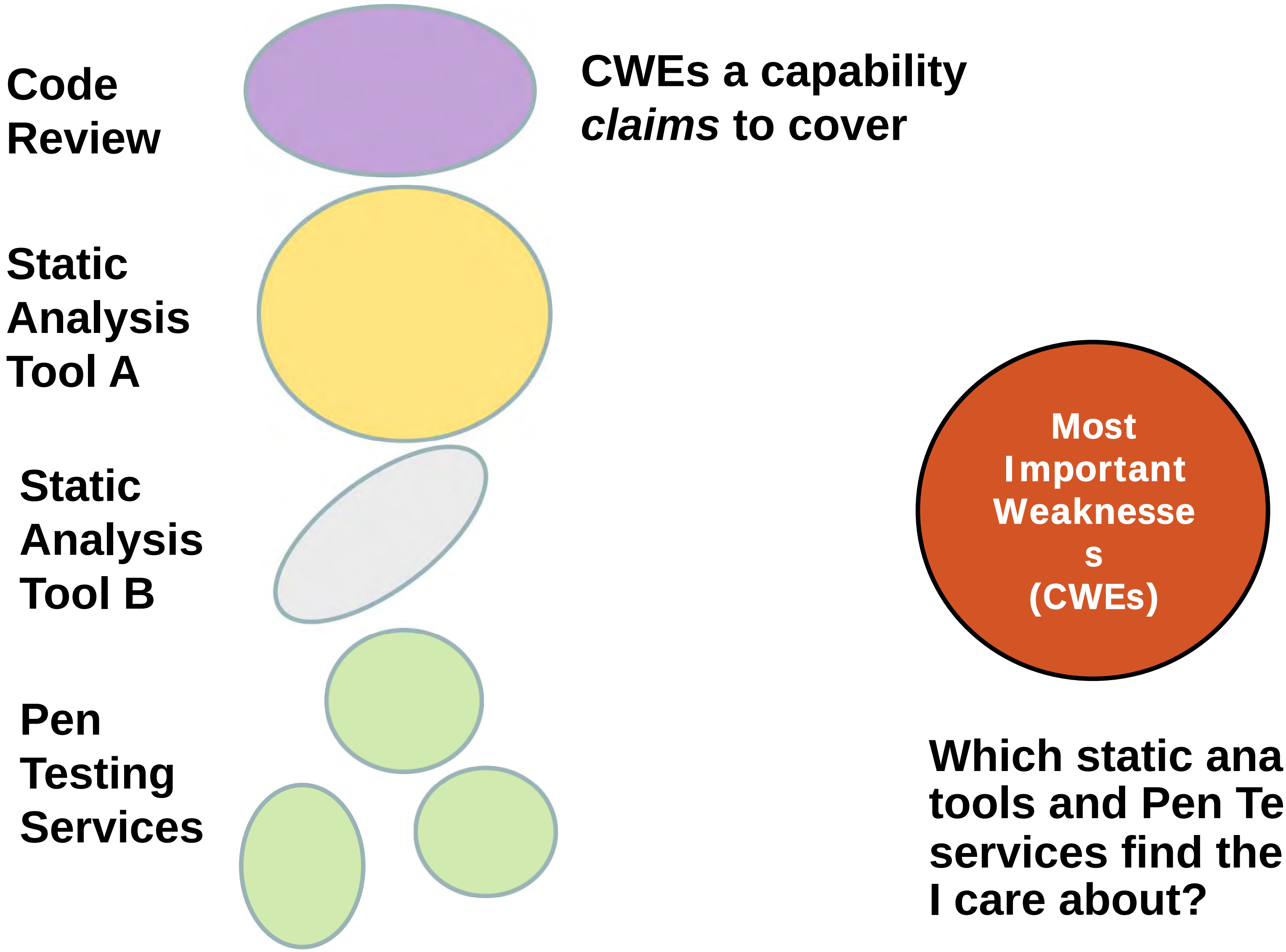
Scope	Effect
Confidentiality	**Technical Impact:** *Read application data* Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.
Access Control	**Technical Impact:** *Bypass protection mechanism* If poor SQL commands are used 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.
Access Control	**Technical Impact:** *Bypass protection mechanism* 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.
Integrity	**Technical Impact:** *Modify application data* 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.

Likelihood of Exploit

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

Utilizing a Priority List of Weaknesses



Code Review

Static Analysis Tool A

Static Analysis Tool B

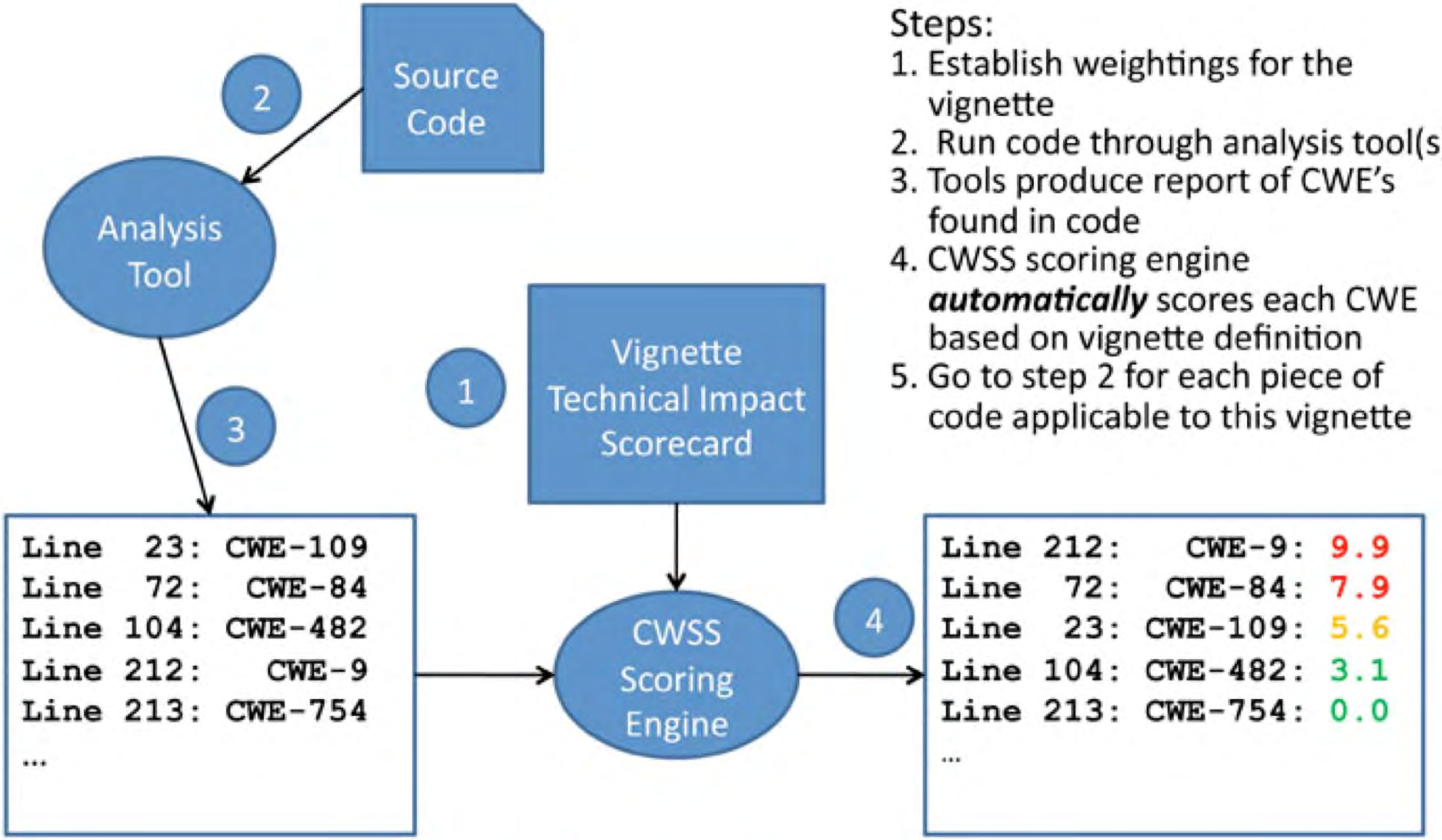
Pen Testing Services

CWEs a capability *claims* to cover

Most Important Weaknesses (CWEs)

Which static analysis tools and Pen Testing services find the CWEs I care about?

