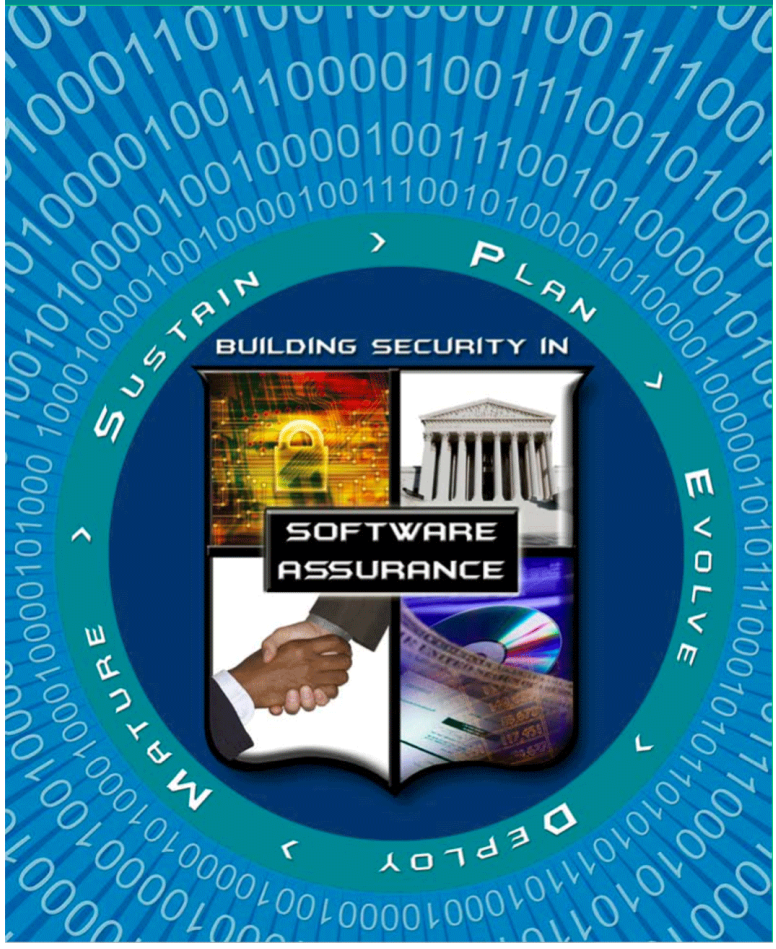
Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses

Software Assurance Pocket Guide Series:
Development, Volume II
Version 2.2, September 26, 2012



Software Assurance (SwA) Pocket Guide Resources

This is a resource for 'getting started' in selecting and adopting relevant practices for engineering, developing, and delivering secure software. As part of the Software Assurance (SwA) Pocket Guide series, this resource is offered for informative use only; it is not intended as directive or presented as being comprehensive since it references and summarizes material in the source documents and on-line resources that provide detailed information. When referencing any part of this document, please provide proper attribution and reference the source documents, when applicable.

This volume of the SwA Pocket Guide series focuses on key practices for mitigating the most egregious exploitable software weaknesses. It identifies mission/business risks attributable to the respective weaknesses, it identifies common attacks that exploit those weaknesses, and provides recommended practices for preventing the weaknesses. It provides insight for how software weaknesses are prioritized to guide training, development and procurement efforts.

At the back of this pocket guide are references, limitation statements, and a listing of topics addressed in the SwA Pocket Guide series. All SwA Pocket Guides and SwA-related documents are freely available for download via the SwA Community Resources and Information Clearinghouse at http://buildsecurityin.us-cert.gov/swa .

Acknowledgements

The SwA Forum and Working Groups function as a stakeholder mega-community that welcomes additional participation in advancing software security and refining SwA-related information resources that are offered free for public use. Input to all SwA resources is encouraged. Please contact Software.Assurance@dhs.gov for comments and inquiries.

The SwA Forum is composed of government, industry, and academic members. The SwA Forum focuses on incorporating SwA considerations in educatoin, acquisition, and development processes relative to potential risk exposures that could be introduced by software and the software supply chain.

Participants in the SwA Forum's Processes & Practices Working Group collaborated with the Technology, Tools and Product Evaluation Working Group in developing the material used in this pocket guide as a step in raising awareness on how to incorporate SwA throughout the Software Development Life Cycle (SDLC).

Lacking common characterization of exploitable software constructs and how they could be attacked, along with associated mitigation practices, previously presented one of the major challenges to realizing software assurance

objectives. As part of the Software Assurance public-private collaboration efforts, the Department of Homeland Security (DHS) National Cyber Security Division (NCSD), together with other Federal partners, has provided sponsorship of Common Weakness Enumeration (CWE) and the Common Attack Pattern Enumeration and Classification (CAPEC) that continue to mature through more widespread use. If not mitigated, these software weaknesses could be sources for future exploitation in the form of new vulnerabilities and vectors for zero-day attacks.

Information contained in this pocket guide is primarily derived from ***"2011 CWE/SANS Top 25 Most Dangerous Software Errors"*** published in the Common Weakness Enumeration (CWE) and SANS websites https://cwe.mitre.org/top25/ and http://www.sans.org/top25errors/ . Material was also contributed from the CERT Secure Coding Practices at http://www.securecoding.cert.org .

Special thanks to the Department of Homeland Security (DHS) National Cyber Security Division's Software Assurance team, the MITRE CWE and CAPEC teams, and the SEI Secure Coding team who provided much of the support to enable the successful completion of this guide and related SwA documents.

Overview

International in scope and free for public use, the Common Weakness Enumeration (CWE) is a community-developed dictionary of software weaknesses. The CWE is a publicly available resource that is collaboratively evolving through public-private contributions. The CWE provides the requisite characterization of exploitable software constructs; improving the education and training of programmers on how to eliminate all-too-common errors before software is delivered and put into operation. This aligns with the "Build Security In" approach to software assurance that emphasizes the need for software to be developed more securely; avoiding security issues in the longer term. The CWE provides a standard means for understanding residual risks; enabling more informed decision-making by suppliers and consumers about the security of software. The attack patterns that can be used to exploit a particular CWE are listed in terms of the Common Attack Pattern Enumeration and Classification (CAPEC) Identifiers in the CAPEC initiative's collection.

The 2011 CWE/SANS Top 25 Most Dangerous Programming Errors is a list of the most egregious programming errors that can lead to serious exploitable software vulnerabilities. These programming errors occur frequently, are often easy to find, and easy to exploit. They are dangerous because they frequently allow attackers to completely take over the software, steal data, or prevent the software from working as intended. Addressing these CWEs will go a long way in securing software, both in development and in operation.

The Top 25 list is the result of collaboration between the SANS Institute, MITRE, and many top software security experts in the US and Europe. It leverages experiences in the development of the SANS Top 20 attack vectors (http://www.sans.org/top20/) and MITRE's Common Weakness Enumeration (CWE) (http://cwe.mitre.org/). MITRE maintains the CWE and CAPEC websites, with support and sponsorship from the US Department of Homeland Security's National Cyber Security Division, presenting detailed descriptions of the top 25 programming errors along with authoritative guidance for mitigating and avoiding them. The CWE site also contains data on more than 800 additional programming errors, design errors, and architecture errors that can lead to exploitable vulnerabilities.

A goal for the CWE Top 25 list is to stop vulnerabilities at the source of exploitable weaknesses by educating programmers on how to eliminate the most egregious programming errors before software is shipped. The list could be used as a tool for education and awareness that helps programmers prevent the kinds of vulnerabilities that plague the software industry. Software consumers could use the same list to help them to ask for more secure software, and software managers and CIOs could use the CWE Top 25 list as a measuring stick of progress in their efforts to secure their software.

While some have shared skepticism about the use of "Top-N" lists, many acknowledge that such lists raise attention and enable change. Focusing only on the list could result in missing the underlying messages. Sound engineering principles are the foundation required to build a strong and reliable infrastructure. Lists, if used to only look at a few issues and flaws, could be misused. Secure software and systems engineering and development hygiene are more than just preventing a few bad practices. What has been missing until now has been a collaboratively developed list of the most egregious programming security defects and appreciation of how software is attacked. There has been no counterpart to the lists that specifically address the programming mistakes and not just the vulnerabilities. This CWE Top 25 list

draws attention to the programmatic problems that lead to exploitable vulnerabilities; enabling discussion to move from talking about the symptoms to addressing the problems. Similar to other Top-N lists, this CWE Top 25 list is not a comprehensive compilation of programming errors. There are many other programming mistakes that can be made, but this provides an effective focus for starting more security-focused risk mitigation efforts. A more comprehensive list of programming errors, specific to the C, C++, and Java programming languages can be found in *The CERT C Secure Coding Standard* [Seacord 09] and on the CERT Secure Coding Wiki.

Some lists have been adopted by organizations as mere checklists, contributing to negative validation, which happens when someone evaluates something against a set of known bad things and assumes it to be safe if those faults are not found. Positive validation, on the other hand, evaluates something against a set of accepted good attributes and presumes it to be dangerous if it doesn't conform. The danger of negative validation is that people should not focus solely on these 25 bad things; yet raising awareness to these most egregious errors in development is positive and valuable. Some software developers have already mapped the CWE Top 25 mitigation and prevention practices with their software development lifecycle, enabling them to better understand that their development practices mitigate the introduction of exploitable vulnerabilities. Some have already sought to use the CWE Top 25 as key criteria in procurement requirements for software developers. Incorporating measurable security criteria and an appreciation of how software can be attacked as a component of any application security procurement language can help focus developers of custom software; providing more accountability for development work because suppliers would be expected to demonstrate that security is a core element of their application development lifecycle, from design and coding through test.

This pocket guide focuses on key practices for preventing and mitigating the most egregious exploitable software weaknesses. The practices are not represented as being complete or comprehensive; yet they do provide a focus for getting started in SwA efforts.

On-line Resources

More practices and details about mitigating exploitable weaknesses are available via on-line resources.

