2011 CWE/SANS Top 25 Most Dangerous Software Errors

http://cwe.mitre.org/top25/

Document version: 1.0.3

Date: September 13, 2011

Project Coordinators:

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

Document Editor:

Steve Christey (MITRE)

Introduction

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

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

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

The 2011 Top 25 makes improvements to the 2010 list, but the spirit and goals remain the same. This year's Top 25 entries are prioritized using inputs from over 20 different organizations, who evaluated each weakness based on prevalence, importance, and likelihood of exploit. It uses the Common Weakness Scoring System (CWSS) to score and rank the final results. The Top 25 list covers a small set of the most effective "Monster Mitigations," which help developers to reduce or eliminate entire groups of the Top 25 weaknesses, as well as many of the hundreds of weaknesses that are documented by CWE.

Table of Contents

Guidance for Using the Top 25

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

User	Activity
Programmers new to security	Read the brief listing, then examine the Monster Mitigations section to see how a small number of changes in your practices can have a big impact on the Top 25. Pick a small number of weaknesses to work with first, and see the Detailed CWE Descriptions for more information on the weakness, which includes code examples and specific mitigations.
Programmers who are experienced in security	Use the general Top 25 as a checklist of reminders, and note the issues that have only recently become more common. Consult the See the On the Cusp page for other weaknesses that did not make the final Top 25; this includes weaknesses that are only starting to grow in prevalence or importance. If you are already familiar with a particular weakness, then consult the Detailed CWE Descriptions and see the "Related CWEs" links for variants that you may not have fully considered. Build your own Monster Mitigations section so that you have a clear understanding of which of your own mitigation practices are the most effective - and where your gaps may lie. Consider building a custom "Top n" list that fits your needs and practices. Consult the Common Weakness Risk Analysis Framework (CWRAF) page for a general framework for building top-N lists, and see Appendix C for a description of how it was done for this year's Top 25. Develop your own nominee list of weaknesses, with your own prevalence and importance factors - and other factors that you may wish - then build a metric and compare the results with your colleagues, which may produce some fruitful discussions.
Software project managers	Treat the Top 25 as an early step in a larger effort towards achieving software security. Strategic possibilities are covered in efforts such as Building Security In Maturity Model (BSIMM), SAFECode, OpenSAMM, Microsoft SDL, and OWASP ASVS. Examine the Monster Mitigations section to determine which approaches may be most suitable to adopt, or establish your own monster mitigations and map out which of the Top 25 are addressed by them. Consider building a custom "Top n" list that fits your needs and practices. Consult the Common Weakness Risk Analysis Framework (CWRAF) page for a general framework for building top-N lists, and see Appendix C for a description of how it was done for this year's Top 25. Develop your own nominee list of weaknesses, with your own prevalence and importance factors - and other factors that you may wish - then build a metric and compare the results with your colleagues, which may produce some fruitful discussions.

Software Testers	Read the brief listing and consider how you would integrate knowledge of these weaknesses into your tests. If you are in a friendly competition with the developers, you may find some surprises in the On the Cusp entries, or even the rest of CWE. For each indvidual CWE entry in the Details section, you can get more information on detection methods from the "technical details" link. Review the CAPEC IDs for ideas on the types of attacks that can be launched against the weakness.
Software customers	Recognize that market pressures often drive vendors to provide software that is rich in features, and security may not be a serious consideration. As a customer, you have the power to influence vendors to provide more secure products by letting them know that security is important to you. Use the Top 25 to help set minimum expectations for due care by software vendors. Consider using the Top 25 as part of contract language during the software acquisition process. The SANS Application Security Procurement Language site offers customer-centric language that is derived from the OWASP Secure Software Contract Annex, which offers a "framework for discussing expectations and negotiating responsibilities" between the customer and the vendor. Other information is available from the DHS Acquisition and Outsourcing Working Group. Consult the Common Weakness Risk Analysis Framework (CWRAF) page for a general framework for building a top-N list that suits your own needs. For the software products that you use, pay close attention to publicly reported vulnerabilities in those products. See if they reflect any of the associated weaknesses on the Top 25 (or your own custom list), and if so, contact your vendor to determine what processes the vendor is undertaking to minimize the risk that these weaknesses will continue to be introduced into the code. See the On the Cusp summary for other weaknesses that did not make the final Top 25; this will include weaknesses that are only starting to grow in prevalence or importance, so they may become your problem in the future.
Educators	Start with the brief listing. Some training materials are also available.
Users of the 2010 Top 25	See the What Changed section; while a lot has changed on the surface, this year's effort is more well-structured.