Scoring Weaknesses Discovered in Code



```
Line  23: CWE-109
Line  72:  CWE-84
Line 104: CWE-482
Line 212:   CWE-9
Line 213: CWE-754
...
```

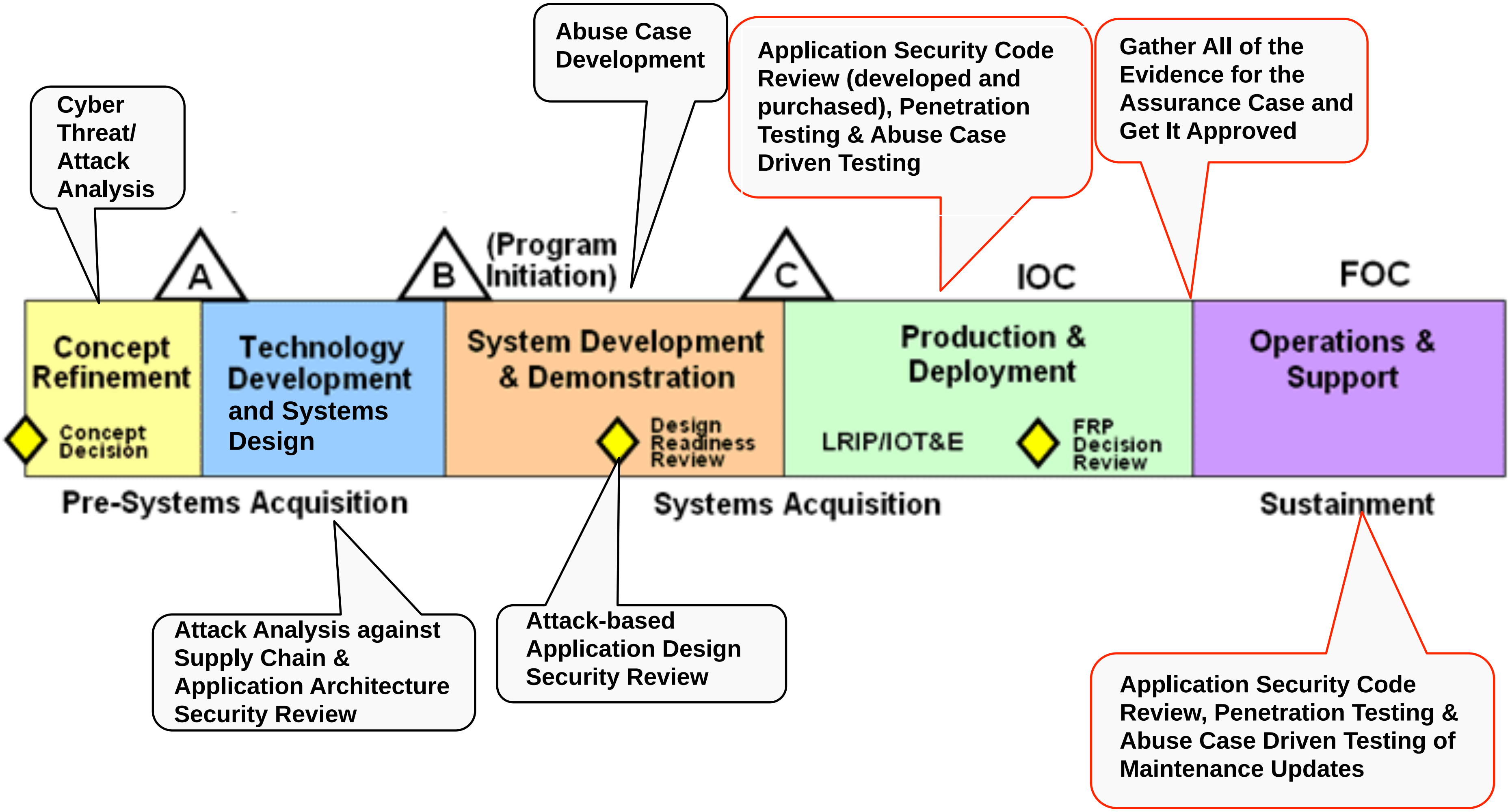
```
Line 212:   CWE-9: 9.9
Line  72:  CWE-84: 7.9
Line  23: CWE-109: 5.6
Line 104: CWE-482: 3.1
Line 213: CWE-754: 0.0
...
```

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

Step 1 is only done once – the rest is automatic

CISQ Assurance & the Systems Dev. Life-Cycle…



Cyber Threat/ Attack Analysis

Abuse Case Development

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

Gather All of the Evidence for the Assurance Case and Get It Approved

Attack Analysis against Supply Chain & Application Architecture Security Review

Attack-based Application Design Security Review

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

* Ideally Insert SwA before RFP release in Analysis of Alternatives

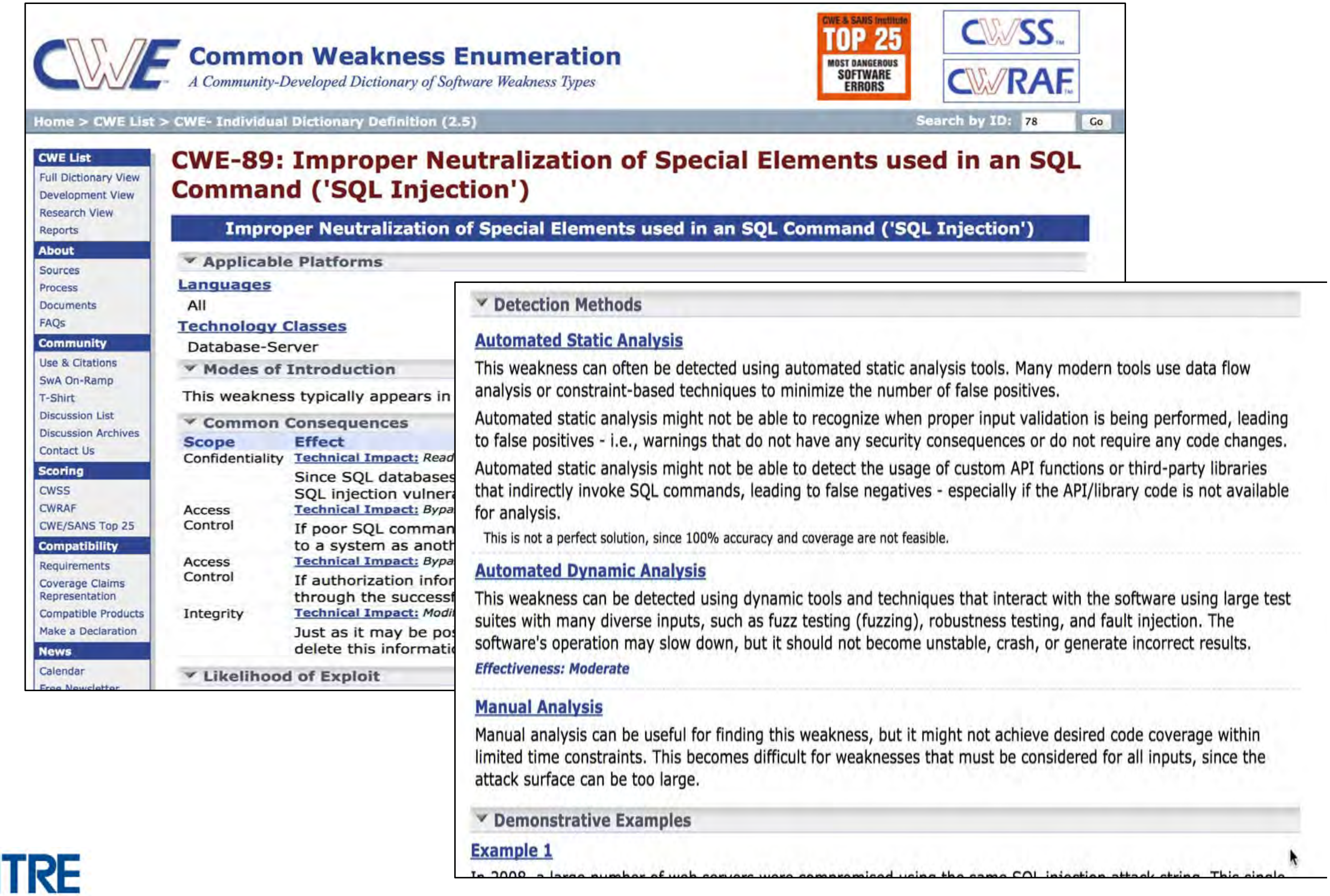
Leveraging and Managing to take Advantage of the Multiple Detection Methods

- **Different assessment methods are effective at finding different types of weaknesses**
- **Some are good at finding the cause and some at finding the effect**

	Static Code Analysis	Penetration Test	Data Security Analysis	Code Review	*Architecture Risk Analysis*
Cross-Site Scripting (XSS)	X	X		X	
SQL Injection	X	X		X	
Insufficient Authorization Controls		X	X	X	*X*
Broken Authentication and Session Management		X	X	X	*X*
Information Leakage		X	X		*X*
Improper Error Handling	X				
Insecure Use of Cryptography		X		X	*X*
Cross Site Request Forgery (CSRF)		X		X	
Denial of Service	X	X	X		*X*
Poor Coding Practices	*X*			*X*	

Detection Methods

Technical Impacts - Common Consequences



CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

▾ Applicable Platforms

Languages

All

Technology Classes

Database-Server

▾ Modes of Introduction

This weakness typically appears in

▾ Common Consequences

Scope	Effect
Confidentiality	Technical Impact: *Read* Since SQL databases SQL injection vulnera
Access Control	Technical Impact: *Bypa* If poor SQL comman to a system as anoth
Access Control	Technical Impact: *Bypa* If authorization infor through the successf
Integrity	Technical Impact: *Modi* Just as it may be pos delete this informatio

▾ Likelihood of Exploit

▾ 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.