- *"2011 CWE/SANS Top 25 Most Dangerous Software Errors" at https://cwe.mitre.org/top25/index.html*
- *SwA Community Resources and Information Clearinghouse (CRIC) at https://buildsecurityin.us-cert.gov/swa*
- *Build Security In (BSI) at https://buildsecurityin.us-cert.gov/*
- *Common Weakness Enumeration (CWE) at https://cwe.mitre.org*
- *SwA On-Ramp at https://cwe.mitre.org/community/swa/*
- *SANS Top 20 2007 Security Risks at http://sans.org/top20/*
- *Common Attack Patterns Enumeration and Classification at https://capec.mitre.org*
- *CERT Secure Coding Wiki at https://www.securecoding.cert.org/*
- *Microsoft Security Development Lifecycle (SDL) at http://msdn.microsoft.com/en-us/security/cc448177.aspx*
- *The Microsoft SDL and the CWE/SANS Top 25 at http://blogs.msdn.com/sdl/archive/2009/01/27/sdl-and-the-cwe-sans-top-25.aspx*
- *New York State "Application Security Procurement Language" at http://www.sans.org/appseccontract*
- *"Enhancing the Development Life Cycle to Produce Secure Software" at https://www.thedacs.com/techs/enhanced_life_cycles*
- *"Software Assurance: A Curriculum Guide to the Common Body of Knowledge to Produce, Acquire, and Sustain Secure Software" and "Towards an Organization for Software System Security Principles and Guidelines" at https://buildsecurityin.us-cert.gov/daisy/bsi/dhs/927-BSI.html*
- *Common Weakness Risk Analysis Framework (CWRAF) at https://cwe.mitre.org/cwraf/*
- *Common Weakness Scoring System (CWSS) at https://cwe.mitre.org/cwss/*

» *"Fundamental Practices for Secure Software Development, 2ND EDITION, A Guide to the Most Effective Secure Development Practices in Use Today", SAFECode, February 8, 2011 at* http://www.safecode.org/publications/SAFECode_Dev_Practices0211.pdf

Background

The 2011 CWE/SANS Top 25 Most Dangerous Programming Errors is a consensus list of the most significant programming errors that can lead to serious software vulnerabilities. They occur frequently, are often easy to find, and easy to exploit. They are dangerous because they will frequently allow attackers to completely take over the software, steal data, or prevent the software from working at all.

The list is the result of collaboration between the MITRE CWE team, many top software security experts in the US and Europe, and the SANS Institute. It leverages experiences in the development of the SANS Top 20 attack vectors (http://www.sans.org/top20/), MITRE's Common Weakness Enumeration (CWE) (http://cwe.mitre.org/), and MITRE's Common Attack Pattern Enumeration and Classification (CAPEC) (https://capec.mitre.org/). With the sponsorship and support of the US Department of Homeland Security's National Cyber Security Division Software Assurance Program, MITRE maintains the CWE and CAPEC websites, presenting detailed descriptions of the top 25 programming errors along with authoritative guidance for mitigating and avoiding them. The CWE site also contains data on more than 800 additional programming errors, design errors, and architecture errors that can lead to exploitable vulnerabilities. See CWE Frequently Asked Questions at http://cwe.mitre.org/about/faq.html .

A goal for the CWE Top 25 list is to stop vulnerabilities at the source by educating programmers on how to eliminate all-too-common mistakes before software is even shipped. The list serves as a tool for education and awareness to help programmers prevent the kinds of vulnerabilities that plague the software industry. Software consumers can use the same list to help them to ask for more secure software. Finally, software managers, testers, and CIOs can use the CWE Top 25 list as a means for selecting the best tools and services for their needs and as a measuring stick of progress in their efforts to secure their software.

Top 25 Common Weaknesses

Table 1 provides the Top 25 CWEs organized into three high-level categories that contain multiple CWE entries:

1. Insecure Interaction Between Components
2. Risky Resource Management
3. Porous Defenses

Table 1 - Top 25 Common Weakness Enumeration (CWE)

Insecure Interaction Between Components	
These weaknesses are related to insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems.	
CWE	**Description**
CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
CWE-352	Cross-Site Request Forgery (CSRF)
CWE-434	Unrestricted Upload of File with Dangerous Type
CWE-601	URL Redirection to Untrusted Site ('Open Redirect')

Risky Resource Management	
These weaknesses are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.	
CWE	**Description**
CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-131	Incorrect Calculation of Buffer Size
CWE-134	Uncontrolled Format String
CWE-190	Integer Overflow or Wraparound
CWE-494	Download of Code Without Integrity Check
CWE-676	Use of Potentially Dangerous Function
CWE-829	Inclusion of Functionality from Untrusted Control Sphere
Porous Defenses	
These weaknesses are related to defensive techniques that are often misused, abused, or just plain ignored.	
CWE	**Description**
CWE-250	Execution with Unnecessary Privileges
CWE-306	Missing Authentication for Critical Function
CWE-307	Improper Restriction of Excessive Authentication Attempts
CWE-311	Missing Encryption of Sensitive Data
CWE-327	Use of a Broken or Risky Cryptographic Algorithm
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-759	Use of a One-Way Hash without a Salt
CWE-798	Use of Hard-coded Credentials
CWE-807	Reliance on Untrusted Inputs in a Security Decision
CWE-862	Missing Authorization
CWE-863	Incorrect Authorization

Selection of the Top 25 CWEs

The Top 25 CWE list was first developed at the end of 2008 and is updated on a yearly basis. Approximately 40 software security experts provided feedback, including software developers, scanning tool vendors, security consultants, government representatives, and university professors. Representation was international. Intermediate versions were created and resubmitted to the reviewers before the list was finalized. More details are provided in the Top 25 Process page at http://cwe.mitre.org/top25/process.html .

To help characterize and prioritize entries in the Top 25 CWE list, a threat model was developed that identified an attacker with solid technical skills and determined enough to invest some time into attacking an organization. Weaknesses in the Top 25 were selected using two primary criteria:

- **Weakness Prevalence:** how often the weakness appears in software that was not developed with security integrated into the software development life cycle (SDLC).
- **Consequences:** the typical consequences of exploiting a weakness if it is present, such as unexpected code execution, data loss, or denial of service.

Prevalence was determined based on estimates from multiple contributors to the Top 25 list, since appropriate statistics were not readily available.

With these criteria, future versions of the Top 25 CWEs will evolve to cover different weaknesses. Other CWEs that represent significant risks were listed as being on the cusp, and they can be viewed at http://cwe.mitre.org/ .

Information about the Weaknesses

The primary audience for CWE information is intended to be software programmers and designers. For each individual CWE entry, additional information is provided.

- CWE ID and name.
- Supporting data fields: supplementary information about the weakness that may be useful for decision-makers to further prioritize the entries.
- Discussion: Short, informal discussion of the nature of the weakness and its consequences.
- Prevention and Mitigations: steps that developers can take to mitigate or eliminate the weakness. Developers may choose one or more of these mitigations to fit their own needs. Note that the effectiveness of these techniques vary, and multiple techniques may be combined for greater defense-in-depth.
- Related CWEs: other CWE entries that are related to the Top 25 weakness. Note: This list is illustrative, not comprehensive.
- Related Attack Patterns: CAPEC entries for attacks that may be successfully conducted against the weakness. Note: the list is not necessarily complete.

See http://cwe.mitre.org for the additional supporting information on each CWE.

Other Supporting Data Fields in CWEs

Each Top 25 entry includes supporting data fields for weakness prevalence and consequences. Each entry also includes the following data fields.

- **Attack Frequency:** how often the weakness occurs in vulnerabilities that are exploited by an attacker.
- **Ease of Detection:** how easy it is for an attacker to find this weakness.
- **Remediation Cost:** the amount of effort required to fix the weakness.
- **Attacker Awareness:** the likelihood that an attacker is going to be aware of this particular weakness, methods for detection, and methods for exploitation.

Associated Mission/Business Risks and Related Attack Patterns

For each common weakness in software, there are associated risks to the mission or business enabled by the software. Moreover, there are common attack patterns that exploit those weaknesses.

Attack patterns are powerful mechanisms that capture and communicate the attacker's perspective. They are descriptions of common methods for exploiting software. They derive from the concept of design patterns applied in a destructive rather than constructive context and are generated from in-depth analysis of specific real-world exploit examples. To assist in enhancing security throughout the software development lifecycle, and to support the needs of developers, testers and educators, the **CWE and Common Attack Pattern Enumeration and Classification (CAPEC)** are co-sponsored by DHS National Cyber Security Division as part of the Software Assurance strategic initiative, and the efforts are managed by MITRE. The CAPEC website provides a publicly available catalog of attack patterns along with a comprehensive schema and classification taxonomy. CAPEC will continue to evolve with public participation and contributions to form a standard mechanism for identifying, collecting, refining, and sharing attack patterns among the software community.

Development teams should use attack patterns to understand the resilience of their software relative to common attacks and misuse. Table 2 lists the Mission/Business risks associated with each CWE, and it lists some of the possible attacks and misuses associated with the relevant CWEs which enable exploitation of the software.

For a full listing and description of all the attacks related to a particular CWE visit the websites for CWE and CAPEC at http://cwe.mitre.org and http://capec.mitre.org .

Table 2 - CWEs and Their Related Attack Patterns and Mission/Business Risks

CWE	Related Attack Pattern	Mission/Business Risks
CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	» CAPEC-23: File System Function Injection, Content Based » CAPEC-64: Using Slashes and URL Encoding Combined to Bypass Validation Logic » CAPEC-76: Manipulating Input to File System Calls » CAPEC-78: Using Escaped Slashes in Alternate Encoding » CAPEC-79: Using Slashes in Alternate Encoding » CAPEC-139: Relative Path Traversal	» DoS: crash / exit / restart » Execute unauthorized code or commands » Modify files or directories » Read files or directories
CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	» CAPEC-6: TCP Header » CAPEC-15: Command Delimiters » CAPEC-43: Exploiting Multiple Input Interpretation Layers » CAPEC-88: OS Command Injection » CAPEC-108: Command Line Execution through SQL Injection	» DoS: crash / exit / restart » Execute unauthorized code or commands » Hide activities » Modify application data » Modify files or directories » Read application data » Read files or directories
CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	» CAPEC-18: Embedding Scripts in Nonscript Elements » CAPEC-19: Embedding Scripts within Scripts » CAPEC-32: Embedding Scripts in HTTP Query Strings » CAPEC-63: Simple Script Injection » CAPEC-85: Client Network Footprinting (using AJAX/XSS) » CAPEC-86: Embedding Script (XSS) in HTTP Headers » CAPEC-91: XSS in IMG Tags » CAPEC-106: Cross Site Scripting through Log Files » CAPEC-198: Cross-Site Scripting in Error Pages » CAPEC-199: Cross-Site Scripting Using Alternate Syntax » CAPEC-209: Cross-Site Scripting Using MIME Type Mismatch » CAPEC-232: Exploitation of Privilege/Trust » CAPEC-243: Cross-Site Scripting in Attributes » CAPEC-244: Cross-Site Scripting via Encoded URI Schemes	» Bypass protection mechanism » Execute unauthorized code or commands » Read application data