Brief Listing of the Top 25

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

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

Rank	Score	ID	Name
[1]	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[3]	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[4]	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[5]	76.9	CWE-	Missing Authentication for Critical Function

		306	
[6]	76.8	CWE-862	Missing Authorization
[7]	75.0	CWE-798	Use of Hard-coded Credentials
[8]	75.0	CWE-311	Missing Encryption of Sensitive Data
[9]	74.0	CWE-434	Unrestricted Upload of File with Dangerous Type
[10]	73.8	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	73.1	CWE-250	Execution with Unnecessary Privileges
[12]	70.1	CWE-352	Cross-Site Request Forgery (CSRF)
[13]	69.3	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	68.5	CWE-494	Download of Code Without Integrity Check
[15]	67.8	CWE-863	Incorrect Authorization
[16]	66.0	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[17]	65.5	CWE-732	Incorrect Permission Assignment for Critical Resource
[18]	64.6	CWE-676	Use of Potentially Dangerous Function
[19]	64.1	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[20]	62.4	CWE-131	Incorrect Calculation of Buffer Size
[21]	61.5	CWE-307	Improper Restriction of Excessive Authentication Attempts
[22]	61.1	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')
[23]	61.0	CWE-134	Uncontrolled Format String
[24]	60.3	CWE-190	Integer Overflow or Wraparound
[25]	59.9	CWE-759	Use of a One-Way Hash without a Salt

CWE-89 - SQL injection - delivers the knockout punch of security weaknesses in 2011. For data-rich software applications, SQL injection is the means to steal the keys to the kingdom. CWE-78, OS command injection, is where the application interacts with the operating system. The classic buffer overflow (CWE-120) comes in third, still pernicious after all these decades. Cross-site scripting (CWE-79) is the bane of web applications everywhere. Rounding out the top 5 is Missing Authentication (CWE-306) for critical functionality.

Category-Based View of the Top 25

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

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

Insecure Interaction Between Components

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

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

Rank	CWE ID	Name
[1]	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
[2]	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
[4]	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
[9]	CWE-434	Unrestricted Upload of File with Dangerous Type
[12]	CWE-352	Cross-Site Request Forgery (CSRF)
[22]	CWE-601	URL Redirection to Untrusted Site ('Open Redirect')

Risky Resource Management

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

Rank	CWE ID	Name
[3]	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
[13]	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')
[14]	CWE-494	Download of Code Without Integrity Check
[16]	CWE-829	Inclusion of Functionality from Untrusted Control Sphere
[18]	CWE-676	Use of Potentially Dangerous Function
[20]	CWE-131	Incorrect Calculation of Buffer Size
[23]	CWE-134	Uncontrolled Format String
[24]	CWE-190	Integer Overflow or Wraparound

Porous Defenses

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

Rank	CWE ID	Name
[5]	CWE-306	Missing Authentication for Critical Function
[6]	CWE-862	Missing Authorization
[7]	CWE-798	Use of Hard-coded Credentials
[8]	CWE-311	Missing Encryption of Sensitive Data

[10]	CWE-807	Reliance on Untrusted Inputs in a Security Decision
[11]	CWE-250	Execution with Unnecessary Privileges
[15]	CWE-863	Incorrect Authorization
[17]	CWE-732	Incorrect Permission Assignment for Critical Resource
[19]	CWE-327	Use of a Broken or Risky Cryptographic Algorithm
[21]	CWE-307	Improper Restriction of Excessive Authentication Attempts
[25]	CWE-759	Use of a One-Way Hash without a Salt

Organization of the Top 25

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

Ranking	The ranking of the weakness in the general list.
Score Summary	A summary of the individual ratings and scores that were given to this weakness, including Prevalence, Importance, and Adjusted Score.
CWE ID and name	CWE identifier and short name of the weakness
Supporting Information	Supplementary information about the weakness that may be useful for decision-makers to further prioritize the entries.
Discussion	Short, informal discussion of the nature of the weakness and its consequences. The discussion avoids digging too deeply into technical detail.
Prevention 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.
General Parent	One or more pointers to more general CWE entries, so you can see the breadth and depth of the problem.
Related Attack Patterns	CAPEC entries for attacks that may be successfully conducted against the weakness. Note: the list is not necessarily complete.
Other pointers	Links to more details including source code examples that demonstrate the weakness, methods for detection, etc.