Effectiveness: Moderate

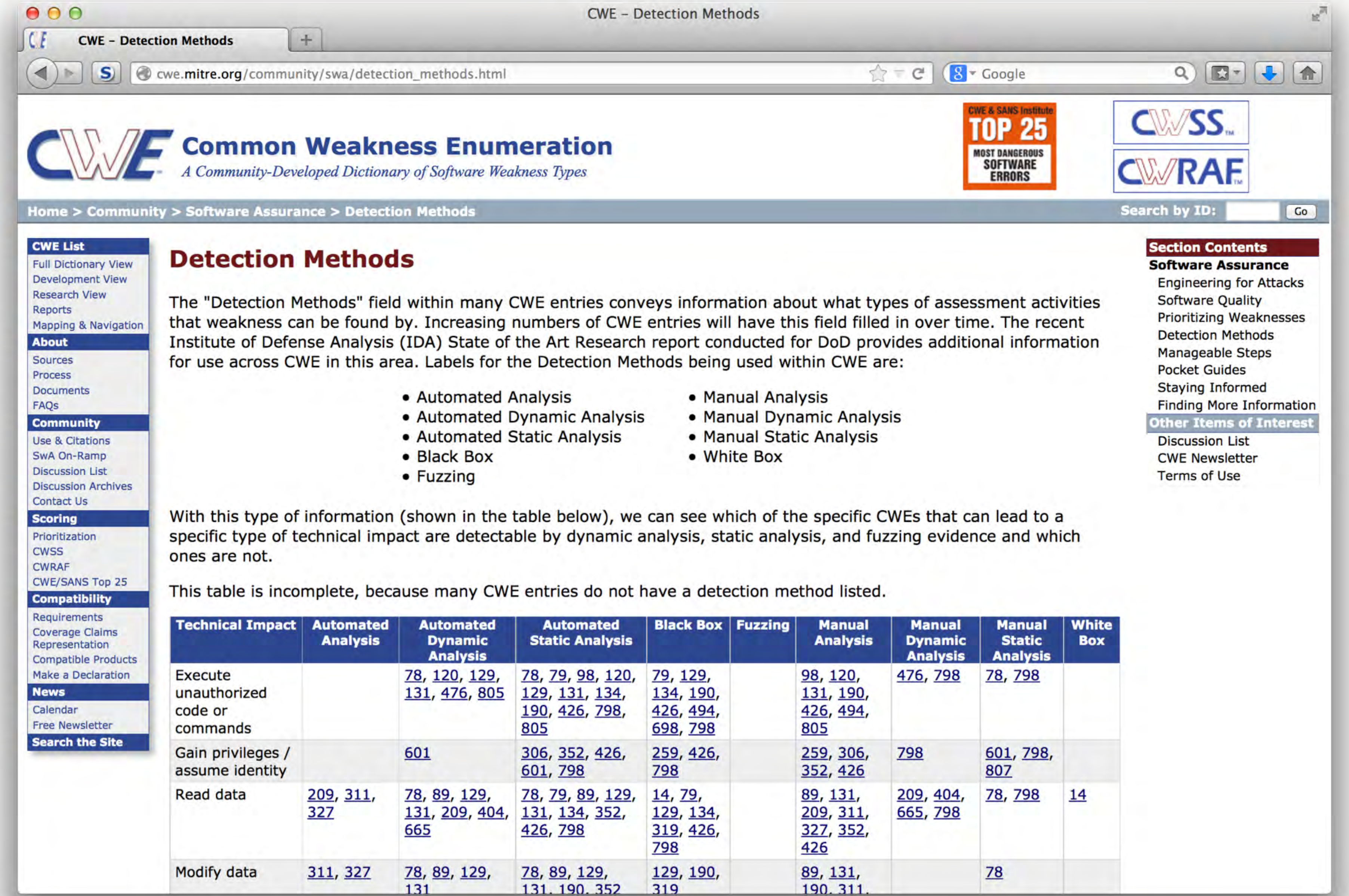
Manual Analysis

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.

▾ Demonstrative Examples

Example 1

New Detection Methods Launched Feb 17



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

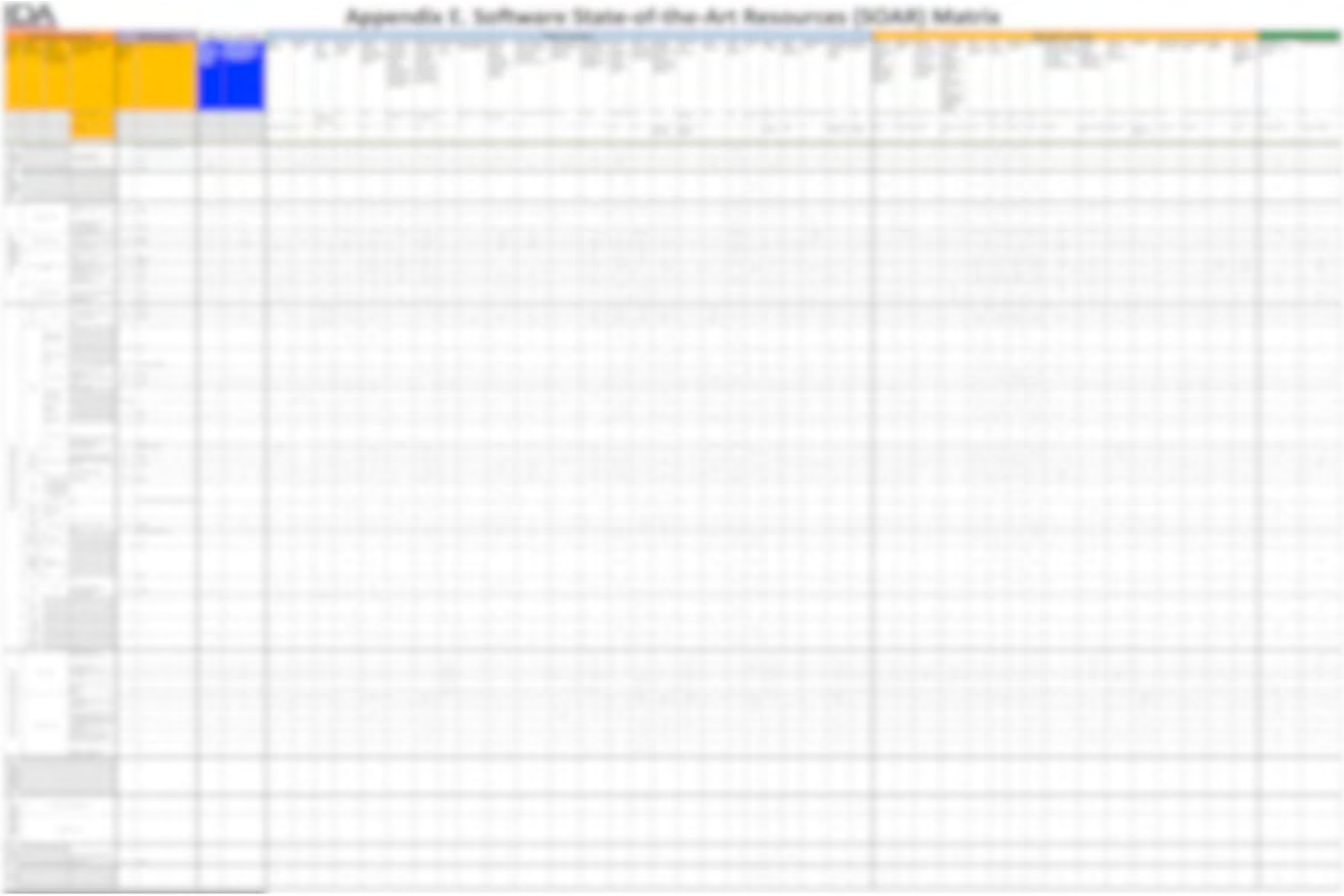
With this type of information (shown in the table below), we can see which of the specific CWEs that can lead to a specific type of technical impact are detectable by dynamic analysis, static analysis, and fuzzing evidence and which ones are not.

