2009 CWE/SANS Top 25 Most Dangerous Programming Errors

http://cwe.mitre.org/top25/archive/2009/2009_cwe_sans_top25.html

Document version: 1.4
Date: October 29, 2009

Project Coordinators:

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

Document Editor:

Steve Christey (MITRE)

Introduction

The 2009 CWE/SANS Top 25 Most Dangerous Programming Errors is a list of the most significant programming errors that can lead to serious software vulnerabilities. They occur frequently, are often easy to find, and easy to exploit. They are dangerous because they will frequently allow attackers to completely take over the software, steal data, or prevent the software from working at all.

The list is the result of collaboration between the SANS Institute, MITRE, and many top software security experts in the US and Europe. It leverages experiences in the development of the SANS Top 20 attack vectors (http://www.sans.org/top20/) and MITRE's Common Weakness Enumeration (CWE) (http://cwe.mitre.org/). 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 also contains data on more than 700 additional programming errors, design errors, and architecture errors that can lead to exploitable vulnerabilities.

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

Table of Contents

Brief Listing of the Top 25

The Top 25 is organized into three high-level categories that contain multiple CWE entries.

Insecure Interaction Between Components

These weaknesses are related to insecure ways in which data is sent and received between separate components, modules, programs, processes, threads, or systems.

- CWE-20: Improper Input Validation
- CWE-116: Improper Encoding or Escaping of Output
- CWE-89: Failure to Preserve SQL Query Structure ('SQL Injection')
- CWE-79: Failure to Preserve Web Page Structure ('Cross-site Scripting')
- CWE-78: Improper Sanitization of Special Elements used in an OS Command ('OS Command Injection')
- CWE-319: Cleartext Transmission of Sensitive Information
- CWE-352: Cross-Site Request Forgery (CSRF)
- CWE-362: Race Condition
- CWE-209: Error Message Information Leak

Risky Resource Management

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.

- CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer
- CWE-642: External Control of Critical State Data
- CWE-73: External Control of File Name or Path
- CWE-426: Untrusted Search Path
- CWE-94: Failure to Control Generation of Code ('Code Injection')
- CWE-494: Download of Code Without Integrity Check
- CWE-404: Improper Resource Shutdown or Release
- CWE-665: Improper Initialization
- CWE-682: Incorrect Calculation

Porous Defenses

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

- CWE-285: Improper Access Control (Authorization)
- CWE-327: Use of a Broken or Risky Cryptographic Algorithm
- CWE-259: Hard-Coded Password
- CWE-732: Incorrect Permission Assignment for Critical Resource
- CWE-330: Use of Insufficiently Random Values
- CWE-250: Execution with Unnecessary Privileges
- CWE-602: Client-Side Enforcement of Server-Side Security

Construction and Selection of the Top 25

The Top 25 list was developed at the end of 2008. Approximately 40 software security experts provided feedback, including software developers, scanning tool vendors, security consultants, government representatives, and university professors. Representation was international. Several intermediate versions were created and resubmitted to the reviewers before the list was finalized. More details are provided in the Top 25 Process page

To help characterize and prioritize entries on the Top 25, a threat model was developed that identifies an attacker who has solid technical skills and is determined enough to invest some time into attacking an organization. More details are provided in Appendix B.

Weaknesses in the Top 25 were selected using two primary criteria:

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

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

With these criteria, future versions of the Top 25 will evolve to cover different weaknesses.

See Appendix A for more details on the selection criteria.

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.

- CWE ID and name
- Supporting data fields: supplementary information about the weakness that may be useful for decision-makers to further prioritize the entries.

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

Other Supporting Data Fields

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

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

See Appendix A for more details.

Insecure Interaction Between Components

CWE-20: Improper Input Validation

Summary

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

Discussion

It's the number one killer of healthy software, so you're just asking for trouble if you don't ensure that your input conforms with expectations. For example, an identifier that you expect to be numeric shouldn't ever contain letters. Nor should the price of a new car be allowed to be a dollar, not even in today's economy. Applications often have more complex validation requirements than these simple examples. Incorrect input validation can lead to vulnerabilities when attackers can modify their inputs in unexpected ways. Many of today's most common vulnerabilities can be eliminated, or at least reduced, using proper input validation.

...View Full Technical Details

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	Understand all the potential areas where untrusted inputs can enter your software: parameters or arguments, cookies, anything read from the network, environment variables, request headers as well as content, URL components, e-mail, files, databases, and any external systems that provide data to the application. Perform input validation at well-defined interfaces.
Architecture and Design	Assume all input is malicious. Use an "accept known good" input validation strategy (i.e., use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks.
	Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if 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.
	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.
Architecture and Design	Do not rely exclusively on blacklist validation to detect malicious input or to encode output (CWE-184). There are too many ways to encode the same character, so you're likely to miss some variants.
Implementation	When your application combines data from multiple sources, perform the validation after the sources have been combined. The individual data elements may pass the validation step but violate the intended restrictions after they have been combined.
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.
Implementation	Directly convert your input type into the expected data type, such as using a conversion function that translates a string into a number. After converting to the expected data type, ensure that the input's values fall within the expected range of allowable values and that multi-field consistencies are maintained.
Implementation	Inputs should be decoded and canonicalized to the application's current internal representation before being validated (CWE-180, CWE-181). Make sure that your application does not inadvertently decode the same input twice (CWE-174). Such errors could be used to bypass whitelist schemes by introducing dangerous inputs after they have been checked. Use libraries such as the OWASP ESAPI Canonicalization control.