Supporting Information

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

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

See Appendix A for more details.

Detailed CWE Descriptions

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

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

Summary

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

Discussion

These days, it seems as if software is all about the data: getting it into the database, pulling it from the database, massaging it into information, and sending it elsewhere for fun and profit. If attackers can influence the SQL that you use to communicate with your database, then suddenly all your fun and profit belongs to them. If you use SQL queries in security controls such as authentication, attackers could alter the logic of those queries to bypass security. They could modify the queries to steal, corrupt, or otherwise change your underlying data. They'll even steal data one byte at a time if they have to, and they have the patience and know-how to do so. In 2011, SQL injection was responsible for the compromises of many high-profile organizations, including Sony Pictures, PBS, MySQL.com, security company HBGary Federal, and many others.

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

Prevention and Mitigations

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

Architecture and Design
If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.
Process SQL queries using prepared statements, parameterized queries, or stored procedures. These features should accept parameters or variables and support strong typing. Do not dynamically construct and execute query strings within these features using "exec" or similar functionality, since you may re-introduce the possibility of SQL injection.

Architecture and Design, Operation
Run your code using the lowest privileges that are required to accomplish the necessary tasks. If possible, create isolated accounts with limited privileges that are only used for a single task. That way, a successful attack will not immediately give the attacker access to the rest of the software or its environment. For example, database applications rarely need to run as the database administrator, especially in day-to-day operations.
Specifically, follow the principle of least privilege when creating user accounts to a SQL database. The database users should only have the minimum privileges necessary to use their account. If the requirements of the system indicate that a user can read and modify their own data, then limit their privileges so they cannot read/write others' data. Use the strictest permissions possible on all database objects, such as execute-only for stored procedures.

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

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

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

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

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

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

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

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

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

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

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

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

Related CWEs

CWE-90	Improper Neutralization of Special Elements used in an LDAP Query ('LDAP Injection')
CWE-564	SQL Injection: Hibernate
CWE-566	Authorization Bypass Through User-Controlled SQL Primary Key
CWE-619	Dangling Database Cursor ('Cursor Injection')

Related Attack Patterns

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

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

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

Effectiveness: Limited

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

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

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

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

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

Implementation
If the program to be executed allows arguments to be specified within an input file or from standard input, then consider using that mode to pass arguments instead of the command line.

Architecture and Design
If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.
Some languages offer multiple functions that can be used to invoke commands. Where possible, identify any function that invokes a command shell using a single string, and replace it with a function that requires individual arguments. These functions typically perform appropriate quoting and filtering of arguments. For example, in C, the system() function accepts a string that contains the entire command to be executed, whereas execl(), execve(), and others require an array of strings, one for each argument. In Windows, CreateProcess() only accepts one command at a time. In Perl, if system() is provided with an array of arguments, then it will quote each of the arguments.

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

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

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

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

Architecture and Design

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

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

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

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

Operation
Use runtime policy enforcement to create a whitelist of allowable commands, then prevent use of any command that does not appear in the whitelist. Technologies such as AppArmor are available to do this.

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

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

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

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

Related CWEs

CWE-88	Argument Injection or Modification

Related Attack Patterns

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

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

Summary

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

Discussion

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

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

Prevention and Mitigations

Requirements
Use a language that does not allow this weakness to occur or provides constructs that make this weakness easier to avoid.
For example, many languages that perform their own memory management, such as Java and Perl, are not subject to buffer overflows. Other languages, such as Ada and C#, typically provide overflow protection, but the protection can be disabled by the programmer.

Be wary that a language's interface to native code may still be subject to overflows, even if the language itself is theoretically safe.

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

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

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

Effectiveness: Defense in Depth

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

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

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

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

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

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

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

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

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

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

Notes: This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

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

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

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

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

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

Architecture and Design, Operation

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

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

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

Effectiveness: Limited

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

Related CWEs

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

Related Attack Patterns

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

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

Summary

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

Discussion

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

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

Prevention and Mitigations

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

Implementation, Architecture and Design
Understand the context in which your data will be used and the encoding that will be expected. This is especially important when transmitting data between different components, or when generating outputs that can contain multiple encodings at the same time, such as web pages or multi-part mail messages. Study all expected communication protocols and data representations to determine the required encoding strategies.
For any data that will be output to another web page, especially any data that was received from external inputs, use the appropriate encoding on all non-alphanumeric characters.