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

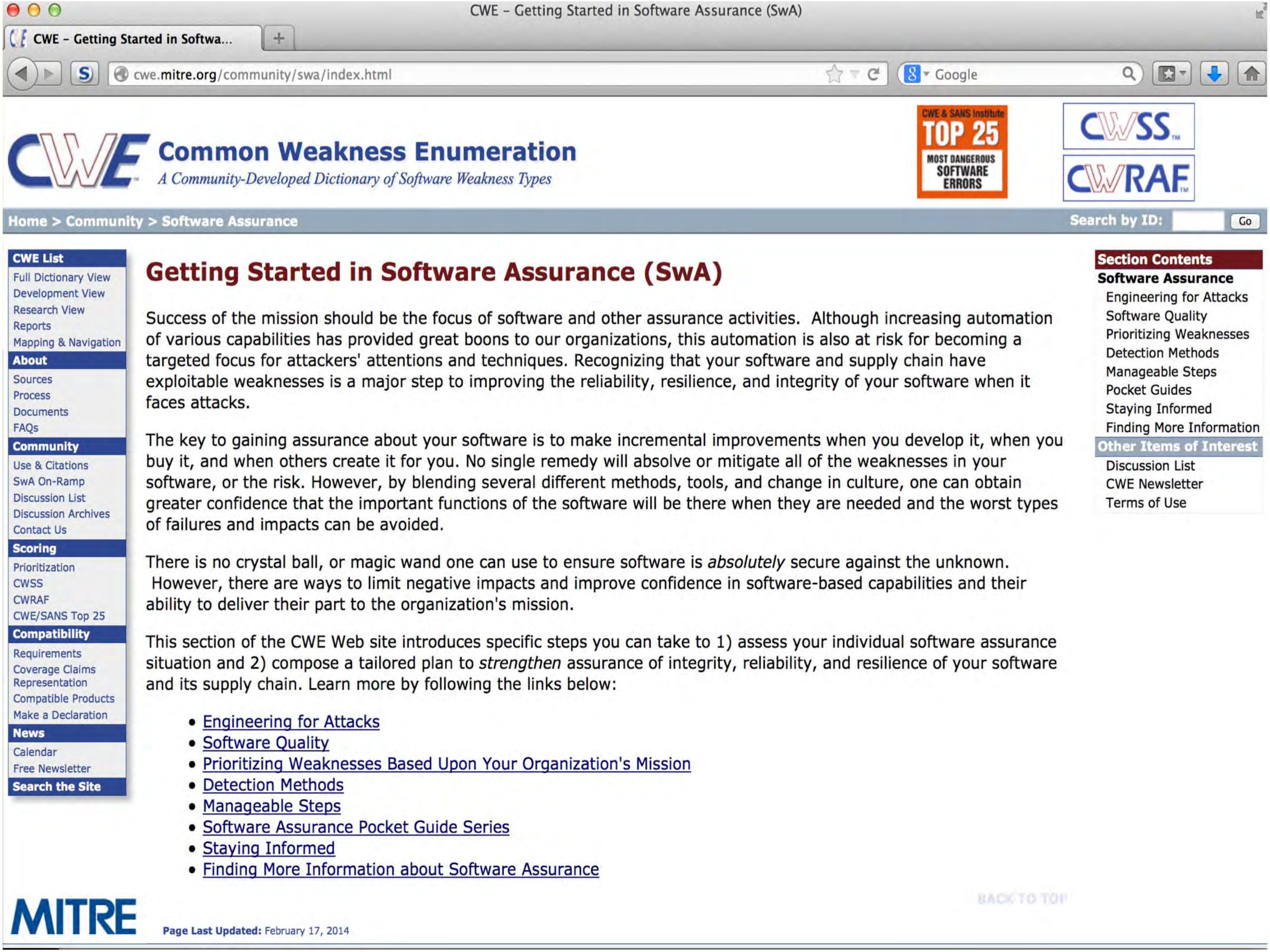
Technical Impact	Automated Analysis	Automated Dynamic Analysis	Automated Static Analysis	Black Box	Fuzzing	Manual Analysis	Manual Dynamic Analysis	Manual Static Analysis	White Box
Execute unauthorized code or commands		78, 120, 129, 131, 476, 805	78, 79, 98, 120, 129, 131, 134, 190, 426, 798, 805	79, 129, 134, 190, 426, 494, 698, 798		98, 120, 131, 190, 426, 494, 805	476, 798	78, 798	
Gain privileges / assume identity		601	306, 352, 426, 601, 798	259, 426, 798		259, 306, 352, 426	798	601, 798, 807	
Read data	209, 311, 327	78, 89, 129, 131, 209, 404, 665	78, 79, 89, 129, 131, 134, 352, 426, 798	14, 79, 129, 134, 319, 426, 798		89, 131, 209, 311, 327, 352, 426	209, 404, 665, 798	78, 798	14
Modify data	311, 327	78, 89, 129, 131	78, 89, 129, 131, 190, 352	129, 190, 319		89, 131, 190, 311,		78	

Technical Impact	Automated Analysis	Automated Dynamic Analysis	Automated Static Analysis	Black Box	Fuzzing	Manual Analysis	Manual Dynamic Analysis	Manual Static Analysis	White Box
Execute unauthorized code or commands		78, 120, 129, 131, 476, 805	78, 79, 98, 120, 129, 131, 134, 190, 426, 798, 805	79, 129, 134, 190, 426, 494, 698, 798		98, 120, 131, 190, 426, 494, 805	476, 798	78, 798	
Gain privileges / assume identity		601	306, 352, 426, 601, 798	259, 426, 798		259, 306, 352, 426	798	601, 798, 807	
Read data	209, 311, 327	78, 89, 129, 131, 209, 404, 665	78, 79, 89, 129, 131, 134, 352, 426, 798	14, 79, 129, 134, 319, 426, 798		89, 131, 209, 311, 327, 352, 426	209, 404, 665, 798	78, 798	14
Modify data	311, 327	78, 89, 129, 131	78, 89, 129, 131, 190, 352	129, 190, 319		89, 131, 190, 311, 327, 352		78	
DoS: unreliable execution		78, 120, 129, 131, 400, 476, 665, 805	78, 120, 129, 131, 190, 352, 400, 426, 805	129, 190, 426, 690	400	120, 131, 190, 352, 426, 805	476, 665	78	
DoS: resource consumption		120, 400, 404, 770, 805	120, 190, 400, 770, 805	190	400, 770	120, 190, 805	404	770	412
Bypass protection mechanism		89, 400, 601, 665	79, 89, 190, 352, 400, 601, 798	14, 79, 184, 190, 733, 798	400	89, 190, 352	665, 798	601, 798, 807	14, 733
Hide activities	327	78	78			327		78	

CWE will leverage the “State of the Art Resource” (SOAR): Software Table of “Verification Methods”



Appendix E. Software State-of-the-Art Resources (SOAR) Matrix



Common Weakness Enumeration

A Community-Developed Dictionary of Software Weakness Types

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:

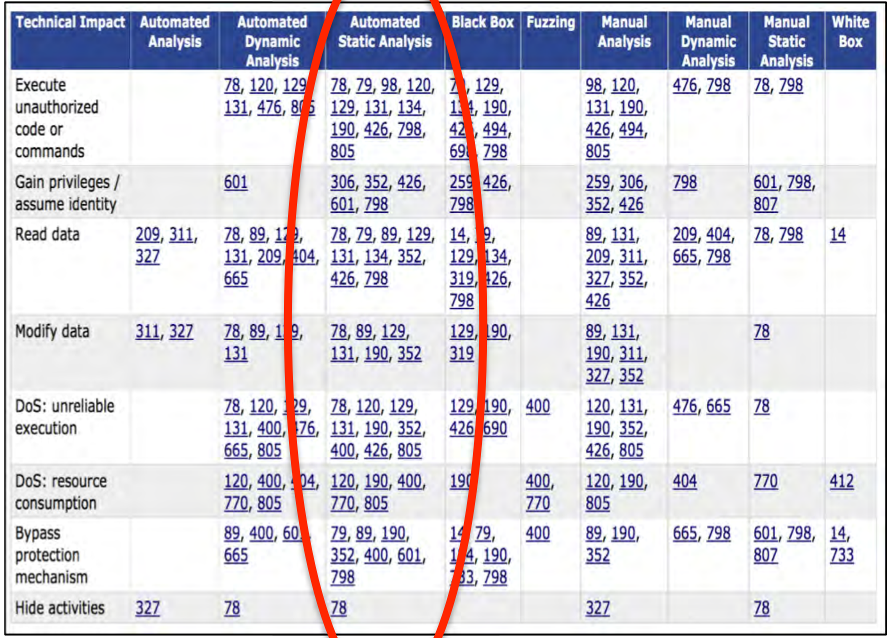
- Engineering for Attacks
- Software Quality
- Prioritizing Weaknesses Based Upon Your Organization's Mission
- Detection Methods
- Manageable Steps
- Software Assurance Pocket Guide Series
- Staying Informed
- Finding More Information about Software Assurance

