Common Weakness Enumeration
A Community-Developed Dictionary of Software Weakness Types

CWE and SANS Institute TOP 25 MOST DANGEROUS SOFTWARE ERRORS

2010 CWE/SANS Top 25 Most Dangerous Software Errors

Document version: 1.08

Date: March 29, 2010

Project Coordinators:

Bob Martin (MITRE)
Mason Brown (SANS)
Alan Paller (SANS)
Dennis Kirby (SANS)

Document Editor:

Steve Christey (MITRE)

Introduction

The 2010 CWE/SANS Top 25 Most Dangerous Software Errors is a list of the most widespread and critical programming errors that can lead to serious software vulnerabilities. They 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 Top 25 list is a tool for education and awareness to help programmers to prevent the kinds of vulnerabilities that plague the software industry, by identifying and avoiding all-too-common mistakes that occur before software is even shipped. Software customers can use the same list to help them to ask for more secure software. Researchers in software security can use the Top 25 to focus on a narrow but important subset of all known security weaknesses. Finally, software managers and CIOs can use the Top 25 list as a measuring stick of progress in their efforts to secure their software.

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/). MITRE maintains the CWE web site, with the support of the US Department of Homeland Security's National Cyber Security Division, presenting detailed descriptions of the top 25 programming errors along with authoritative guidance for mitigating and avoiding them. The CWE site contains data on more than 800 programming errors, design errors, and architecture errors that can lead to exploitable vulnerabilities.

The 2010 Top 25 makes substantial improvements to the 2009 list, but the spirit and goals remain the same. The structure of the list has been modified to distinguish mitigations and general secure programming principles from more concrete weaknesses. This year's Top 25 entries are prioritized using inputs from over 20 different organizations, who evaluated each weakness based on prevalence and importance. The new version introduces focus profiles that allow developers and other users to select the parts of the Top 25 that are most relevant to their concerns. The new list also adds a small set of the most effective "Monster Mitigations," which help developers to reduce or eliminate entire groups of the Top 25 weaknesses, as well as many of the other 800 weaknesses that are documented by CWE. Finally, many high-level weaknesses from the 2009 list have been replaced with lower-level variants that are more actionable.

Table of Contents

Guidance for Using the Top 25

Here is some guidance for different types of users of the Top 25.

User	Activity
Programmers new to security	Read the brief listing, then examine the Monster Mitigations section to see how a small number of changes in your practices can have a big impact on the Top 25. Review the focus profiles, to determine which set of issues you want to concentrate on. If 25 entries are too much to start with, select an individual focus profile, such as Weaknesses by Language or Educational Emphasis. Choose the prioritization scheme that most fits your needs, such as the profile that prioritizes importance over prevalence, which may be useful to software customers. If your software is web-based, see the comparison between the Top 25 and the OWASP Top Ten 2010 RC1. Pick a small number of weaknesses to work with first, and see the Detailed CWE Descriptions for more information on the weakness, which includes code examples and specific mitigations.
Programmers who are experienced in security	Use the general Top 25 as a checklist of reminders, and note the issues that have only recently become more common. Consult the focus profile for Established Secure Developers to see the issues that major vendors are still wrestling with. See the On the Cusp profile for other weaknesses that did not make the final Top 25; this includes weaknesses that are only starting to grow in prevalence or importance. If you are already familiar with a particular weakness, then consult the Detailed CWE Descriptions and see the "Related CWEs" links for variants that you may not have fully considered. Build your own Monster Mitigations section so that you have a clear understanding of which of your own mitigation practices are the most effective - and where your gaps may lie. If you are considering building a custom "Top n" list that fits your needs and practices, consult Appendix C to see how it was done for this year's Top 25. Develop your own nominee list of weaknesses, with your own prevalence and importance factors - and other factors that you may wish - then build a metric and compare the results with your colleagues, which may produce some fruitful discussions.
	Treat the Top 25 as an early step in a larger effort towards achieving software security. Strategic possibilities are covered in efforts such as BSIMM, SAFECode, OpenSAMM, Microsoft SDL, and ASVS.

Software project managers	Peruse the Focus Profiles to find the profile that is best for you. Especially examine the profiles that identify which automated and manual techniques are useful for detecting weaknesses, and whether they are best fixed in the design or implementation phases of the software development lifecycle. Review the profile that provides a breakdown by language. Also review the profile that ranks weaknesses primarily by importance since that may be the most important to software customers. Examine the Monster Mitigations section to determine which approaches may be most suitable to adopt, or establish your own monster mitigations and map out which of the Top 25 are addressed by them. Consider building a custom "Top n" list that fits your needs and practices; consult Appendix C on the construction and scoring of the Top 25. Develop your own nominee list of weaknesses, with your own prevalence and importance factors - or other factors that are more critical to you - then score your results and compare them with your colleagues, which may produce some fruitful discussions.
Software Testers	Read the brief listing and consider how you would integrate knowledge of these weaknesses into your tests. Review the focus profiles to help you decide which set of issues you need to concentrate on. If you are in a friendly competition with the developers, you may find some surprises in the On the Cusp entries, or even the rest of CWE. Pay close attention to the focus profile for Automated vs. Manual Analysis, which will give you hints about which methods will be most effective. For each indvidual CWE entry in the Details section, you can get more information on detection methods from the "technical details" link. Review the CAPEC IDs for ideas on the types of attacks that can be launched against the weakness.
Software customers	Recognize that market pressures often drive vendors to provide software that is rich in features, and security may not be a serious consideration. As a customer, you have the power to influence vendors to provide more secure products by letting them know that security is important to you. Use the Top 25 to help set minimum expectations for due care by software vendors. Consider using the Top 25 as part of contract language during the software acquisition process. The SANS Application Security Procurement Language site offers customer-centric language that is derived from the OWASP Secure Software Contract Annex, which offers a "framework for discussing expectations and negotiating responsibilities" between the customer and the vendor. Other information is available from the DHS Acquisition and Outsourcing Working Group. Consult the Focus Profile in which weaknesses are ranked by Importance to see which weaknesses were scored to be the most important, independent of how prevalent they are. For the software products that you use, pay close attention to publicly reported vulnerabilities in those products. See if they reflect any of the associated weaknesses on the Top 25, and if so, contact your vendor to determine what processes the vendor is undertaking to minimize the risk that these weaknesses will continue to be introduced into the code. Use the Design and Implementation profile to determine which weaknesses are only fixable in the design phase. This often means that if a design-related weakness is discovered in the software you buy, it could take a long time for the vendor to provide a fix. See the On the Cusp summary for other weaknesses that did not make the final Top 25; this will include weaknesses that are only starting to grow in prevalence or importance, so they may become your problem in the future.
Educators	Read the brief listing, then review the focus profiles, to determine which set of issues you may want to cover, especially the Educational Emphasis profile. Some training materials are also available.
	See the What Changed section; while a lot has changed on the surface, this year's

Users of the 2009 Top 25	effort is more well-structured. Consult the focus profiles and select what's most appropriate for you, or create your own.

Brief Listing of the Top 25

This is a brief listing of the Top 25 items, using the general ranking.

NOTE: 16 other weaknesses were considered for inclusion in the Top 25, but their general scores were not high enough. They are listed in the On the Cusp focus profile.

Rank	Score	ID	Name
[1]	346	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[2]	330	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[3]	273	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	261	CWE-352	Cross-Site Request Forgery (CSRF)
[5]	219	CWE-285	Improper Authorization
[6]	202	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[7]	197	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[8]	194	CWE-434	Unrestricted Upload of File with Dangerous Type
[9]	188	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[10]	188	CWE-311	Missing Encryption of Sensitive Data
[11]	176	CWE-798	Use of Hard-coded Credentials
[12]	158	CWE-805	Buffer Access with Incorrect Length Value
[13]	157	CWE-98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')
[14]	156	CWE-129	Improper Validation of Array Index
[15]	155	CWE-754	Improper Check for Unusual or Exceptional Conditions
[16]	154	CWE-209	Information Exposure Through an Error Message
[17]	154	CWE-190	Integer Overflow or Wraparound
[18]	153	CWE-131	Incorrect Calculation of Buffer Size
[19]	147	CWE-306	Missing Authentication for Critical Function
[20]	146	CWE-	Download of Code Without Integrity Check

		494	
[21]	145	CWE-732	Incorrect Permission Assignment for Critical Resource
[22]	145	CWE-770	Allocation of Resources Without Limits or Throttling
[23]	142	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[24]	141	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[25]	138	CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

Cross-site scripting and SQL injection are the 1-2 punch of security weaknesses in 2010. Even when a software package doesn't primarily run on the web, there's a good chance that it has a web-based management interface or HTML-based output formats that allow cross-site scripting. For data-rich software applications, SQL injection is the means to steal the keys to the kingdom. The classic buffer overflow comes in third, while more complex buffer overflow variants are sprinkled in the rest of the Top 25.

Category-Based View of the Top 25

This section sorts the entries into the three high-level categories that were used in the 2009 Top 25:

- Insecure Interaction Between Components
- Risky Resource Management
- Porous Defenses

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.

For each weakness, its ranking in the general list is provided in square brackets.

Rank	**CWE ID**	**Name**
[1]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[2]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[4]	CWE-352	Cross-Site Request Forgery (CSRF)
[8]	CWE-434	Unrestricted Upload of File with Dangerous Type
[9]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[17]	CWE-209	Information Exposure Through an Error Message
[23]	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[25]	CWE-362	Race Condition

Risky Resource Management

The weaknesses in this category are related to ways in which software does not properly manage the creation, usage, transfer, or destruction of important system resources.

Rank	CWE ID	Name
[3]	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[7]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[12]	CWE-805	Buffer Access with Incorrect Length Value
[13]	CWE-754	Improper Check for Unusual or Exceptional Conditions
[14]	CWE-98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')
[15]	CWE-129	Improper Validation of Array Index
[16]	CWE-190	Integer Overflow or Wraparound
[18]	CWE-131	Incorrect Calculation of Buffer Size
[20]	CWE-494	Download of Code Without Integrity Check
[22]	CWE-770	Allocation of Resources Without Limits or Throttling

Porous Defenses

The weaknesses in this category are related to defensive techniques that are often misused, abused, or just plain ignored.

Rank	CWE ID	Name
[5]	CWE-285	Improper Access Control (Authorization)
[6]	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[10]	CWE-311	Missing Encryption of Sensitive Data
[11]	CWE-798	Use of Hard-coded Credentials
[19]	CWE-306	Missing Authentication for Critical Function
[21]	CWE-732	Incorrect Permission Assignment for Critical Resource
[24]	CWE-327	Use of a Broken or Risky Cryptographic Algorithm

Focus Profiles

The prioritization of items in the general Top 25 list is just that - general. The rankings, and even the selection of which items should be included, can vary widely depending on context. Ideally, each organization can decide how to rank weaknesses based on its own criteria, instead of relying on a single general-purpose list.

A separate document provides several "focus profiles" with their own criteria for selection and ranking, which may be more useful than the general list.

