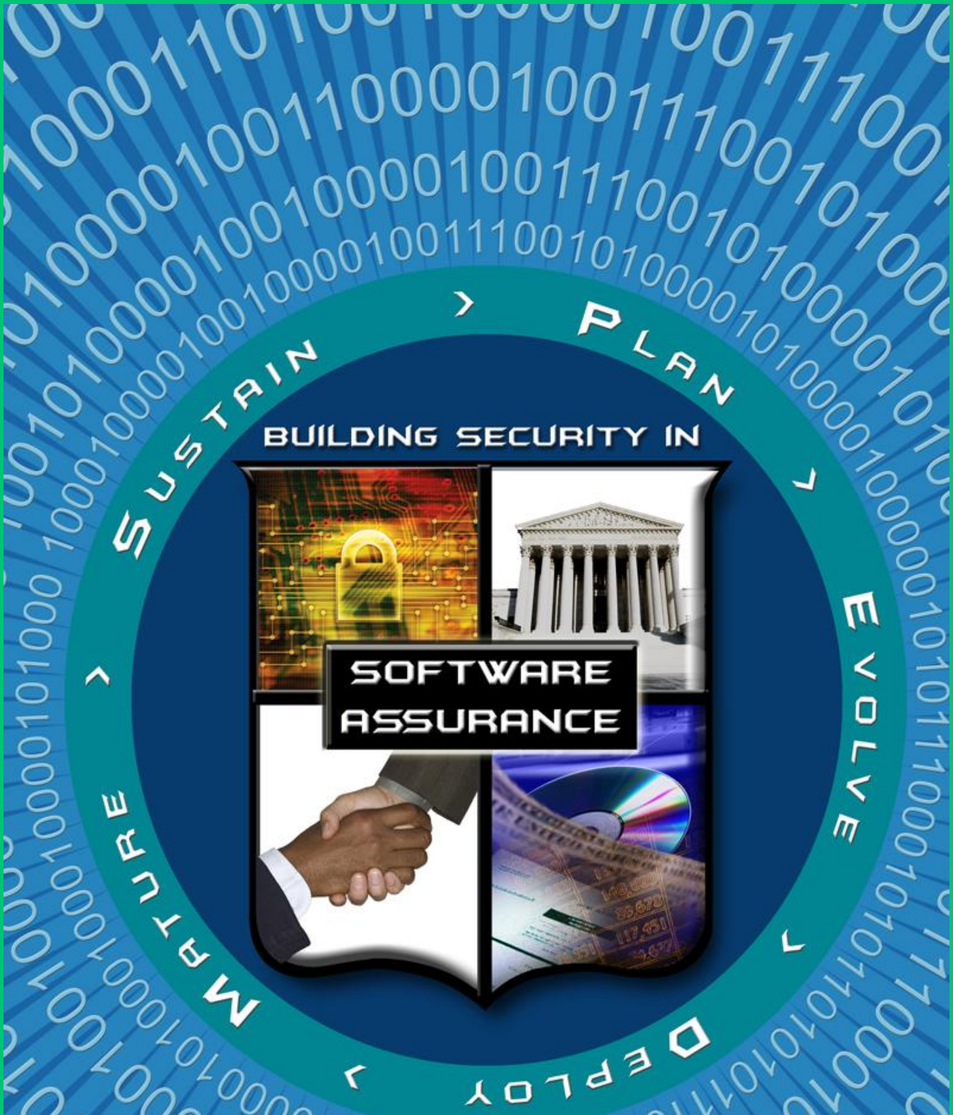
Key Practices for Mitigating the Most Egregious Exploitable Software Weaknesses

Software Assurance Pocket Guide Series:
Development, Volume II
Version 1.3, May 24, 2009



Software Assurance (SwA) Pocket Guide Resources

This is a resource for 'getting started' in selecting and adopting relevant practices for 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 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.

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

SwA Pocket Guides are developed and socialized by the SwA community as a collaborative effort to obtain a common look and feel and are not produced by individual entities. SwA Forum and Working Groups function as a stakeholder meta-community that welcomes additional participation in advancing software security and refining. All SwA-related information resources that are produced are offered free for public use. Inputs to documents for the online resources are invited. Please contact Software.Assurance@dhs.gov for comments and inquiries. For the most up to date pocket guides, check the website at https://buildsecurityin.us-cert.gov/swa/.

The SwA Forum and Working Groups are composed of government, industry, and academic members and focuses on incorporating SwA considerations in the acquisition and development processes relative to potential risk exposures that could be introduced by software and the 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 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) that continues to mature through more wide-spread use.

Information contained in this pocket guide is primarily derived from ***"2009 CWE/SANS Top 25 Most Dangerous Programming Errors"*** published in the Common Weakness Enumeration (CWE) and SANS websites http://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 team 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 which 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.

MITRE manages the CWE 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 website also contains data on more than 700 additional programming errors, design errors, and architecture errors that can lead to exploitable vulnerabilities.

The 2009 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/). A goal for the Top 25 CWE list is to stop vulnerabilities at the source 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 ask for more secure software, and software managers and CIOs could use the Top 25 CWE list as a measuring stick of progress in their efforts to secure their software.

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.

- *"2009 CWE/SANS Top 25 Most Dangerous Programming Errors" at http://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 http://cwe.mitre.org.*
- *SANS Top 20 2007 Security Risks at http://sans.org/top20/.*
- *Common Attack Patterns Enumeration and Classification at http://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.*

Background

The 2009 CWE/SANS Top 25 Most Dangerous Programming Errors is a 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 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/). With the sponsorship and support of the US Department of Homeland Security's National Cyber Security Division, MITRE maintains the CWE website, 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 700 additional programming errors, design errors, and architecture errors that can lead to exploitable vulnerabilities.

Use of the Top 25 CWE List

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 engineering and development hygiene is 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. There has been no counterpart to the lists that specifically address the programming mistakes and not just the vulnerabilities. This Top 25 CWE 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 Top 25 CWE 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 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 Top 25 CWE mitigation and prevention practices with their respective software development life cycles, enabling them to better understand that their development practices mitigate the introduction of exploitable vulnerabilities. Some have already sought to use the Top 25 CWE as key criteria in procurement requirements for software developers. Incorporating measurable security criteria 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 life cycle, from design and coding through test.