	Consider performing repeated canonicalization until your input does not change any more. This will avoid double-decoding and similar scenarios, but it might inadvertently modify inputs that are allowed to contain properly-encoded dangerous content.
Implementation	When exchanging data between components, ensure that both components are using the same character encoding. Ensure that the proper encoding is applied at each interface. Explicitly set the encoding you are using whenever the protocol allows you to do so.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Related CWEs

CWE-184	Incomplete Blacklist
CWE-74	Injection
CWE-79	Cross-site Scripting (XSS)
CWE-89	SQL injection
CWE-95	Eval Injection

Related Attack Patterns

CAPEC-IDs: [view all]
3, 7, 8, 9, 10, 13, 14, 18, 22, 24, 28, 31, 32, 42, 43, 45, 46, 47, 52, 53, 63, 64, 66, 67, 71, 72, 73, 78, 79, 80, 81, 83, 85, 86, 88, 91, 99, 101, 104, 106, 108, 109, 110

CWE-116: Improper Encoding or Escaping of Output

Summary

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

Discussion

Computers have a strange habit of doing what you say, not what you mean. Insufficient output encoding is the often-ignored sibling to poor input validation, but it is at the root of most injection-based attacks, which are all the rage these days. An attacker can modify the commands that you intend to send to other components, possibly leading to a complete compromise of your application - not to mention exposing the other components to exploits that the attacker would not be able to launch directly. This turns "do what I mean" into "do what the attacker says." When

your program generates outputs to other components in the form of structured messages such as queries or requests, it needs to separate control information and metadata from the actual data. This is easy to forget, because many paradigms carry data and commands bundled together in the same stream, with only a few special characters enforcing the boundaries. An example is Web 2.0 and other frameworks that work by blurring these lines. This further exposes them to attack.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Use languages, libraries, or frameworks that make it easier to generate properly encoded output.
	Examples include the ESAPI Encoding control.
	Alternately, use built-in functions, but consider using wrappers in case those functions are discovered to have a vulnerability.
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.
	For example, stored procedures can enforce database query structure and reduce the likelihood of SQL injection.
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.
Architecture and Design	In some cases, input validation may be an important strategy when output encoding is not a complete solution. For example, you may be providing the same output that will be processed by multiple consumers that use different encodings or representations. In other cases, you may be required to allow user-supplied input to contain control information, such as limited HTML tags that support formatting in a wiki or bulletin board. When this type of requirement must be met, use an extremely strict whitelist to limit which control sequences can be used. Verify that the resulting syntactic structure is what you expect. Use your normal encoding methods for the remainder of the input.
Architecture and Design	Use input validation as a defense-in-depth measure to reduce the likelihood of output encoding errors (see CWE-20).
Requirements	Fully specify which encodings are required by components that will be communicating with each other.
Implementation	When exchanging data between components, ensure that both components are using the same character encoding. Ensure that the proper encoding is applied at each interface. Explicitly set the encoding you are using whenever the protocol allows you to do so.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Related CWEs

CWE-74	Injection
CWE-78	OS command injection
CWE-79	Cross-site Scripting (XSS)
CWE-88	Argument Injection
CWE-89	SQL injection
CWE-93	CRLF Injection

Related Attack Patterns

CAPEC-IDs: [view all]
18, 63, 73, 81, 85, 86, 104

CWE-89: Improper Sanitization 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 they can do nasty things where they get all the fun and profit. 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.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Use languages, libraries, or frameworks that make it easier to generate properly encoded output.
	For example, consider using persistence layers such as Hibernate or Enterprise Java Beans, which can provide significant protection against SQL injection if used properly.
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	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 in spite of the risk, use proper encoding and escaping of inputs. 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). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks.
	Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if 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.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Operation	Use an application firewall that can detect attacks against this weakness. This might not catch all attacks, and it might require some effort for customization. However, it can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.

Related CWEs

CWE-564	SQL Injection: Hibernate
CWE-566	Access Control Bypass Through User-Controlled SQL Primary Key
CWE-619	Cursor Injection
CWE-90	LDAP Injection

Related Attack Patterns

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

CWE-79: Failure to Preserve Web Page Structure ('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.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Use languages, libraries, or frameworks that make it easier to generate properly encoded output.

	Examples include Microsoft's Anti-XSS library, the OWASP ESAPI Encoding module, and Apache Wicket.
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	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. This encoding will vary depending on whether the output is part of the HTML body, element attributes, URIs, JavaScript sections, Cascading Style Sheets, etc. Note that HTML Entity Encoding is only appropriate for the HTML body.
Implementation	Use and specify a strong character encoding such as ISO-8859-1 or UTF-8. When an encoding is not specified, the web browser may choose a different encoding by guessing which encoding is actually being used by the web page. This can open you up to subtle XSS attacks related to that encoding. 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.
Implementation	Assume all input is malicious. Use an "accept known good" input validation strategy (i.e., use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks.
	Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if 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.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Use the XSS Cheat Sheet (see references) to launch a wide variety of attacks against your web application. The Cheat Sheet contains many subtle XSS variations that are specifically targeted against weak XSS defenses.
Operation	Use an application firewall that can detect attacks against this weakness. This might not catch all attacks, and it might require some effort for customization. However, it can be beneficial in cases in which the code cannot be fixed (because it is controlled by a third party), as an emergency prevention measure while more comprehensive software assurance measures are applied, or to provide defense in depth.