Name	Description
On the Cusp: Weaknesses that Did Not Make the 2010 Top 25	From the original nominee list of 41 submitted CWE entries, the Top 25 was selected. This "On the Cusp" profile includes the remaining 16 weaknesses that did not make it into the final Top 25.
Educational Emphasis	This profile ranks weaknesses that are important from an educational perspective within a school or university context. It focuses on the CWE entries that graduating students should know, including historically important weaknesses.
Weaknesses by Language	This profile specifies which weaknesses appear in which programming languages. Notice that most weaknesses are actually language-independent, although they may be more prevalent in one language or another.
Weaknesses Typically Fixed in Design or Implementation	This profile lists weaknesses that are typically fixed in design or implementation.
Automated vs. Manual Analysis	This profile highlights which weaknesses can be detected using automated versus manual analysis. Currently, there is very little public, authoritative information about the efficacy of these methods and their utility. There are many competing opinions, even among experts. As a result, these ratings should only be treated as guidelines, not rules.
For Developers with Established Software Security Practices	This profile is for developers who have already established security in their practice. It uses votes from the major developers who contributed to the Top 25.
Ranked by Importance - for Software Customers	This profile ranks weaknesses based primarily on their importance, as determined from the base voting data that was used to create the general list. Prevalence is included in the scores, but it has much less weighting than importance.
Weaknesses by Technical Impact	This profile lists weaknesses based on their technical impact, i.e., what an attacker can accomplish by exploiting each weakness.

Organization of the Top 25

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

Ranking	The ranking of the weakness in the general list.
Score Summary	A summary of the individual ratings and scores that were given to this weakness, including Prevalence, Importance, and Adjusted Score.
CWE ID and name	CWE identifier and short name of the weakness
Supporting Information	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. The discussion avoids digging too deeply into technical detail.
Prevention	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

and Mitigations	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.
General Parent	One or more pointers to more general CWE entries, so you can see the breadth and depth of the problem.
Related Attack Patterns	CAPEC entries for attacks that may be successfully conducted against the weakness. Note: the list is not necessarily complete.
Other pointers	Links to more details including source code examples that demonstrate the weakness, methods for detection, etc.

Supporting Information

Each Top 25 entry includes supporting data fields for weakness prevalence, technical impact, and other information. Each entry also includes the following data fields.

Field	Description
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 Appendix A for more details.

Detailed CWE Descriptions

This section provides details for each individual CWE entry, along with links to additional information. See the Organization of the Top 25 section for an explanation of the various fields.

1 CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

Summary

Weakness Prevalence	High	Consequences	Code execution, Security bypass
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

Cross-site scripting (XSS) is one of the most prevalent, obstinate, and dangerous vulnerabilities in web applications. It's pretty much inevitable when you combine the stateless nature of HTTP, the mixture of data and script in HTML, lots of data passing between web sites, diverse encoding schemes, and feature-rich web browsers. If you're not careful, attackers can inject Javascript or other browser-executable content into a web page that your application generates. Your web page is then accessed by other users, whose browsers execute that malicious script as if it came from you (because, after all, it *did* come from you). Suddenly, your web site is serving code that you didn't write. The attacker can use a variety of techniques to get the input directly into your server, or use an unwitting victim as the middle man in a technical version of the "why do you keep hitting yourself?" game.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples of libraries and frameworks that make it easier to generate properly encoded output include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.

Implementation, Architecture and Design
Understand the context in which your 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.

Parts of the same output document may require different encodings, which will vary depending on whether the output is in the:

HTML body

Element attributes (such as src="XYZ")

URIs

JavaScript sections

Cascading Style Sheets and style property

etc. Note that HTML Entity Encoding is only appropriate for the HTML body.

Consult the XSS Prevention Cheat Sheet [REF-16] for more details on the types of encoding and escaping that are needed.

Architecture and Design, Implementation
Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.
Effectiveness: Limited

Notes: This technique has limited effectiveness, but can be helpful when it is possible to store client state and sensitive information on the server side instead of in cookies, headers, hidden form fields, etc.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Architecture and Design
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.

Implementation
For every web page that is generated, use and specify a character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can cause the web browser to treat certain sequences as special, opening up the client to subtle XSS attacks. See CWE-116 for more mitigations related to encoding/escaping.

Implementation
With Struts, you should write all data from form beans with the bean's filter attribute set to true.

Implementation
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.
Effectiveness: Defense in Depth

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

When dynamically constructing web pages, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. All input should be validated and cleansed, not just parameters that the user is supposed to specify, but all data in the request, including hidden fields, cookies, headers, the URL itself, and so forth. A common mistake that leads to continuing XSS vulnerabilities is to validate only fields that are expected to be redisplayed by the site. It is common to see data from the request that is reflected by the application server or the application that the development team did not anticipate. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the HTTP request is recommended.

Note that proper output encoding, escaping, and quoting is the most effective solution for preventing XSS, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent XSS, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, in a chat application, the heart emoticon ("<3") would likely pass the validation step, since it is commonly used. However, it cannot be directly inserted into the web page because it contains the "<" character, which would need to be escaped or otherwise handled. In this case, stripping the "<" might reduce the risk of XSS, but it would produce incorrect behavior because the emoticon would not be recorded. This might seem to be a minor inconvenience, but it would be more important in a mathematical forum that wants to represent inequalities.

Even if you make a mistake in your validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address.

Ensure that you perform input validation at well-defined interfaces within the application. This will help protect the application even if a component is reused or moved elsewhere.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Operation
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.
Effectiveness: Moderate

Notes: An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.

Related CWEs

CWE-82	Improper Neutralization of Script in Attributes of IMG Tags in a Web Page
CWE-85	Doubled Character XSS Manipulations
CWE-87	Improper Neutralization of Alternate XSS Syntax
CWE-692	Incomplete Blacklist to Cross-Site Scripting

Related Attack Patterns

CAPEC-IDs: [view all]
18, 19, 32, 63, 85, 86, 91, 106, 198, 199, 209, 232, 243, 244, 245, 246, 247

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

Summary

Weakness Prevalence	High	Consequences	Data loss, Security bypass
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

These days, it seems as if software is all about the data: getting it into the database, pulling it from the database, massaging it into information, and sending it elsewhere for fun and profit. If attackers can influence the SQL that you use to communicate with your database, then suddenly all your fun and profit belongs to them. If you use SQL queries in security controls such as authentication, attackers could alter the logic of those queries to bypass security. They could modify the queries to steal, corrupt, or otherwise change your underlying data. They'll even steal data one byte at a time if they have to, and they have the patience and know-how to do so.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, consider using persistence layers such as Hibernate or Enterprise Java Beans, which can provide significant protection against SQL injection if used properly.

Architecture and Design
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 you may re-introduce the possibility of SQL injection.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.
Specifically, 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.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Implementation
If you need to use dynamically-generated query strings or commands in spite of the risk, properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).
Instead of building your own implementation, such features may be available in the database or programming language. For example, the Oracle DBMS_ASSERT package can check or enforce that parameters have certain properties that make them less vulnerable to SQL injection. For MySQL, the mysql_real_escape_string() API function is available in both C and PHP.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

When constructing SQL query strings, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. This will indirectly limit the scope of an attack, but this technique is less important than proper output encoding and escaping.

Note that proper output encoding, escaping, and quoting is the most effective solution for preventing SQL injection, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent SQL injection, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, the name "O'Reilly" would likely pass the validation step, since it is a common last name in the English language. However, it cannot be directly inserted into the database because it contains the "'" apostrophe character, which would need to be escaped or otherwise handled. In this case, stripping the apostrophe might reduce the risk of SQL injection, but it would produce incorrect behavior because the wrong name would be recorded.

When feasible, it may be safest to disallow meta-characters entirely, instead of escaping them. This will provide some defense in depth. After the data is entered into the database, later processes may neglect to escape meta-characters before use, and you may not have control over those processes.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Implementation
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success.
If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.

In the context of SQL Injection, error messages revealing the structure of a SQL query can help attackers tailor successful attack strings.

Operation
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.
Effectiveness: Moderate

Notes: An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.

Related CWEs

CWE-90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
CWE-564	SQL Injection: Hibernate
CWE-566	Authorization Bypass Through User-Controlled SQL Primary Key

CWE-619	Dangling Database Cursor ('Cursor Injection')

Related Attack Patterns

CAPEC-IDs: [view all]
7, 66, 108, 109, 110

3 CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')

Summary

Weakness Prevalence	High	Consequences	Code execution, Denial of service, Data loss
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

Buffer overflows are Mother Nature's little reminder of that law of physics that says: if you try to put more stuff into a container than it can hold, you're going to make a mess. The scourge of C applications for decades, buffer overflows have been remarkably resistant to elimination. However, copying an untrusted input without checking the size of that input is the simplest error to make in a time when there are much more interesting mistakes to avoid. That's why this type of buffer overflow is often referred to as "classic." It's decades old, and it's typically one of the first things you learn about in Secure Programming 101.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Requirements
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and 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.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples include the Safe C String Library (SafeStr) by Messier and Viega, and the Strsafe.h library from Microsoft. These libraries provide safer versions of overflow-prone string-handling functions.

Notes: This is not a complete solution, since many buffer overflows are not related to strings.

Build and Compilation
Run or compile your software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows.
For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.

Effectiveness: Defense in Depth

Notes: This is not necessarily a complete solution, since these mechanisms can only detect certain types of overflows. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Implementation
Consider adhering to the following rules when allocating and managing an application's memory:
Double check that your buffer is as large as you specify.

When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string.

Check buffer boundaries if accessing the buffer in a loop and make sure you are not in danger of writing past the allocated space.

If necessary, truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be

syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Operation
Use a feature like Address Space Layout Randomization (ASLR).
Effectiveness: Defense in Depth

Notes: This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Operation
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent.
Effectiveness: Defense in Depth

Notes: 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. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Build and Compilation, Operation
Most mitigating technologies at the compiler or OS level to date address only a subset of buffer overflow problems and rarely provide complete protection against even that subset. It is good practice to implement strategies to increase the workload of an attacker, such as leaving the attacker to guess an unknown value that changes every program execution.

Implementation
Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.
Effectiveness: Moderate

Notes: This approach is still susceptible to calculation errors, including issues such as off-by-one errors (CWE-193) and incorrectly calculating buffer lengths (CWE-131).

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Related CWEs

CWE-129	Improper Validation of Array Index
CWE-131	Incorrect Calculation of Buffer Size

Related Attack Patterns

CAPEC-IDs: [view all]
8, 9, 10, 14, 24, 42, 44, 45, 46, 47, 67, 92, 100

4 CWE-352: Cross-Site Request Forgery (CSRF)

Summary

Weakness Prevalence	High	Consequences	Data loss, Code execution
Remediation Cost	High	Ease of Detection	Moderate
Attack Frequency	Often	Attacker Awareness	Medium

Discussion

You know better than to accept a package from a stranger at the airport. It could contain dangerous contents. Plus, if anything goes wrong, then it's going to look as if you did it, because you're the one with the package when you board the plane. Cross-site request forgery is like that strange package, except the attacker tricks a user into activating a request that goes to your site. Thanks to scripting and the way the web works in general, the user might not even be aware that the request is being sent. But once the request gets to your server, it looks as if it came from the user, not the attacker. This might not seem like a big deal, but the attacker has essentially masqueraded as a legitimate user and gained all the potential access that the user has. This is especially handy when the user has administrator privileges, resulting in a complete compromise of your application's functionality. When combined with XSS, the result can be extensive and devastating. If you've heard about XSS worms that stampede through very large web sites in a matter of minutes, there's usually CSRF feeding them.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, use anti-CSRF packages such as the OWASP CSRFGuard.