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

HTML body

Element attributes (such as src="XYZ")

URIs

JavaScript sections

Cascading Style Sheets and style property

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

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

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

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

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

Architecture and Design
If available, use structured mechanisms that automatically enforce the separation between data and code. These mechanisms may be able to provide the relevant quoting, encoding, and validation automatically, instead of relying on the developer to provide this capability at every point where output is generated.

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

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

Implementation
To help mitigate XSS attacks against the user's session cookie, set the session cookie to be HttpOnly. In browsers that support the HttpOnly feature (such as more recent versions of Internet Explorer and Firefox), this attribute can prevent the user's session cookie from being accessible to malicious client-side scripts that use document.cookie. This is not a complete solution, since HttpOnly is not supported by all browsers. More importantly, XMLHTTPRequest and other powerful browser technologies provide read access to HTTP headers, including the Set-Cookie header in which the HttpOnly flag is set.
Effectiveness: Defense in Depth

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

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

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

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

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

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

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

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

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

Related CWEs

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

Related Attack Patterns

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

5 CWE-306: Missing Authentication for Critical Function

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

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

Related CWEs

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

Related Attack Patterns

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

6 CWE-862: Missing 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. In May 2011, Citigroup revealed that it had been compromised by hackers who were able to steal details of hundreds of thousands of bank accounts by changing the account information that was present in fields in the URL; authorization would check that the user had the rights to access the account being specified. Earlier, a similar missing-authorization attack was used to steal private information of iPad owners from an AT&T site.

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

Prevention and Mitigations

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

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

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

Architecture and Design
For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page. Users should not be able to access any unauthorized functionality or information by simply requesting direct access to that page.
One way to do this is to ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.

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

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
1, 17, 58, 122, 180

7 CWE-798: Use of Hard-coded Credentials

Summary

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

Discussion

Hard-coding a secret password or cryptograpic key into your program is bad manners, even though it makes it extremely convenient - for skilled reverse engineers. While it might shrink your testing and support budgets, it can reduce the security of your customers to dust. If the password is the same across all your software, then every customer becomes vulnerable if (rather, when) your password becomes known. Because it's hard-coded, it's usually a huge pain for sysadmins to fix. And you know how much they love inconvenience at 2 AM when their network's being hacked - about as much as you'll love responding to hordes of angry customers and reams of bad press if your little secret should get out. Most of the CWE Top 25 can be explained away as an honest mistake; for this issue, though, many customers won't see it that way. The high-profile Stuxnet worm, which caused operational problems in an Iranian nuclear site, used hard-coded credentials in order to spread. Another way that hard-coded credentials arise is through unencrypted or obfuscated storage in a configuration file, registry key, or other location that is only intended to be accessible to an administrator. While this is much more polite than burying it in a binary program where it can't be modified, it becomes a Bad Idea to expose this file to outsiders through lax permissions or other means.

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

Prevention and Mitigations

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

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

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

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

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

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

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

Related CWEs

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

Related Attack Patterns

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

8 CWE-311: Missing Encryption of Sensitive Data

Summary

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

Discussion

Whenever sensitive data is being stored or transmitted anywhere outside of your control, attackers may be looking for ways to get to it. Thieves could be anywhere - sniffing your packets, reading your databases, and sifting through your file systems. If your software sends sensitive information across a network, such as private data or authentication credentials, that information crosses many different nodes in transit to its final destination. Attackers can sniff this data right off the wire, and it doesn't require a lot of effort. All they need to do is control one node along the path to the final destination, control any node within the same networks of those transit nodes, or plug into an available interface. If your software stores sensitive information on a local file or database, there may be other ways for attackers to get at the file. They may benefit from lax permissions, exploitation of another vulnerability, or physical theft of the disk. You know those massive credit card thefts you keep hearing about? Many of them are due to unencrypted storage. In 2011, many breaches of customer emails and passwords made the attacker's job easier by storing critical information without any encryption. Once the attacker got access to the database, it was game over. In June 2011, the LulzSec group grabbed headlines by grabbing and publishing unencrypted data.