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
19, 32, 85, 86, 91

CWE-78: Improper Sanitization 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.

...View Full Technical Details

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	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 commands can be executed by your software.
	Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection.
	This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.
	Be careful to avoid CWE-243 and other weaknesses related to jails.
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 command locally in the session's state instead of sending it out to the client in a hidden form field.
Architecture and Design	Use languages, libraries, or frameworks that make it easier to generate properly encoded output.
	Examples include the ESAPI Encoding control.
Implementation	Properly quote arguments and escape any special characters within those arguments. If some special characters are still needed, wrap the arguments in quotes, and escape all other characters that do not pass a strict whitelist. Be careful of argument injection (CWE-88).
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.
Implementation	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). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks.
	Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the input for further processing. As an example of business rule logic, "boat" may be syntactically valid because it only contains alphanumeric characters, but it is not valid if 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.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
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).
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.
System Configuration	Assign permissions to the software system that prevent the user from accessing/ opening privileged files. Run the application with the lowest privileges possible (CWE-250).

Related CWEs

CWE-88	Argument Injection

Related Attack Patterns

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

CWE-319: Cleartext Transmission of Sensitive Information

Summary

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

Discussion

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. Trying to obfuscate traffic using schemes like Base64 and URL encoding doesn't offer any protection, either; those encodings are for normalizing communications, not scrambling data to make it unreadable.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Encrypt the data with a reliable encryption scheme before transmitting.
Implementation	When using web applications with SSL, use SSL for the entire session from login to logout, not just for the initial login page.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.

	Attach the monitor to the process, trigger the feature that sends the data, and look for the presence or absence of common cryptographic functions in the call tree. Monitor the network and determine if the data packets contain readable commands. Tools exist for detecting if certain encodings are in use. If the traffic contains high entropy, this might indicate the usage of encryption.
Operation	Configure servers to use encrypted channels for communication, which may include SSL or other secure protocols.

Related CWEs

CWE-312	Plaintext Storage of Sensitive Information
CWE-614	Sensitive Cookie in HTTPS Session Without "Secure" Attribute

Related Attack Patterns

CAPEC-IDs: [view all]
65, 102

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.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Use anti-CSRF packages such as the OWASP CSRFGuard.

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).
	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.
	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.
	Note that this can probably be bypassed using XSS (CWE-79).
Architecture and Design	Use the ESAPI Session Management control.
	This control includes a component for CSRF.
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.
	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.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
	Use OWASP CSRFTester to identify potential issues.

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
62, 111

CWE-362: 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.

...View Full Technical Details

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 multi-threading, only use thread-safe functions on shared variables.
Implementation	Use atomic operations on shared variables. Be wary of innocent-looking constructs like "x++". This is actually non-atomic, since it involves a read followed by a write.
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.
Testing	Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
	Insert breakpoints or delays in between relevant code statements to artificially expand the race window so that it will be easier to detect.

Testing	Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If you trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.

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	Race Condition in Checking for Certificate Revocation
CWE-421	Race Condition During Access to Alternate Channel

Related Attack Patterns

CAPEC-IDs: [view all]
26, 29

CWE-209: Error Message Information Leak

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.

...View Full Technical Details

Prevention and Mitigations

Implementation	Ensure that error messages only contain minimal information that are useful to their intended audience, and nobody else. The messages need to strike the balance between being too cryptic and not being cryptic enough. They should not necessarily reveal the methods that were used to determine the error. Such detailed information can help an attacker craft another attack that now will pass through the validation filters.
	If errors must be tracked in some detail, capture them in log messages - but consider what could occur if the log messages can be viewed by attackers. Avoid recording highly sensitive information such as passwords in any form. Avoid inconsistent messaging that might accidentally tip off an attacker about internal state, such as whether a username is valid or not.
Implementation	Handle exceptions internally and do not display errors containing potentially sensitive information to a user.
Build and Compilation	Debugging information should not make its way into a production release.
Testing	Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If you trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.
Testing	Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
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 Leak
CWE-210	Product-Generated Error Message Information Leak
CWE-538	File and Directory Information Leaks

Related Attack Patterns

CAPEC-IDs: [view all]
7, 54

Risky Resource Management

CWE-119: Failure to Constrain Operations within the Bounds of a Memory Buffer

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

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. One reason is that they aren't just about using strcpy() incorrectly, or improperly checking the length of your inputs. Attack and detection techniques continue to improve, and today's buffer overflow variants aren't always obvious at first or even second glance. You may think that you're completely immune to buffer overflows because you write your code in higher-level languages instead of C. But what is your favorite "safe" language's interpreter written in? What about the native code you call? What languages are the operating system API's written in? How about the software that runs Internet infrastructure? Thought so.