Another example is the ESAPI Session Management control, which includes a component for CSRF.

Implementation
Ensure that your application is free of cross-site scripting issues (CWE-79), because most CSRF defenses can be bypassed using attacker-controlled script.

Architecture and Design
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).
Notes: Note that this can be bypassed using XSS (CWE-79).

Architecture and Design
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.
Notes: Note that this can be bypassed using XSS (CWE-79).

Architecture and Design
Use the "double-submitted cookie" method as described by Felten and Zeller.
This technique requires Javascript, so it may not work for browsers that have Javascript disabled.

Notes: Note that this can probably be bypassed using XSS (CWE-79).

Architecture and Design
Do not use the GET method for any request that triggers a state change.

Implementation
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.
Notes: 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.

Related CWEs

CWE-346	Origin Validation Error
CWE-441	Unintended Proxy/Intermediary

Related Attack Patterns

CAPEC-IDs: [view all]
62, 111

5 CWE-285: Improper Authorization

Summary

Weakness Prevalence	High	Consequences	Security bypass
Remediation Cost	Low to Medium	Ease of Detection	Moderate

Attack Frequency	Often	Attacker Awareness	High

Discussion

Suppose you're hosting a house party for a few close friends and their guests. You invite everyone into your living room, but while you're catching up with one of your friends, one of the guests raids your fridge, peeks into your medicine cabinet, and ponders what you've hidden in the nightstand next to your bed. Software faces similar authorization problems that could lead to more dire consequences. If you don't ensure that your software's users are only doing what they're allowed to, then attackers will try to exploit your improper authorization and exercise unauthorized functionality that you only intended for restricted users.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully mapping roles with data and functionality. Use role-based access control (RBAC) to enforce the roles at the appropriate boundaries.
Note that this approach may not protect against horizontal authorization, i.e., it will not protect a user from attacking others with the same role.

Architecture and Design
Ensure that you perform access control checks related to your business logic. These checks may be different than the access control checks that you apply to more generic resources such as files, connections, processes, memory, and database records. For example, a database may restrict access for medical records to a specific database user, but each record might only be intended to be accessible to the patient and the patient's doctor.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, consider using authorization frameworks such as the JAAS Authorization Framework and the OWASP ESAPI Access Control feature.

Architecture and Design
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.

System Configuration, Installation
Use the access control capabilities of your operating system and server environment and define your access control lists accordingly. Use a "default deny" policy when defining these ACLs.

Related CWEs

CWE-425	Direct Request ('Forced Browsing')
CWE-639	Authorization Bypass Through User-Controlled Key
CWE-732	Incorrect Permission Assignment for Critical Resource
CWE-749	Exposed Dangerous Method or Function

Related Attack Patterns

CAPEC-IDs: [view all]
1, 13, 17, 39, 45, 51, 59, 60, 76, 77, 87, 104

6 CWE-807: Reliance on Untrusted Inputs in a Security Decision

Summary

Weakness Prevalence	High	Consequences	Security bypass
Remediation Cost	Medium	Ease of Detection	Moderate
Attack Frequency	Often	Attacker Awareness	High

Discussion

In countries where there is a minimum age for purchasing alcohol, the bartender is typically expected to verify the purchaser's age by checking a driver's license or other legally acceptable proof of age. But if somebody looks old enough to drink, then the bartender may skip checking the license altogether. This is a good thing for underage customers who happen to look older. Driver's licenses may require close scrutiny to identify fake licenses, or to determine if a person is using someone else's license. Software developers often rely on untrusted inputs in the same way, and when these inputs are used to decide whether to grant access to restricted resources, trouble is just around the corner.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Architecture and Design
Store state information and sensitive data on the server side only.
Ensure that the system definitively and unambiguously keeps track of its own state and user state and has rules defined for legitimate state transitions. Do not allow any application user to affect state directly in any way other than through legitimate actions leading to state transitions.

If information must be stored on the client, do not do so without encryption and integrity checking, or otherwise having a mechanism on the server side to catch tampering. Use a message authentication code (MAC) algorithm, such as Hash Message Authentication Code (HMAC). Apply this against the state or sensitive data that you have to expose, which can guarantee the integrity of the data - i.e., that the data has not been modified. Ensure that you use an algorithm with a strong hash function (CWE-328).

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
With a stateless protocol such as HTTP, use a framework that maintains the state for you.

Examples include ASP.NET View State 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.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.

Architecture and Design, Implementation
Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.
Identify all inputs that are used for security decisions and determine if you can modify the design so that you do not have to rely on submitted inputs at all. For example, you may be able to keep critical information about the user's session on the server side instead of recording it within external data.

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
232

7 CWE-22: Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')

Summary

Weakness Prevalence	Widespread	Consequences	Code execution, Data loss, Denial of service
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

While data is often exchanged using files, sometimes you don't intend to expose every file on your system while doing so. When you use an outsider's input while constructing a filename, the resulting path could point outside of the intended directory. An attacker could combine multiple

".." or similar sequences to cause the operating system to navigate out of the restricted directory, and into the rest of the system.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if the input is only expected to contain colors such as "red" or "blue."

Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). A blacklist is likely to miss at least one undesirable input, especially if the code's environment changes. This can give attackers enough room to bypass the intended validation. However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

When validating filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a whitelist of allowable file extensions, which will help to avoid CWE-434.

Do not rely exclusively on a filtering mechanism that removes potentially dangerous characters. This is equivalent to a blacklist, which may be incomplete (CWE-184). For example, filtering "/" is insufficient protection if the filesystem also supports the use of "\" as a directory separator. Another possible error could occur when the filtering is applied in a way that still produces dangerous data (CWE-182). For example, if "../" sequences are removed from the ".../...//" string in a sequential fashion, two instances of "../" would be removed from the original string, but the remaining characters would still form the "../" string.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Implementation
Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180). Make sure that the application does not decode the same input twice (CWE-174). Such errors could be used to bypass whitelist validation schemes by introducing dangerous inputs after they have been checked.
Use a built-in path canonicalization function (such as realpath() in C) that produces the canonical version of the pathname, which effectively removes ".." sequences and symbolic links (CWE-23, CWE-59). This includes:

realpath() in C

getCanonicalPath() in Java

GetFullPath() in ASP.NET

realpath() or abs_path() in Perl

realpath() in PHP

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Operation
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.
Effectiveness: Moderate

Notes: An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.
For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap provide this capability.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Architecture and Design, Operation
Store library, include, and utility files outside of the web document root, if possible. Otherwise, store them in a separate directory and use the web server's access control capabilities to prevent attackers from directly requesting them. One common practice is to define a fixed constant in each calling program, then check for the existence of the constant in the library/include file; if the constant does not exist, then the file was directly requested, and it can exit immediately.
This significantly reduces the chance of an attacker being able to bypass any protection mechanisms that are in the base program but not in the include files. It will also reduce your attack surface.

Implementation
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success.
If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.

In the context of path traversal, error messages which disclose path information can help attackers craft the appropriate attack strings to move through the file system hierarchy.

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
23, 64, 76, 78, 79, 139

8 CWE-434: Unrestricted Upload of File with Dangerous Type

Summary

Weakness Prevalence	Common	Consequences	Code execution
Remediation Cost	Medium	Ease of Detection	Moderate
Attack Frequency	Sometimes	Attacker Awareness	Medium

Discussion

You may think you're allowing uploads of innocent images (rather, images that won't damage your system - the Interweb's not so innocent in some places). But the name of the uploaded file could contain a dangerous extension such as .php instead of .gif, or other information (such as content type) may cause your server to treat the image like a big honkin' program. So, instead of seeing the latest paparazzi shot of your favorite Hollywood celebrity in a compromising position, you'll be the one whose server gets compromised.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
Generate your own filename for an uploaded file instead of the user-supplied filename, so that no external input is used at all.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Architecture and Design
Consider storing the uploaded files outside of the web document root entirely. Then, use other mechanisms to deliver the files dynamically.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be

syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

For example, limiting filenames to alphanumeric characters can help to restrict the introduction of unintended file extensions.

Architecture and Design
Define a very limited set of allowable extensions and only generate filenames that end in these extensions. Consider the possibility of XSS (CWE-79) before you allow .html or .htm file types.

Implementation
Ensure that only one extension is used in the filename. Some web servers, including some versions of Apache, may process files based on inner extensions so that "filename.php.gif" is fed to the PHP interpreter.

Implementation
When running on a web server that supports case-insensitive filenames, ensure that you perform case-insensitive evaluations of the extensions that are provided.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Implementation
Do not rely exclusively on sanity checks of file contents to ensure that the file is of the expected type and size. It may be possible for an attacker to hide code in some file segments that will still be executed by the server. For example, GIF images may contain a free-form comments field.

Implementation
Do not rely exclusively on the MIME content type or filename attribute when determining how to render a file. Validating the MIME content type and ensuring that it matches the extension is only a partial solution.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
1, 122

9 CWE-78: Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')

Summary

Weakness Prevalence	Medium		Consequences	Code execution
Remediation Cost	Medium		Ease of Detection	Easy
Attack Frequency	Often		Attacker Awareness	High

Discussion

Your software is often the bridge between an outsider on the network and the internals of your operating system. When you invoke another program on the operating system, but you allow untrusted inputs to be fed into the command string that you generate for executing that program, then you are inviting attackers to cross that bridge into a land of riches by executing their own commands instead of yours.

Prevention and Mitigations

Architecture and Design
If at all possible, use library calls rather than external processes to recreate the desired functionality.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Architecture and Design
For any data that will be used to generate a command to be executed, keep as much of that data out of external control as possible. For example, in web applications, this may require storing the data locally in the session's state instead of sending it out to the client in a hidden form field.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, consider using the ESAPI Encoding control or a similar tool, library, or framework. These will help the programmer encode outputs in a manner less prone to error.

Implementation
If you need to use dynamically-generated query strings or commands in spite of the risk, properly quote arguments and escape any special characters within those arguments. The most conservative approach is to escape or filter all characters that do not pass an extremely strict whitelist (such as everything that is not alphanumeric or white space). If some special characters are still needed, such as white space, wrap each argument in quotes after the escaping/filtering step. Be careful of argument injection (CWE-88).

Implementation
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.

Architecture and Design
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.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

When constructing OS command strings, use stringent whitelists that limit the character set based on the expected value of the parameter in the request. This will indirectly limit the scope of an attack, but this technique is less important than proper output encoding and escaping.

Note that proper output encoding, escaping, and quoting is the most effective solution for preventing OS command injection, although input validation may provide some defense-in-depth. This is because it effectively limits what will appear in output. Input validation will not always prevent OS command injection, especially if you are required to support free-form text fields that could contain arbitrary characters. For example, when invoking a mail program, you might need to allow the subject field to contain otherwise-dangerous inputs like ";" and ">" characters, which would need to be escaped or otherwise handled. In this case, stripping the character might reduce the risk of OS command injection, but it would produce incorrect behavior because the subject field would not be recorded as the user intended. This might seem to be a minor inconvenience, but it could be more important when the program relies on well-structured subject lines in order to pass messages to other components.