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

Prevention and Mitigations

Requirements
Clearly specify which data or resources are valuable enough that they should be protected by encryption. Require that any transmission or storage of this data/resource should use well-vetted encryption algorithms.

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

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

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

Identify the separate needs and contexts for encryption:

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

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

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

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

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

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

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

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

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

Related CWEs

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

Related Attack Patterns

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

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

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

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

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

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

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

provided.

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

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

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

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

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

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

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

Effectiveness: Limited

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

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
1, 122

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

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

Examples include ASP.NET View State and the OWASP ESAPI Session Management feature.

Be careful of language features that provide state support, since these might be provided as a convenience to the programmer and may not be considering security.

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

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

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

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
232

11 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.

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

Prevention and Mitigations

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

Architecture and Design
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 to avoid CWE-271. Avoid weaknesses such as CWE-288 and CWE-420 by protecting all possible communication channels that could interact with your privileged code, such as a secondary socket that you only intend to be accessed by administrators.

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
When you drop privileges, ensure that you have dropped them successfully to avoid CWE-273. As protection mechanisms in the environment get stronger, privilege-dropping calls may fail even if it seems like they would always succeed.

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.

Operation, System Configuration
Ensure that your software runs properly under the Federal Desktop Core Configuration (FDCC) or an equivalent hardening configuration guide, which many organizations use to limit the attack surface and potential risk of deployed software.

Related CWEs

CWE-272	Least Privilege Violation
CWE-273	Improper Check for Dropped Privileges
CWE-653	Insufficient Compartmentalization

Related Attack Patterns

CAPEC-IDs: [view all]
69, 104

12 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 (like Facebook), there's usually CSRF feeding them.

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

Prevention and Mitigations

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

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

Implementation

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

Architecture and Design
Generate a unique nonce for each form, place the nonce into the form, and verify the nonce upon receipt of the form. Be sure that the nonce is not predictable (CWE-330).
Notes: Note that this can be bypassed using XSS (CWE-79).

Architecture and Design
Identify especially dangerous operations. When the user performs a dangerous operation, send a separate confirmation request to ensure that the user intended to perform that operation.
Notes: Note that this can be bypassed using XSS (CWE-79).

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

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

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

Implementation
Check the HTTP Referer header to see if the request originated from an expected page. This could break legitimate functionality, because users or proxies may have disabled sending the Referer for privacy reasons.
Notes: Note that this can be bypassed using XSS (CWE-79). An attacker could use XSS to generate a spoofed Referer, or to generate a malicious request from a page whose Referer would be allowed.

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
62, 111

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

Summary

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

Discussion

While data is often exchanged using files, sometimes you don't intend to expose every file on your system while doing so. When you use an outsider's input while constructing a filename, the resulting path could point outside of the intended directory. An attacker could combine multiple ".." or similar sequences to cause the operating system to navigate out of the restricted directory, and into the rest of the system.

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

Prevention and Mitigations

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

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

When validating filenames, use stringent whitelists that limit the character set to be used. If feasible, only allow a single "." character in the filename to

avoid weaknesses such as CWE-23, and exclude directory separators such as "/" to avoid CWE-36. Use a whitelist of allowable file extensions, which will help to avoid CWE-434.

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

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

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

realpath() in C

getCanonicalPath() in Java

GetFullPath() in ASP.NET

realpath() or abs_path() in Perl

realpath() in PHP

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

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

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

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

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

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

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

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

Effectiveness: Limited

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

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

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

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

Operation, Implementation
If you are using PHP, configure your application so that it does not use register_globals. During implementation, develop your application so that it does

not rely on this feature, but be wary of implementing a register_globals emulation that is subject to weaknesses such as CWE-95, CWE-621, and similar issues.

Related CWEs

None.

Related Attack Patterns

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

14 CWE-494: Download of Code Without Integrity Check

Summary

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

Discussion

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

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

Prevention and Mitigations

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

Architecture and Design, Operation
Encrypt the code with a reliable encryption scheme before transmitting.
This will only be a partial solution, since it will not detect DNS spoofing and it will not prevent your code from being modified on the hosting site.

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

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

Use code signing technologies such as Authenticode. See references.