Page Last Updated: February 17, 2014

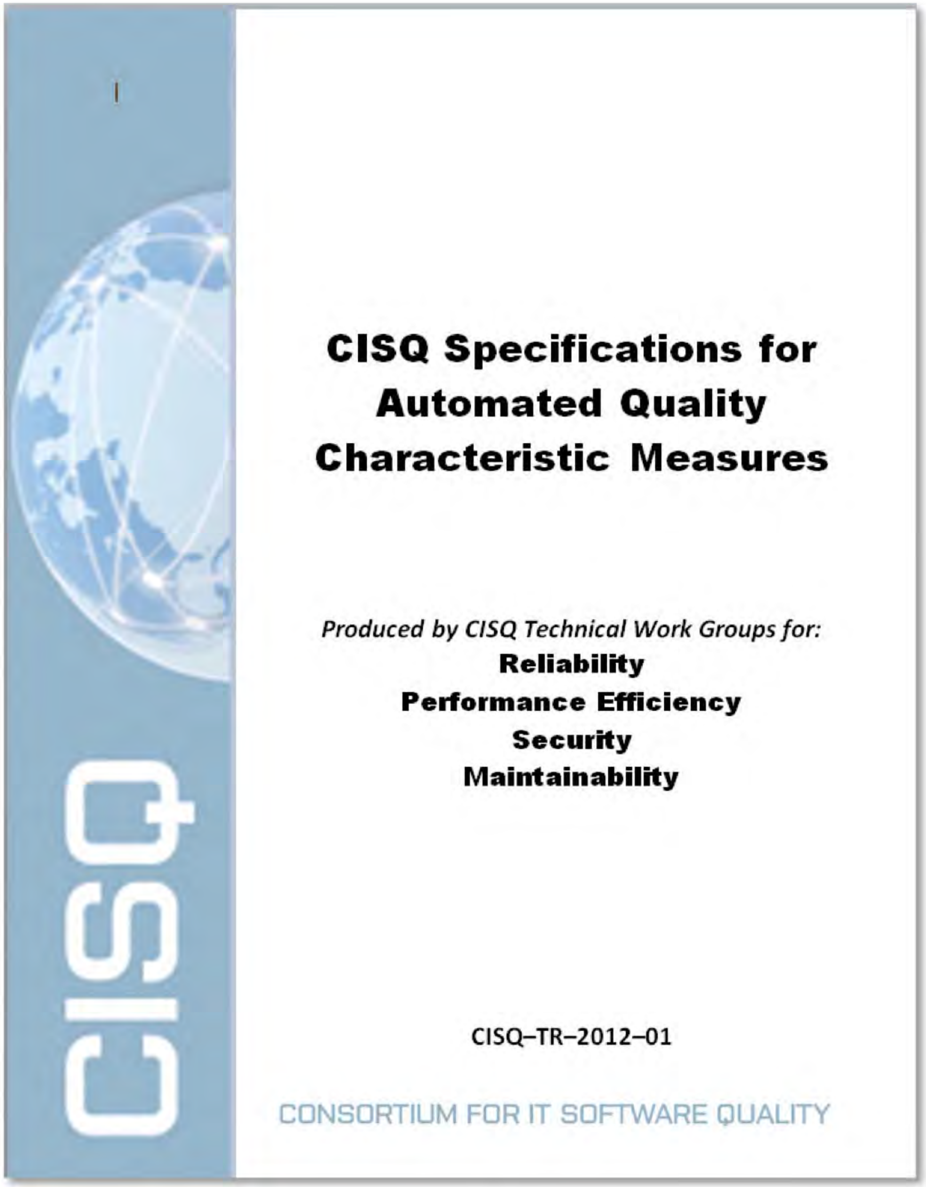
CISQ Security Measure

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



Technical Impact	Automated Analysis	Automated Dynamic Analysis	Automated Static Analysis	Black Box	Fuzzing	Manual Analysis	Manual Dynamic Analysis	Manual Static Analysis	White Box
Execute unauthorized code or commands		78, 120, 129, 131, 476, 805	78, 79, 98, 120, 129, 131, 134, 190, 426, 798, 805	7[illegible], 129, 134, 190, 426, 494, 698, 798		98, 120, 131, 190, 426, 494, 805	476, 798	78, 798	
Gain privileges / assume identity		601	306, 352, 426, 601, 798	259, 426, 798		259, 306, 352, 426	798	601, 798, 807	
Read data	209, 311, 327	78, 89, 129, 131, 209, 404, 665	78, 79, 89, 129, 131, 134, 352, 426, 798	14, [illegible]9, 129, 134, 319, 426, 798		89, 131, 209, 311, 327, 352, 426	209, 404, 665, 798	78, 798	14
Modify data	311, 327	78, 89, 129, 131	78, 89, 129, 131, 190, 352	129, 190, 319		89, 131, 190, 311, 327, 352		78	
DoS: unreliable execution		78, 120, 129, 131, 400, 476, 665, 805	78, 120, 129, 131, 190, 352, 400, 426, 805	129, 190, 426, 690	400	120, 131, 190, 352, 426, 805	476, 665	78	
DoS: resource consumption		120, 400, 404, 770, 805	120, 190, 400, 770, 805	190	400, 770	120, 190, 805	404	770	412
Bypass protection mechanism		89, 400, 601, 665	79, 89, 190, 352, 400, 601, 798	14, 79, 134, 190, 733, 798	400	89, 190, 352	665, 798	601, 798, 807	14, 733
Hide activities	327	78	78			327		78	



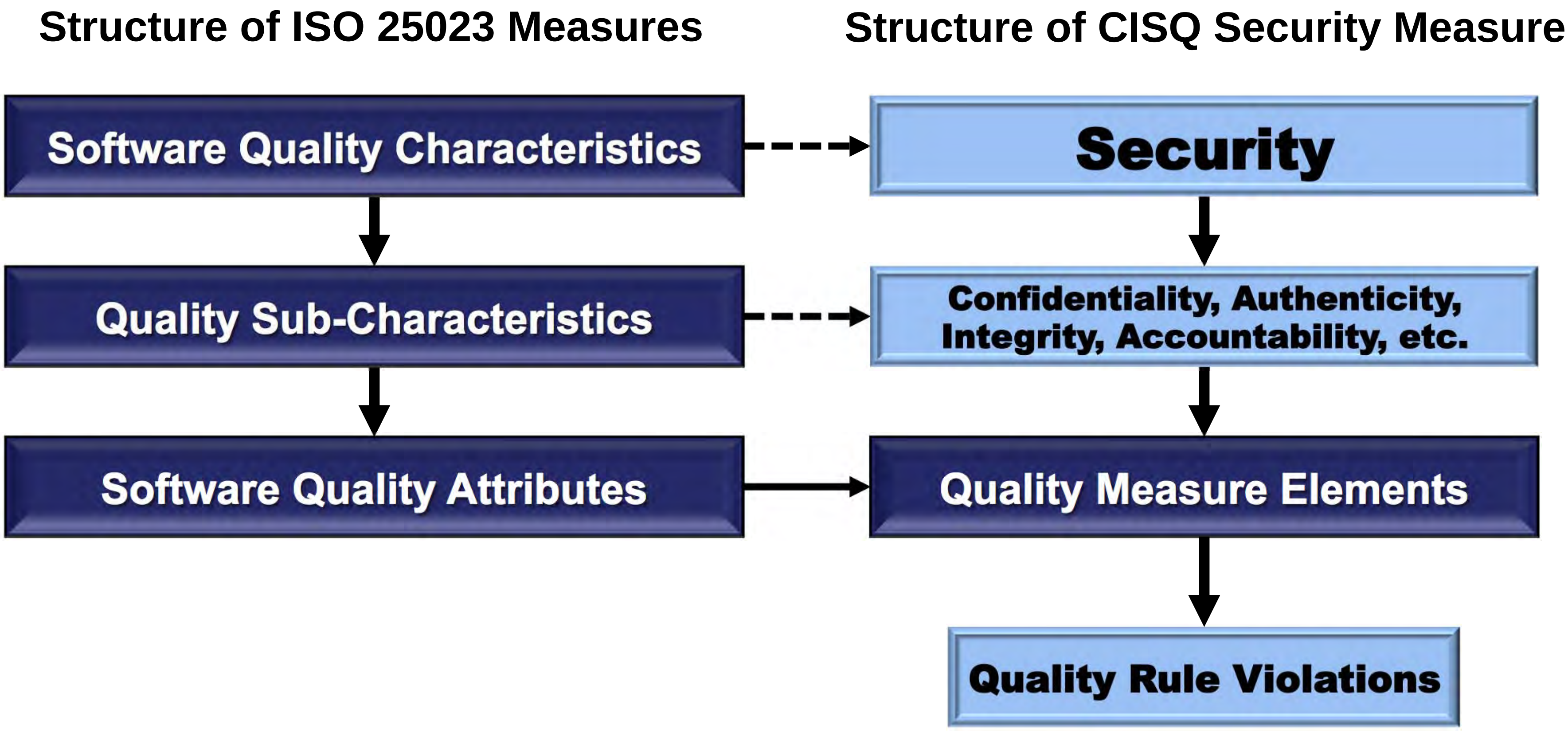
CISQ Specifications for Automated Quality Characteristic Measures

Produced by CISQ Technical Work Groups for:
Reliability
Performance Efficiency
Security
Maintainability

CISQ–TR–2012–01

CONSORTIUM FOR IT SOFTWARE QUALITY

Measuring Security by Violated Rules



Structure of ISO 25023 Measures

Structure of CISQ Security Measure

Software Quality Characteristics → Security

Quality Sub-Characteristics → Confidentiality, Authenticity, Integrity, Accountability, etc.

Software Quality Attributes → Quality Measure Elements

Quality Measure Elements → Quality Rule Violations

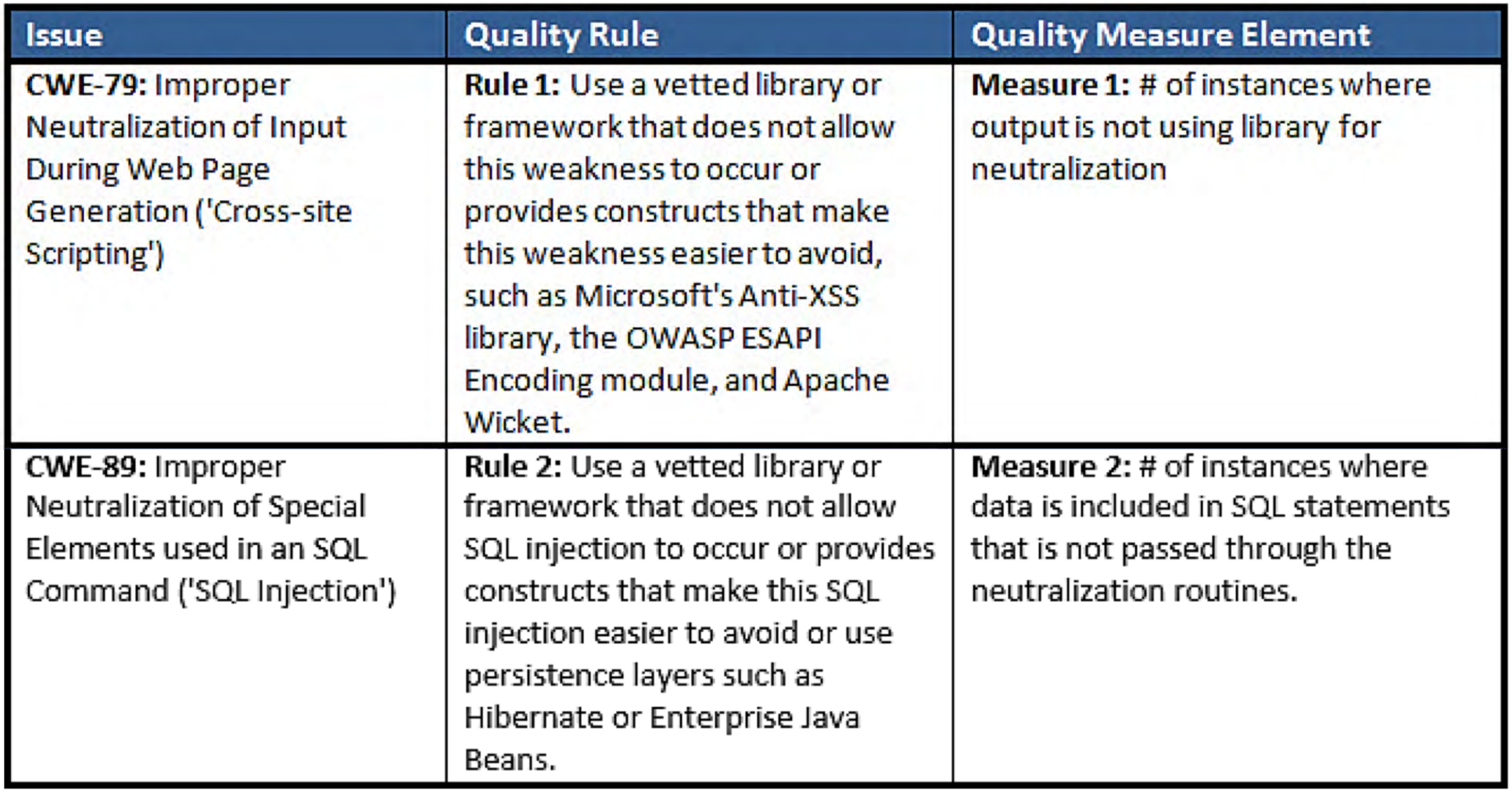
- Cross-site scripting
- SQL injection
- Buffer overflow
- OS command injection
- Unvalidated array
- Etc.