...View Full Technical Details

Prevention and Mitigations

Requirements	Use a language with features that can automatically mitigate or eliminate buffer overflows.
	For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer.
	Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe.
Architecture and Design	Use languages, libraries, or frameworks that make it easier to manage buffers without exceeding their boundaries.
	Examples include the Safe C String Library (SafeStr) by Messier and Viega, and the Strsafe.h library from Microsoft. These libraries provide safer versions of overflow-prone string-handling functions. This is not a complete solution, since many buffer overflows are not related to strings.
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.
	This is not necessarily a complete solution, since these mechanisms can only detect certain types of overflows. In addition, a buffer overflow attack can still cause a denial of service, since the typical response is to exit the application.
Implementation	Programmers should adhere to the following rules when allocating and managing their 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 calling this function 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.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Operation	Use a feature like Address Space Layout Randomization (ASLR). This is not a complete solution. However, it forces the attacker to guess an unknown value that changes every program execution.
Operation	Use a CPU and operating system that offers Data Execution Protection (NX) or its equivalent. This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required.

Related CWEs

CWE-120	Classic Buffer Overflow
CWE-129	Unchecked Array Indexing
CWE-130	Failure to Handle Length Parameter Inconsistency
CWE-131	Incorrect Calculation of Buffer Size
CWE-415	Double Free
CWE-416	Use After Free

Related Attack Patterns

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

CWE-642: External Control of Critical State Data

Summary

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

Discussion

There are many ways to store user state data without the overhead of a database. Unfortunately, if you store that data in a place where an attacker can modify it, this

also reduces the overhead for a successful compromise. For example, the data could be stored in configuration files, profiles, cookies, hidden form fields, environment variables, registry keys, or other locations, all of which can be modified by an attacker. In stateless protocols such as HTTP, some form of user state information must be captured in each request, so it is exposed to an attacker out of necessity. If you perform any security-critical operations based on this data (such as stating that the user is an administrator), then you can bet that somebody will modify the data in order to trick your application into doing something you didn't intend.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Understand all the potential locations that are accessible to attackers. For example, some programmers assume that cookies and hidden form fields cannot be modified by an attacker, or they may not consider that environment variables can be modified before a privileged program is invoked.
Architecture and Design	Do not keep state information on the client without using encryption and integrity checking, or otherwise having a mechanism on the server side to catch state tampering. Use a message authentication code (MAC) algorithm, such as Hash Message Authentication Code (HMAC). Apply this against the state data that 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	Store state information on the server side only. Ensure that the system definitively and unambiguously keeps track of its own state and user state and has rules defined for legitimate state transitions. Do not allow any application user to affect state directly in any way other than through legitimate actions leading to state transitions.
Architecture and Design	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	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.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Related CWEs

CWE-472	External Control of Assumed-Immutable Web Parameter
CWE-565	Use of Cookies in Security Decision

Related Attack Patterns

CAPEC-IDs: [view all]
21, 31, 167

CWE-73: External Control of File Name or Path

Summary

Weakness Prevalence	High	Consequences	Code execution Data loss
Remediation Cost	Medium	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. Other file-related attacks are simplified by external control of a filename, such as symbolic link following, which causes your application to read or modify files that the attacker can't access directly. The same applies if your program is running with raised privileges and it accepts filenames as input. And if you allow an outsider to specify an arbitrary URL from which you'll download code and execute it, you're just asking for worms.

...View Full Technical Details

Prevention and Mitigations

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

	Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection.
	This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.
	Be careful to avoid CWE-243 and other weaknesses related to jails.
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	Assume all input is malicious. Use an "accept known good" input validation strategy (i.e., use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks.
	For filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a whitelist of allowable file extensions, which will help to avoid CWE-434.
Implementation	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).
Installation	Use OS-level permissions and run as a low-privileged user to limit the scope of any successful attack.
Operation	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.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Related CWEs

CWE-22	Path Traversal
CWE-434	Unrestricted File Upload
CWE-59	Link Following
CWE-98	Remote File Inclusion

Related Attack Patterns

CAPEC-IDs: [view all]
13, 64, 72, 76, 78, 79, 80

CWE-426: Untrusted Search Path

Summary

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

Discussion

Your software depends on you, or its environment, to provide a search path so that it knows where it can find critical resources such as code libraries or configuration files. If the search path is under attacker control, then the attacker can modify it to point to resources of the attacker's choosing. This causes the software to access the wrong resource at the wrong time. The same risk exists if a single search path element could be under attacker control, such as the current working directory.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Hard-code your search path to a set of known-safe values, or allow them to be specified by the administrator in a configuration file. Do not allow these settings to be modified by an external party. Be careful to avoid related weaknesses such as CWE-427 and CWE-428.
Implementation	When invoking other programs, specify those programs using fully-qualified pathnames.
Implementation	Sanitize your environment before invoking other programs. This includes the PATH environment variable, LD_LIBRARY_PATH and other settings that identify the location of code libraries, and any application-specific search paths.
Implementation	Check your search path before use and remove any elements that are likely to be unsafe, such as the current working directory or a temporary files directory.
Implementation	Use other functions that require explicit paths. Making use of any of the other readily available functions that require explicit paths is a safe way to avoid this problem. For example, system() in C does not require a full path since the shell can take care of it, while execl() and execv() require a full path.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.
	Attach the monitor to the process and look for library functions and system calls that suggest when a search path is being used. One pattern is when the program performs multiple accesses of the same file but in different directories, with repeated failures until the proper filename is found. Library calls such as getenv() or their equivalent can be checked to see if any path-related variables are being accessed.