Table 2 - CWEs and Their Related Attack Patterns and Mission/Business Risks		
CWE	**Related Attack Pattern**	**Mission/Business Risks**
	» CAPEC-245: Cross-Site Scripting Using Doubled Characters, e.g. %3C %3Cscript » CAPEC-246: Cross-Site Scripting Using Flash » CAPEC-247: Cross-Site Scripting with Masking through Invalid Characters in Identifiers	
CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	» CAPEC-7: Blind SQL Injection » CAPEC-66: SQL Injection » CAPEC-108: Command Line Execution through SQL Injection » CAPEC-109: Object Relational Mapping Injection » CAPEC-110: SQL Injection through SOAP Parameter Tampering » CAPEC-470: Expanding Control over the Operating System from the Database	» Bypass protection mechanism » Modify application data » Read application data
CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	» CAPEC-8: Buffer Overflow in an API Call » CAPEC-9: Buffer Overflow in Local Command-Line Utilities » CAPEC-10: Buffer Overflow via Environment Variables » CAPEC-14: Client-side Injection-induced Buffer Overflow » CAPEC-24: Filter Failure through Buffer Overflow » CAPEC-42: MIME Conversion » CAPEC-44: Overflow Binary Resource File » CAPEC-45: Buffer Overflow via Symbolic Links » CAPEC-46: Overflow Variables and Tags » CAPEC-47: Buffer Overflow via Parameter Expansion » CAPEC-67: String Format Overflow in syslog() » CAPEC-92: Forced Integer Overflow » CAPEC-100: Overflow Buffers	» DoS: crash / exit / restart » DoS: resource consumption (CPU) » Execute unauthorized code or commands
CWE-131 : Incorrect Calculation of Buffer Size	» CAPEC-47: Buffer Overflow via Parameter Expansion » CAPEC-100: Overflow Buffers	» DoS: crash / exit / restart » Execute unauthorized code or commands » Modify memory » Read memory
CWE-134 : Uncontrolled Format String	» CAPEC-67: String Format Overflow in syslog() » CAPEC-135: Format String Injection	» Execute unauthorized code or commands » Read memory
CWE-190 : Integer Overflow or Wraparound	» CAPEC-92: Forced Integer Overflow	» Bypass protection mechanism » DoS: crash / exit / restart

Table 2 - CWEs and Their Related Attack Patterns and Mission/Business Risks		
CWE	**Related Attack Pattern**	**Mission/Business Risks**
		» DoS: instability » DoS: resource consumption (CPU) » DoS: resource consumption (memory) » Execute unauthorized code or commands » Modify memory
CWE-250 : Execution with Unnecessary Privileges	» CAPEC-69: Target Programs with Elevated Privileges » CAPEC-104: Cross Zone Scripting » CAPEC-470: Expanding Control over the Operating System from the Database	» DoS: crash / exit / restart » Execute unauthorized code or commands » Gain privileges / assume identity » Read application data
CWE-306 : Missing Authentication for Critical Function	» CAPEC-12: Choosing a Message/ Channel Identifier on a Public/ Multicast Channel » CAPEC-36: Using Unpublished Web Service APIs » CAPEC-40: Manipulating Writeable Terminal Devices » CAPEC-62: Cross Site Request Forgery (aka Session Riding) » CAPEC-225: Exploitation of Authentication	» Gain privileges / assume identity » Other
CWE-307 : Improper Restriction of Excessive Authentication Attempts	» CAPEC-16: Dictionary-based Password Attack » CAPEC-49: Password Brute Forcing » CAPEC-55: Rainbow Table Password Cracking » CAPEC-70: Try Common(default) Usernames and Passwords » CAPEC-112: Brute Force	» Bypass protection mechanism
CWE-311 : Missing Encryption of Sensitive Data	» CAPEC-31: Accessing/Intercepting/ Modifying HTTP Cookies » CAPEC-37: Lifting Data Embedded in Client Distributions » CAPEC-65: Passively Sniff and Capture Application Code Bound for Authorized Client » CAPEC-117: Data Interception Attacks » CAPEC-155: Screen Temporary Files for Sensitive Information » CAPEC-157: Sniffing Attacks » CAPEC-167: Lifting Sensitive Data from the Client » CAPEC-204: Lifting cached, sensitive data embedded in client distributions (thick or thin) » CAPEC-205: Lifting credential(s)/ key material embedded in client distributions (thick or thin) » CAPEC-258: Passively Sniffing and Capturing Application Code Bound	» Modify application data » Read application data

Table 2 - CWEs and Their Related Attack Patterns and Mission/Business Risks		
CWE	**Related Attack Pattern**	**Mission/Business Risks**
	for an Authorized Client During Dynamic Update » CAPEC-259: Passively Sniffing and Capturing Application Code Bound for an Authorized Client During Patching » CAPEC-260: Passively Sniffing and Capturing Application Code Bound for an Authorized Client During Initial Distribution » CAPEC-383: Harvesting Usernames or UserIDs via Application API Event Monitoring » CAPEC-384: Application API Message Manipulation via Man-in-the-Middle » CAPEC-385: Transaction or Event Tampering via Application API Manipulation » CAPEC-386: Application API Navigation Remapping » CAPEC-387: Navigation Remapping To Propagate Malicoius Content » CAPEC-388: Application API Button Hijacking » CAPEC-389: Content Spoofing Via Application API Manipulation	
CWE-327 : Use of a Broken or Risky Cryptographic Algorithm	» CAPEC-20: Encryption Brute Forcing » CAPEC-97: Cryptanalysis » CAPEC-459: Creating a Rogue Certificate Authority Certificate	» Hide activities » Modify application data » Read application data
CWE-352 : Cross-Site Request Forgery (CSRF)	» CAPEC-62: Cross Site Request Forgery (aka Session Riding) » CAPEC-111: JSON Hijacking (aka JavaScript Hijacking) » CAPEC-462: Cross-Domain Search Timing » CAPEC-467: Cross Site Identification	» Bypass protection mechanism » DoS: crash / exit / restart » Gain privileges / assume identity » Modify application data » Read application data
CWE-434 : Unrestricted Upload of File with Dangerous Type	» CAPEC-1: Accessing Functionality Not Properly Constrained by ACLs » CAPEC-122: Exploitation of Authorization	» Execute unauthorized code or commands
CWE-494 : Download of Code Without Integrity Check	» CAPEC-184: Software Integrity Attacks » CAPEC-185: Malicious Software Download » CAPEC-186: Malicious Software Update » CAPEC-187: Malicious Automated Software Update	» Alter execution logic » Execute unauthorized code or commands » Other
CWE-601 : URL Redirection to Untrusted Site ('Open Redirect')	» CAPEC-194: Fake the Source of Data	» Bypass protection mechanism » Gain privileges / assume identity

Table 2 - CWEs and Their Related Attack Patterns and Mission/Business Risks		
CWE	**Related Attack Pattern**	**Mission/Business Risks**
		» Other
CWE-676 : Use of Potentially Dangerous Function	» CAPEC-113: API Abuse/Misuse	» Quality degradation » Unexpected state » Varies by context
CWE-732 : Incorrect Permission Assignment for Critical Resource	» CAPEC-1: Accessing Functionality Not Properly Constrained by ACLs » CAPEC-17: Accessing, Modifying or Executing Executable Files » CAPEC-60: Reusing Session IDs (aka Session Replay) » CAPEC-61: Session Fixation » CAPEC-62: Cross Site Request Forgery (aka Session Riding) » CAPEC-122: Exploitation of Authorization » CAPEC-127: Directory Indexing » CAPEC-180: Exploiting Incorrectly Configured Access Control Security Levels » CAPEC-232: Exploitation of Privilege/Trust » CAPEC-234: Hijacking a privileged process	» Gain privileges / assume identity » Modify application data » Other » Read application data » Read files or directories
CWE-759 : Use of a One-Way Hash without a Salt	» CAPEC-20: Encryption Brute Forcing » CAPEC-55: Rainbow Table Password Cracking » CAPEC-97: Cryptanalysis	» Bypass protection mechanism » Gain privileges / assume identity
CWE-798 : Use of Hard-coded Credentials	» CAPEC-70: Try Common(default) Usernames and Passwords » CAPEC-188: Reverse Engineering » CAPEC-189: Software Reverse Engineering » CAPEC-190: Reverse Engineer an Executable to Expose Assumed Hidden Functionality or Content » CAPEC-191: Read Sensitive Strings Within an Executable » CAPEC-205: Lifting credential(s)/ key material embedded in client distributions (thick or thin)	» Bypass protection mechanism » Execute unauthorized code or commands » Gain privileges / assume identity » Other » Read application data
CWE-807 : Reliance on Untrusted Inputs in a Security Decision	» CAPEC-232: Exploitation of Privilege/Trust	» Bypass protection mechanism » Gain privileges / assume identity » Varies by context
CWE-829 : Inclusion of Functionality from Untrusted Control Sphere	» CAPEC-38: Leveraging/ Manipulating Configuration File Search Paths » CAPEC-101: Server Side Include (SSI) Injection » CAPEC-103: Clickjacking » CAPEC-111: JSON Hijacking (aka JavaScript Hijacking) » CAPEC-175: Code Inclusion » CAPEC-181: Flash File Overlay	» Execute unauthorized code or commands

Table 2 - CWEs and Their Related Attack Patterns and Mission/Business Risks		
CWE	**Related Attack Pattern**	**Mission/Business Risks**
	» CAPEC-184: Software Integrity Attacks » CAPEC-185: Malicious Software Download » CAPEC-193: PHP Remote File Inclusion » CAPEC-222: iFrame Overlay » CAPEC-251: Local Code Inclusion » CAPEC-252: PHP Local File Inclusion » CAPEC-253: Remote Code Inclusion	
CWE-862 : Missing Authorization	» CAPEC-1: Accessing Functionality Not Properly Constrained by ACLs » CAPEC-17: Accessing, Modifying or Executing Executable Files » CAPEC-58: Restful Privilege Elevation » CAPEC-122: Exploitation of Authorization » CAPEC-180: Exploiting Incorrectly Configured Access Control Security Levels	» Bypass protection mechanism » Gain privileges / assume identity » Modify application data » Modify files or directories » Read application data » Read files or directories
CWE-863 : Incorrect Authorization	» CAPEC-1: Accessing Functionality Not Properly Constrained by ACLs » CAPEC-17: Accessing, Modifying or Executing Executable Files » CAPEC-58: Restful Privilege Elevation » CAPEC-122: Exploitation of Authorization » CAPEC-180: Exploiting Incorrectly Configured Access Control Security Levels	» Bypass protection mechanism » Gain privileges / assume identity » Modify application data » Modify files or directories » Read application data » Read files or directories