Even if you make a mistake in your validation (such as forgetting one out of 100 input fields), appropriate encoding is still likely to protect you from injection-based attacks. As long as it is not done in isolation, input validation is still a useful technique, since it may significantly reduce your attack surface, allow you to detect some attacks, and provide other security benefits that proper encoding does not address.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.

Operation
Run the code in an environment that performs automatic taint propagation and prevents any command execution that uses tainted variables, such as Perl's "-T" switch. This will force you to perform validation steps that remove the taint, although you must be careful to correctly validate your inputs so that you do not accidentally mark dangerous inputs as untainted (see CWE-183 and CWE-184).

Implementation
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the

balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success.
If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.

In the context of OS Command Injection, error information passed back to the user might reveal whether an OS command is being executed and possibly which command is being used.

Operation
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.

Operation
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.
Effectiveness: Moderate

Notes: An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.

Related CWEs

CWE-88	Argument Injection or Modification

Related Attack Patterns

CAPEC-IDs: [view all]
6, 15, 43, 88, 108

10 CWE-311: Missing Encryption of Sensitive Data

Summary

Weakness Prevalence	High	Consequences	Data loss
Remediation Cost	Medium	Ease of Detection	Easy
Attack Frequency	Sometimes	Attacker Awareness	High

Discussion

Whenever sensitive data is being stored or transmitted anywhere outside of your control, attackers may be looking for ways to get to it. Thieves could be anywhere - sniffing your packets, reading your databases, and sifting through your file systems. If your software sends sensitive information across a network, such as private data or authentication credentials, that information crosses many different nodes in transit to its final destination. Attackers can sniff this data right off the wire, and it doesn't require a lot of effort. All they need to do is control one node along the path to the final destination, control any node within the same networks of those transit nodes, or plug into an available interface. If your software stores sensitive information on a local file or database, there may be other ways for attackers to get at the file. They may benefit from lax permissions, exploitation of another vulnerability, or physical theft of the disk. You know those massive credit card thefts you keep hearing about? Many of them are due to unencrypted storage.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Requirements
Clearly specify which data or resources are valuable enough that they should be protected by encryption. Require that any transmission or storage of

this data/resource should use well-vetted encryption algorithms.

Architecture and Design
Using threat modeling or other techniques, assume that your data can be compromised through a separate vulnerability or weakness, and determine where encryption will be most effective. Ensure that data you believe should be private is not being inadvertently exposed using weaknesses such as insecure permissions (CWE-732).

Architecture and Design
Ensure that encryption is properly integrated into the system design, including but not necessarily limited to:
Encryption that is needed to store or transmit private data of the users of the system

Encryption that is needed to protect the system itself from unauthorized disclosure or tampering

Identify the separate needs and contexts for encryption:

One-way (i.e., only the user or recipient needs to have the key). This can be achieved using public key cryptography, or other techniques in which the encrypting party (i.e., the software) does not need to have access to a private key.

Two-way (i.e., the encryption can be automatically performed on behalf of a user, but the key must be available so that the plaintext can be automatically recoverable by that user). This requires storage of the private key in a format that is recoverable only by the user (or perhaps by the operating system) in a way that cannot be recovered by others.

Architecture and Design
Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis.
For example, US government systems require FIPS 140-2 certification.

Do not develop your own cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If your algorithm can be compromised if attackers find out how it works, then it is especially weak.

Periodically ensure that you aren't using obsolete cryptography. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once regarded as strong.

Architecture and Design
Compartmentalize your system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area.

Implementation, Architecture and Design
When you use industry-approved techniques, you need to use them correctly. Don't cut corners by skipping resource-intensive steps (CWE-325). These steps are often essential for preventing common attacks.

Implementation
Use naming conventions and strong types to make it easier to spot when sensitive data is being used. When creating structures, objects, or other complex entities, separate the sensitive and non-sensitive data as much as possible.
Effectiveness: Defense in Depth

Notes: This makes it easier to spot places in the code where data is being used that is unencrypted.

Related CWEs

CWE-312	Cleartext Storage of Sensitive Information
CWE-319	Cleartext Transmission of Sensitive Information

Related Attack Patterns

CAPEC-IDs: [view all]
31, 37, 65, 117, 155, 157, 167, 204, 205, 258, 259, 260, 383, 384, 385, 386, 387, 388, 389

11 CWE-798: Use of Hard-coded Credentials

Summary

Weakness Prevalence	Medium	Consequences	Security bypass
Remediation Cost	Medium to High	Ease of Detection	Moderate
Attack Frequency	Rarely	Attacker Awareness	High

Discussion

Hard-coding a secret password or cryptograpic key into your program is bad manners, even though it makes it extremely convenient - for skilled reverse engineers. While it might shrink your testing and support budgets, it can reduce the security of your customers to dust. If the password is the same across all your software, then every customer becomes vulnerable if (rather, when) your password becomes known. Because it's hard-coded, it's usually a huge pain for sysadmins to fix. And you know how much they love inconvenience at 2 AM when their network's being hacked

- about as much as you'll love responding to hordes of angry customers and reams of bad press if your little secret should get out. Most of the CWE Top 25 can be explained away as an honest mistake; for this issue, though, customers won't see it that way. Another way that hard-coded credentials arise is through unencrypted or obfuscated storage in a configuration file, registry key, or other location that is only intended to be accessible to an administrator. While this is much more polite than burying it in a binary program where it can't be modified, it becomes a Bad Idea to expose this file to outsiders through lax permissions or other means.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
For outbound authentication: store passwords, keys, and other credentials outside of the code in a strongly-protected, encrypted configuration file or database that is protected from access by all outsiders, including other local users on the same system. Properly protect the key (CWE-320). If you cannot use encryption to protect the file, then make sure that the permissions are as restrictive as possible.
In Windows environments, the Encrypted File System (EFS) may provide some protection.

Architecture and Design
For inbound authentication: Rather than hard-code a default username and password, key, or other authentication credentials for first time logins, utilize a "first login" mode that requires the user to enter a unique strong password or key.

Architecture and Design
If the software must contain hard-coded credentials or they cannot be removed, perform access control checks and limit which entities can access the feature that requires the hard-coded credentials. For example, a feature might only be enabled through the system console instead of through a network connection.

Architecture and Design
For inbound authentication using passwords: apply strong one-way hashes to your passwords and store those hashes in a configuration file or database with appropriate access control. That way, theft of the file/database still requires the attacker to try to crack the password. When handling an incoming password during authentication, take the hash of the password and compare it to the hash that you have saved.
Use randomly assigned salts for each separate hash that you generate. This increases the amount of computation that an attacker needs to conduct a brute-force attack, possibly limiting the effectiveness of the rainbow table method.

Architecture and Design
For front-end to back-end connections: Three solutions are possible, although none are complete.
The first suggestion involves the use of generated passwords or keys that are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals.

Next, the passwords or keys should be limited at the back end to only performing actions valid for the front end, as opposed to having full access.

Finally, the messages sent should be tagged and checksummed with time sensitive values so as to prevent replay-style attacks.

Related CWEs

CWE-259	Use of Hard-coded Password
CWE-321	Use of Hard-coded Cryptographic Key

Related Attack Patterns

CAPEC-IDs: [view all]
70, 188, 189, 190, 191, 192, 205

12 CWE-805: Buffer Access with Incorrect Length Value

Summary

Weakness Prevalence	Common	Consequences	Code execution, Denial of service, Data loss
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

A popular insult is: "Take a long walk off a short pier." One programming equivalent for this insult is to access memory buffers using an incorrect length value. Whether you're reading or writing data as you march down that pier, once you've passed the boundaries of the buffer, you'll wind up in deep water.

Prevention and Mitigations

Requirements
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and 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.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Examples include the Safe C String Library (SafeStr) by Messier and Viega, and the Strsafe.h library from Microsoft. These libraries provide safer versions of overflow-prone string-handling functions.

Notes: This is not a complete solution, since many buffer overflows are not related to strings.

Build and Compilation
Run or compile your software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows.
For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.

Effectiveness: Defense in Depth

Notes: This is not necessarily a complete solution, since these mechanisms can only detect certain types of overflows. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Implementation
Consider adhering to the following rules when allocating and managing an application's memory:
Double check that your buffer is as large as you specify.

When using functions that accept a number of bytes to copy, such as strncpy(), be aware that if the destination buffer size is equal to the source buffer size, it may not NULL-terminate the string.

Check buffer boundaries if accessing the buffer in a loop and make sure you are not in danger of writing past the allocated space.

If necessary, truncate all input strings to a reasonable length before passing them to the copy and concatenation functions.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Operation
Use a feature like Address Space Layout Randomization (ASLR).
Effectiveness: Defense in Depth

Notes: This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Operation
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent.
Effectiveness: Defense in Depth

Notes: 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. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Related CWEs

CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-126	Buffer Over-read
CWE-129	Improper Validation of Array Index
CWE-131	Incorrect Calculation of Buffer Size

Related Attack Patterns

CAPEC-IDs: [view all]
100

13 CWE-98: Improper Control of Filename for Include/Require Statement in PHP Program ('PHP File Inclusion')

Summary

Weakness Prevalence	Common	Consequences	Code execution, Data loss
Remediation Cost	Low to Medium	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

Not a lot of Top 25 weaknesses are unique to a single programming language, but that just goes to show how special this one is. The idea was simple enough: you can make a lot of smaller parts of a document (or program), then combine them all together into one big document (or program) by "including" or "requiring" those smaller pieces. This is a common enough way to build programs. Combine this with the common tendency to allow attackers to influence the location of the document (or program) - perhaps even on an attacker-controlled web site, if you're unlucky enough - then suddenly the attacker can read any document (or run any program) on your web server. This feature has been removed or significantly limited in later versions of PHP, but despite the evidence that everything changes on the Internet every 2 years, code is forever.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.
For example, ID 1 could map to "inbox.txt" and ID 2 could map to "profile.txt". Features such as the ESAPI AccessReferenceMap provide this capability.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

For filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a whitelist of allowable file extensions, which will help to avoid CWE-434.

Architecture and Design, Operation
Store library, include, and utility files outside of the web document root, if possible. Otherwise, store them in a separate directory and use the web server's access control capabilities to prevent attackers from directly requesting them. One common practice is to define a fixed constant in each calling program, then check for the existence of the constant in the library/include file; if the constant does not exist, then the file was directly requested, and it can exit immediately.
This significantly reduces the chance of an attacker being able to bypass any protection mechanisms that are in the base program but not in the include files. It will also reduce your attack surface.

Architecture and Design, Implementation
Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.
Many file inclusion problems occur because the programmer assumed that certain inputs could not be modified, especially for cookies and URL components.

Operation
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.
Effectiveness: Moderate

Notes: An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Operation, Implementation
Develop and run your code in the most recent versions of PHP available, preferably PHP 6 or later. Many of the highly risky features in earlier PHP interpreters have been removed, restricted, or disabled by default.

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.
Often, programmers do not protect direct access to files intended only to be included by core programs. These include files may assume that critical variables have already been initialized by the calling program. As a result, the use of register_globals combined with the ability to directly access the include file may allow attackers to conduct file inclusion attacks. This remains an extremely common pattern as of 2009.