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

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

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

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

Effectiveness: Limited

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

Related CWEs

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

Related Attack Patterns

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

15 CWE-863: Incorrect Authorization

Summary

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

Discussion

While the lack of authorization is more dangerous (see elsewhere in the Top 25), incorrect authorization can be just as problematic. Developers may attempt to control access to certain resources, but implement it in a way that can be bypassed. For example, once a person has logged in to a web application, the developer may store the permissions in a cookie. By modifying the cookie, the attacker can access other resources. Alternately, the developer might perform authorization by delivering code that gets executed in the web client, but an attacker could use a customized client that removes the check entirely.

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

Prevention and Mitigations

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

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

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

Architecture and Design
For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page. Users should not be able to access any unauthorized functionality or information by simply requesting direct access to that page.
One way to do this is to ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.

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

Related CWEs

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

CWE-749	Exposed Dangerous Method or Function

Related Attack Patterns

CAPEC-IDs: [view all]
1, 17, 58, 122, 180

16 CWE-829: Inclusion of Functionality from Untrusted Control Sphere

Summary

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

Discussion

The idea seems simple enough (not to mention cool enough): you can make a lot of smaller parts of a document (or program), then combine them all together into one big document (or program) by "including" or "requiring" those smaller pieces. This is a common enough way to build programs. Combine this with the common tendency to allow attackers to influence the location of some of these pieces - perhaps even from the attacker's own server - then suddenly you're importing somebody else's code. In these Web 2.0 days, maybe it's just "the way the Web works," but not if security is a consideration.

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

Prevention and Mitigations

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

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

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

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

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

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

Effectiveness: Limited

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

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

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

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

to avoid CWE-434.

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

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

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

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

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
35, 38, 101, 103, 111, 175, 181, 184, 185, 186, 187, 193, 222, 251, 252, 253

17 CWE-732: Incorrect Permission Assignment for Critical Resource

Summary

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

Discussion

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

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

Prevention and Mitigations

Implementation
When using a critical resource such as a configuration file, check to see if the resource has insecure permissions (such as being modifiable by any regular user), and generate an error or even exit the software if there is a possibility that the resource could have been modified by an unauthorized party.

Architecture and Design
Divide your application into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully defining distinct user

groups, privileges, and/or roles. Map these against data, functionality, and the related resources. Then set the permissions accordingly. This will allow you to maintain more fine-grained control over your resources.
Effectiveness: Moderate

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

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

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

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

Effectiveness: Moderate

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

Implementation, Installation
During program startup, explicitly set the default permissions or umask to the most restrictive setting possible. Also set the appropriate permissions during program installation. This will prevent you from inheriting insecure permissions from any user who installs or runs the program.
Effectiveness: High

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

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

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

Operation, System Configuration
Ensure that your software runs properly under the Federal Desktop Core Configuration (FDCC) or an equivalent hardening configuration guide, which many organizations use to limit the attack surface and potential risk of deployed software.

Related CWEs

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

Related Attack Patterns

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

18 CWE-676: Use of Potentially Dangerous Function

Summary

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

Discussion

Safety is critical when handling power tools. The programmer's toolbox is chock full of power tools, including library or API functions that make assumptions about how they will be used, with no guarantees of safety if they are abused. If potentially-dangerous functions are not used properly, then things can get real messy real quick.

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

Prevention and Mitigations

Build and Compilation, Implementation
Identify a list of prohibited API functions and prohibit developers from using these functions, providing safer alternatives. In some cases, automatic code analysis tools or the compiler can be instructed to spot use of prohibited functions, such as the "banned.h" include file from Microsoft's SDL.

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]

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

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

Architecture and Design
Carefully manage and protect cryptographic keys (see CWE-320). If the keys can be guessed or stolen, then the strength of the cryptography itself is irrelevant.

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

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

steps are often essential for preventing common attacks.

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
20, 97

20 CWE-131: Incorrect Calculation of Buffer Size

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Effectiveness: Defense in Depth

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

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

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

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

Notes: This is not a complete solution, since buffer overflows could be used to overwrite nearby variables to modify the software's state in dangerous ways. In addition, it cannot be used in cases in which self-modifying code is required. Finally, an attack could still cause a denial of service, since the typical response is to exit the application.

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

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

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

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

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

Effectiveness: Limited

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

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
47, 100

21 CWE-307: Improper Restriction of Excessive Authentication Attempts

Summary

Weakness Prevalence		Consequences	

Remediation Cost		Ease of Detection	
Attack Frequency		Attacker Awareness	

Discussion

An often-used phrase is "If at first you don't succeed, try, try again." Attackers may try to break into your account by writing programs that repeatedly guess different passwords. Without some kind of protection against brute force techniques, the attack will eventually succeed. You don't have to be advanced to be persistent.