Key Practices

The key practices documented in "2011 CWE/SANS Top 25 Most Dangerous Programming Errors" focus on preventing and mitigating dangerous programming errors. Some of the Key Practices specified in the pocket guide are derived from mitigation recommendations that were common across many of the CWEs in the CWE Top 25, and others came from approaches described on the CERT Secure Coding Wiki. Additional information on preventing the various weaknesses is available in the CERT Secure Coding Wiki at https://www.securecoding.cert.org/ and other websites listed under On-Line Resources of this SwA Pocket Guide. Development teams are also encouraged to use the CAPEC attack patterns to gain understanding of how their software can be attacked, as well as considering how they can engineer their software to better handle such attacks. They are also encouraged to use the CAPEC attack patterns to develop tests that can determine the resilience of their code relative to the common attacks used to exploit software weaknesses. In this SwA Pocket Guide the key practices are grouped in tables according to Software Development Life Cycle (SDLC) phases:

1. Requirements, Architecture, and Design (Table 3) ;
2. Build, Compilation, Implementation, Testing, and Documentation (Table 4) ;
3. Installation, Operation and System Configuration (Table 5) , and

4. Associated CERT Coding Rules (Table 6) .

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.	
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap provide this capability.[R.22.3]	
If at all possible, use library calls rather than external processes to recreate the desired functionality.	CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
For any data that will be used to generate a command to be executed, keep as much of that data out of external control as possible. For example, in web applications, this may require storing the data locally in the session's state instead of sending it out to the client in a hidden form field.	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using the ESAPI Encoding control [R.78.8] or a similar tool, library, or framework. These will help the programmer encode outputs in a manner less prone to error.	
If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. Some languages offer multiple functions that can be used to invoke commands. Where possible, identify any function that invokes a command shell using a single string, and replace it with a function that requires individual arguments. These functions typically perform appropriate quoting and filtering of arguments. For example, in C, the system() function accepts a string that contains the entire command to be executed, whereas execl(), execve(), and others require an array of strings, one for each argument. In Windows, CreateProcess() only accepts one command at a time. In Perl, if system() is provided with an array of arguments, then it will quote each of the arguments.	

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.	CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.	
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using persistence layers such as Hibernate or Enterprise Java Beans, which can provide significant protection against SQL injection if used properly.	CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. Process SQL queries using prepared statements, parameterized queries, or stored procedures. These features should accept parameters or variables and support strong typing. Do not dynamically construct and execute query strings within these features using "exec" or similar functionality, since this may re-introduce the possibility of SQL injection. [R.89.3]	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.	
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and	CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
C#, typically provide overflow protection, but the protection can be disabled by the programmer. Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Examples include the Safe C String Library (SafeStr) by Messier and Viega [R.120.4], and the Strsafe.h library from Microsoft [R.120.3]. These libraries provide safer versions of overflow-prone string-handling functions.	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	CWE-131 : Incorrect Calculation of Buffer Size
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Use libraries or frameworks that make it easier to handle numbers without unexpected consequences, or buffer allocation routines that automatically track buffer size. Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++). [R.131.1]	
Choose a language that is not subject to this flaw.	CWE-134 : Uncontrolled Format String
Ensure that all protocols are strictly defined, such that all out-of-bounds behavior can be identified simply, and require strict conformance to the protocol.	CWE-190 : Integer Overflow or Wraparound
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. If possible, choose a language or compiler that performs automatic bounds checking.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Use libraries or frameworks that make it easier to handle numbers without unexpected consequences. Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++). [R.190.5]	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass	

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
Identify the functionality that requires additional privileges, such as access to privileged operating system resources. Wrap and centralize this functionality if possible, and isolate the privileged code as much as possible from other code [R.250.2]. Raise your privileges as late as possible, and drop them as soon as possible to avoid CWE-271. Avoid weaknesses such as CWE-288 and CWE-420 by protecting all possible communication channels that could interact with your privileged code, such as a secondary socket that you only intend to be accessed by administrators.	CWE-250 : Execution with Unnecessary Privileges
Divide your software into anonymous, normal, privileged, and administrative areas. Identify which of these areas require a proven user identity, and use a centralized authentication capability. Identify all potential communication channels, or other means of interaction with the software, to ensure that all channels are appropriately protected. Developers sometimes perform authentication at the primary channel, but open up a secondary channel that is assumed to be private. For example, a login mechanism may be listening on one network port, but after successful authentication, it may open up a second port where it waits for the connection, but avoids authentication because it assumes that only the authenticated party will connect to the port. In general, if the software or protocol allows a single session or user state to persist across multiple connections or channels, authentication and appropriate credential management need to be used throughout.	CWE-306 : Missing Authentication for Critical Function
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
Where possible, avoid implementing custom authentication routines and consider using authentication capabilities as provided by the surrounding framework, operating system, or environment. These may make it easier to provide a clear separation between authentication tasks and authorization tasks. In environments such as the World Wide Web, the line between authentication and authorization is sometimes blurred. If custom authentication routines are required instead of those provided by the server, then these routines must be applied to every single page, since these pages could be requested directly.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using libraries with authentication capabilities such as OpenSSL or the ESAPI Authenticator [R.306.3].	
Common protection mechanisms include: Disconnecting the user after a small number of failed attempts Implementing a timeout Locking out a targeted account Requiring a computational task on the user's part.	CWE-307 : Improper Restriction of Excessive Authentication Attempts

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Consider using libraries with authentication capabilities such as OpenSSL or the ESAPI Authenticator. [R.307.1]	
Clearly specify which data or resources are valuable enough that they should be protected by encryption. Require that any transmission or storage of this data/resource should use well-vetted encryption algorithms.	CWE-311 : Missing Encryption of Sensitive Data
Using threat modeling or other techniques, assume that your data can be compromised through a separate vulnerability or weakness, and determine where encryption will be most effective. Ensure that data you believe should be private is not being inadvertently exposed using weaknesses such as insecure permissions (CWE-732). [R.311.1]	
Ensure that encryption is properly integrated into the system design, including but not necessarily limited to: Encryption that is needed to store or transmit private data of the users of the system Encryption that is needed to protect the system itself from unauthorized disclosure or tampering Identify the separate needs and contexts for encryption: One-way (i.e., only the user or recipient needs to have the key). This can be achieved using public key cryptography, or other techniques in which the encrypting party (i.e., the software) does not need to have access to a private key. Two-way (i.e., the encryption can be automatically performed on behalf of a user, but the key must be available so that the plaintext can be automatically recoverable by that user). This requires storage of the private key in a format that is recoverable only by the user (or perhaps by the operating system) in a way that cannot be recovered by others.	
Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis. For example, US government systems require FIPS 140-2 certification. Do not develop your own cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If your algorithm can be compromised if attackers find out how it works, then it is especially weak. Periodically ensure that you aren't using obsolete cryptography. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once regarded as strong.	
Compartmentalize your system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area.	
Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis. For example, US government systems require FIPS 140-2 certification.	CWE-327 : Use of a Broken or Risky Cryptographic Algorithm

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
Do not develop your own cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If your algorithm can be compromised if attackers find out how it works, then it is especially weak. Periodically ensure that you aren't using obsolete cryptography. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once regarded as strong. [R.327.4]	
Design your software so that you can replace one cryptographic algorithm with another. This will make it easier to upgrade to stronger algorithms.	
Carefully manage and protect cryptographic keys (see CWE-320). If the keys can be guessed or stolen, then the strength of the cryptography itself is irrelevant.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Industry-standard implementations will save you development time and may be more likely to avoid errors that can occur during implementation of cryptographic algorithms. Consider the ESAPI Encryption feature.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, use anti-CSRF packages such as the OWASP CSRFGuard. [R.352.3] Another example is the ESAPI Session Management control, which includes a component for CSRF. [R.352.9]	CWE-352 : Cross-Site Request Forgery (CSRF)
Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330). [R.352.5]	
Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation.	
Use the "double-submitted cookie" method as described by Felten and Zeller: When a user visits a site, the site should generate a pseudorandom value and set it as a cookie on the user's machine. The site should require every form submission to include this value as a form value and also as a cookie value. When a POST request is sent to the site, the request should only be considered valid if the form value and the cookie value are the same. Because of the same-origin policy, an attacker cannot read or modify the value stored in the cookie. To successfully submit a form on behalf of the user, the attacker would have to correctly guess the pseudorandom value. If the pseudorandom value is cryptographically strong, this will be prohibitively difficult. This technique requires Javascript, so it may not work for browsers that have Javascript disabled. [R.352.4]	
Do not use the GET method for any request that triggers a state change.	