A goal for the Top 25 CWE 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 could use the same list to help them to ask for more secure software. Finally, software managers and CIOs can use the Top 25 list 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, and
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-20:	Improper Input Validation.
CWE-116:	Improper Encoding or Escaping of Output.
CWE-89:	Failure to Preserve SQL Query Structure (aka 'SQL Injection').
CWE-79:	Failure to Preserve Web Page Structure (aka 'Cross-site Scripting').
CWE-78:	Failure to Preserve OS Command Structure (aka 'OS Command Injection').
CWE-319:	Cleartext Transmission of Sensitive Information
CWE-352:	Cross-Site Request Forgery (CSRF).
CWE-362:	Race Condition.
CWE-209:	Error Message Information Leak.
Risky Resource Management	

Table 1 - Top 25 Common Weakness Enumeration (CWE) continue	
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-119:	Failure to Constrain Operations within the Bounds of a Memory Buffer.
CWE-642:	External Control of Critical State Data.
CWE-73:	External Control of File Name or Path.
CWE-426:	Untrusted Search Path.
CWE-94:	Failure to Control Generation of Code (aka 'Code Injection').
CWE-494:	Download of Code Without Integrity Check.
CWE-404:	Improper Resource Shutdown or Release.
CWE-665:	Improper Initialization.
CWE-682:	Incorrect Calculation.
Porous Defenses	
These weaknesses are related to defensive techniques that are often misused, abused, or just plain ignored.	
CWE	**Description**
CWE-285:	Improper Access Control (Authorization).
CWE-327:	Use of a Broken or Risky Cryptographic Algorithm.
CWE-259:	Hard-Coded Password.
CWE-732:	Insecure Permission Assignment for Critical Resource.
CWE-330:	Use of Insufficiently Random Values.
CWE-250:	Execution with Unnecessary Privileges.
CWE-602:	Client-Side Enforcement of Server-Side Security.

Selection of the Top 25 CWEs

The Top 25 CWE list was first developed at the end of 2008. 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). (Note: Prevalence was determined based on estimates from multiple contributors to the Top 25 list, since appropriate statistics were not readily available.)

»» **Consequences:** the typical consequences of exploiting a weakness if it is present, such as unexpected code execution, data loss, or denial of service.

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

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

- 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.

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.

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

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 life cycle, 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-20: Improper Input Validation	» Overflow (various types, CAPEC IDs: 8,9, 10, 24, 45, 46, 47, 67). » Subverting Environment Variable Values (CAPEC ID:13). » Injection (various types, CAPEC IDs: 7, 88, 63, 83, 101). » Fuzzing (CAPEC ID: 28). » Embedding Scripts (various types, CAPEC IDs: 18, 32).	» Gain control of compromised system. » Crash of compromised system (denial of service). » Allow execution of malicious/arbitrary code. » Access or modification sensitive data. » Escalate privileges. » System resources modification.
CWE-116: Improper Encoding or Escaping of Output	» Embedding Scripts (various types, CAPEC IDs: 18, 86). » Simple Script Injection(CAPEC ID:63). » User-Controlled Filename (CAPEC ID:73). » Web Logs Tampering (CAPEC ID:81). » Client Network Footprinting (CAPEC ID:85).	» Allow execution of malicious/arbitrary code. » Access or modification of sensitive data. » Escalate privileges. » Render compromised system unusable (AKA denial of service). » Leak information.
CWE-89: Failure to Preserve SQL Query Structure (aka 'SQL Injection')	» Blind SQL Injection (CAPEC ID:7). » SQL Injection (CAPEC ID:66).	» Allow execution of malicious/arbitrary code. » Access or modification of sensitive data. » Leak information.
CWE-79: Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')	» Embedding Scripts (various types, CAPEC IDs: 19, 32, 86). » Client Network Footprinting (using AJAX/XSS, CAPEC ID:85). » XSS in IMG Tags (CAPEC ID:91).	» Allow execution of malicious/arbitrary code. » Escalate privileges. » Leak information.
CWE-78: Failure to Preserve OS Command Structure (aka 'OS Command Injection')	» Argument Injection (CAPEC ID:6). » Command Delimiters (CAPEC ID:15). » Exploiting Multiple Input Interpretation Layers (CAPEC ID:43). » Command Injection (CAPEC ID:88).	» Allow execution of malicious/arbitrary code. » Modify data. » Leak information. » Escalate privileges.
CWE-319: Cleartext Transmission of Sensitive Information	» Passively Sniff and Capture Application Code Bound for Authorized Client (CAPEC ID:65).	» Leak information. » Escalate privileges.
CWE-352: Cross-Site Request Forgery (CSRF)	» Cross Site Request Forgery (aka Session Riding ,	» Leak information.

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

CWE	Related Attack Pattern	Mission/Business Risks
	CAPEC ID:62).	» Escalate privileges. » Modify data.
CWE-362: Race Condition	» Leveraging Race Conditions (CAPEC ID:26). » Leveraging Time-of-Check and Time-of-Use (TOCTOU) Race Conditions (CAPEC ID:29).	» Escalate privileges. » Modify data. » Allow execution of malicious/arbitrary code. » Leak information. » Render system unusable (AKA denial of service).
CWE-209: Error Message Information Leak	» Blind SQL Injection (CAPEC ID:7). » Probing an Application Through Targeting its Error Reporting (CAPEC ID:54).	» Modify data. » Allow execution of malicious/arbitrary code. » Leak information.
CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer	» Overflow (various types, CAPEC IDs: 8, 9, 14, 24, 44, 45, 46, 47,100). » MIME Conversion (CAPEC ID:42).	» Gain control of the system. » Crash the system (denial of service). » Allow execution of malicious/arbitrary code. » Access or modification of sensitive data. » Leak information. » Escalate privileges.
CWE-642: External Control of Critical State Data	» [Input needed] » Manipulate User State (CAPEC ID:74).	» Gain control or access to state data.
CWE-73: External Control of File Name or Path	» Subverting Environment Variable Values (CAPEC ID:13). » Using Slashes and Encoding (various types, CAPEC IDs: 64, 78, 79). » URL Encoding (CAPEC ID:72). » Manipulating Input to File System Calls (CAPEC ID:76). » Using UTF-8 Encoding to Bypass Validation Logic (CAPEC ID:80).	» Allow execution of malicious/arbitrary code. » Leak information. » Escalate privileges. » Render system unusable (AKA denial of service). » Modify data.
CWE-426: Untrusted Search Path	» [Input needed] » Manipulating Input to File System Calls (CAPEC ID:76).	» [Input needed] » Allow execution of malicious/arbitrary code. » Escalate privileges.
CWE-94: Failure to Control Generation of Code (aka 'Code Injection')	» Leverage Executable Code in Nonexecutable Files (CAPEC ID:35). » Manipulating User-Controlled Variables (CAPEC ID:77).	» Allow execution of malicious/arbitrary code. » Modify data. » Escalate privileges.
CWE-494: Download of Code Without Integrity Check	» [Input needed	» [Input needed]