Operation
Set allow_url_fopen to false, which limits the ability to include files from remote locations.
Effectiveness: High

Notes: Be aware that some versions of PHP will still accept ftp:// and other URI schemes. In addition, this setting does not protect the code from path traversal attacks (CWE-22), which are frequently successful against the same vulnerable code that allows remote file inclusion.

Related CWEs

CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
CWE-73	External Control of File Name or Path

Related Attack Patterns

CAPEC-IDs: [view all]
193

14 CWE-129: Improper Validation of Array Index

Summary

Weakness Prevalence	Common	Consequences	Code execution, Denial of service, Data loss
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Sometimes	Attacker Awareness	Medium

Discussion

If you use untrusted inputs when calculating an index into an array, then an attacker could provide an index that is outside the boundaries of the array. If you've allocated an array of 100 objects or structures, and an attacker provides an index that is -23 or 978, then "unexpected behavior" is the euphemism for what happens next.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Architecture and Design
Use an input validation framework such as Struts or the OWASP ESAPI Validation API. If you use Struts, be mindful of weaknesses covered by the CWE-101 category.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.
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.

Requirements
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, Ada allows the programmer to constrain the values of a variable and languages such as Java and Ruby will allow the programmer to handle exceptions when an out-of-bounds index is accessed.

Operation
Use a feature like Address Space Layout Randomization (ASLR).
Effectiveness: Defense in Depth

Notes: This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Operation
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent.
Effectiveness: Defense in Depth

Notes: 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. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

When accessing a user-controlled array index, use a stringent range of values that are within the target array. Make sure that you do not allow negative values to be used. That is, verify the minimum as well as the maximum of the range of acceptable values.

Implementation
Be especially careful to validate your input when you invoke code that crosses language boundaries, such as from an interpreted language to native code. This could create an unexpected interaction between the language boundaries. Ensure that you are not violating any of the expectations of the language with which you are 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.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Related CWEs

CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-131	Incorrect Calculation of Buffer Size

Related Attack Patterns

CAPEC-IDs: [view all]
100

15 CWE-754: Improper Check for Unusual or Exceptional Conditions

Summary

Weakness Prevalence	High	Consequences	Denial of service, Security bypass, Data loss, Code execution
Remediation Cost	Low	Ease of Detection	Moderate
Attack Frequency	Often	Attacker Awareness	High

Discussion

Murphy's Law says that anything that can go wrong, will go wrong. Yet it's human nature to always believe that bad things could never happen, at least not to you. Security-wise, it pays to be cynical. If you always expect the worst, then you'll be better prepared for attackers who seek to inflict their worst. By definition, they're trying to use your software in ways you don't want.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Requirements
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Choose languages with features such as exception handling that force the programmer to anticipate unusual conditions that may generate exceptions. Custom exceptions may need to be developed to handle unusual business-logic conditions. Be careful not to pass sensitive exceptions back to the user (CWE-209, CWE-248).

Implementation
Check the results of all functions that return a value and verify that the value is expected.
Effectiveness: High

Notes: Checking the return value of the function will typically be sufficient, however beware of race conditions (CWE-362) in a concurrent environment.

Implementation
If using exception handling, catch and throw specific exceptions instead of overly-general exceptions (CWE-396, CWE-397). Catch and handle exceptions as locally as possible so that exceptions do not propagate too far up the call stack (CWE-705). Avoid unchecked or uncaught exceptions where feasible (CWE-248).
Effectiveness: High

Notes: Using specific exceptions, and ensuring that exceptions are checked, helps programmers to anticipate and appropriately handle many unusual events that could occur.

Implementation
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success.
If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.

Exposing additional information to a potential attacker in the context of an exceptional condition can help the attacker determine what attack vectors are most likely to succeed beyond DoS.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Notes: Performing extensive input validation does not help with handling unusual conditions, but it will minimize their occurrences and will make it more difficult for attackers to trigger them.

Architecture and Design, Implementation
If the program must fail, ensure that it fails gracefully (fails closed). There may be a temptation to simply let the program fail poorly in cases such as low memory conditions, but an attacker may be able to assert control before the software has fully exited. Alternately, an uncontrolled failure could cause cascading problems with other downstream components; for example, the program could send a signal to a downstream process so the process immediately knows that a problem has occurred and has a better chance of recovery.

Architecture and Design
Use system limits, which should help to prevent resource exhaustion. However, the software should still handle low resource conditions since they may

still occur.

Related CWEs

CWE-252	Unchecked Return Value
CWE-476	NULL Pointer Dereference

Related Attack Patterns

CAPEC-IDs: [view all]

16 CWE-209: Information Exposure Through an Error Message

Summary

Weakness Prevalence	High	Consequences	Data loss
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

If you use chatty error messages, then they could disclose secrets to any attacker who dares to misuse your software. The secrets could cover a wide range of valuable data, including personally identifiable information (PII), authentication credentials, and server configuration. Sometimes, they might seem like harmless secrets that are convenient for your users and admins, such as the full installation path of your software. Even these little secrets can greatly simplify a more concerted attack that yields much bigger rewards, which is done in real-world attacks all the time. This is a concern whether you send temporary error messages back to the user or if you permanently record them in a log file.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Implementation
Ensure that error messages only contain minimal details that are useful to the intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can be used to refine the original attack to increase the chances of success.
If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.

Implementation
Handle exceptions internally and do not display errors containing potentially sensitive information to a user.

Implementation
Use naming conventions and strong types to make it easier to spot when sensitive data is being used. When creating structures, objects, or other complex entities, separate the sensitive and non-sensitive data as much as possible.
Effectiveness: Defense in Depth

Notes: This makes it easier to spot places in the code where data is being used that is unencrypted.

Implementation, Build and Compilation
Debugging information should not make its way into a production release.

System Configuration
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.

System Configuration
Create default error pages or messages that do not leak any information.

Related CWEs

CWE-204	Response Discrepancy Information Exposure
CWE-210	Information Exposure Through Generated Error Message
CWE-538	File and Directory Information Exposure

Related Attack Patterns

CAPEC-IDs: [view all]
7, 54, 214, 215

17	CWE-190: Integer Overflow or Wraparound

Summary

Weakness Prevalence	Common	Consequences	Denial of service, Code execution, Data loss
Remediation Cost	Low	Ease of Detection	Easy
Attack Frequency	Sometimes	Attacker Awareness	High

Discussion

In the real world, 255+1=256. But to a computer program, sometimes 255+1=0, or 0-1=65535, or maybe 40,000+40,000=14464. You don't have to be a math whiz to smell something fishy. Actually, this kind of behavior has been going on for decades, and there's a perfectly rational and incredibly boring explanation. Ultimately, it's buried deep in the DNA of computers, who can't count to infinity even if it sometimes feels like they take that long to complete an important task. When programmers forget that computers don't do math like people, bad things ensue - anywhere from crashes, faulty price calculations, infinite loops, and execution of code.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Requirements
Ensure that all protocols are strictly defined, such that all out-of-bounds behavior can be identified simply, and require strict conformance to the protocol.

Requirements
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
If possible, choose a language or compiler that performs automatic bounds checking.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Use libraries or frameworks that make it easier to handle numbers without unexpected consequences.

Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++).

Implementation
Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.
Use unsigned integers where possible. This makes it easier to perform sanity checks for integer overflows. If you must use signed integers, make sure that your range check includes minimum values as well as maximum values.

Implementation
Understand your programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation.
Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Implementation
Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.

Related CWEs

CWE-191	Integer Underflow (Wrap or Wraparound)

Related Attack Patterns

CAPEC-IDs: [view all]
92

18 CWE-131: Incorrect Calculation of Buffer Size

Summary

Weakness Prevalence	High	Consequences	Code execution, Denial of service, Data loss
Remediation Cost	Low	Ease of Detection	Easy to Moderate
Attack Frequency	Often	Attacker Awareness	High

Discussion

In languages such as C, where memory management is the programmer's responsibility, there are many opportunities for error. If the programmer does not properly calculate the size of a buffer, then the buffer may be too small to contain the data that the programmer plans to write - even if the input was properly validated. Any number of problems could produce the incorrect calculation, but when all is said and done, you're going to run head-first into the dreaded buffer overflow.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Implementation
If you allocate a buffer for the purpose of transforming, converting, or encoding an input, make sure that you allocate enough memory to handle the largest possible encoding. For example, in a routine that converts "&" characters to "&amp;" for HTML entity encoding, you will need an output buffer that is at least 5 times as large as the input buffer.

Implementation
Understand your programming language's underlying representation and how it interacts with numeric calculation (CWE-681). Pay close attention to byte size discrepancies, precision, signed/unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation.
Also be careful to account for 32-bit, 64-bit, and other potential differences that may affect the numeric representation.

Implementation
Perform input validation on any numeric input by ensuring that it is within the expected range. Enforce that the input meets both the minimum and maximum requirements for the expected range.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Implementation
When processing structured incoming data containing a size field followed by raw data, ensure that you identify and resolve any inconsistencies between the size field and the actual size of the data (CWE-130).

Implementation
When allocating memory that uses sentinels to mark the end of a data structure - such as NUL bytes in strings - make sure you also include the sentinel in your calculation of the total amount of memory that must be allocated.

Implementation
Replace unbounded copy functions with analogous functions that support length arguments, such as strcpy with strncpy. Create these if they are not available.
Effectiveness: Moderate

Notes: This approach is still susceptible to calculation errors, including issues such as off-by-one errors (CWE-193) and incorrectly calculating buffer lengths (CWE-131).

Additionally, this only addresses potential overflow issues. Resource consumption / exhaustion issues are still possible.

Implementation
Use sizeof() on the appropriate data type to avoid CWE-467.

Implementation
Use the appropriate type for the desired action. For example, in C/C++, only use unsigned types for values that could never be negative, such as height, width, or other numbers related to quantity. This will simplify your sanity checks and will reduce surprises related to unexpected casting.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Use libraries or frameworks that make it easier to handle numbers without unexpected consequences, or buffer allocation routines that automatically track buffer size.

Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++).

Build and Compilation
Run or compile your software using features or extensions that automatically provide a protection mechanism that mitigates or eliminates buffer overflows.
For example, certain compilers and extensions provide automatic buffer overflow detection mechanisms that are built into the compiled code. Examples include the Microsoft Visual Studio /GS flag, Fedora/Red Hat FORTIFY_SOURCE GCC flag, StackGuard, and ProPolice.

Effectiveness: Defense in Depth

Notes: This is not necessarily a complete solution, since these mechanisms can only detect certain types of overflows. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Operation
Use a feature like Address Space Layout Randomization (ASLR).
Effectiveness: Defense in Depth

Notes: This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution. In addition, an attack could still cause a denial of service, since the typical response is to exit the application.