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
Generate your own filename for an uploaded file instead of the user-supplied filename, so that no external input is used at all.[R.434.1] [R.434.2]	CWE-434 : Unrestricted Upload of File with Dangerous Type
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.	
Consider storing the uploaded files outside of the web document root entirely. Then, use other mechanisms to deliver the files dynamically. [R.434.2]	
Define a very limited set of allowable extensions and only generate filenames that end in these extensions. Consider the possibility of XSS (CWE-79) before you allow .html or .htm file types.	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Speficially, it may be helpful to use tools or frameworks to perform integrity checking on the transmitted code. If you are providing the code that is to be downloaded, such as for automatic updates of your software, then use cryptographic signatures for your code and modify your download clients to verify the signatures. Ensure that your implementation does not contain CWE-295, CWE-320, CWE-347, and related weaknesses. Use code signing technologies such as Authenticode. See references [R.494.1] [R.494.2] [R.494.3].	CWE-494 : Download of Code Without Integrity Check
Use an intermediate disclaimer page that provides the user with a clear warning that they are leaving your site. Implement a long timeout before the redirect occurs, or force the user to click on the link. Be careful to avoid XSS problems (CWE-79) when generating the disclaimer page.	CWE-601 : URL Redirection to Untrusted Site ('Open Redirect')
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. For example, ID 1 could map to "/login.asp" and ID 2 could map to "http://www.example.com/". Features such as the ESAPI AccessReferenceMap provide this capability. [R.601.4]	
Ensure that no externally-supplied requests are honored by requiring that all redirect requests include a unique nonce generated by the application [R.601.1]. Be sure that the nonce is not predictable (CWE-330).	
Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully defining distinct user groups, privileges, and/or roles. Map these against data, functionality, and the related resources. Then set the permissions accordingly. This will allow you to maintain more fine-grained control over your resources. [R.732.2]	CWE-732 : Incorrect Permission Assignment for Critical Resource

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
Generate a random salt each time you process a new password. Add the salt to the plaintext password before hashing it. When you store the hash, also store the salt. Do not use the same salt for every password that you process (CWE-760). [R.759.3]	CWE-759 : Use of a One-Way Hash without a Salt
Use one-way hashing techniques that allow you to configure a large number of rounds, such as bcrypt. This may increase the expense when processing incoming authentication requests, but if the hashed passwords are ever stolen, it significantly increases the effort for conducting a brute force attack, including rainbow tables. With the ability to configure the number of rounds, you can increase the number of rounds whenever CPU speeds or attack techniques become more efficient.	
For outbound authentication: store passwords, keys, and other credentials outside of the code in a strongly-protected, encrypted configuration file or database that is protected from access by all outsiders, including other local users on the same system. Properly protect the key (CWE-320). If you cannot use encryption to protect the file, then make sure that the permissions are as restrictive as possible [R.798.1]. In Windows environments, the Encrypted File System (EFS) may provide some protection.	CWE-798 : Use of Hard-coded Credentials
For inbound authentication: Rather than hard-code a default username and password, key, or other authentication credentials for first time logins, utilize a "first login" mode that requires the user to enter a unique strong password or key.	
If the software must contain hard-coded credentials or they cannot be removed, perform access control checks and limit which entities can access the feature that requires the hard-coded credentials. For example, a feature might only be enabled through the system console instead of through a network connection.	
For inbound authentication using passwords: apply strong one-way hashes to your passwords and store those hashes in a configuration file or database with appropriate access control. That way, theft of the file/database still requires the attacker to try to crack the password. When handling an incoming password during authentication, take the hash of the password and compare it to the hash that you have saved. Use randomly assigned salts for each separate hash that you generate. This increases the amount of computation that an attacker needs to conduct a brute-force attack, possibly limiting the effectiveness of the rainbow table method.	
For front-end to back-end connections: Three solutions are possible, although none are complete. The first suggestion involves the use of generated passwords or keys that are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals. Next, the passwords or keys should be limited at the back end to only performing actions valid for the front end, as opposed to having full access. Finally, the messages sent should be tagged and checksummed with time sensitive values so as to prevent replay-style attacks.	
Store state information and sensitive data on the server side only. Ensure that the system definitively and unambiguously keeps track of its own state and user state and has rules defined for legitimate state transitions. Do not allow any application user to affect state directly in any way other than through legitimate actions leading to state transitions.	CWE-807 : Reliance on Untrusted Inputs in a Security Decision

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
If information must be stored on the client, do not do so without encryption and integrity checking, or otherwise having a mechanism on the server side to catch tampering. Use a message authentication code (MAC) algorithm, such as Hash Message Authentication Code (HMAC) [R.807.2]. Apply this against the state or sensitive data that you have to expose, which can guarantee the integrity of the data - i.e., that the data has not been modified. Ensure that you use an algorithm with a strong hash function (CWE-328).	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. With a stateless protocol such as HTTP, use a framework that maintains the state for you. Examples include ASP.NET View State [R.807.3] and the OWASP ESAPI [R.807.4] Session Management feature. Be careful of language features that provide state support, since these might be provided as a convenience to the programmer and may not be considering security.	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.	CWE-829 : Inclusion of Functionality from Untrusted Control Sphere
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap provide this capability [R.829.1].	
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.	
Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully mapping roles with data and functionality. Use role-based access control (RBAC) [R.862.1] to enforce the roles at the appropriate boundaries. Note that this approach may not protect against horizontal authorization, i.e., it will not protect a user from attacking others with the same role.	CWE-862 : Missing Authorization
Ensure that you perform access control checks related to your business logic. These checks may be different than the access control checks that you apply to more generic resources such as files, connections, processes, memory, and database records. For example, a database may restrict access for medical records to a specific database user, but each record might only be intended to be accessible to the patient and the patient's doctor [R.862.2].	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.	

Table 3 – Requirements, Architecture, and Design	
Prevention and Mitigation Practices	**CWE**
For example, consider using authorization frameworks such as the JAAS Authorization Framework [R.862.5] and the OWASP ESAPI Access Control feature [R.862.4].	
For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page. Users should not be able to access any unauthorized functionality or information by simply requesting direct access to that page. One way to do this is to ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.	
Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully mapping roles with data and functionality. Use role-based access control (RBAC) [R.863.1] to enforce the roles at the appropriate boundaries. Note that this approach may not protect against horizontal authorization, i.e., it will not protect a user from attacking others with the same role.	CWE-863 : Incorrect Authorization
Ensure that you perform access control checks related to your business logic. These checks may be different than the access control checks that you apply to more generic resources such as files, connections, processes, memory, and database records. For example, a database may restrict access for medical records to a specific database user, but each record might only be intended to be accessible to the patient and the patient's doctor.	
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. For example, consider using authorization frameworks such as the JAAS Authorization Framework [R.863.4] and the OWASP ESAPI Access Control feature [R.863.5].	
For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page. Users should not be able to access any unauthorized functionality or information by simply requesting direct access to that page. One way to do this is to ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue."	CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). A blacklist is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When validating filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a whitelist of allowable file extensions, which will help to avoid CWE-434. Do not rely exclusively on a filtering mechanism that removes potentially dangerous characters. This is equivalent to a blacklist, which may be incomplete (CWE-184). For example, filtering "/" is insufficient protection if the filesystem also supports the use of "\" as a directory separator. Another possible error could occur when the filtering is applied in a way that still produces dangerous data (CWE-182). For example, if "../" sequences are removed from the ".../...//" string in a sequential fashion, two instances of "../" would be removed from the original string, but the remaining characters would still form the "../" string.	
Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180). Make sure that the application does not decode the same input twice (CWE-174). Such errors could be used to bypass whitelist validation schemes by introducing dangerous inputs after they have been checked. Use a built-in path canonicalization function (such as realpath() in C) that produces the canonical version of the pathname, which effectively removes ".." sequences and symbolic links (CWE-23, CWE-59). This includes: realpath() in C getCanonicalPath() in Java GetFullPath() in ASP.NET realpath() or abs_path() in Perl realpath() in PHP	
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success. If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not. In the context of path traversal, error messages which disclose path information can help attackers craft the appropriate attack strings to move through the file system hierarchy.	
If you need to use dynamically-generated query strings or commands in spite of the risk, properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist (such as everything that is not alphanumeric or white space).	CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).	
If the program to be executed allows arguments to be specified within an input file or from standard input, then consider using that mode to pass arguments instead of the command line.	
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." When constructing OS command strings, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. This will indirectly limit the scope of an attack, but this technique is less important than proper output encoding and escaping. Note that proper output encoding, escaping, and quoting is the most effective solution for preventing OS command injection, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent OS command injection, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, when invoking a mail program, you might need to allow the subject field to contain otherwise-dangerous inputs like ";" and ">" characters, which would need to be escaped or otherwise handled. In this case, stripping the character might reduce the risk of OS command injection, but it would produce incorrect behavior because the subject field would not be recorded as the user intended. This might seem to be a minor inconvenience, but it could be more important when the program relies on well-structured subject lines in order to pass messages to other components. Even if you make a mistake in your validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address.	
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success. If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
In the context of OS Command Injection, error information passed back to the user might reveal whether an OS command is being executed and possibly which command is being used.	
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/ escaping.	CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
With Struts, you should write all data from form beans with the bean's filter attribute set to true.	
To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.	
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." When dynamically constructing web pages, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. All input should be validated and cleansed, not just parameters that the user is supposed to specify, but all data in the request, including hidden fields, cookies, headers, the URL itself, and so forth. A common mistake that leads to continuing XSS vulnerabilities is to validate only fields that are expected to be redisplayed by the site. It is common to see data from the request that is reflected by the application server or the application that the development team did not anticipate. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the HTTP request is recommended. Note that proper output encoding, escaping, and quoting is the most effective solution for preventing XSS, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent XSS, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, in a chat application, the heart emoticon ("<3") would likely pass the validation step, since it is commonly used. However, it cannot be directly inserted into the web page because it contains the "<" character, which	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
would need to be escaped or otherwise handled. In this case, stripping the "<" might reduce the risk of XSS, but it would produce incorrect behavior because the emoticon would not be recorded. This might seem to be a minor inconvenience, but it would be more important in a mathematical forum that wants to represent inequalities. Even if you make a mistake in your validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address. Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.	
If you need to use dynamically-generated query strings or commands in spite of the risk, properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88). Instead of building your own implementation, such features may be available in the database or programming language. For example, the Oracle DBMS_ASSERT package can check or enforce that parameters have certain properties that make them less vulnerable to SQL injection. For MySQL, the mysql_real_escape_string() API function is available in both C and PHP.	CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." When constructing SQL query strings, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. This will indirectly limit the scope of an attack, but this technique is less important than proper output encoding and escaping. Note that proper output encoding, escaping, and quoting is the most effective solution for preventing SQL injection, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent SQL injection, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, the name "O'Reilly" would likely pass the validation step, since it is a common last name in the English language. However, it cannot be directly inserted into the database because it contains the "'" apostrophe character, which would need to be escaped or otherwise handled. In this case, stripping the apostrophe might reduce the risk of SQL injection, but it would produce incorrect behavior because the wrong name would be recorded.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
When feasible, it may be safest to disallow meta-characters entirely, instead of escaping them. This will provide some defense in depth. After the data is entered into the database, later processes may neglect to escape meta-characters before use, and you may not have control over those processes.	
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success. If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not. In the context of SQL Injection, error messages revealing the structure of a SQL query can help attackers tailor successful attack strings.	
Run or compile your software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows. For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.	CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
Consider adhering to the following rules when allocating and managing an application's memory: Double check that your buffer is as large as you specify. When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string. Check buffer boundaries if accessing the buffer in a loop and make sure you are not in danger of writing past the allocated space. If necessary, truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.	
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.	
If you allocate a buffer for the purpose of transforming, converting, or encoding an input, make sure that you allocate enough memory to handle the largest possible encoding. For example, in a routine that converts "&" characters to "&" for HTML entity encoding, you will need an output buffer that is at least 5 times as large as the input buffer.	CWE-131 : Incorrect Calculation of Buffer Size
Understand your programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation. [R.131.7] Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation.	
Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.	
When processing structured incoming data containing a size field followed by raw data, ensure that you identify and resolve any inconsistencies between the size field and the actual size of the data (CWE-130).	
When allocating memory that uses sentinels to mark the end of a data structure - such as NUL bytes in strings - make sure you also include the sentinel in your calculation of the total amount of memory that must be allocated.	
Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.	
Use sizeof() on the appropriate data type to avoid CWE-467.	
Use the appropriate type for the desired action. For example, in C/C++, only use unsigned types for values that could never be negative, such as height, width, or other numbers related to quantity. This will simplify your sanity checks and will reduce surprises related to unexpected casting.	
Run or compile your software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows. For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.	
Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.	
Ensure that all format string functions are passed a static string which cannot be controlled by the user and that the proper number of arguments are always sent to that	CWE-134 : Uncontrolled Format String