Related CWEs

CWE-427	Uncontrolled Search Path Element
CWE-428	Unquoted Search Path or Element

Related Attack Patterns

CAPEC-IDs: [view all]
38

CWE-94: Failure to Control Generation of Code ('Code Injection')

Summary

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

Discussion

For ease of development, sometimes you can't beat using a couple lines of code to employ lots of functionality. It's even cooler when you manage the code dynamically. While it's tough to deny the sexiness of dynamically-generated code, attackers find it equally appealing. It becomes a serious vulnerability when your code is directly callable by unauthorized parties, if external inputs can affect which code gets executed, or (horror of horrors) if those inputs are fed directly into the code itself. The implications are obvious: all your code are belong to them.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Refactor your program so that you do not have to dynamically generate code.
Architecture and Design	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 code can be executed by your software.

	Examples include the Unix chroot jail and AppArmor. In general, managed code may provide some protection.
	This may not be a feasible solution, and it only limits the impact to the operating system; the rest of your application may still be subject to compromise.
	Be careful to avoid CWE-243 and other weaknesses related to jails.
Implementation	Assume all input is malicious. Use an "accept known good" input validation strategy (i.e., use a whitelist). Reject any input that does not strictly conform to specifications, or transform it into something that does. Use a blacklist to reject any unexpected inputs and detect potential attacks.
	To reduce the likelihood of code injection, use stringent whitelists that limit which constructs are allowed. If you are dynamically constructing code that invokes a function, then verifying that the input is alphanumeric might be insufficient. An attacker might still be able to reference a dangerous function that you did not intend to allow, such as system(), exec(), or exit().
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
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).

Related CWEs

CWE-470	Unsafe Reflection
CWE-95	Eval Injection
CWE-96	Static Code Injection
CWE-98	Remote File Inclusion

Related Attack Patterns

CAPEC-IDs: [view all]
35, 77

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.

...View Full Technical Details

Prevention and Mitigations

Implementation	Perform proper forward and reverse DNS lookups to detect DNS spoofing. 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	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 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.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.
	Attach the monitor to the process and also sniff the network connection. Trigger features related to product updates or plugin installation, which is likely to force a code download. Monitor when files are downloaded and separately executed, or if they are otherwise read back into the process. Look for evidence of cryptographic library calls that use integrity checking.

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

CWE-404: Improper Resource Shutdown or Release

Summary

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

Discussion

When your precious system resources have reached their end-of-life, you need to dispose of them correctly. Otherwise, your environment will become heavily congested or contaminated. This applies to memory, files, cookies, data structures, sessions, communication pipes, and so on. Attackers can exploit improper shutdown to maintain control over those resources well after you thought you got rid of them. This can lead to significant resource consumption because nothing actually gets released back to the system. If you don't wash your garbage before you dispose of it, attackers may sift through it, looking for gems in the form of sensitive data that should have been wiped. They could also reuse the resources, which may seem like the right thing to do in a "Green" world, except in the virtual world, those resources may still have significant value.

...View Full Technical Details

Prevention and Mitigations

Requirements	Use a language with features that can automatically mitigate or eliminate resource-shutdown weaknesses.
	For example, languages such as Java, Ruby, and Lisp perform automatic garbage collection that releases memory for objects that have been deallocated.
Implementation	It is good practice to be responsible for freeing all resources you allocate and to be consistent with how and where you free memory in a function. If you allocate memory that you intend to free upon completion of the function, you must be sure to free the memory at all exit points for that function including error conditions.
Implementation	Memory should be allocated/freed using matching functions such as malloc/free, new/delete, and new[]/delete[].
Implementation	When releasing a complex object or structure, ensure that you properly dispose of all of its member components, not just the object itself.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Testing	Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If you trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.

Related CWEs

CWE-14	Compiler Removal of Code to Clear Buffers
CWE-226	Sensitive Information Uncleared Before Release
CWE-262	Not Using Password Aging
CWE-299	Failure to Check for Certificate Revocation
CWE-401	Memory Leak
CWE-415	Double Free
CWE-416	Use After Free
CWE-568	finalize() Method Without super.finalize()
CWE-590	Free of Invalid Pointer Not on the Heap

Related Attack Patterns

CAPEC-IDs: [view all]
118, 119, 125, 130, 131

CWE-665: Improper Initialization

Summary

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

Discussion

Just as you should start your day with a healthy breakfast, proper initialization helps to ensure that your software will run without fainting in the middle of an important event. If you don't properly initialize your data and variables, an attacker might be able to do the initialization for you, or extract sensitive information that remains from previous sessions. When those variables are used in security-critical operations, such as making an authentication decision, then they could be modified to bypass your security. Incorrect initialization can occur anywhere, but it is probably most prevalent in rarely-encountered conditions that cause your code to inadvertently skip

initialization, such as obscure errors.