Operation
Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent.
Effectiveness: Defense in Depth

Notes: 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. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

Implementation
Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Related CWEs

CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
CWE-129	Improper Validation of Array Index
CWE-805	Buffer Access with Incorrect Length Value

Related Attack Patterns

CAPEC-IDs: [view all]
47, 100

19 CWE-306: Missing Authentication for Critical Function

Summary

Weakness Prevalence	Common	Consequences	Security bypass
Remediation Cost	Low to High	Ease of Detection	Moderate
Attack Frequency	Sometimes	Attacker Awareness	High

Discussion

In countless action movies, the villain breaks into a high-security building by crawling through heating ducts or pipes, scaling elevator shafts, or hiding under a moving cart. This works because the pathway into the building doesn't have all those nosy security guards asking for identification. Software may expose certain critical functionality with the assumption that nobody would think of trying to do anything but break in through the front door. But attackers know how to case a joint and figure out alternate ways of getting into a system.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Architecture and Design
Divide your software into anonymous, normal, privileged, and administrative areas. Identify which of these areas require a proven user identity, and use a centralized authentication capability.
Identify all potential communication channels, or other means of interaction with the software, to ensure that all channels are appropriately protected. Developers sometimes perform authentication at the primary channel, but open up a secondary channel that is assumed to be private. For example, a login mechanism may be listening on one network port, but after successful authentication, it may open up a second port where it waits for the connection, but avoids authentication because it assumes that only the authenticated party will connect to the port.

In general, if the software or protocol allows a single session or user state to persist across multiple connections or channels, authentication and appropriate credential management need to be used throughout.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Architecture and Design
Where possible, avoid implementing custom authentication routines and consider using authentication capabilities as provided by the surrounding framework, operating system, or environment. These may make it easier to provide a clear separation between authentication tasks and authorization tasks.
In environments such as the World Wide Web, the line between authentication and authorization is sometimes blurred. If custom authentication routines are required instead of those provided by the server, then these routines must be applied to every single page, since these pages could be requested directly.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, consider using libraries with authentication capabilities such as OpenSSL or the ESAPI Authenticator.

Related CWEs

CWE-302	Authentication Bypass by Assumed-Immutable Data
CWE-307	Improper Restriction of Excessive Authentication Attempts

Related Attack Patterns

CAPEC-IDs: [view all]
12, 36, 40, 62, 225

20 CWE-494: Download of Code Without Integrity Check

Summary

Weakness Prevalence	Medium	Consequences	Code execution
Remediation Cost	Medium to High	Ease of Detection	Moderate
Attack Frequency	Rarely	Attacker Awareness	Low

Discussion

You don't need to be a guru to realize that if you download code and execute it, you're trusting that the source of that code isn't malicious. Maybe you only access a download site that you trust, but attackers can perform all sorts of tricks to modify that code before it reaches you. They can hack the download site, impersonate it with DNS spoofing or cache poisoning, convince the system to redirect to a different site, or even modify the code in transit as it crosses the network. This scenario even applies to cases in which your own product downloads and installs its own updates. When this happens, your software will wind up running code that it doesn't expect, which is bad for you but great for attackers.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Implementation

Perform proper forward and reverse DNS lookups to detect DNS spoofing.
Notes: This is only a partial solution since it will not prevent your code from being modified on the hosting site or in transit.

Architecture and Design, Operation
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 your code from being modified on the hosting site.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid. Speficially, it may be helpful to use tools or frameworks to perform integrity checking on the transmitted code.

If you are providing the code that is to be downloaded, such as for automatic updates of your software, then use cryptographic signatures for your code and modify your download clients to verify the signatures. Ensure that your implementation does not contain CWE-295, CWE-320, CWE-347, and related weaknesses.

Use code signing technologies such as Authenticode. See references.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Limited

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Related CWEs

CWE-247	Reliance on DNS Lookups in a Security Decision
CWE-292	Trusting Self-reported DNS Name
CWE-346	Origin Validation Error
CWE-350	Improperly Trusted Reverse DNS

Related Attack Patterns

CAPEC-IDs: [view all]
184, 185, 186, 187

21 CWE-732: Incorrect Permission Assignment for Critical Resource

Summary

Weakness Prevalence	Medium	Consequences	Data loss, Code execution
Remediation Cost	Low to High	Ease of Detection	Easy
Attack Frequency	Often	Attacker Awareness	High

Discussion

It's rude to take something without asking permission first, but impolite users (i.e., attackers) are willing to spend a little time to see what they can get away with. If you have critical programs, data stores, or configuration files with permissions that make your resources readable or writable by the world - well, that's just what they'll become. While this issue might not be considered during implementation or design, sometimes that's where the solution needs to be applied. Leaving it up to a harried sysadmin to notice and make the appropriate changes is far from optimal, and sometimes impossible.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Implementation
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.

Architecture and Design
Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully defining distinct user groups, privileges, and/or roles. Map these against data, functionality, and the related resources. Then set the permissions accordingly. This will allow you to maintain more fine-grained control over your resources.
Effectiveness: Moderate

Notes: This can be an effective strategy. However, in practice, it may be difficult or time consuming to define these areas when there are many different resources or user types, or if the applications features change rapidly.

Architecture and Design, Operation
Run your code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by your software.
OS-level examples include the Unix chroot jail, AppArmor, and SELinux. In general, managed code may provide some protection. For example, java.io.FilePermission in the Java SecurityManager allows you to specify restrictions on file operations.

This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.

Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness: Moderate

Notes: The effectiveness of this mitigation depends on the prevention capabilities of the specific sandbox or jail being used and might only help to reduce the scope of an attack, such as restricting the attacker to certain system calls or limiting the portion of the file system that can be accessed.

Implementation, Installation
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 you from inheriting insecure permissions from any user who installs or runs the program.
Effectiveness: High

System Configuration
For all configuration files, executables, and libraries, make sure that they are only readable and writable by the software's administrator.
Effectiveness: High

Documentation
Do not suggest insecure configuration changes in your documentation, especially if those configurations can extend to resources and other software that are outside the scope of your own software.

Installation
Do not assume that the system administrator will manually change the configuration to the settings that you recommend in the manual.

Operation, System Configuration
Ensure that your 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.

Related CWEs

CWE-276	Incorrect Default Permissions
CWE-277	Insecure Inherited Permissions
CWE-279	Incorrect Execution-Assigned Permissions
CWE-285	Improper Authorization

Related Attack Patterns

CAPEC-IDs: [view all]
1, 17, 60, 61, 62, 122, 180, 232, 234

22 CWE-770: Allocation of Resources Without Limits or Throttling

Summary

Weakness Prevalence	High	Consequences	Denial of service
Remediation Cost	Low to High	Ease of Detection	Moderate
Attack Frequency	Rarely	Attacker Awareness	High

Discussion

Suppose you work at a pizza place. If someone calls in and places an order for a thousand pizzas (with anchovies) to be delivered immediately, you'd quickly put a stop to that nonsense. But a computer program, if left to its own devices, would happily try to fill that order. While software often runs under hard limits of the system (memory, disk space, CPU) - it's not particularly polite when it uses all these resources to the exclusion of everything else. And often, only a little bit is ever expected to be allocated to any one person or task. The lack of control over resource allocation is an avenue for attackers to cause a denial of service against other users of your software, possibly the entire system - and in some cases, this can be leveraged to conduct other more devastating attacks.

Technical Details | Code Examples | Detection Methods | References

Prevention and Mitigations

Requirements
Clearly specify the minimum and maximum expectations for capabilities, and dictate which behaviors are acceptable when resource allocation reaches limits.

Architecture and Design
Limit the amount of resources that are accessible to unprivileged users. Set per-user limits for resources. Allow the system administrator to define these limits. Be careful to avoid CWE-410.

Architecture and Design
Design throttling mechanisms into the system architecture. The best protection is to limit the amount of resources that an unauthorized user can cause to be expended. A strong authentication and access control model will help prevent such attacks from occurring in the first place, and it will help the administrator to identify who is committing the abuse. The login application should be protected against DoS attacks as much as possible. Limiting the database access, perhaps by caching result sets, can help minimize the resources expended. To further limit the potential for a DoS attack, consider tracking the rate of requests received from users and blocking requests that exceed a defined rate threshold.

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Notes: This will only be applicable to cases where user input can influence the size or frequency of resource allocations.

Architecture and Design
For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side, in order to avoid CWE-602. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.

Architecture and Design
Mitigation of resource exhaustion attacks requires that the target system either:
recognizes the attack and denies that user further access for a given amount of time, typically by using increasing time delays

uniformly throttles all requests in order to make it more difficult to consume resources more quickly than they can again be freed.

The first of these solutions is an issue in itself though, since it may allow attackers to prevent the use of the system by a particular valid user. If the attacker impersonates the valid user, he may be able to prevent the user from accessing the server in question.

The second solution can be difficult to effectively institute -- and even when properly done, it does not provide a full solution. It simply requires more resources on the part of the attacker.

Architecture and Design
Ensure that protocols have specific limits of scale placed on them.

Architecture and Design, Implementation
If the program must fail, ensure that it fails gracefully (fails closed). There may be a temptation to simply let the program fail poorly in cases such as low memory conditions, but an attacker may be able to assert control before the software has fully exited. Alternately, an uncontrolled failure could cause cascading problems with other downstream components; for example, the program could send a signal to a downstream process so the process immediately knows that a problem has occurred and has a better chance of recovery.
Ensure that all failures in resource allocation place the system into a safe posture.

Implementation
For system resources when using C, consider using the getrlimit() function included in the sys/resources library in order to determine how many files are currently allowed to be opened for the process.

Operation
Use resource-limiting settings provided by the operating system or environment. For example, setrlimit() can be used to set limits for certain types of resources. However, this is not available on all operating systems.
Ensure that your application performs the appropriate error checks and error handling in case resources become unavailable (CWE-703).

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
82, 99, 119, 121, 125, 130, 147, 197, 227, 228, 229

23 CWE-601: URL Redirection to Untrusted Site ('Open Redirect')

Summary

Weakness Prevalence	High	Consequences	Code execution, Data loss, Denial of service
Remediation Cost	Medium	Ease of Detection	Easy
Attack Frequency	Sometimes	Attacker Awareness	Medium

Discussion

While much of the power of the World Wide Web is in sharing and following links between web sites, typically there is an assumption that a user should be able to click on a link or perform some other action before being sent to a different web site. Many web applications have implemented redirect features that allow attackers to specify an arbitrary URL to link to, and the web client does this automatically. This may be another of those features that are "just the way the web works," but if left unchecked, it could be useful to attackers in a couple important ways. First, the victim could be autoamtically redirected to a malicious site that tries to attack the victim through the web browser. Alternately, a phishing attack could be conducted, which tricks victims into visiting malicious sites that are posing as legitimate sites. Either way, an uncontrolled redirect will send your users someplace that they don't want to go.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Implementation
Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a whitelist of acceptable inputs that strictly conform to specifications. Reject any input that does not strictly conform to specifications, or transform it into something that does. Do not rely exclusively on looking for malicious or malformed inputs (i.e., do not rely on a blacklist). However, blacklists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.
When performing input validation, consider all potentially relevant properties, including length, type of input, the full range of acceptable values, missing or extra inputs, syntax, consistency across related fields, and conformance to business rules. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if you are expecting colors such as "red" or "blue."

Use a whitelist of approved URLs or domains to be used for redirection.

Architecture and Design
Use an intermediate disclaimer page that provides the user with a clear warning that they are leaving your site. Implement a long timeout before the redirect occurs, or force the user to click on the link. Be careful to avoid XSS problems (CWE-79) when generating the disclaimer page.