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
function as well. If at all possible, use functions that do not support the %n operator in format strings. [R.134.1] [R.134.2]	
Heed the warnings of compilers and linkers, since they may alert you to improper usage.	
Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range. Use unsigned integers where possible. This makes it easier to perform sanity checks for integer overflows. If you must use signed integers, make sure that your range check includes minimum values as well as maximum values.	CWE-190 : Integer Overflow or Wraparound
Understand your programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation. [R.190.3] Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation.	
Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.	
Perform extensive input validation for any privileged code that must be exposed to the user and reject anything that does not fit your strict requirements.	CWE-250 : Execution with Unnecessary Privileges
When you drop privileges, ensure that you have dropped them successfully to avoid CWE-273. As protection mechanisms in the environment get stronger, privilege-dropping calls may fail even if it seems like they would always succeed.	
If circumstances force you to run with extra privileges, then determine the minimum access level necessary. First identify the different permissions that the software and its users will need to perform their actions, such as file read and write permissions, network socket permissions, and so forth. Then explicitly allow those actions while denying all else [R.250.2]. Perform extensive input validation and canonicalization to minimize the chances of introducing a separate vulnerability. This mitigation is much more prone to error than dropping the privileges in the first place.	
Use naming conventions and strong types to make it easier to spot when sensitive data is being used. When creating structures, objects, or other complex entities, separate the sensitive and non-sensitive data as much as possible.	CWE-311 : Missing Encryption of Sensitive Data
Ensure that your application is free of cross-site scripting issues (CWE-79), because most CSRF defenses can be bypassed using attacker-controlled script.	CWE-352 : Cross-Site Request Forgery (CSRF)
Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.	
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not	CWE-434 : Unrestricted Upload of File with Dangerous Type

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." For example, limiting filenames to alphanumeric characters can help to restrict the introduction of unintended file extensions.	
Ensure that only one extension is used in the filename. Some web servers, including some versions of Apache, may process files based on inner extensions so that "filename.php.gif" is fed to the PHP interpreter.[R.434.1] [R.434.2]	
When running on a web server that supports case-insensitive filenames, ensure that you perform case-insensitive evaluations of the extensions that are provided.	
Do not rely exclusively on sanity checks of file contents to ensure that the file is of the expected type and size. It may be possible for an attacker to hide code in some file segments that will still be executed by the server. For example, GIF images may contain a free-form comments field.	
Do not rely exclusively on the MIME content type or filename attribute when determining how to render a file. Validating the MIME content type and ensuring that it matches the extension is only a partial solution.	
Perform proper forward and reverse DNS lookups to detect DNS spoofing.	CWE-494 : Download of Code Without Integrity Check
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." Use a whitelist of approved URLs or domains to be used for redirection.	CWE-601 : URL Redirection to Untrusted Site ('Open Redirect')
When using a critical resource such as a configuration file, check to see if the resource has insecure permissions (such as being modifiable by any regular user) [R.732.1], and generate an error or even exit the software if there is a possibility that the resource could have been modified by an unauthorized party.	CWE-732 : Incorrect Permission Assignment for Critical Resource
Do not suggest insecure configuration changes in your documentation, especially if those configurations can extend to resources and other software that are outside the scope of your own software.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation	
Prevention and Mitigation Practices	**CWE**
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue." For filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a whitelist of allowable file extensions, which will help to avoid CWE-434.	CWE-829 : Inclusion of Functionality from Untrusted Control Sphere

Table 5 – Installation, Operation and System Configuration	
Prevention and Mitigation Practices	**CWE**
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.	CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
Run the code in an environment that performs automatic taint propagation and prevents any command execution that uses tainted variables, such as Perl's "-T" switch. This will force you to perform validation steps that remove the taint, although you must be careful to correctly validate your inputs so that you do not accidentally mark dangerous inputs as untainted (see CWE-183 and CWE-184).	CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
Use runtime policy enforcement to create a whitelist of allowable commands, then prevent use of any command that does not appear in the whitelist. Technologies such as AppArmor are available to do this.	
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.	
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.	CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.	CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Table 5 – Installation, Operation and System Configuration	
Prevention and Mitigation Practices	**CWE**
Use a feature like Address Space Layout Randomization (ASLR). [R.120.5] [R.120.7]	CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent. [R.120.7] [R.120.9]	
Use a feature like Address Space Layout Randomization (ASLR). [R.131.3][R.131.5]	CWE-131 : Incorrect Calculation of Buffer Size
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent. [R.131.4][R.131.5]	
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.	CWE-601 : URL Redirection to Untrusted Site ('Open Redirect')
For all configuration files, executables, and libraries, make sure that they are only readable and writable by the software's administrator.	CWE-732 : Incorrect Permission Assignment for Critical Resource
Do not assume that the system administrator will manually change the configuration to the settings that you recommend in the manual.	
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.	CWE-829 : Inclusion of Functionality from Untrusted Control Sphere

Table 6 – Associated CERT Coding Rules	
Prevention and Mitigation Practices	**CWE**
FIO02-C: Canonicalize path names originating from untrusted sources	CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
FIO02-CPP: Canonicalize path names originating from untrusted sources	
ENV03-C: Sanitize the environment when invoking external programs	CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
ENV04-C: Do not call system() if you do not need a command processor	
STR02-C: Sanitize data passed to complex subsystems	
IDS07-J: Do not pass untrusted, unsanitized data to the Runtime.exec() method	
STR02-CPP: Sanitize data passed to complex subsystems	
ENV03-CPP: Sanitize the environment when invoking external programs	
ENV04-CPP: Do not call system() if you do not need a command processor	
No associated CERT coding rules listed for this CWE entry.	CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
No associated CERT coding rules listed for this CWE entry.	CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Table 6 – Associated CERT Coding Rules	
Prevention and Mitigation Practices	**CWE**
STR35-C: Do not copy data from an unbounded source to a fixed-length array	CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
STR35-CPP: Do not copy data from an unbounded source to a fixed-length array	
MEM35-C: Allocate sufficient memory for an object	CWE-131 : Incorrect Calculation of Buffer Size
MEM35-CPP: Allocate sufficient memory for an object	
FIO30-C: Exclude user input from format strings	CWE-134 : Uncontrolled Format String
FIO30-C: Exclude user input from format strings	
IDS06-J: Exclude user input from format strings	
FIO30-CPP: Exclude user input from format strings	
INT03-C: Use a secure integer library	CWE-190 : Integer Overflow or Wraparound
INT30-C: Ensure that unsigned integer operations do not wrap	
INT32-C: Ensure that operations on signed integers do not result in overflow	
INT35-C: Evaluate integer expressions in a larger size before comparing or assigning	
MEM07-C: Ensure that the arguments to calloc(), when multiplied, can be represented as	
MEM35-C: Allocate sufficient memory for an object	
INT03-CPP: Use a secure integer library	
INT30-CPP: Ensure that unsigned integer operations do not wrap	
INT32-CPP: Ensure that operations on signed integers do not result in overflow	
INT35-CPP: Evaluate integer expressions in a larger size before comparing or assigning	
MEM07-CPP: Ensure that the arguments to calloc(), when multiplied, can be represented as	
MEM35-CPP: Allocate sufficient memory for an object	
SER09-J: Minimize privileges before deserializing from a privilege context	CWE-250 : Execution with Unnecessary Privileges
No associated CERT coding rules listed for this CWE entry.	CWE-306 : Missing Authentication for Critical Function
No associated CERT coding rules listed for this CWE entry.	CWE-307 : Improper Restriction of Excessive Authentication Attempts
MSC00-J: Use SSLSocket rather than Socket for secure data exchange	CWE-311 : Missing Encryption of Sensitive Data
MSC02-J: Generate strong random numbers	CWE-327 : Use of a Broken or Risky Cryptographic Algorithm
MSC30-CPP: Do not use the rand() function for generating pseudorandom numbers	
MSC32-CPP: Ensure your random number generator is properly seeded	
No associated CERT coding rules listed for this CWE entry.	CWE-352 : Cross-Site Request Forgery (CSRF)

Table 6 – Associated CERT Coding Rules	
Prevention and Mitigation Practices	**CWE**
No associated CERT coding rules listed for this CWE entry.	CWE-434 : Unrestricted Upload of File with Dangerous Type
SEC06-J: Do not rely on the default automatic signature verification provided by	CWE-494 : Download of Code Without Integrity Check
No associated CERT coding rules listed for this CWE entry.	CWE-601 : URL Redirection to Untrusted Site ('Open Redirect')
ERR07-C: Prefer functions that support error checking over equivalent functions that	CWE-676 : Use of Potentially Dangerous Function
FIO01-C: Be careful using functions that use file names for identification	
INT06-C: Use strtol() or a related function to convert a string token to an integer	
INT06-CPP: Use strtol() or a related function to convert a string token to an integer	
FIO01-CPP: Be careful using functions that use file names for identification	
FIO03-J: Create files with appropriate access permission	CWE-732 : Incorrect Permission Assignment for Critical Resource
SEC01-J: Do not allow tainted variables in privileged blocks	
ENV03-J: Do not grant dangerous combinations of permissions	
FIO06-CPP: Create files with appropriate access permissions	
FIO06-C: Create files with appropriate access permissions	
No associated CERT coding rules listed for this CWE entry.	CWE-759 : Use of a One-Way Hash without a Salt
MSC03-J: Never hard code sensitive information	CWE-798 : Use of Hard-coded Credentials
ENV03-CPP: Sanitize the environment when invoking external programs	CWE-807 : Reliance on Untrusted Inputs in a Security Decision
SEC09-J: Do not base security checks on untrusted sources	
No associated CERT coding rules listed for this CWE entry.	CWE-829 : Inclusion of Functionality from Untrusted Control Sphere
No associated CERT coding rules listed for this CWE entry.	CWE-862 : Missing Authorization
No associated CERT coding rules listed for this CWE entry.	CWE-863 : Incorrect Authorization