■ ISO structure

□ Examples from CISQ measures

CISQ

Example Security Issue→Rule→Measure



Issue	Quality Rule	Quality Measure Element
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	**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.	**Measure 1:** # of instances where output is not using library for neutralization
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	**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.	**Measure 2:** # of instances where data is included in SQL statements that is not passed through the neutralization routines.

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

TABLE OF CONTENTS

Departments

Mitigating Risks of Counterfeit and Tainted

CROSSTALK
MITIGATING RISKS
COUNTERFEIT
TAINTED COMPO

CROSSTALK

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Non-Malicious Taint

Bad Hygiene is as Dangerous 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 a malicious saboteur is the cause. It does not matter if it is malicious logic inserted at the factory or inserted through an update after fielding. 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 from bad software hygiene, like inadvertent coding flaws or weak architectural constructs are as dangerous to the mission as malicious acts. Enormous energies are put into hygiene and quality in the medical and food industries to address any source of taint. 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 initially 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 to both the software and hardware components within the ICT. Because of the way ICT hardware items are maintained, the supply chain for ongoing sustainment support of the software is often disconnected from the support for the hardware (e.g., continued software maintenance contracts with third parties other than the original manufacturer). As a result, supply chain assurance regarding software requires a slightly unique approach within the larger world of SCRM.

Some may want to focus on just "low hanging fruit" like banning suspect products by the 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 misconception that "adding" software assurance to the mix of supply chain concerns and activities will add too much complexity, thereby making SCRM even harder to perform. Some organizations and sectors are already developing standards of care and due-diligence that directly address these unintended and bad hygiene types of issues. That said, such practices for avoiding the bad hygiene issues that make software unfit for its intended purpose are not the norm across most of the industries involved in creating and supporting software-based products. Mitigating risk to the mission is a critical objective and including software assurance as a fundamental aspect of SCRM for ICT equipment is a critical component of delivering mission assurance.

During the past several decades, software-based ICT capabilities have become the basis of almost every aspect of today's cyber commerce, governance, national security, and recreation. Software-based devices are in our homes, vehicles, communications, and toys. Unfortunately software, the basis of these cyber capabilities, can be unpredictable since there are now underlying rules software has to follow as opposed to the rest of our material world which is constrained by the laws of gravity, chemistry, and physics with core factors like Plank's Constant. This is even more true given the variety and level of skills and training of those who create and evolve cyber capabilities. The result is that for the foreseeable future there will remain a need to address the types of quality and integrity problems that leave software unreliable, attackable, and brittle directly. This includes addressing the problems that allow malware and exploitable vulnerabilities to be accidentally inserted into products during development, packaging, or updates due to poor software hygiene practices.

Computer language specifications are historically vague and loosely written. (Note: ISO/IEC JTC1 SC22 issued a Technical Report [1] with guidance for selecting languages and using languages more secure and reliably.) There is often a lack of concern for resilience, robustness, and security in the variety of development tools used to build and deploy software. And there are gaps in the skills and education of those that manage, specify, create, test, and field these software-based products.

Additionally, software-based products are available to attackers who study them and then make these products do things their creators never intended. Traditionally this has led to calls for improved security functionality and more rigorous review, testing, and management. However, that approach fails to account for the core differences between the engineering of software-based products and other engineering disciplines. Those differences are detailed later in this article.

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 software-based products that comprise this expanding cyber world. ICT systems must be designed to withstand attacks and offer resilience through better integrity, avoidance of known weaknesses in code, architecture, and design. Additionally, ICT systems should be created with designed-in protection capabilities to address unforeseen attacks by making them intrinsically more rugged and resilient so that there are fewer ways to impact the system. This same concern has been expressed by Congress with the inclusion of a definition of "Software Assurance" in Public Law 112-239 Section 933 [2] where they directed DoD to specifically address software assurance of its systems.