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

CWE	Related Attack Pattern	Mission/Business Risks
	» Accessing, Modifying or Executing Executable Files (CAPEC ID: 17)]	» Allow execution of malicious/arbitrary code. » Modify data.
CWE-404: Improper Resource Shutdown or Release	» [Input needed] » Reusing Session IDs (aka Session Replay) (CAPEC ID:60) » Manipulating Input to File System Calls (CAPEC ID:76) » Manipulating User-Controlled Variables (CAPEC ID:77)	» [Input needed] » Render system unusable (AKA denial of service).
CWE-665: Improper Initialization	» [Input needed] » Poison Web Service Registry (CAPEC ID:51) » Session Credential Falsification through Prediction (CAPEC ID:59)	» [Input needed] » System Crash. » Render system unusable (AKA denial of service).
CWE-682: Incorrect Calculation	» [Input needed] » Subverting Environment Variable Values (CAPEC ID:13)	» [Input needed] » Render system unusable (AKA denial of service). » Resource consumption. » Execution of arbitrary code.
CWE-327: Use of a Broken or Risky Cryptographic Algorithm	» Cryptanalysis (CAPEC ID:97).	» Escalate privileges. » Leak information. » Modify data.
CWE-285: Improper Access Control (Authorization)	» Accessing Functionality Not Properly Constrained by ACLs (CAPEC ID:1). » Subverting Environment Variable Values (CAPEC ID:13). » Accessing, Modifying or Executing Executable Files (CAPEC ID:17). » Forceful Browsing (CAPEC ID:87). » Manipulating Opaque Client-based Data Tokens (CAPEC ID:39). » Buffer Overflow via Symbolic Links (CAPEC ID:45). » Poison Web Service Registry (CAPEC ID:51). » Session Credential Falsification through Prediction (CAPEC ID:59). » Reusing Session IDs (aka Session Replay) (CAPEC ID:60). » Manipulating Input to File System Calls (CAPEC ID:76). » Manipulating User-Controlled Variables (CAPEC ID:77).	» Escalate privileges. » Render system unusable (AKA denial of service). » Allow execution of malicious/arbitrary code. » Leak information. » Modify data.
CWE-259: Hard-Coded Password	» [Input needed] » Manipulating Input to File System Calls (CAPEC ID:76) » Target Programs with Elevated Privileges (CAPEC	» [Input needed] » Unathorized access. » Escalate privileges. » Gain control of the system.

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

CWE	Related Attack Pattern	Mission/Business Risks
	ID:69)	
CWE-732: Insecure Permission Assignment for Critical Resource	» Reusing Session IDs (aka Session Replay, CAPEC ID:60). » Session Fixation (CAPEC ID:61). » Cross Site Request Forgery (aka Session Riding, CAPEC ID:62).	» Escalate privileges. » Leak information. » Modify data.
CWE-330: Use of Insufficiently Random Values	» Session Credential Falsification through Prediction (CAPEC ID:59).	» Escalate privileges.
CWE-250: Execution with Unnecessary Privileges	» Target Programs with Elevated Privileges (CAPEC ID:69).	» Escalate privileges. » Render system unusable (AKA denial of service). » Allow execution of malicious/arbitrary code.
CWE-602: Client-Side Enforcement of Server-Side Security	» [Input needed] » Manipulating Opaque Client-based Data Tokens (CAPEC ID:39)	» [Input needed] » Allow execution of malicious/arbitrary code » Escalate privileges » Leak information

Key Practices

The key practices documented in "2009 CWE/SANS Top 25 Most Dangerous Programming Errors" focus on preventing and mitigating those dangerous programming errors. Some of the Key Practices specified in the pocket guide are derived from CERT Secure Coding Wiki. Additional information on preventing the various weaknesses is found 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 encouraged to use the CAPEC scripts to 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), and
3. Installation, Operation and System Configuration (Table 5).

Table 3 – Requirements, Architecture , and Design Phases

Prevention and Mitigation Practices	CWE
Use an input validation framework such as Struts or the OWASP ESAPI Validation API. When using Struts, be mindful of weaknesses covered by the CWE-101 category.	**CWE-20: Improper Input Validation**
Understand all the potential areas where untrusted inputs can enter the software: parameters or arguments, cookies, anything read from the network, environment variables, request headers as well as content, URL components, e-mail, files, databases, and any external systems that provide data to the application. Perform input validation at well-defined interfaces.	
Assume all input is malformed or malicious. Use an "accept known good" input validation strategy (that is, use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks. Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if colors such as "red" or "blue" are expected.	

Table 3 – Requirements, Architecture , and Design Phases	
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 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. Even though client-side checks provide minimal benefits with respect to server-side security, they are still useful. First, they can support intrusion detection. If the server receives input that should have been rejected by the client, then it may be an indication of an attack. Second, client-side error-checking can provide helpful feedback to the user about the expectations for valid input. Third, there may be a reduction in server-side processing time for accidental input errors, although this is typically a small savings.	
Do not rely exclusively on blacklist validation to detect malicious input or to encode output (CWE-184). There are too many ways to encode the same character, so some variants are likely to be missed.	
Implement the CERT C guidelines: FIO30-C Exclude User Input from Format Strings.	
Implement the CERT C++ guidelines: FIO30-CPP Exclude User Input from Format Strings.	
Implement the CERT Sun Microsystems Java guidelines: FIO06-J Validate User Input and FIO35-J Exclude User from Format Strings.	
Use languages, libraries, or frameworks that make it easier to generate properly encoded output. Examples include the ESAPI Encoding control. Alternately, use built-in functions, but consider using wrappers in case those functions are discovered to have any vulnerabilities.	**CWE-116: Improper Encoding or Escaping of Output**
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. For example, stored procedures can enforce database query structure and reduce the likelihood of SQL injection.	
Understand the context in which the data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.	
In some cases, input validation may be an important strategy when output encoding is not a complete solution. For example, the same output could be processed by multiple consumers that use different encodings or representations. In other cases, it may be required to allow user-supplied input to contain control information, such as limited HTML tags that support formatting in a wiki or bulletin board. When this type of requirement must be met, use an extremely strict whitelist to limit which control sequences can be used. Verify that the resulting syntactic structure is what is expected. Use the normal encoding methods for the remainder of the input.	
Use input validation as a defense-in-depth measure to reduce the likelihood of output encoding errors (see CWE-20).	
Fully specify which encodings are required by components that will be communicating with each other.	
Implement the CERT C guidelines: MSC09-C. Character Encoding – Use Subset of ASCII for Safety, and MSC10-C. Character Encoding – UTF8 Related Issues.	
Implement the CERT C++ guidelines: MSC09-CPP Character Encoding – Use Subset of ASCII for Safety, and MSC10-CPP. Character Encoding – UTF8 Related Issues.	
Implement the CERT Sun Microsystems Java guidelines: FIO05-J. Document character encoding while performing file IO, FIO32-J. Do not serialize sensitive data, INT00-J. Provide methods to read and write little-endian data, and INT01-J. Provide mechanisms to handle unsigned data when required.	
Use languages, libraries, or frameworks that make it easier to generate properly encoded output. 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: Failure to Preserve SQL Query Structure (aka '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.	