...View Full Technical Details

Prevention and Mitigations

Requirements	Use a language with features that can automatically mitigate or eliminate weaknesses related to initialization.
	For example, in Java, if the programmer does not explicitly initialize a variable, then the code could produce a compile-time error (if the variable is local) or automatically initialize the variable to the default value for the variable's type. In Perl, if explicit initialization is not performed, then a default value of undef is assigned, which is interpreted as 0, false, or an equivalent value depending on the context in which the variable is accessed.
Architecture and Design	Identify all variables and data stores that receive information from external sources, and apply input validation to make sure that they are only initialized to expected values.
Implementation	Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage.
Implementation	Pay close attention to complex conditionals that affect initialization, since some conditions might not perform the initialization.
Implementation	Avoid race conditions (CWE-362) during initialization routines.
Build and Compilation	Run or compile your software with settings that generate warnings about uninitialized variables or data.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Stress-test the software by calling it simultaneously from a large number of threads or processes, and look for evidence of any unexpected behavior. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Identify error conditions that are not likely to occur during normal usage and trigger them. For example, run the program under low memory conditions, run with insufficient privileges or permissions, interrupt a transaction before it is completed, or disable connectivity to basic network services such as DNS. Monitor the software for any unexpected behavior. If you trigger an unhandled exception or similar error that was discovered and handled by the application's environment, it may still indicate unexpected conditions that were not handled by the application itself.

Related CWEs

CWE-453	Insecure Default Variable Initialization
CWE-454	External Initialization of Trusted Variables
CWE-456	Missing Initialization

Related Attack Patterns

CAPEC-IDs: [view all]
26, 29, 172

CWE-682: Incorrect Calculation

Summary

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

Discussion

Computers can perform calculations whose results don't seem to make mathematical sense. For example, if you are multiplying two large, positive numbers, the result might be a much smaller number due to an integer overflow. In other cases, the calculation might be impossible for the program to perform, such as a divide-by-zero. When attackers have some control over the inputs that are used in numeric calculations, this weakness can actually have security consequences. It could cause you to make incorrect security decisions. It might cause you to allocate far more resources than you intended - or maybe far fewer, as in the case of integer overflows that trigger buffer overflows due to insufficient memory allocation. It could violate business logic, such as a calculation that produces a negative price. Finally, denial of service is also possible, such as a divide-by-zero that triggers a program crash.

...View Full Technical Details

Prevention and Mitigations

Implementation	Understand your programming language's underlying representation and how it interacts with numeric calculation. Pay close attention to byte size discrepancies, precision, signed/ unsigned distinctions, truncation, conversion and casting between types, "not-a-number" calculations, and how your language handles numbers that are too large or too small for its underlying representation.
Implementation	Perform input validation on any numeric inputs by ensuring that they are within the expected range.
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.
Implementation	Use languages, libraries, or frameworks that make it easier to handle numbers without unexpected consequences.
	Examples include safe integer handling packages such as SafeInt (C++) or IntegerLib (C or C++).
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Related CWEs

CWE-131	Incorrect Calculation of Buffer Size
CWE-135	Incorrect Calculation of Multi-Byte String Length
CWE-190	Integer Overflow or Wraparound
CWE-193	Off-by-one Error
CWE-369	Divide By Zero
CWE-467	Use of sizeof() on a Pointer Type
CWE-681	Incorrect Conversion between Numeric Types

Related Attack Patterns

CAPEC-IDs: [view all]
124, 128, 129

Porous Defenses

CWE-285: Improper Access Control (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.

...View Full Technical Details

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 may be different than the access control checks that you apply to the resources that support your business logic.
Architecture and Design	Use 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.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
System Configuration	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-749	Insecure Exposed Methods

Related Attack Patterns

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

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.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	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.
Architecture and Design	Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations.
	For example, US government systems require FIPS 140-2 certification.
	As with all cryptographic mechanisms, the source code should be available for analysis.
	Periodically ensure that 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 which 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 languages, libraries, or frameworks that make it easier to use strong cryptography.
	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	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.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

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]
97

CWE-259: Hard-Coded Password

Summary

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

Discussion

Hard-coding a secret account and password into your software's authentication module is 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.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	For outbound authentication: store passwords outside of the code in a strongly-protected, encrypted configuration file or database that is protected from access by all outsiders, including other local users on the same system. Properly protect the key (CWE-320). If you cannot use encryption to protect the file, then make sure that the permissions are as restrictive as possible.
Architecture and Design	For inbound authentication: Rather than hard-code a default username and password for first time logins, utilize a "first login" mode that requires the user to enter a unique strong password.
Architecture and Design	Perform access control checks and limit which entities can access the feature that requires the hard-coded password. For example, a feature might only be enabled through the system console instead of through a network connection.
Architecture and Design	For inbound authentication: 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 which are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals.