Architecture and Design
When the set of acceptable objects, such as filenames or URLs, is limited or known, create a mapping from a set of fixed input values (such as numeric IDs) to the actual filenames or URLs, and reject all other inputs.
For example, ID 1 could map to "/login.asp" and ID 2 could map to "http://www.example.com/". Features such as the ESAPI AccessReferenceMap provide this capability.

Architecture and Design, Implementation
Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, reverse DNS lookups, query results, request headers, URL components, e-mail, files, filenames, databases, and any external systems that provide data to the application. Remember that such inputs may be obtained indirectly through API calls.
Many open redirect problems occur because the programmer assumed that certain inputs could not be modified, such as cookies and hidden form fields.

Operation
Use an application firewall that can detect attacks against this weakness. It can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.
Effectiveness: Moderate

Notes: An application firewall might not cover all possible input vectors. In addition, attack techniques might be available to bypass the protection mechanism, such as using malformed inputs that can still be processed by the component that receives those inputs. Depending on functionality, an application firewall might inadvertently reject or modify legitimate requests. Finally, some manual effort may be required for customization.

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
194

24 CWE-327: Use of a Broken or Risky Cryptographic Algorithm

Summary

Weakness Prevalence	High	Consequences	Data loss, Security bypass
Remediation Cost	Medium to High	Ease of Detection	Moderate
Attack Frequency	Rarely	Attacker Awareness	Medium

Discussion

If you are handling sensitive data or you need to protect a communication channel, you may be using cryptography to prevent attackers from reading it. You may be tempted to develop your own encryption scheme in the hopes of making it difficult for attackers to crack. This kind of grow-your-own cryptography is a welcome sight to attackers. Cryptography is just plain hard. If brilliant mathematicians and computer scientists worldwide can't get it right (and they're always breaking their own stuff), then neither can you. You might think you created a brand-new algorithm that nobody will figure out, but it's more likely that you're reinventing a wheel that falls off just before the parade is about to start.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
Select a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations. As with all cryptographic mechanisms, the source code should be available for analysis.
For example, US government systems require FIPS 140-2 certification.

Do not develop your own cryptographic algorithms. They will likely be exposed to attacks that are well-understood by cryptographers. Reverse engineering techniques are mature. If your algorithm can be compromised if attackers find out how it works, then it is especially weak.

Periodically ensure that you aren't using obsolete cryptography. Some older algorithms, once thought to require a billion years of computing time, can now be broken in days or hours. This includes MD4, MD5, SHA1, DES, and other algorithms that were once regarded as strong.

Architecture and Design
Design your software so that you can replace one cryptographic algorithm with another. This will make it easier to upgrade to stronger algorithms.

Architecture and Design
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.

Architecture and Design
Use a vetted library or framework that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
Industry-standard implementations will save you development time and may be more likely to avoid errors that can occur during implementation of cryptographic algorithms. Consider the ESAPI Encryption feature.

Implementation, Architecture and Design
When you use industry-approved techniques, you need to use them correctly. Don't cut corners by skipping resource-intensive steps (CWE-325). These steps are often essential for preventing common attacks.

Related CWEs

CWE-320	Key Management Errors
CWE-329	Not Using a Random IV with CBC Mode
CWE-331	Insufficient Entropy
CWE-338	Use of Cryptographically Weak PRNG

Related Attack Patterns

CAPEC-IDs: [view all]
20, 97

25 CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')

Summary

Weakness Prevalence	Medium	Consequences	Denial of service, Code execution, Data loss
Remediation Cost	Medium to High	Ease of Detection	Moderate
Attack Frequency	Sometimes	Attacker Awareness	High

Discussion

Traffic accidents occur when two vehicles attempt to use the exact same resource at almost exactly the same time, i.e., the same part of the road. Race conditions in your software aren't much different, except an attacker is consciously looking to exploit them to cause chaos or get your application to cough up something valuable. In many cases, a race condition can involve multiple processes in which the attacker has full control over one process. Even when the race condition occurs between multiple threads, the attacker may be able to influence when some of those threads execute. Your only comfort with race conditions is that data corruption and denial of service are the norm. Reliable techniques for code execution haven't been developed - yet. At least not for some kinds of race conditions. Small comfort indeed. The impact can be local or global, depending on what the race condition affects - such as state variables or security logic - and whether it occurs within multiple threads, processes, or systems.

Technical Details | *Code Examples* | *Detection Methods* | *References*

Prevention and Mitigations

Architecture and Design
In languages that support it, use synchronization primitives. Only wrap these around critical code to minimize the impact on performance.

Architecture and Design
Use thread-safe capabilities such as the data access abstraction in Spring.

Architecture and Design
Minimize the usage of shared resources in order 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).

Implementation
When using multithreading and operating on shared variables, only use thread-safe functions.

Implementation
Use atomic operations on shared variables. Be wary of innocent-looking constructs such as "x++". This may appear atomic at the code layer, but it is actually non-atomic at the instruction layer, since it involves a read, followed by a computation, followed by a write.

Implementation
Use a mutex if available, but be sure to avoid related weaknesses such as CWE-412.

Implementation
Avoid double-checked locking (CWE-609) and other implementation errors that arise when trying to avoid the overhead of synchronization.

Implementation
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.

Implementation
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.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.

Related CWEs

CWE-364	Signal Handler Race Condition
CWE-366	Race Condition within a Thread
CWE-367	Time-of-check Time-of-use (TOCTOU) Race Condition
CWE-370	Missing Check for Certificate Revocation after Initial Check
CWE-421	Race Condition During Access to Alternate Channel

Related Attack Patterns

CAPEC-IDs: [view all]
26, 29

Monster Mitigations

These mitigations will be effective in eliminating or reducing the severity of the Top 25. These mitigations will also address many weaknesses that are not even on the Top 25. If you adopt these mitigations, you are well on your way to making more secure software.

A Monster Mitigation Matrix is also available to show how these mitigations apply to weaknesses in the Top 25.

ID	Description
M1	Establish and maintain control over all of your inputs.
M2	Establish and maintain control over all of your outputs.
M3	Lock down your environment.
M4	Assume that external components can be subverted, and your code can be read by anyone.
M5	Use industry-accepted security features instead of inventing your own.
GP1	(general) Use libraries and frameworks that make it easier to avoid introducing weaknesses.
GP2	(general) Integrate security into the entire software development lifecycle.
GP3	(general) Use a broad mix of methods to comprehensively find and prevent weaknesses.
GP4	(general) Allow locked-down clients to interact with your software.

Appendix A: Selection Criteria and Supporting Fields

Entries on the 2010 Top 25 were selected using two primary criteria: weakness prevalence and importance.

Prevalence

Prevalence is effectively an average of values that were provided by voting contributors to the 2010 Top 25 list. This reflects the voter's assessment of how often the issue is encountered in their environment. For example, software vendors evaluated prevalence relative to their own software; consultants evaluated prevalence based on their experience in evaluating other people's software.

Acceptable ratings were:

Widespread	This weakness is encountered more frequently than almost all other weaknesses. Note: for selection on the general list, the "Widespread" rating could not be used more than 4 times.
High	This weakness is encountered very often, but it is not widespread.
Common	This weakness is encountered periodically.
Limited	This weakness is encountered rarely, or never.

Importance

Importance is effectively an average of values that were provided by voting contributors to the 2010 Top 25 list. This reflects the voter's assessment of how important the issue is in their environment.

Ratings for Importance were:

Critical	This weakness is more important than any other weakness, or it is one of the most important. It should be addressed as quickly as possible, and might require dedicating resources that would normally be assigned to other tasks. (Example: a buffer overflow might receive a Critical rating in unmanaged code because of the possibility of code execution.) Note: for selection on the general list, the "Critical" rating could not be used more than 4 times.
High	This weakness should be addressed as quickly as possible, but it is less important than the most critical weaknesses. (Example: in some threat models, an error message information leak may be given high importance because it can simplify many other attacks.)
Medium	This weakness should be addressed, but only after High and Critical level weaknesses have been addressed.
Low	It is not urgent to address the weakness, or it is not important at all.

Additional Fields

Each listed CWE entry also includes several additional fields, whose values are defined below.

Consequences

When this weakness occurs in software to form a vulnerability, what are the typical consequences of exploiting it?

Code execution	an attacker can execute code or commands
Data loss	an attacker can steal, modify, or corrupt sensitive data
Denial of service	an attacker can cause the software to fail or slow down, preventing legitimate users from being able to use it
Security bypass	an attacker can bypass a security protection mechanism; the consequences vary depending on what the mechanism is intended to protect

Attack Frequency

How often does this weakness occur in vulnerabilities that are targeted by a skilled, determined attacker?

Consider an "exposed host" which is either: an Internet-facing server, an Internet-using client, a multi-user system with untrusted users, or a multi-tiered system that crosses organizational or trust boundaries. Also consider that a skilled, determined attacker can combine attacks on multiple systems in order to reach a target host.

Often	an exposed host is likely to see this attack on a daily basis.
Sometimes	an exposed host is likely to see this attack more than once a month.
Rarely	an exposed host is likely to see this attack less often than once a month.

Ease of Detection

How easy is it for the skilled, determined attacker to find this weakness, whether using black-box or white-box methods, manual or automated?

Easy	automated tools or techniques exist for detecting this weakness, or it can be found quickly using simple manipulations (such as typing "<script>" into form fields to detect obvious XSS).
Moderate	only partial support using automated tools or techniques; might require some understanding of the program logic; might only exist in rare situations that might not be under direct attacker control (such as low memory conditions).

Difficult	requires time-consuming, manual methods or intelligent semi-automated support, along with attacker expertise.

Remediation Cost

How resource-intensive is it to fix this weakness when it occurs? This cannot be quantified in a general way, since each developer is different. For the purposes of this list, the cost is defined as:

Low	code change in a single block or function
Medium	code or algorithmic change, probably local to a single file or component
High	requires significant change in design or architecture, or the vulnerable behavior is required by downstream components, e.g. a design problem in a library function

This selection does not take into account other cost factors, such as procedural fixes, testing, training, patch deployment, QA, etc.

Attacker Awareness

The likelihood that a skilled, determined attacker is going to be aware of this particular weakness, methods for detection, and methods for exploitation. This assumes that the attacker knows which configuration or environment is used.

High	the attacker is capable of detecting this type of weakness and writing reliable exploits for popular platforms or configurations.
Medium	the attacker is aware of the weakness through regular monitoring of security mailing lists or databases, but has not necessarily explored it closely, and automated exploit frameworks or techniques are not necessarily available.
Low	the attacker either is not aware of the issue, does not pay close attention to it, or the weakness requires special technical expertise that the attacker does not necessarily have (but could potentially acquire).

Related CWEs

This lists some CWE entries that are related to the given entry. This includes lower-level variants, or CWEs that can occur when the given entry is also present.

The list of Related CWEs is illustrative, not complete.

Related Attack Patterns

This provides a list of attack patterns that can successfully detect or exploit the given weakness. This is provided in terms of Common Attack Pattern Enumeration and Classification (CAPEC) IDs.

Appendix B: What Changed in the 2010 Top 25

The release of the 2009 Top 25 resulted in extensive feedback from developers, product managers, security industry professionals, and others. MITRE and SANS used this feedback to make several significant improvements to the 2010 Top 25, although it retains the same spirit and goals as last year's effort.