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
MIT-10	Run or compile your software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows. For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice. 1. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 2. CWE-131 : Incorrect Calculation of Buffer Size
MIT-11	Use a feature like Address Space Layout Randomization (ASLR).[R.XX.A] [R.XX.B]

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
	1. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 2. CWE-131 : Incorrect Calculation of Buffer Size
MIT-12	Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent.[R.XX.A] [R.XX.B] 1. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 2. CWE-131 : Incorrect Calculation of Buffer Size
MIT-13	Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available. 1. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 2. CWE-131 : Incorrect Calculation of Buffer Size
MIT-14	Store state information and sensitive data on the server side only. Ensure that the system definitively and unambiguously keeps track of its own state and user state and has rules defined for legitimate state transitions. Do not allow any application user to affect state directly in any way other than through legitimate actions leading to state transitions. If information must be stored on the client, do not do so without encryption and integrity checking, or otherwise having a mechanism on the server side to catch tampering. Use a message authentication code (MAC) algorithm, such as Hash Message Authentication Code (HMAC) [R.XX.A]. Apply this against the state or sensitive data that you have to expose, which can guarantee the integrity of the data - i.e., that the data has not been modified. Ensure that you use an algorithm with a strong hash function (CWE-328). 1. CWE-807 : Reliance on Untrusted Inputs in a Security Decision
MIT-15	For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 4. CWE-434 : Unrestricted Upload of File with Dangerous Type 5. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 6. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 7. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere 8. CWE-131 : Incorrect Calculation of Buffer Size 9. CWE-190 : Integer Overflow or Wraparound 10. CWE-306 : Missing Authentication for Critical Function 11. CWE-807 : Reliance on Untrusted Inputs in a Security Decision
MIT-16	If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 4. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
	5. CWE-807 : Reliance on Untrusted Inputs in a Security Decision
MIT-17	Run your code using the lowest privileges that are required to accomplish the necessary tasks [R.XX.A]. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-434 : Unrestricted Upload of File with Dangerous Type 4. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 5. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 6. CWE-494 : Download of Code Without Integrity Check 7. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere 8. CWE-131 : Incorrect Calculation of Buffer Size 9. CWE-250 : Execution with Unnecessary Privileges
MIT-18	Identify the functionality that requires additional privileges, such as access to privileged operating system resources. Wrap and centralize this functionality if possible, and isolate the privileged code as much as possible from other code [R.XX.A]. Raise your privileges as late as possible, and drop them as soon as possible to avoid CWE-271. Avoid weaknesses such as CWE-288 and CWE-420 by protecting all possible communication channels that could interact with your privileged code, such as a secondary socket that you only intend to be accessed by administrators. 1. CWE-250 : Execution with Unnecessary Privileges
MIT-19	When you drop privileges, ensure that you have dropped them successfully to avoid CWE-273. As protection mechanisms in the environment get stronger, privilege-dropping calls may fail even if it seems like they would always succeed. 1. CWE-250 : Execution with Unnecessary Privileges
MIT-20	Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180). Make sure that the application does not decode the same input twice (CWE-174). Such errors could be used to bypass whitelist validation schemes by introducing dangerous inputs after they have been checked. 1. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
MIT-21	When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 4. CWE-434 : Unrestricted Upload of File with Dangerous Type 5. CWE-601 : URL Redirection to Untrusted Site ('Open Redirect') 6. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 7. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 8. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
MIT-22	Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software. OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails. 1. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 2. CWE-434 : Unrestricted Upload of File with Dangerous Type 3. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 4. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 5. CWE-494 : Download of Code Without Integrity Check 6. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere 7. CWE-131 : Incorrect Calculation of Buffer Size 8. CWE-732 : Incorrect Permission Assignment for Critical Resource
MIT-24	When there is a need to store or transmit sensitive data, use strong, up-to-date cryptographic algorithms to encrypt that data. Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and use well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis. For example, US government systems require FIPS 140-2 certification. Do not develop your own cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If your algorithm can be compromised if attackers find out how it works, then it is especially weak. Periodically ensure that you aren't using obsolete cryptography. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once regarded as strong. [R.XX.A] 1. CWE-311 : Missing Encryption of Sensitive Data 2. CWE-327 : Use of a Broken or Risky Cryptographic Algorithm
MIT-25	When you use industry-approved techniques, you need to use them correctly. Don't cut corners by skipping resource-intensive steps (CWE-325). These steps are often essential for preventing common attacks. 1. CWE-311 : Missing Encryption of Sensitive Data 2. CWE-327 : Use of a Broken or Risky Cryptographic Algorithm 3. CWE-759 : Use of a One-Way Hash without a Salt
MIT-26	Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system. 1. CWE-131 : Incorrect Calculation of Buffer Size 2. CWE-190 : Integer Overflow or Wraparound
MIT-27	If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
	2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
MIT-28	If you need to use dynamically-generated query strings or commands in spite of the risk, properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88). 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
MIT-29	Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 4. CWE-601 : URL Redirection to Untrusted Site ('Open Redirect') 5. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 6. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere
MIT-3	Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. 1. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 2. CWE-190 : Integer Overflow or Wraparound
MIT-30	For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping. 1. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
MIT-31	To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set. 1. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
MIT-32	Run the code in an environment that performs automatic taint propagation and prevents any command execution that uses tainted variables, such as Perl's "-T" switch. This will force you to perform validation steps that remove the taint, although you must be careful to correctly validate your inputs so that you do not accidentally mark dangerous inputs as untainted (see CWE-183 and CWE-184). 1. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
MIT-33	Use naming conventions and strong types to make it easier to spot when sensitive data is being used. When creating structures, objects, or other complex entities, separate the sensitive and non-sensitive data as much as possible. 1. CWE-311 : Missing Encryption of Sensitive Data
MIT-34	Store library, include, and utility files outside of the web document root, if possible. Otherwise, store them in a separate directory and use the web server's access control capabilities to prevent attackers from directly requesting them. One common practice is to define a fixed constant in each calling program, then check for the existence of the constant in the library/include file; if the constant does not exist, then the file was directly requested, and it can exit immediately. This significantly reduces the chance of an attacker being able to bypass any protection mechanisms that are in the base program but not in the include files. It will also reduce your attack surface. 1. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 2. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere
MIT-36	Understand your programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation. [R.XX.A] Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation. 1. CWE-131 : Incorrect Calculation of Buffer Size 2. CWE-190 : Integer Overflow or Wraparound
MIT-37	Ensure that your software runs properly under the Federal Desktop Core Configuration (FDCC) [R.XX.A] or an equivalent hardening configuration guide, which many organizations use to limit the attack surface and potential risk of deployed software. 1. CWE-250 : Execution with Unnecessary Privileges 2. CWE-732 : Incorrect Permission Assignment for Critical Resource
MIT-39	Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success. If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
MIT-4	Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 4. CWE-352 : Cross-Site Request Forgery (CSRF)

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
	5. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 6. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 7. CWE-494 : Download of Code Without Integrity Check 8. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere 9. CWE-131 : Incorrect Calculation of Buffer Size 10. CWE-190 : Integer Overflow or Wraparound 11. CWE-306 : Missing Authentication for Critical Function 12. CWE-862 : Missing Authorization 13. CWE-807 : Reliance on Untrusted Inputs in a Security Decision 14. CWE-863 : Incorrect Authorization 15. CWE-327 : Use of a Broken or Risky Cryptographic Algorithm 16. CWE-307 : Improper Restriction of Excessive Authentication Attempts
MIT-42	Perform proper forward and reverse DNS lookups to detect DNS spoofing. 1. CWE-494 : Download of Code Without Integrity Check
MIT-5	Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue." Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). A blacklist is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. 1. CWE-89 : Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection') 2. CWE-78 : Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection') 3. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 4. CWE-434 : Unrestricted Upload of File with Dangerous Type 5. CWE-601 : URL Redirection to Untrusted Site ('Open Redirect') 6. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow') 7. CWE-22 : Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal') 8. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere
MIT-6	Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls. 1. CWE-79 : Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting') 2. CWE-601 : URL Redirection to Untrusted Site ('Open Redirect') 3. CWE-829 : Inclusion of Functionality from Untrusted Control Sphere 4. CWE-807 : Reliance on Untrusted Inputs in a Security Decision
MIT-8	Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.

Table 7 - Shared Mitigations	
Mitigation	**CWE Entries**
	1. CWE-131 : Incorrect Calculation of Buffer Size 2. CWE-190 : Integer Overflow or Wraparound
MIT-9	Consider adhering to the following rules when allocating and managing an application's memory: 1. CWE-120 : Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Creating Custom Top-N Lists using CWSS and CWRAF

The CWE/SANS Top 25 is a great starting point, but each organization has its own set of business priorities, threat environment, and risk tolerance. For those who understand those issues, a more refined and custom Top-N for their business, and what software is doing for their business, is possible through the Common Weakness Scoring System (CWSS) (https://cwe.mitre.org/cwss/) and the Common Weakness Risk Analysis Framework (CWRAF) (https://cwe.mitre.org/cwraf/). The mechanisms in CWSS and CWRAF minimize this difficulty by letting organizations model their own business impact considerations into a risk-scoring mechanism.

CWSS provides the mechanism for scoring software's weaknesses in a consistent, flexible, open manner while considering the context and reflecting the weaknesses' impacts against that context. It aims to provide a consistent approach for tools and services prioritizing their static- and dynamic-analysis findings while addressing government, academia, and industry stakeholder needs.