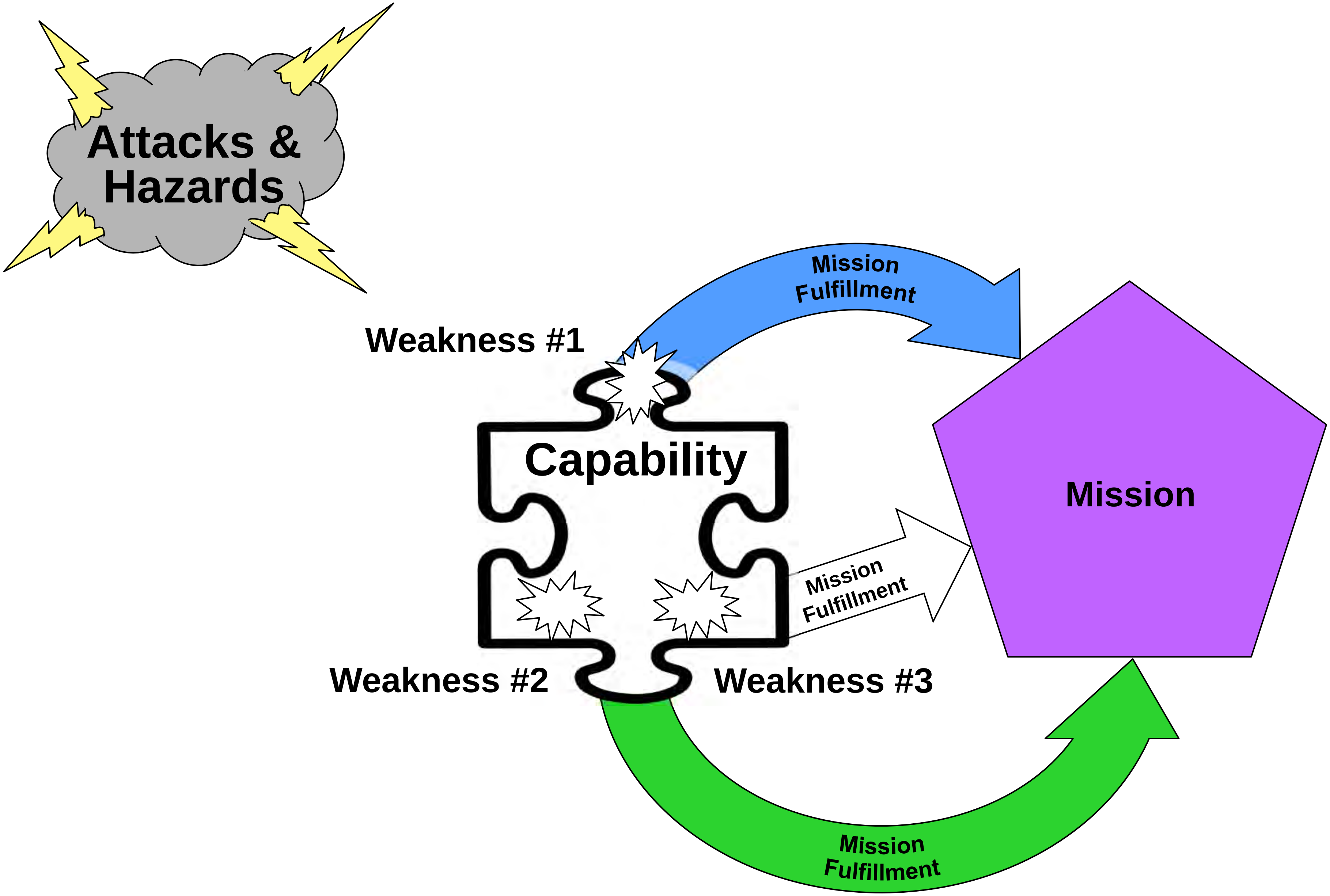
Defining "Taint" and Software Assurance

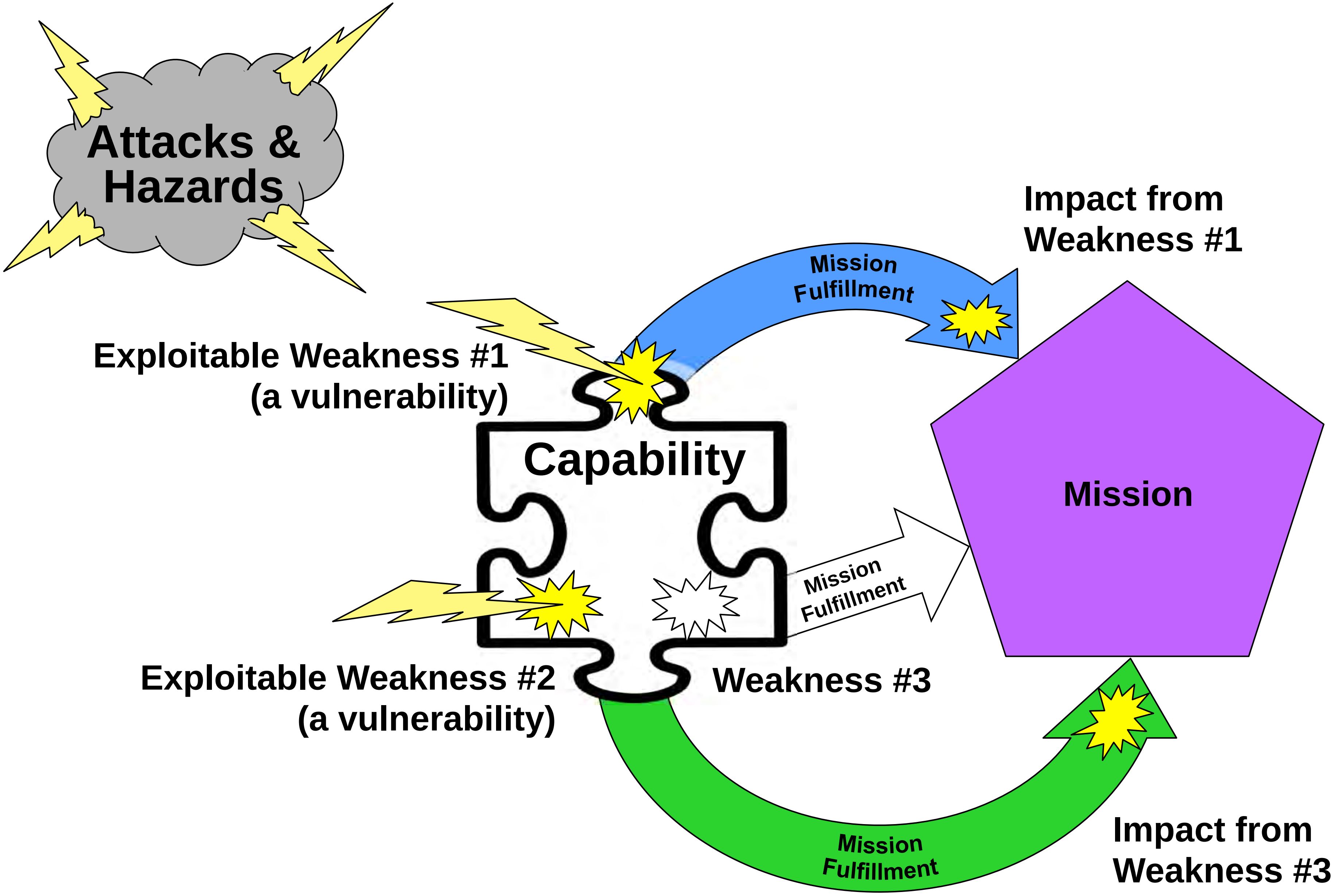
While there is no concrete definition of what "taint" specifically means within the cyber realm, we would be remiss not to look to the general use of the term, as well as synonyms and antonyms. Merriam Webster [3] provides a useful point-of-departure, as shown in Table 1 below.

1: Weakne Technical Impacts by Detection Meth

2: Matching Coverage Claims to Your Organization's Needs

ddressed using general engineering and process improve-
methodologies. However, it is clear that software fails from
other than these causes. As discussed above, software
s no laws unless their creators impose them and can fail
individual implementation mistakes or through the intro-
n of weaknesses or malicious logic.
software developers or systems engineering practitioners
he training and experience to recognize, consider, and
these weaknesses. Few (if any) tools or procedures are
ble to review and test for all weaknesses in a systematic
er. Developers are rarely provided with criteria about what
of problems are possible, and what their presence could
to the fielded software system and its users.
nanage these risks we cannot just expect to come up
he "right security requirements." We also need to provide a
dology that assists in gaining assurance through the gath-
of evidence and showing how that information provides
ance and confidence that the system development process
ssed the removal or mitigation of weaknesses that could
exploitable vulnerabilities. The changes in revision 4 of
al Institute of Standards and Technology (NIST) Special
ation 800-53 [13] directly bring assurance into the secu-
sture equation.


Attacks &
Hazards
Weakness #1
Mission
Fulfillment
Capability
Mission
Mission
Fulfillment
Weakness #2
Weakness #3
Mission
Fulfillment



Attacks & Hazards

Impact from Weakness #1

Mission Fulfillment

Exploitable Weakness #1 (a vulnerability)

Capability

Mission

Mission Fulfillment

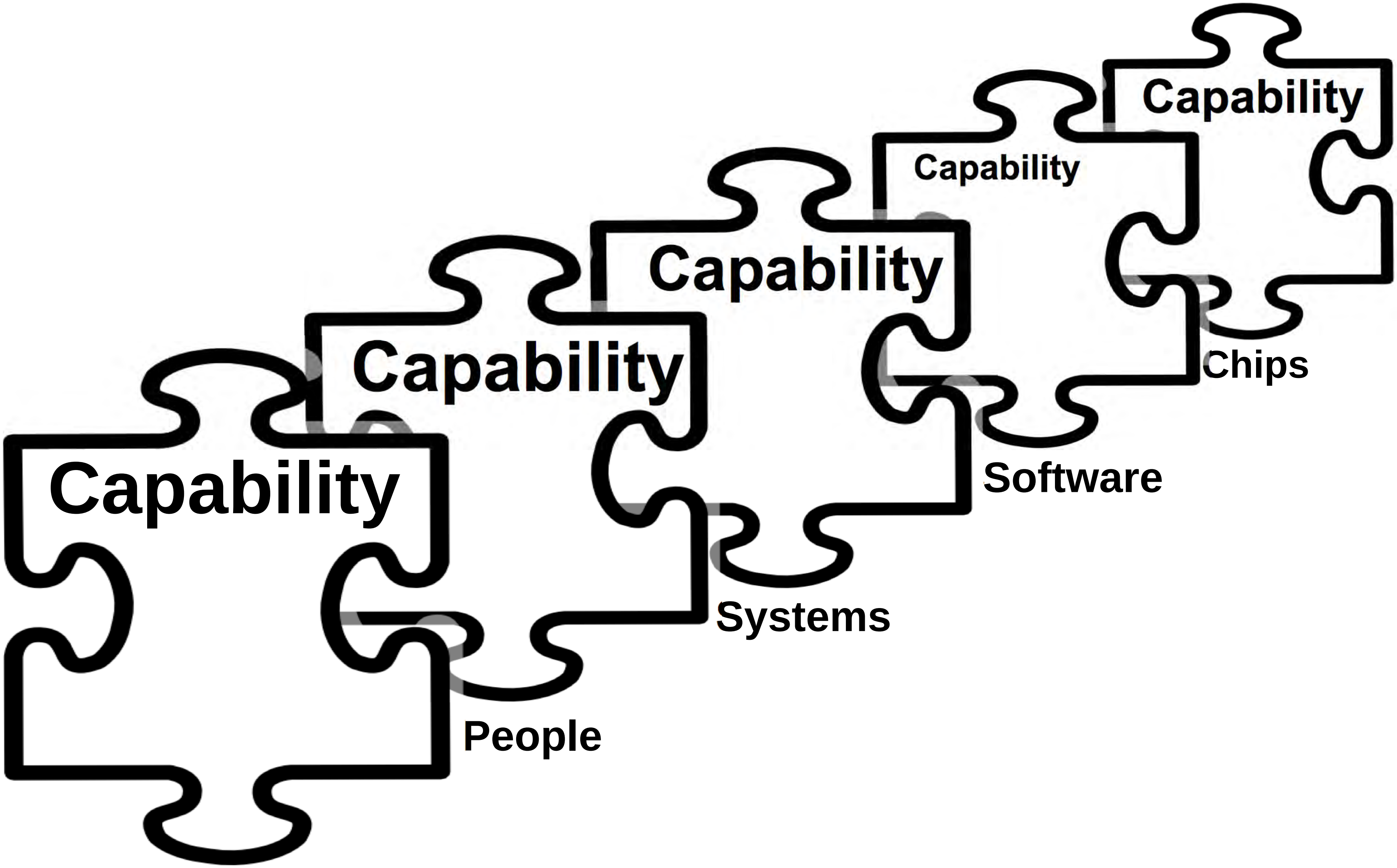
Exploitable Weakness #2 (a vulnerability)