The general nature of the 2009 list sometimes made it difficult for readers to identify narrower selections of weaknesses that were most relevant to their concerns. As a result, "focus profiles" were created to address several different use-cases.

The 2009 list did not have any ranked items, largely due to the process that was used to create it, and the lack of quantitative data regarding the frequency and severity of weaknesses in real-world code. The 2009 list also attempted to provide guidance based on a single threat model, which is not necessarily the best choice for a general-purpose list. In 2010, while more quantitative weakness data was available, it was not necessarily representative; the sample size was too small; and the associated data sets did not always have varying levels of detail. The 2010 list was built using a survey of 28 organizations, who ranked potential weaknesses based on their prevalence and importance, which provides some quantitative support to the final rankings.

With input from some Top 25 contributors, it was decided to give less emphasis to the root cause weaknesses, and more on the resultant issues. These root cause problems gave the perception of overlapping concepts, and because they were often applicable to other entries on the 2009 Top 25, they were sometimes considered to be too high-level. As a result, sometimes fundamental design problems were removed from the 2009 list, or otherwise captured elsewhere. Many of these root cause weaknesses were used as an early basis for the Monster Mitigations section.

In other cases, higher-level weaknesses were replaced with more specific, actionable ones.

There was a significant demand for improved details on mitigating each weakness. These improvements continued incrementally throughout 2009. For 2010, more consistent, detailed, and comprehensive mitigations are provided. To avoid getting too lost in the details, a summary of "Monster Mitigations" is also provided, which shows product managers and developers a way to relate general practices to the weaknesses that they address. These monster mitigations help developers to reduce or eliminate entire groups of the Top 25 weaknesses, as well as many of the other 800 weaknesses that are documented in the Common Weakness Enumeration (CWE).

Table of changes between 2009 and 2010

This table summarizes the most important changes of the Top 25 between 2009 and 2010.

2009	2010
CWE-20	high-level root cause; now covered in Monster Mitigations
CWE-116	high-level root cause; now covered in Monster Mitigations
CWE-602	high-level root cause; now covered in Monster Mitigations
CWE-250	high-level root cause; now covered in Monster Mitigations
CWE-119	high-level class; replaced with lower-level CWE-120, CWE-129, CWE-131, and CWE-805
CWE-259	Replaced with higher-level CWE-798
CWE-73	high-level root cause; now covered in Monster Mitigations
CWE-642	high-level root cause; now covered in Monster Mitigations
CWE-94	high-level; CWE name and description also caused improper interpretation of the types of issues it intended to cover.
CWE-404	high-level; replaced by children CWE-772 and CWE-672
CWE-682	high-level; replaced by children CWE-131 and CWE-190
CWE-319	replaced with its parent, CWE-311

While a number of high-level weaknesses were either replaced with lower-level entries or relocated to the mitigations section, some items remain on the list, specifically CWE-285 and

CWE-732. Lower-level children are not easily available, probably a reflection of the fact that access control is often domain-specific.

Appendix C: Construction, Selection, and Scoring of the Top 25

The 2010 version of the Top 25 list builds on the original 2009 version. Approximately 40 software security experts provided feedback, including software developers, scanning tool vendors, security consultants, government representatives, and university professors. Representation was international.

The primary means of communication was through a private discussion list, with the most activity occurring over a period of about 6 weeks. In 2009, there were multiple iterations of drafts. This year, discussion was more focused on one or two areas at a time, and drafts of smaller sections were posted for review.

Many Top 25 contributors advocated using a quantitative, data-driven approach. However, while there is more data available in 2010 than there was in 2009, it still does not have sufficient scale or precision. It is still heartening to see more raw data being generated.

The construction and development of the Top 25 occurred over the following phases. Note that these phases are approximate, since there many activities overlapped.

Preparation of the Nominee List

- Top 25 participants were asked to re-evaluate the 2009 Top 25. For each entry, they were asked whether to "Keep" or "Remove" the entry in the 2010 list.
- Top 25 participants were also given a list of the "On the Cusp" items from the 2009 Top 25, as well as additional entries that have appeared more frequently in CVE data in recent years
- Top 25 participants could suggest new entries for addition - whether from "On the Cusp" or their own nominees.
- The 2009 Top 25 was restructured to move some original entries to the mitigations section, and to provide lower-level entries instead of abstract ones. In some cases, this forced the creation of new CWE entries (See Appendix B for details).
- If there was active advocacy for any potential entry, then it was added to the Nominee List.

Selection of Factors for Evaluation

After some brief discussion, it was decided to use two factors, Prevalence and Importance. While stringent definitions were originally desired for each, more flexible definitions were created to allow for diverse roles within the software security community.

Prevalence would be evaluated according to this criterion: For each "project" (whether a software package, pen test, educational effort, etc.), how often does this weakness occur or otherwise pose a problem?

Ratings for Prevalence were:

Widespread	This weakness is encountered more frequently than almost all other weaknesses. Note: for selection on the general list, the "Widespread" rating could not be used more than 4 times.
High	This weakness is encountered very often, but it is not widespread.
Common	This weakness is encountered periodically.
Limited	This weakness is encountered rarely, or never.

For Importance, the criterion was: "If this weakness appears in software, what priority do you use when making recommendations to your consumer base? (e.g. to fix, mitigate, educate)."

Ratings for Importance were:

Critical	This weakness is more important than any other weakness, or it is one of the most important. It should be addressed as quickly as possible, and might require dedicating resources that would normally be assigned to other tasks. (Example: a buffer overflow might receive a Critical rating in unmanaged code because of the possibility of code execution.) Note: for selection on the general list, the "Critical" rating could not be used more than 4 times.
High	This weakness should be addressed as quickly as possible, but it is less important than the most critical weaknesses. (Example: in some threat models, an error message information leak may be given high importance because it can simplify many other attacks.)
Medium	This weakness should be addressed, but only after High and Critical level weaknesses have been addressed.
Low	It is not urgent to address the weakness, or it is not important at all.

Nominee List Creation and Voting

After the factors were decided and the final Nominee List was created (with a total of 41 entries), participants were given a voting ballot containing these nominees. For each nominee entry, the ballot provided space for the voter to provide a Prevalence rating, an Importance rating, and any associated comments.

- Participants were given approximately one week to evaluate the items and vote on them. This was ultimately extended for three additional days.
- Since the voting process was conducted by manually filling out a form, some inconsistencies and incomplete results occurred. This would trigger an exchange of email until the final ballot was selected. The main area of contention was the limitation of 4 ratings each for "Critical" importance and "Widespread" prevalence.
- Voting rules were refined to ensure that each organization only submitted one ballot, to avoid the possibility that a small number of organizations could bias the results too much. Generally, organizations that needed to merge multiple ballots foundthe exercise informative in clarifying their own perspectivesamongst each other.

Selection of Metrics

- During the voting period, participants were provided with several possible methods for scoring voting ballots and devising metrics. Some proposals added the two Prevalence and Importance factors together. One proposal used the squares of each factor. Other proposals used different weights or value ranges. Some proposed metrics suggested using higher values for "Widespread" prevalence and "Critical" importance, since these were artificially limited to 4 ratings for each factor per voter.
- The selected metrics were then evaluated by a skilled statistician for validity using several methods including chi-square.
- The final selection of a metric took place once the validation was complete.

Selection of Final List (General Ranking)

After the selection of the metric, and the remaining votes were tallied, the Final List was selected using the following process.

- For each weakness in the Nominee List, all associated votes were collected. Each single vote had a Prevalence and Importance rating. A sub-score was created for this vote using the selected metric. For each weakness, the sub-scores were all collected and added together.
- The Nominee List was sorted based on the aggregate scores.
- The weaknesses with the 25 highest scores were selected from the sorted list.
- The remaining weaknesses were added to the "On the Cusp" list.
- Some of the originally-proposed metrics were considered for use in additional Focus Profiles.

Challenges

- Voters wanted to select ranges of values, which reflects the diversity of opinions and contexts that occurs in the regular work for many Top 25 participants. This poses special challenges for developing a general-purpose metric.
- Some prevalence data was available. Generally this data was too high-level, but sometimes it was too low-level. Many Top 25 contributors advocated using more precise statistics, but such statistics were not readily available, in terms of depth and coverage. Most vulnerability tracking efforts work at high levels of abstraction. For example, CVE trend data can track buffer overflows, but public vulnerability reports rarely mention the specific bug that led to the overflow. Some software vendors may track weaknesses at low levels, but even if they are willing to share that information, it represents a fairly small data point. There is some hope that in future years, the appropriate type of data will become more readily available.

Appendix D: Comparison to OWASP Top Ten 2010 RC1

The OWASP Top Ten 2010 RC1, released in late 2009, is a valuable document for developers. Its focus is on web applications, and it characterizes problems in terms of risk, instead of weaknesses. It also uses different metrics for selection.

In general, the CWE/SANS 2010 Top 25 covers more weaknesses, including those that rarely appear in web applications, such as buffer overflows.

The following list identifies each Top Ten category along with its associated CWE entries.

OWASP Top Ten 2010 RC1	2010 Top 25
A1 - Injection	CWE-89 (SQL injection), CWE-78 (OS Command injection)
A2 - Cross Site Scripting (XSS)	CWE-79 (Cross-site scripting)
A3 - Broken Authentication and Session Management	CWE-306, CWE-307, CWE-798
A4 - Insecure Direct Object References	CWE-285
A5 - Cross Site Request Forgery (CSRF)	CWE-352
A6 - Security Misconfiguration	No direct mappings; CWE-209 is frequently the result of misconfiguration.
A7 - Failure to Restrict URL Access	CWE-285
A8 - Unvalidated Redirects and Forwards	CWE-601
A9 - Insecure Cryptographic Storage	CWE-327, CWE-311
A10 - Insufficient Transport Layer Protection	CWE-311

Appendix E: Other Resources for the Top 25

While this is the primary document, other supporting documents are available:

- SANS Announcement for the Top 25
- SANS Application Security Blog - commentary on individual Top 25 entries
- Supporting quotes for the Top 25
- List of contributors
- On the Cusp - list of weaknesses that almost made it
- CWE View for the 2010 Top 25
- Frequently Asked Questions (FAQ)
- Description of the process for creating the Top 25
- Change log for earlier draft versions

Top 25 Documents & Podcasts

Changes to This Document

Version	Date	Description
1.08	March 29, 2011	Updated details to be consistent with the release of CWE 1.12, primarily with mitigations and name changes.
1.07	December 13, 2010	Updated details to be consistent with the release of CWE 1.11, primarily with mitigations and some name changes.
1.06	September 27, 2010	Updated details to be consistent with the release of CWE 1.10, primarily with mitigations.
1.05	June 29, 2010	Modified official name of the list, replacing "Programming" with "Software"
1.04	June 21, 2010	Updated details to be consistent with the release of CWE 1.9, primarily with mitigations.
1.03	April 5, 2010	Updated details to be consistent with the release of CWE 1.8.1, primarily with improved mitigations and additional CAPEC mappings.
1.02	Feb 25, 2010	Fixed "Bypass security" impact in CWE-306; added pointers to SANS Application Security blog.
1.01	Feb 17, 2010	Added links to Microsoft SDL, added page numbers to PDF
1.0	Feb 16, 2010	Initial version