	Next, the passwords used should be limited at the back end to only performing actions valid for the front end, as opposed to having full access.
	Finally, the messages sent should be tagged and checksummed with time sensitive values so as to prevent replay style attacks.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.
	Attach the monitor to the process and perform a login. Using disassembled code, look at the associated instructions and see if any of them appear to be comparing the input to a fixed string or value.

Related CWEs

CWE-256	Plaintext Storage of a Password
CWE-257	Storing Passwords in a Recoverable Format
CWE-260	Password in Configuration File
CWE-321	Use of Hard-coded Cryptographic Key

Related Attack Patterns

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

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.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	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.
Implementation	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.
System Configuration	For all configuration files, executables, and libraries, make sure that they are only readable and writable by the software's administrator.
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.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.
	Attach the monitor to the process and watch for library functions or system calls on OS resources such as files, directories, and shared memory. Examine the arguments to these calls to infer which permissions are being used.
	Note that this technique is only useful for permissions issues related to system resources. It is not likely to detect application-level business rules that are related to permissions, such as if a user of a blog system marks a post as "private," but the blog system inadvertently marks it as "public."
Testing	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	Insecure Default Permissions
CWE-277	Insecure Inherited Permissions

CWE-279	Insecure Execution-assigned Permissions

Related Attack Patterns

CAPEC-IDs: [view all]
60, 61, 62

CWE-330: Use of Insufficiently Random Values

Summary

Weakness Prevalence	Medium	Consequences	Security bypass Data loss
Remediation Cost	Medium	Ease of Detection	Easy to Difficult
Attack Frequency	Rarely	Attacker Awareness	Medium

Discussion

Imagine how quickly a Las Vegas casino would go out of business if gamblers could predict the next roll of the dice, spin of the wheel, or turn of the card. If you use security features that require good randomness, but you don't provide it, then you'll have attackers laughing all the way to the bank. You may depend on randomness without even knowing it, such as when generating session IDs or temporary filenames. Pseudo-Random Number Generators (PRNG) are commonly used, but a variety of things can go wrong. Once an attacker can determine which algorithm is being used, he or she can guess the next random number often enough to launch a successful attack after a relatively small number of tries. After all, if you were in Vegas and you figured out that a game with 1000-to-1 odds could be knocked down to 10-1 odds after you paid close attention for a couple games, wouldn't it be worthwhile to keep playing until you hit the jackpot?

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Use a well-vetted algorithm that is currently considered to be strong by experts in the field, and select well-tested implementations with adequate length seeds.
	In general, if a pseudo-random number generator is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts.
	Pseudo-random number generators can produce predictable numbers if the generator is known and the seed can be guessed. A 256-bit seed is a good starting point for producing a "random enough" number.
Implementation	Consider a PRNG that re-seeds itself as needed from high quality pseudo-random output sources, such as hardware devices.
Testing	Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.

Testing	Perform FIPS 140-2 tests on data to catch obvious entropy problems.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.
	Attach the monitor to the process and look for library functions that indicate when randomness is being used. Run the process multiple times to see if the seed changes. Look for accesses of devices or equivalent resources that are commonly used for strong (or weak) randomness, such as /dev/urandom on Linux. Look for library or system calls that access predictable information such as process IDs and system time.

Related CWEs

CWE-329	Not Using a Random IV with CBC Mode
CWE-331	Insufficient Entropy
CWE-334	Small Space of Random Values
CWE-336	Same Seed in PRNG
CWE-337	Predictable Seed in PRNG
CWE-338	Use of Cryptographically Weak PRNG
CWE-341	Predictable from Observable State

Related Attack Patterns

CAPEC-IDs: [view all]
59, 112

CWE-250: Execution with Unnecessary Privileges

Summary

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

Discussion

Spider Man, the well-known comic superhero, lives by the motto "With great power comes great responsibility." Your software may need special privileges to perform

certain operations, but wielding those privileges longer than necessary can be extremely risky. When running with extra privileges, your application has access to resources that the application's user can't directly reach. For example, you might intentionally launch a separate program, and that program allows its user to specify a file to open; this feature is frequently present in help utilities or editors. The user can access unauthorized files through the launched program, thanks to those extra privileges. Command execution can happen in a similar fashion. Even if you don't launch other programs, additional vulnerabilities in your software could have more serious consequences than if it were running at a lower privilege level.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	Identify the functionality that requires additional privileges, such as access to privileged operating system resources. Wrap and centralize this functionality if possible, and isolate the privileged code as much as possible from other code. Raise your privileges as late as possible, and drop them as soon as possible. Avoid weaknesses such as CWE-288 and CWE-420 by protecting all possible communication channels that could interact with your privileged code, such as a secondary socket that you only intend to be accessed by administrators.
Implementation	Perform extensive input validation for any privileged code that must be exposed to the user and reject anything that does not fit your strict requirements.
Implementation	Ensure that you drop privileges as soon as possible (CWE-271), and make sure that you check to ensure that privileges have been dropped successfully (CWE-273).
Implementation	If circumstances force you to run with extra privileges, then determine the minimum access level necessary. First identify the different permissions that the software and its users will need to perform their actions, such as file read and write permissions, network socket permissions, and so forth. Then explicitly allow those actions while denying all else. Perform extensive input validation and canonicalization to minimize the chances of introducing a separate vulnerability. This mitigation is much more prone to error than dropping the privileges in the first place.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.
Testing	Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic.
	Attach the monitor to the process and perform a login. Look for library functions and system calls that indicate when privileges are being raised or dropped. Look for accesses of resources that are restricted to normal users.
	Note that this technique is only useful for privilege issues related to system resources. It is not likely to detect application-level business rules that are related to privileges, such as if a blog system allows a user to delete a blog entry without first checking that the user has administrator privileges.
Testing	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-272	Least Privilege Violation
CWE-273	Failure to Check Whether Privileges Were Dropped Successfully
CWE-653	Insufficient Compartmentalization