Table 3 – Requirements, Architecture , and Design Phases

Prevention and Mitigation Practices	CWE
Follow the principle of least privilege when creating user accounts to a SQL database. The database users should only have the minimum privileges necessary to use their account. If the requirements of the system indicate that a user can read and modify their own data, then limit their privileges so they cannot read/write others' data. Use the strictest permissions possible on all database objects, such as execute-only for stored procedures.	
For any security checks that are performed on the client-side, ensure that these checks are duplicated on the server-side 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.	
Implement the CERT Sun Microsystems Java guideline: FIO06-J. Validate user input.	
Use languages, libraries, or frameworks that make it easier to generate properly encoded output. Examples include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.	**CWE-79: Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')**
For any security checks that are performed on the client-side, ensure that these checks are duplicated on the server-side 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 at all possible, use library calls rather than external processes to recreate the desired functionality.	**CWE-78: Failure to Preserve OS Command Structure (aka 'OS Command Injection')**
Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which commands can be executed by the software. Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails.	
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 command locally in the session's state instead of sending it out to the client in a hidden form field.	
Use languages, libraries, or frameworks that make it easier to generate properly encoded output. Examples include the ESAPI Encoding control.	
Encrypt the data with a reliable encryption scheme before transmitting.	**CWE-319: Cleartext Transmission of Sensitive Information**
Use anti-Cross Site Request Forgery (CSRF) packages such as the OWASP CSRFGuard.	**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). Note that this can be bypassed using XSS (CWE-79).	
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. Note that this can be bypassed using XSS (CWE-79).	
Use the "double-submitted cookie" method as described by Felten and Zeller: http://freedom-to-tinker.com/sites/default/files/csrf.pdf . Note that this can probably be bypassed using XSS (CWE-79).	
Use the ESAPI Session Management control. This control includes a component for CSRF.	
Do not use the GET method for any request that triggers a state change.	
Use synchronization primitives in languages that support it. Only wrap these around critical code to minimize the impact on performance.	**CWE-362: Race Condition**
Use thread-safe capabilities such as the data access abstraction in Spring.	
Minimize the usage of shared resources to remove as much complexity as possible from the control flow and to reduce the likelihood of unexpected conditions occurring. Additionally, this will minimize the amount of synchronization necessary and may even help to reduce the likelihood of a denial of service where an attacker may be able to repeatedly trigger a critical section (CWE-400).	

Table 3 – Requirements, Architecture , and Design Phases	
Prevention and Mitigation Practices	**CWE**
Use a language with features that can automatically mitigate or eliminate buffer overflows. 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 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.	**CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer**
Use languages, libraries, or frameworks that make it easier to manage buffers without exceeding their boundaries. Examples include the Safe C String Library (SafeStr) by Messier and Viega, and the Strsafe.h library from Microsoft. These libraries provide safer versions of overflow-prone string-handling functions. This is not a complete solution, since many buffer overflows are not related to strings.	
Implement the CERT C guidelines: ARR30-C. Guarantee that array indices are within the valid range, ARR32-C. Ensure size arguments for variable length arrays are in a valid range, ARR33-C. Guarantee that copies are made into storage of sufficient size, and ARR35-C. Do not allow loops to iterate beyond the end of an array.	
Implement CERT C++ guidelines: ARR30-CPP. Guarantee that array and vector indices are within the valid range, ARR33-CPP. Guarantee that copies are made into storage of sufficient size, and ARR35-CPP. Do not allow loops to iterate beyond the end of an array or container.	
Understand all the potential locations that are accessible to attackers. For example, some programmers assume that cookies and hidden form fields cannot be modified by an attacker, or they may not consider that environment variables can be modified before a privileged program is invoked.	**CWE-642: External Control of Critical State Data**
Do not keep state information on the client without using encryption and integrity checking, or otherwise having a mechanism on the server-side to catch state tampering. Use a message authentication code (MAC) algorithm, such as Hash Message Authentication Code (HMAC). Apply this against the state data that must be exposed, which can guarantee the integrity of the data - that is, that the data has not been modified. Use an algorithm with a strong hash function (CWE-328).	
Implement the CERT Sun Microsystems Java guidelines: FIO00-J. Validate deserialized objects and OBJ36-J. Provide mutable classes with a clone method.	
Store state information 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.	
With a stateless protocol such as HTTP, use a framework that maintains the state automatically. Examples include ASP.NET View State and the OWASP ESAPI 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 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 filenames is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames, 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.	**CWE-73: External Control of File Name or Path**
Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict all access to files within a particular directory. Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails.	
For any security checks that are performed on the client-side, ensure that these checks are duplicated on the server-side 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.	