CWRAF uses the core scoring mechanisms from CWSS to let software developers and consumers prioritize their own target list of software weaknesses across their unique portfolio of software applications and projects, focusing on those with the greatest potential to harm their business. To reduce risk, organizations can select appropriate tools and techniques, focus staff training, and define contracting details to ensure outsourced efforts also address the prioritized issues.

CWRAF and CWSS let users create top-n lists for their particular software and business domains, missions, and technology groups. In conjunction with other activities, CWSS and CWRAF help developers and consumers introduce more robust and resilient software into their operational environments.

Key Discussion Points Between Developers and Consumers, Acquirers, and Project Management

Improving software assurance requires a more explicit dialog between consumers, acquirers, project managers, and developers on an ongoing basis. Here are some discussion points that will hopefully provide a better understanding of what you and others are doing and need to do to help improve the security and resilience of your software.

1. Design/Development Practices
    a. Which process model or standard is used that specifies the activities/practices that are followed (e.g. BSIMM, OpenSAMM)?
    b. Which security-related frameworks are used, such as ESAPI or built-in frameworks?
    c. Which SDLC activities are used to directly prevent or mitigate vulnerabilities in the application software? (e.g. threat modeling, automated code analysis (static or dynamic), etc).
    d. Which security controls have been utilized to mitigate specific problems (e.g. authentication, authorization, cryptography)

e. Which secure coding rules/practices are followed? (e.g. CERT, MISRA, ISO SC-22, custom). How is conformance enforced (e.g. automated tools during checkin)?

f. What differences, if any, exist between the secure development practices for legacy code, versus newly developed code?

g. Which "Top N" vulnerability/attack lists do your development practices actively attempt to address (e.g. CWE Top 25, OWASP Top Ten, custom Top-N list)?

2. Third-Party Software Management

   a. Which third-party libraries are used by the software?

   b. How does the development team keep current with third-party libraries so that it does not use code with known vulnerabilities?

   c. How are third-party code changes and vulnerabilities tracked/monitored?

   d. Which third-party libraries were independently examined for vulnerabilities before being included in the software?

3. Detection and Analysis

   a. Which standardized analysis/testing methodologies are used to evaluate the software? (e.g. OWASP ASVS, OSSTMM)

   b. Has an independent 3rd-party review been performed against the software? Did the review cover code implementation, design, architecture, or installation settings?

   d. What tools are used for automated code analysis? Static or dynamic? White box or black box?

   e. Which manual analysis techniques were used?

   f What specifications, data formats, and protocols are used? Were any test case suites or fuzzing tools used to evaluate the implementation (e.g. PROTOS)?

   g. What is the attack surface of the software (in privileged code and overall)? What metrics are used? Can the attack surface or attack patterns be described in terms of CAPEC?

   h. Which parts of the code have been most recently reviewed?

   i. Which parts of the software contain legacy code whose analysis has been skipped?

4. Compiler/Environment

   a. Which compiler settings are used to reduce or eliminate risk for key weaknesses (e.g. /GS switch)?

   b. Were any compiler warnings disabled or ignored when compiling the code? If so, which ones and why?

   c. Was the code compiled using safe libraries?

   d. Which OS features are used to reduce or eliminate the risk of important weaknesses (e.g. DEP, ASLR)?

5. Configuration/Installation

   a. Is the product installed "secure by default"?

   b. Is the product installed so that critical executables, libraries, configuration files, registry keys, etc. cannot be modified by untrusted parties?

   c. Does the software run with limited privileges? If not, how is privilege separation performed?

   d. How does the documentation cover security-relevant settings for administrators to use to lock down the software?

   e. Does the software work under FDCC/USGCB configurations, and/or other secure configurations?

   f. How does the software restrict access to network ports?

6. Vulnerability Response

   a. Is a security response center set up to handle incoming vulnerability reports from external parties?

b. How easy is it for independent researchers and non-customers to report exploitable weaknesses and vulnerabilities relative to this software?

c. Are emergency procedures in place to quickly fix issues that are first discovered being exploited in the wild?

d. Are procedures in place to handle when vulnerabilities are publicly disclosed without notifying the developer or giving sufficient time to produce a patch)?

e. Is there a sufficiently comprehensive set of information sources that are monitored for reported vulnerabilities in your own software, in third-party products, and competitor/analogous products?

f. When a new weakness is found by an outside party, how are the software and associated development practices reviewed and modified to ensure that similar weaknesses are also detected and removed?

7. Vulnerability Disclosure

    a. How are consumers of the software notified about new vulnerabilities found in the code?

    b. For vulnerabilities that are publicly disclosed by other parties without a patch, is there a policy to provide public commentary before a patch is available?

    c. Which details are disclosed to customers? What is disclosed to the general public?

    d. Are any credits or compensation provided to independent vulnerability researchers?

8. What kind of evidence or proof can be offered regarding these claims?

Using Tools and Other Capabilities to Identify the Top 25

Developers and third-party analysts can use CWE-compatible tools that can map to CWE items in the CWE Top 25. With the advancing maturity and increasing adoption of CWE, most vendors of software analysis tools and services express their findings of weaknesses in code, design, and architecture using CWE identifiers. This common language for expressing weaknesses has eliminated much of the ambiguity and confusion surrounding exactly what the tool or service has found. At the same time, different vendors take different approaches as to how they look for weaknesses and what weaknesses they look for. The CWE Coverage Claims Representation (CCR) is a means for software analysis vendors to convey to their customers exactly which CWE-identified weaknesses they claim to be able to locate in software. The word claim is emphasized since neither the CCR itself nor the CWE Compatibility Program verify or otherwise vet these statements of coverage. The CWE Effectiveness Program will eventually fulfill this role of verification.

The main goal of the CCR is to facilitate the communication of unambiguous statements of the intention of a tool or service to discover specific, CWE-identified weaknesses in software. These statements of claim are intended to allow the providers of software analysis tools and services and the consumers of those tools and services to share a single, unambiguous understanding of the scope of software weakness analysis. CCR wants users of tools and services to be aware and informed of the coverage of the tools and services they make use of in analyzing their software, and when specific classes of weaknesses or individual weaknesses are of specific concern, they can make sure their tools and services are at least trying to find them. Having a mis-match between an organization's focus and the capabilities of their tools and services is not something to be discovered after using and depending on them, but rather is something that should be addressed in the initial discussions and exploration of bringing those capabilities to bear for the organization.

It is anticipated that the CCR will also foster innovation in the technology of software analysis tools and services by allowing vendors to clearly state their intentions with respect to weakness discovery and understand more clearly when there is a need for targeting additional weaknesses to address their customer's concerns. Currently, a tool that does a very deep analysis on a small subset of the entire set of CWE-defined weaknesses may be judged as inadequate by potential customers since, by definition, it fails to discover a broad set of weaknesses. However, with the CCR, the tool provider could supply a CCR document for that tool, clearly setting expectations as to the set of weaknesses that the tool attempts to discover. Tool consumers could then evaluate tools based on what specific CWE-identified weaknesses those tools claim to discover and how that coverage fits within their needs, rather than comparing it to the entire set of CWE-defined weaknesses.

Conclusion

The Software Assurance Pocket Guide Series is developed in collaboration with the SwA Forum and Working Groups and provides summary material in a more consumable format. The series provides informative material for SwA initiatives that seek to reduce software vulnerabilities, minimize exploitation, and address ways to improve the routine development, acquisition and deployment of trustworthy software products. Together, these activities will enable more secure and reliable software that supports mission requirements across enterprises and the critical infrastructure.

For additional information or contribution to future material and/or enhancements of this pocket guide, please consider joining any of the SwA Working Groups and/or send comments to Software.Assurance@dhs.gov . SwA Forums are open to all participants and free of charge. Please visit https://buildsecurityin.us-cert.gov for further information.

No Warranty

This material is furnished on an "as-is" basis for information only. The authors, contributors, and participants of the SwA Forum and Working Groups, their employers, the U.S. Government, other participating organizations, all other entities associated with this information resource, and entities and products mentioned within this pocket guide make no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose, completeness or merchantability, exclusivity, or results obtained from use of the material. No warranty of any kind is made with respect to freedom from patent, trademark, or copyright infringement. Reference or use of any trademarks is not intended in any way to infringe on the rights of the trademark holder. No warranty is made that use of the information in this pocket guide will result in software that is secure. Examples are for illustrative purposes and are not intended to be used as is or without undergoing analysis.

Reprints

Any Software Assurance Pocket Guide may be reproduced and/or redistributed in its original configuration, within normal distribution channels (including but not limited to on-demand Internet downloads or in various archived/ compressed formats).

Anyone making further distribution of these pocket guides via reprints may indicate on the pocket guide that their organization made the reprints of the document, but the pocket guide should not be otherwise altered.

These resources have been developed for information purposes and should be available to all with interests in software security.

For more information, including recommendations for modification of SwA pocket guides, please contact Software.Assurance@dhs.gov or visit the Software Assurance Community Resources and Information Clearinghouse: https://buildsecurityin.us-cert.gov/swa to download this document either format (4"x8" or 8.5"x11").

Software Assurance (SwA) Pocket Guide Series

SwA is primarily focused on software security and mitigating risks attributable to software; better enabling resilience in operations. SwA Pocket Guides are provided; with some yet to be published. All are offered as informative resources; not comprehensive in coverage. All are intended as resources for 'getting started' with various aspects of software assurance. The planned coverage of topics in the SwA Pocket Guide Series is listed:

SwA in Acquisition & Outsourcing

I. Contract Language for Integrating Software Security into the Acquisition Life Cycle
II. Software Supply Chain Risk Management & Due-Diligence

SwA in Development

I. Integrating Security into the Software Development Life Cycle
II. Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses
III. Risk-based Software Security Testing
IV. Requirements & Analysis for Secure Software
V. Architecture & Design Considerations for Secure Software
VI. Secure Coding & Software Construction
VII. Security Considerations for Technologies, Methodologies & Languages

SwA Life Cycle Support

I. SwA in Education, Training & Certification
II. Secure Software Distribution, Deployment, & Operations
III. Code Transparency & Software Labels
IV. Assurance Case Management
V. Assurance Process Improvement & Benchmarking
VI. Secure Software Environment & Assurance Ecosystem

SwA Measurement & Information Needs

I. Making Software Security Measurable
II. Practical Measurement Framework for SwA & InfoSec
III. SwA Business Case & Return on Investment

SwA Pocket Guides and related documents are freely available for download via the DHS NCSD Software Assurance Community Resources and Information Clearinghouse at https://buildsecurityin.us-cert.gov/swa .