Related Attack Patterns

CAPEC-IDs: [view all]
69, 104

CWE-602: Client-Side Enforcement of Server-Side Security

Summary

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

Discussion

Rich clients can make attackers richer, and customers poorer, if you trust the clients to perform security checks on behalf of your server. Remember that underneath that fancy GUI, it's just code. Attackers can reverse engineer your client and write their own custom clients that leave out certain inconvenient features like all those pesky security controls. The consequences will vary depending on what your security checks are protecting, but some of the more common targets are authentication, authorization, and input validation. If you've implemented security in your servers, then you need to make sure that you're not solely relying on the clients to enforce it.

...View Full Technical Details

Prevention and Mitigations

Architecture and Design	For any security checks that are performed on the client side, ensure that these checks are duplicated on the server side. Attackers can bypass the client-side checks by modifying values after the checks have been performed, or by changing the client to remove the client-side checks entirely. Then, these modified values would be submitted to the server.
	Even though client-side checks provide minimal benefits with respect to server-side security, they are still useful. First, they can support intrusion detection. If the server receives input that should have been rejected by the client, then it may be an indication of an attack. Second, client-side error-checking can provide helpful feedback to the user about the expectations for valid input. Third, there may be a reduction in server-side processing time for accidental input errors, although this is typically a small savings.

Architecture and Design	If some degree of trust is required between the two entities, then use integrity checking and strong authentication to ensure that the inputs are coming from a trusted source. Design the product so that this trust is managed in a centralized fashion, especially if there are complex or numerous communication channels, in order to reduce the risks that the implementer will mistakenly omit a check in a single code path.
Testing	Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.
Testing	Use tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Related CWEs

CWE-20	Insufficient Input Validation
CWE-642	External Control of Critical State Data

Related Attack Patterns

CAPEC-IDs: [view all]
21, 31, 122, 162, 202, 207, 208

Appendix A: Selection Criteria and Supporting Fields

Entries on the CWE Top 25 were selected using two primary criteria: weakness prevalence and severity. The severity is reflected in the common consequences of the weakness.

Weakness Prevalence

How often this weakness appears in software that was not developed with security integrated into the software development life cycle (SDLC). The prevalence only applies to applications that are potentially susceptible. For example, the prevalence of SQL injection is only evaluated with respect to applications that use a database.

The prevalence value is determined based on estimates from multiple contributors to the Top 25 list, including CVE vulnerability trend data. Top 25 contributors advocated using more precise statistics, but such statistics are 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 they may be reluctant to share such information.

- High: the weakness is likely to occur at least once in over 50% of potentially affected software.
- Medium: the weakness is likely to occur at least once in 10% to 50% of potentially affected software.

- Low: the weakness is likely to occur in less than 10% of potentially affected software.

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 the 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).
- 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 consumers, e.g. a design problem in a library function

This selection does not take into account other cost factors, such as procedural fixes, 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

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.

Appendix B: Threat Model for the Skilled, Determined Attacker

Selection for the CWE Top 25 assumes that the user wants to make it difficult and time-consuming for a skilled, determined attacker to break into the software.

Though many kinds of attackers exist, it is assumed that the attacker has most of the following characteristics.

Skill:

- Has a solid technical understanding of well-documented vulnerabilities;
- Can detect and exploit those vulnerabilities with some success, using black box and white box methods;
- Can learn new vulnerabilities and attack techniques without significant effort; and
- Can combine attacks on multiple systems to gain deeper access into the targeted organization.

Determination:

- Seeks to steal confidential data or take over an entire software package for its computing capability, independent of the motive (financial, military, political, or other);
- Is not necessarily part of a large or well-funded group, but may collaborate with a small number of other individuals; and
- Is willing to invest at least 20 hours to attack a single software package.

Informally, the attacker's skills and determination exceed that of a "script kiddie" but are less than that of a nation-state or criminal organization.

Note that the model does not consider denial of service to be a primary motivation for the attacker. This is contrary to the model that may be followed in some areas, such as critical infrastructure protection and e-commerce, in which system downtime may have catastrophic consequences.

Also note that this model was developed late in the review period for the Top 25, so it did not influence the selection of the Top 25 items significantly. However, it is included here to give some context for how the values for other supporting fields were derived. Authors of future "top N" lists should consider making their threat model more explicit, which can ensure that the prioritization is appropriate for the desired environments.

Appendix C: Other Resources for the Top 25

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

- SANS Announcement for the Top 25
- Supporting quotes for the Top 25
- List of contributors
- On the Cusp - list of weaknesses that almost made it
- CWE View for the 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