Table 3 – Requirements, Architecture , and Design Phases

Prevention and Mitigation Practices	CWE
Implement the CERT C guideline: ENV02-C. Beware of multiple environment variables with the same effective name.	
Implement the CERT C++ guideline: ENV02-CPP. Beware of multiple environment variables with the same effective name.	
Hard-code the search path to a set of known-safe values, or allow them to be specified by the administrator in a configuration file. Do not allow these settings to be modified by an external party. Be careful to avoid related weaknesses such as CWE-427 and CWE-428.	**CWE-426: Untrusted Search Path**
Implement the CERT C guideline: ENV02-C. Beware of multiple environment variables with the same effective name.	
Implement the CERT C++ guideline: ENV02-CPP. Beware of multiple environment variables with the same effective name.	
Refactor the program so that there is no need to dynamically generate code.	**CWE-94: Failure to Control Generation of Code (aka 'Code Injection')**
Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which code can be executed by the software. Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection. This may not be a feasible solution, and it only limits the impact to the operating system; the rest of the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails.	
Encrypt the code with a reliable encryption scheme before transmitting. This will only be a partial solution, since it will not detect DNS spoofing and it will not prevent the code from being modified on the hosting site.	**CWE-494: Download of Code Without Integrity Check**
Use integrity checking on the transmitted code. If the code must be downloaded, such as for automatic updates, use cryptographic signatures for the code and modify the download clients to verify the signatures. Ensure that the implementation does not contain CWE-295, CWE-320, CWE-347, and related weaknesses. Use code signing technologies such as Authenticode.	
Implement the CERT Sun Microsystems Java guidelines: SEC05-J. Digitally sign all artifacts before deployment and SEC08-J. Consider signing and sealing objects before transit.	
Use a language with features that can automatically mitigate or eliminate resource-shutdown weaknesses. For example, languages such as Java, Ruby, and Lisp perform automatic garbage collection that releases memory for objects that have been deallocated.	**CWE-404: Improper Resource Shutdown or Release**
Use a language with features that can automatically mitigate or eliminate weaknesses related to initialization. For example, in Java, if the programmer does not explicitly initialize a variable, then the code could produce a compile-time error (if the variable is local) or automatically initialize the variable to the default value for the variable's type. In Perl, if explicit initialization is not performed, then a default value of undef is assigned, which is interpreted as 0, false, or an equivalent value depending on the context in which the variable is accessed.	**CWE-665: Improper Initialization**
Identify all variables and data stores that receive information from external sources, and apply input validation to make sure that they are only initialized to expected values.	
Divide the 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) to enforce the roles at the appropriate boundaries. Note that this approach may not protect against horizontal attacks. That is, it will not protect a user from attacking others with the same role.	**CWE-285: Improper Access Control (Authorization)**
Perform access control checks related to the business logic. These may be different than the access control checks that are applied to the resources that support the business logic.	
Use authorization frameworks such as the JAAS Authorization Framework and the OWASP ESAPI Access Control feature.	

Table 3 – Requirements, Architecture , and Design Phases

Prevention and Mitigation Practices	CWE
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.	
Do not develop custom cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If the algorithm can be compromised once attackers find out how it works, then it is especially weak.	**CWE-327: Use of a Broken or Risky Cryptographic Algorithm**
Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. For example, US government systems require FIPS 140-2 certification. As with all cryptographic mechanisms, the source code should be available for analysis. Periodically ensure that the cryptography being used is not obsolete. 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 which were once regarded as strong.	
Design the software so that one cryptographic algorithm can be replaced 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 languages, libraries, or frameworks that make it easier to use strong cryptography. Industry-standard implementations will save development time and may be more likely to avoid errors that can occur during implementation of cryptographic algorithms. Consider the ESAPI Encryption feature.	
For outbound authentication: store passwords 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 encryption cannot be used to protect the file, then make sure that the permissions are as restrictive as possible.	**CWE-259: Hard-Coded Password**
For inbound authentication: Rather than hard-code a default username and password for first time logins, utilize a “first login” mode that requires the user to enter a unique strong password.	
Perform access control checks and limit which entities can access the feature that requires the hard-coded password. For example, a feature might only be enabled through the system console instead of through a network connection.	
Implement the CERT Sun Microsystems Java guideline: FIO36-J. Never hardcode sensitive information.	
For inbound authentication: apply strong one-way hashes to the 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 has been saved. Use randomly assigned salts for each separate generated hash. 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 which 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 used 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.	
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), 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: Insecure Permission Assignment for Critical Resource**
Divide the 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 more fine-grained control over those resources.	
Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations with adequate length seeds. In general, if a pseudo-random number generator is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts. Pseudo-random number generators can produce predictable numbers if the generator is known and the seed can be guessed. A 256-bit seed is a good starting point for producing a “random enough” number.	**CWE-330: Use of Insufficiently Random Values**

Table 3 – Requirements, Architecture , and Design Phases

Prevention and Mitigation Practices	CWE
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. Raise the privileges as late as possible, and drop them as soon as possible. Avoid weaknesses such as CWE-288 and CWE-420 by protecting all possible communication channels that could interact with privileged code, such as a secondary socket that is only intended for access by administrators.	**CWE-250: Execution with Unnecessary Privileges**
For any security checks that are performed on the client-side, ensure that these checks are duplicated on the server-side. 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. Even though client-side checks provide minimal benefits with respect to server-side security, they are still useful. First, they can support intrusion detection. If the server receives input that should have been rejected by the client, then it may be an indication of an attack. Second, client-side error-checking can provide helpful feedback to the user about the expectations for valid input. Third, there may be a reduction in server-side processing time for accidental input errors, although this is typically a small savings.	**CWE-602: Client-Side Enforcement of Server-Side Security**
If some degree of trust is required between the two entities, then use integrity checking and strong authentication to ensure that the inputs are coming from a trusted source. Design the product so that this trust is managed in a centralized fashion, especially if there are complex or numerous communication channels. This will reduce the risks that the implementer will mistakenly omit a check in a single code path.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases

Prevention and Mitigation Practices	CWE
When the application combines data from multiple sources, perform the validation after the sources have been combined. The individual data elements may pass the validation step but violate the intended restrictions after they have been combined.	**CWE-20: Improper Input Validation**
Be especially careful to validate input when invoking code that crosses language boundaries, such as from an interpreted language to native code. This could create an unexpected interaction between the language boundaries. Do not violate any of the expectations of the language with which the code is interfacing. For example, even though Java may not be susceptible to buffer overflows, providing a large argument in a call to native code might trigger an overflow.	
Directly convert the input into the expected data type, such as using a conversion function that translates a string into a number. After converting to the expected data type, ensure that the input's values fall within the expected range of allowable values and that multi-field consistencies are maintained.	
Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180, CWE-181). Make sure that the application does not inadvertently decode the same input twice (CWE-174). Such errors could be used to bypass whitelist schemes by introducing dangerous inputs after they have been checked. Use libraries such as the OWASP ESAPI Canonicalization control. Consider performing repeated canonicalization until the input does not change any more. This will avoid double-decoding and similar scenarios, but it might inadvertently modify inputs that are allowed to contain properly-encoded dangerous content.	
Implement the CERT Sun Microsystems Java guidelines: FIO06-J. Validate user input, FIO00-J. Validate deserialized objects, and FIO35-J. Exclude user input from format strings.	
Implement the CERT C guidelines: INT06-C. Use strtol() or a related function to convert a string token to an integer, ERR07-C. Prefer functions that support error checking over equivalent functions that don't, APP00-C. Functions should validate their parameters, and MEM10-C. Define and use a pointer validation function.	
Implement the CERT C++ guidelines: MSC08-CPP. Library functions should validate their parameters, INT06-CPP. Use strtol() or a related function to convert a string token to an integer, and MEM10-CPP. Define and use a pointer validation function.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
When exchanging data between components, ensure that both components are using the same character encoding. Ensure that the proper encoding is applied at each interface. Explicitly set the encoding being used whenever the protocol allows it.	**CWE-116: Improper Encoding or Escaping of Output**