Weakness #3

Mission Fulfillment

Impact from Weakness #3

Many Capabilities Support the Mission



Capability

Capability

Capability

Capability

Capability

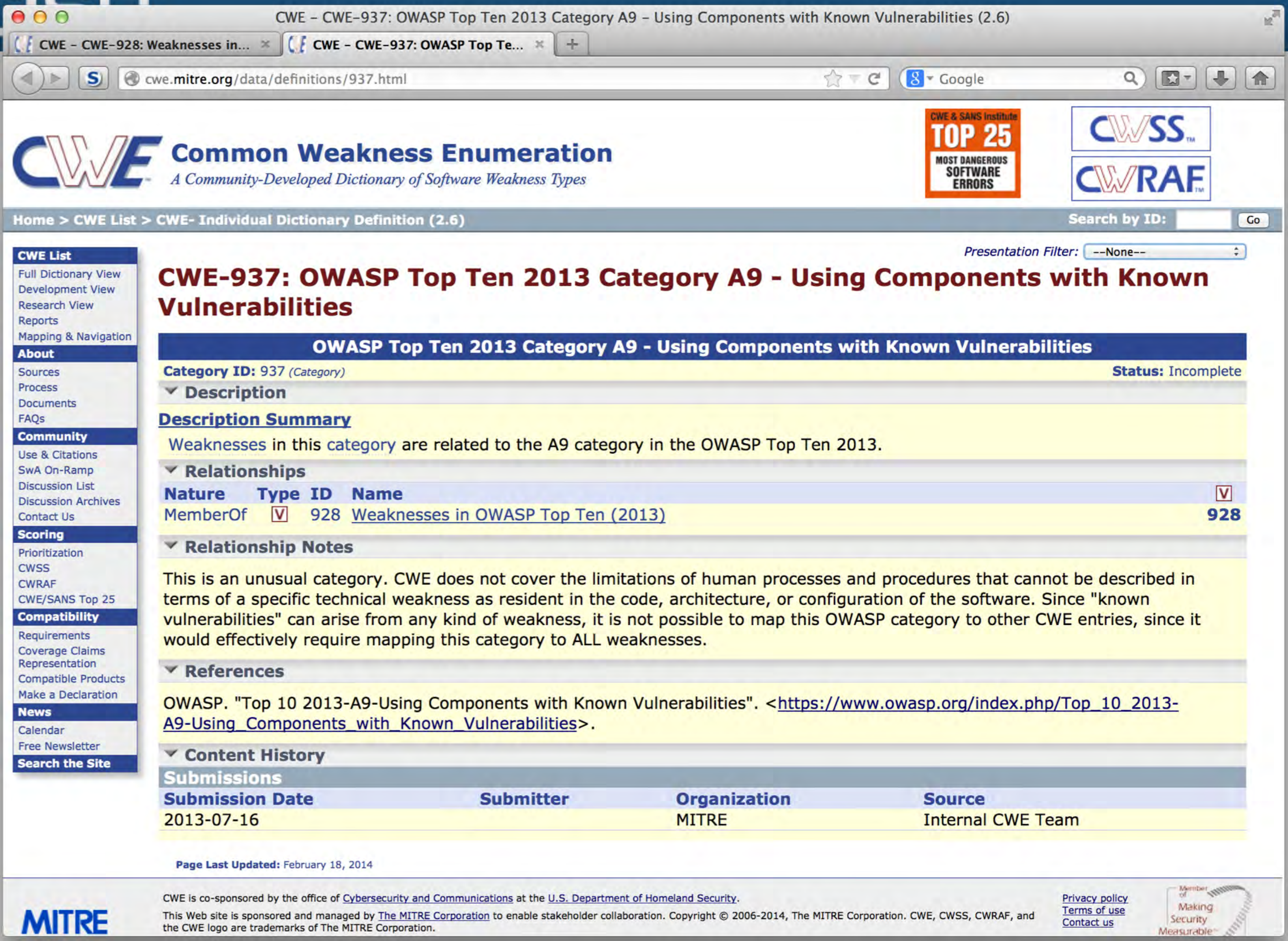
Chips

Software

Systems

People

Supply Chain Activities



Common Weakness Enumeration

A Community-Developed Dictionary of Software Weakness Types

Home > CWE List > CWE- Individual Dictionary Definition (2.6)

Search by ID: Go

CWE List
- Full Dictionary View
- Development View
- Research View
- Reports
- Mapping & Navigation

About
- Sources
- Process
- Documents
- FAQs

Community
- Use & Citations
- SwA On-Ramp
- Discussion List
- Discussion Archives
- Contact Us

Scoring
- Prioritization
- CWSS
- CWRAF
- CWE/SANS Top 25

Compatibility
- Requirements
- Coverage Claims Representation
- Compatible Products
- Make a Declaration

News
- Calendar
- Free Newsletter

Search the Site

Presentation Filter: --None--

CWE-937: OWASP Top Ten 2013 Category A9 - Using Components with Known Vulnerabilities

OWASP Top Ten 2013 Category A9 - Using Components with Known Vulnerabilities

Category ID: 937 *(Category)* **Status:** Incomplete

Description

Description Summary

Weaknesses in this category are related to the A9 category in the OWASP Top Ten 2013.

Relationships

Nature	Type	ID	Name	V
MemberOf	V	928	Weaknesses in OWASP Top Ten (2013)	928

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

OWASP. "Top 10 2013-A9-Using Components with Known Vulnerabilities". <https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities>.

Content History

Submissions

Submission Date	Submitter	Organization	Source
2013-07-16		MITRE	Internal CWE Team

Page Last Updated: February 18, 2014

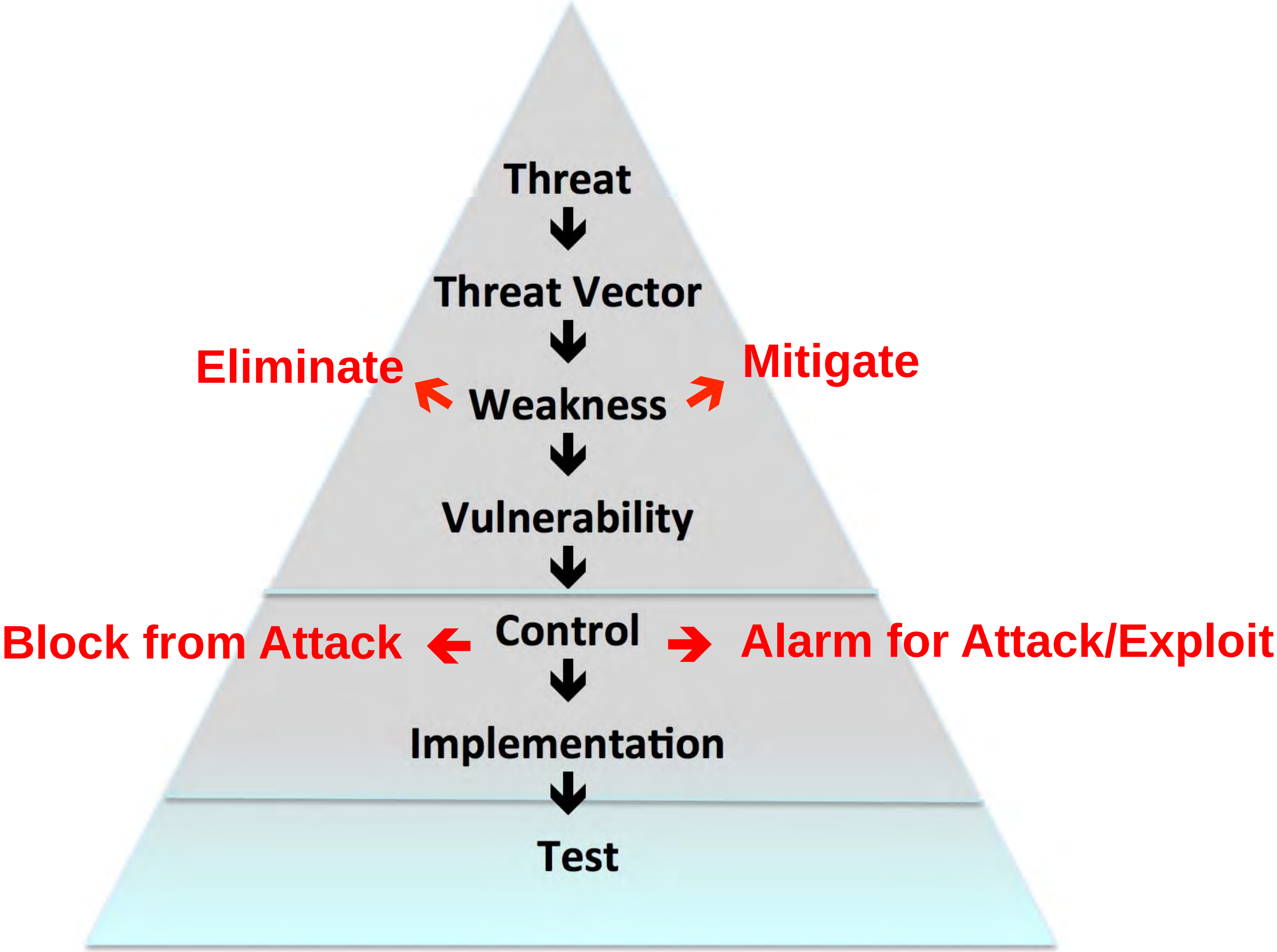
MITRE

CWE is co-sponsored by the office of Cybersecurity and Communications at the U.S. Department of Homeland Security.

Privacy policy
Terms of use
Contact us

CISQ

Assurance on the Management of Weaknesses



Threat
↓
Threat Vector
↓
Eliminate ← Weakness → Mitigate
↓
Vulnerability
↓
Block from Attack ← Control → Alarm for Attack/Exploit
↓
Implementation
↓
Test

CWE

CVE

CAPEC