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
When using dynamically-generated query strings in spite of the risk, use proper encoding and escaping of inputs. Instead of building a custom 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: **Failure to Preserve SQL Query Structure (aka 'SQL Injection')**
Assume all input is malformed or malicious. Use an "accept known good" input validation strategy (that is, use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks. Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if colors such as "red" or "blue" are expected.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
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 the application is 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. 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 it might not be possible to control those processes.	
Understand the context in which the data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies. For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters. This encoding will vary depending on whether the output is part of the HTML body, element attributes, URIs, JavaScript sections, Cascading Style Sheets, etc. Note that HTML Entity Encoding is only appropriate for the HTML body.	CWE-79: **Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')**
Use and specify a strong 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 expose the application to subtle XSS attacks related to that encoding. See CWE-116 for more mitigations related to encoding/escaping.	
Assume all input is malformed or malicious. Use an "accept known good" input validation strategy (that is, use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks. Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if colors such as "red" or "blue" are expected.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
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 the application is 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 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 a mistake is made in the validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to offer protection 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 the attack surface, allow the application to detect some attacks, and provide other security benefits that proper encoding does not address.	
Ensure that input validation is performed at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Use the XSS Cheat Sheet (see references) to launch a wide variety of attacks against the web application. The Cheat Sheet contains many subtle XSS variations that are specifically targeted against weak XSS defenses.	
With Struts, 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.	
Properly quote arguments and escape any special characters within those arguments. If some special characters are still needed, wrap the arguments in quotes, and escape all other characters that do not pass a strict whitelist. Be careful of argument injection (CWE-88).	**CWE-78: Failure to Preserve OS Command Structure (aka 'OS Command Injection)**
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.	
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 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Assume all input is malformed or malicious. Use an “accept known good” input validation strategy (that is, use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks. Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, “boat” may be syntactically valid because it only contains alphanumeric characters, but it is not valid if colors such as “red” or “blue” are expected.	
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 the application is required to support free-form text fields that could contain arbitrary characters. For example, when invoking a mail program, the subject field might 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 to pass messages to other components. Even if a mistake is made in the validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to offer protection 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 attack surface, allow the application to detect some attacks, and provide other security benefits that proper encoding does not address.	
Implement the CERT C guidelines: ENV03-C. Sanitize the environment when invoking external programs, STR02-C. Sanitize data passed to complex subsystems, and ENV04-C. Do not call system() if you do not need a command processor.	
Implement the CERT C++ guidelines: ENV03-CPP. Sanitize the environment when invoking external programs, STR02-CPP. Sanitize data passed to complex subsystems, and ENV04-CPP. Do not call system() if you do not need a command processor.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software’s operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
When using web applications with SSL, use SSL for the entire session from login to logout, not just for the initial login page.	CWE-319: **Cleartext Transmission of Sensitive Information**
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Use monitoring tools that examine the software’s process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process, trigger the feature that sends the data, and look for the presence or absence of common cryptographic functions in the call tree. Monitor the network and determine if the data packets contain readable commands. Tools exist for detecting if certain encodings are in use. If the traffic contains high entropy, this might indicate the usage of encryption.	
Ensure that the 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. Note that this can be bypassed using XSS (CWE-79). An attacker could use XSS to generate a spoofed Referer, or to generate a malicious request from a page whose Referer would be allowed.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules. Use OWASP CSRFTester to identify potential issues.	
When using multi-threading, only use thread-safe functions on shared variables.	**CWE-362: Race Condition**
Use atomic operations on shared variables. Be wary of innocent-looking constructs like "x++". This is actually non-atomic, since it involves a read followed by a write.	
Use a mutex if available, but be sure to avoid related weaknesses such as CWE-412.	
Avoid double-checked locking (CWE-609) and other implementation errors that arise when trying to avoid the overhead of synchronization.	
Disable interrupts or signals over critical parts of the code, but also make sure that the code does not go into a large or infinite loop.	
Use the volatile type modifier for critical variables to avoid unexpected compiler optimization or reordering. This does not necessarily solve the synchronization problem, but it can help.	
Implement the CERT C guideline: FIO31-C. Do not simultaneously open the same file multiple times.	
Implement the CERT C++ guideline: FIO31-CPP. Do not simultaneously open the same file multiple times.	
Implement the CERT Sun Microsystems Java guidelines: FIO31-J. Create a copy of mutable inputs, CON30-J. Synchronize access to shared mutable variables, CON01-J. Avoid using ThreadGroup APIs, OBJ32-J. Do not allow partially initialized objects to be accessed, MSC00-J. Eliminate class initialization cycles, CON32-J. Prefer notifyAll() to notify(), and FIO37-J. Create and delete temporary files safely.	
Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results. Insert breakpoints or delays in between relevant code statements to artificially expand the race window so that it will be easier to detect.	
Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If these actions trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.	
Ensure that error messages only contain minimal information that are useful to their 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 help an attacker craft another attack that now will pass through the validation filters. 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.	**CWE-209: Error Message Information Leak**
Handle exceptions internally and do not display errors containing potentially sensitive information to a user.	
Implement the CERT Sun Microsystems Java guidelines: EXC01-J. Do not allow exceptions to transmit sensitive information, EXC05-J. Use a class dedicated to reporting exceptions, and FIO35-J. Exclude user input from format strings.	
Debugging information should not make its way into a production release.	
Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If these actions trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases

Prevention and Mitigation Practices	CWE
Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	**CWE-602: Client-Side Enforcement of Server-Side Security**
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Run or compile the 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. This is not necessarily a complete solution, since these mechanisms can only detect certain types of overflows. In addition, a buffer overflow attack can still cause a denial of service, since the typical response is to exit the application.	**CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer**
Programmers should adhere to the following rules when allocating and managing their application's memory: » Double check that the buffer is as large as originally specified. » 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. » If calling these functions in a loop, check buffer boundaries and make sure there is no 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.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Implement the CERT C guidelines: MEM09-C. Do not assume memory allocation routines initialize memory, FIO37-C. Do not assume character data has been read, STR33-C. Size wide character strings correctly, ARR34-C. Ensure that array types in expressions are compatible, ENV01-C. Do not make assumptions about the size of an environment variable, ARR33-C. Guarantee that copies are made into storage of sufficient size, ARR35-C. Do not allow loops to iterate beyond the end of an array, ARR00-C. Understand how arrays work, STR32-C. Null-terminate byte strings as required, and STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator.	
Implement the CERT C++ guidelines: ENV01-CPP. Do not make assumptions about the size of an environment variable, FIO37-CPP. Do not assume character data has been read, STR32-CPP. Null-terminate character arrays as required, MEM09-CPP. Do not assume memory allocation routines initialize memory, ARR33-CPP. Guarantee that copies are made into storage of sufficient size, ARR00-CPP. Understand how arrays and vectors work, STR31-CPP. Guarantee that storage for character arrays has sufficient space for character data and the null terminator, ARR35-CPP. Do not allow loops to iterate beyond the end of an array or container.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	**CWE-642: External Control of Critical State Data**
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Assume all input is malformed or malicious. Use an "accept known good" input validation strategy (that is, use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks. 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-73: External Control of File Name or Path**
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).	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Implement the CERT C guidelines: FIO01-C. Be careful using functions that use file names for identification and FIO02-C. Canonicalize path names originating from untrusted sources.	
Implement the CERT C++ guidelines: FIO01-CPP. Be careful using functions that use file names for identification and FIO02-CPP. Canonicalize path names originating from untrusted sources.	
Implement the CERT Sun Microsystems Java guideline: FIO01-J. Canonicalize path names originating from untrusted sources.	
Sanitize the environment before invoking other programs. This includes the PATH environment variable, LD_LIBRARY_PATH and other settings that identify the location of code libraries, and any application-specific search paths.	**CWE-426: Untrusted Search Path**
Check the search path before use and remove any elements that are likely to be unsafe, such as the current working directory or a temporary files directory.	
Use other functions that require explicit paths. Making use of any of the other readily available functions that require explicit paths is a safe way to avoid this problem. For example, system() in C does not require a full path since the shell can take care of it, while execl() and execv() require a full path.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and look for library functions and system calls that suggest when a search path is being used. One pattern is when the program performs multiple accesses of the same file but in different directories, with repeated failures until the proper filename is found. Library calls such as getenv() or their equivalent can be checked to see if any path-related variables are being accessed.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Assume all input is malformed or malicious. Use an "accept known good" input validation strategy (that is, use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks. To reduce the likelihood of code injection, use stringent whitelists that limit which constructs are allowed. If the code is dynamically constructed to invoke a function, then verifying that the input is alphanumeric might be insufficient. An attacker might still be able to reference a dangerous function that is not intended, such as system(), exec(), or exit().	**CWE-94: Failure to Control Generation of Code (aka 'Code Injection')**
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Perform proper forward and reverse DNS lookups to detect DNS spoofing. This is only a partial solution since it will not prevent the code from being modified on the hosting site or in transit.	**CWE-494: Download of Code Without Integrity Check**
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and also sniff the network connection. Trigger features related to product updates or plugin installation, which is likely to force a code download. Monitor when files are downloaded and separately executed, or if they are otherwise read back into the process. Look for evidence of cryptographic library calls that use integrity checking.	
It is good practice to be responsible for freeing all resources the software allocates and to be consistent with how and where the memory is freed in a function. If the software allocates memory that is intended to be freed upon completion of the function, free the memory at all exit points for that function, including error conditions.	**CWE-404: Improper Resource Shutdown or Release**
Memory should be allocated/freed using matching functions such as malloc/free, new/delete, and new[]/delete[].	
When releasing a complex object or structure, ensure that all of its member components are disposed of properly, not just the object itself.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If these actions trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.	
Implement the CERT C guideline: FIO42-C. Ensure files are properly closed when they are no longer needed.	
Implement the CERT C++ guideline: FIO42-CPP. Ensure files are properly closed when they are no longer needed.	
Explicitly initialize all variables and other data stores, either during declaration or just before the first usage.	**CWE-665: Improper Initialization**
Pay close attention to complex conditionals that affect initialization, since some conditions might not perform the initialization.	
Avoid race conditions (CWE-362) during initialization routines.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Run or compile the software with settings that generate warnings about uninitialized variables or data.	
Implement the CERT C guidelines: MEM09-C. Do not assume memory allocation routines initialize memory and ARR02-C. Explicitly specify array bounds, even if implicitly defined by an initializer.	
Implement the CERT C++ guidelines: ARR02-CPP. Explicitly specify array bounds, even if implicitly defined by an initializer, MEM09-CPP. Do not assume memory allocation routines initialize memory.	
Implement the CERT Sun Microsystems Java guidelines: FIO34-J. Ensure all resources are properly closed when they are no longer needed, CON35-J. Do not try to force thread shutdown, EXC04-J. Prevent against inadvertent calls to System.exit() or forced shutdown, and EXC30-J. Do not exit abruptly from a finally block.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If these actions trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.	
Understand the programming language's underlying representation and how it interacts with numeric calculation. Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how the language handles numbers that are too large or too small for its underlying representation.	**CWE-682: Incorrect Calculation**
Perform input validation on any numeric inputs by ensuring that they are within the expected range.	
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.	
Use languages, 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++).	
Implement the CERT C guidelines: FLP33-C. Convert integers to floating point for floating point operations , INT10-C. Do not assume a positive remainder when using the % operator, INT07-C. Use only explicitly signed or unsigned char type for numeric values, INT13-C. Use bitwise operators only on unsigned operands, and FLP32-C. Prevent or detect domain and range errors in math functions.	
Implement the CERT C++ guidelines: INT10-CPP. Do not assume a positive remainder when using the % operator, FLP33-CPP. Convert integers to floating point for floating point operations, INT07-CPP. Use only explicitly signed or unsigned char type for numeric values, INT13-CPP. Use bitwise operators only on unsigned operands, and FLP32-CPP. Prevent or detect domain and range errors in math functions.	
Implement the CERT Sun Microsystems Java guidelines: INT34-J. Perform explicit range checking to ensure integer operations do not overflow, INT30-J. Be careful while casting integers to narrower types. FLP00-J. Consider avoiding floating point numbers when precise computation is needed, INT33-J. Be careful while casting numeric types to wider floating-point types, and FLP01-J. Take care in rearranging floating point expressions.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.	
When using industry-approved techniques, use them correctly. Don't cut corners by skipping resource-intensive steps (CWE-325). These steps are often essential for preventing common attacks.	**CWE-327: Use of a Broken or Risky Cryptographic**

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases

Prevention and Mitigation Practices	CWE
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	**Algorithm**
Implement the CERT C guidelines: MSC30-C. Do not use the rand() function for generating pseudorandom numbers, MSC32-C. Ensure your random number generator is properly seeded.	
Implement the CERT C++ guidelines: MSC30-CPP. Do not use the rand() function for generating pseudorandom numbers, MSC32-CPP. Ensure your random number generator is properly seeded.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	**CWE-285: Improper Access Control (Authorization)**
During program startup, explicitly set the default permissions or umask to the most restrictive setting possible. Also set the appropriate permissions during program installation. This will prevent the software from inheriting insecure permissions from any user who installs or runs the program.	**CWE-732: Insecure Permission Assignment for Critical Resource**
Do not suggest insecure configuration changes in the documentation, especially if those configurations can extend to resources that are outside the scope of the software.	
Implement the CERT Sun Microsystems Java guidelines: SEC01-J. Be careful using doPrivileged.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and watch for library functions or system calls on OS resources such as files, directories, and shared memory. Examine the arguments to these calls to infer which permissions are being used. Note that this technique is only useful for permissions issues related to system resources. It is not likely to detect application-level business rules that are related to permissions, such as if a user of a blog system marks a post as "private," but the blog system inadvertently marks it as "public."	
Ensure that the software runs properly under the Federal Desktop Core Configuration (FDCC) or an equivalent hardening configuration guide, which many organizations use to limit the attack surface and potential risk of deployed software.	
Consider a PRNG that re-seeds itself as needed from high quality pseudo-random output sources, such as hardware devices.	**CWE-330: Use of Insufficiently Random Values**
Implement the CERT C guidelines: MSC30-C. Do not use the rand() function for generating pseudorandom numbers, MSC32-C. Ensure your random number generator is properly seeded.	
Implement the CERT C++ guidelines: MSC30-CPP. Do not use the rand() function for generating pseudorandom numbers, MSC32-CPP. Ensure your random number generator is properly seeded.	
Implement the CERT Sun Microsystems Java guidelines: MSC30-J. Generate truly random numbers.	
Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.	
Perform FIPS 140-2 tests on data to catch obvious entropy problems.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	

Table 4 – Build, Compilation, Implementation, Testing, and Documentation Phases	
Prevention and Mitigation Practices	**CWE**
Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and look for library functions that indicate when randomness is being used. Run the process multiple times to see if the seed changes. Look for accesses of devices or equivalent resources that are commonly used for strong (or weak) randomness, such as /dev/urandom on Linux. Look for library or system calls that access predictable information such as process IDs and system time.	
Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and perform a login. Using disassembled code, look at the associated instructions and see if any of them appear to be comparing the input to a fixed string or value.	**CWE-259: Hard-Coded Password**
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Perform extensive input validation for any privileged code that must be exposed to the user and reject anything that does not fit strict requirements.	**CWE-250: Execution with Unnecessary Privileges**
Drop privileges as soon as possible (CWE-271) and check that privileges have been dropped successfully (CWE-273).	
If circumstances require running 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. 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.	
Implement the CERT C guidelines: POS02-C. Follow the principle of least privilege, POS36-C. Observe correct revocation order while relinquishing privileges, and POS37-C. Ensure that privilege relinquishment is successful.	
Implement the CERT Sun Microsystems Java guidelines: SEC31-J. Never grant AllPermission to untrusted code, SEC32-J. Do not grant ReflectPermission with action suppressAccessChecks.	
Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.	
Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was developed by a third party, or when verifying that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and perform a login. Look for library functions and system calls that indicate when privileges are being raised or dropped. Look for accesses of resources that are restricted to normal users. Note that this technique is only useful for privilege issues related to system resources. It is not likely to detect application-level business rules that are related to privileges, such as if a blog system allows a user to delete a blog entry without first checking that the user has administrator privileges.	
Ensure that the software runs properly under the Federal Desktop Core Configuration (FDCC) or an equivalent hardening configuration guide, which many organizations use to limit the attack surface and potential risk of deployed software.	

Table 5 – Installation, Operation, and System Configuration Phases	
Prevention and Mitigation Practices	**CWE**

Table 5 – Installation, Operation, and System Configuration Phases	
Prevention and Mitigation Practices	**CWE**
Use an application firewall that can detect attacks against this weakness. This might not catch all attacks, and it might require some effort for customization. However, 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: Failure to Preserve SQL Query Structure (aka 'SQL Injection')**
Use an application firewall that can detect attacks against this weakness. This might not catch all attacks, and it might require some effort for customization. However, 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: Failure to Preserve Web Page Structure (aka 'Cross-site Scripting')**
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 forces the application to perform validation steps that remove the taint. Be careful to correctly validate the input so that no dangerous input is accidentally marked as untainted (see CWE-183 and CWE-184).	**CWE-78: Failure to Preserve OS Command Structure (aka '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.	
Assign permissions to the software system that prevent the user from accessing/opening privileged files. Run the application with the lowest privileges possible (CWE-250).	
When using PHP, configure the application so that it does not use register_globals. During implementation, develop the 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.	**CWE-642: External Control of Critical State Data**
Configure servers to use encrypted channels for communication, which may include SSL or other secure protocols.	**CWE-319: Cleartext Transmission of Sensitive Information**
Where available, configure the environment to use less verbose error messages. For example, in PHP, disable the display_errors setting during configuration, or at runtime using the error_reporting() function.	**CWE-209: Error Message Information Leak**
Create default error pages or messages that do not leak any information.	
Use a feature like Address Space Layout Randomization (ASLR). This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution.	**CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer**
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent. This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required.	
Use OS-level permissions and run as a low-privileged user to limit the scope of any successful attack.	**CWE-73: External Control of File Name or Path**
When using PHP, configure the application so that it does not use register_globals. During implementation, develop the 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.	
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. Perform validation steps that remove the taint, and must be careful to correctly validate the input so that no dangerous input is accidentally marked as untainted (see CWE-183 and CWE-184).	**CWE-94: Failure to Control Generation of Code (aka 'Code Injection')**
Use the access control capabilities of the operating system and server environment. Define access control lists (ACLs) accordingly. Use a "default deny" policy when defining these ACLs.	**CWE-285: Improper Access Control (Authorization)**
For all configuration files, executables, and libraries, make sure that they are only readable and writable by the software's administrator.	**CWE-732: Insecure Permission Assignment for Critical Resource**
Do not assume that the system administrator will manually change the configuration to the settings that are recommended in the manual.	

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

Reprints

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. Software Assurance in Acquisition and Contract Language
II. Software Supply Chain Risk Management & Due-Diligence

SwA in Development

I. Integrating Security in 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
VII. Penetration Testing throughout the Life Cycle

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.