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

Prevention and Mitigations

Architecture and Design
Common protection mechanisms include:
Disconnecting the user after a small number of failed attempts

Implementing a timeout

Locking out a targeted account

Requiring a computational task on the user's part.

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

Related CWEs

CWE-302	Authentication Bypass by Assumed-Immutable Data
CWE-306	Missing Authentication for Critical Function

Related Attack Patterns

CAPEC-IDs: [view all]
16, 49, 55, 70, 112

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

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

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

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

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

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
194

23 CWE-134: Uncontrolled Format String

Summary

Weakness Prevalence		Consequences	
Remediation Cost		Ease of Detection	
Attack Frequency		Attacker Awareness	

Discussion

The mantra is that successful relationships depend on communicating clearly, and this applies to software, too. Format strings are often used to send or receive well-formed data. By controlling a format string, the attacker can control the input or output in unexpected ways - sometimes, even, to execute code.

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

Prevention and Mitigations

Requirements
Choose a language that is not subject to this flaw.

Implementation

Ensure that all format string functions are passed a static string which cannot be controlled by the user and that the proper number of arguments are always sent to that function as well. If at all possible, use functions that do not support the %n operator in format strings.

Build: Heed the warnings of compilers and linkers, since they may alert you to improper usage.

Related CWEs

None.

Related Attack Patterns

CAPEC-IDs: [view all]
67

24 CWE-190: Integer Overflow or Wraparound

Summary

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

Discussion

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

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

Prevention and Mitigations

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

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

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

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

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

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

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

Implementation
Examine compiler warnings closely and eliminate problems with potential security implications, such as signed / unsigned mismatch in memory

operations, or use of uninitialized variables. Even if the weakness is rarely exploitable, a single failure may lead to the compromise of the entire system.

Related CWEs

CWE-191	Integer Underflow (Wrap or Wraparound)

Related Attack Patterns

CAPEC-IDs: [view all]
92

25 CWE-759: Use of a One-Way Hash without a Salt

Summary

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

Discussion

Salt might not be good for your diet, but it can be good for your password security. Instead of storing passwords in plain text, a common practice is to apply a one-way hash, which effectively randomizes the output and can make it more difficult if (or when?) attackers gain access to your password database. If you don't add a little salt to your hash, then the health of your application is in danger.

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

Prevention and Mitigations

Architecture and Design
Generate a random salt each time you process a new password. Add the salt to the plaintext password before hashing it. When you store the hash, also store the salt. Do not use the same salt for every password that you process (CWE-760).

Architecture and Design
Use one-way hashing techniques that allow you to configure a large number of rounds, such as bcrypt. This may increase the expense when processing incoming authentication requests, but if the hashed passwords are ever stolen, it significantly increases the effort for conducting a brute force attack, including rainbow tables. With the ability to configure the number of rounds, you can increase the number of rounds whenever CPU speeds or attack techniques become more efficient.

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

Related CWEs

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

Related Attack Patterns

CAPEC-IDs: [view all]
16, 20, 49, 55, 97

Monster Mitigations

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

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

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

See the Monster Mitigation Matrix that maps these mitigations to Top 25 weaknesses.

Appendix A: Selection Criteria and Supporting Fields

Entries on the 2011 Top 25 were selected using three primary criteria: weakness prevalence, importance, and likelihood of exploit.

Prevalence

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

Acceptable ratings were:

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

Importance

Importance is effectively an average of values that were provided by voting contributors to the 2011 Top 25 list. This reflects the voter's assessment of how important the issue is in their environment.

Ratings for Importance were:

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

	attacks.)
Medium	This weakness should be addressed, but only after High and Critical level weaknesses have been addressed.
Low	It is not urgent to address the weakness, or it is not important at all.

Additional Fields

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

Consequences

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

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

Attack Frequency

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

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

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

Ease of Detection

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

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

Remediation Cost

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

Low	code change in a single block or function

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

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

Attacker Awareness

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

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

Related CWEs

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

The list of Related CWEs is illustrative, not complete.

Related Attack Patterns

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

Appendix B: What Changed in the 2011 Top 25

The release of the 2009 and 2010 Top 25 efforts resulted in extensive feedback from developers, product managers, security industry professionals, and others. MITRE and SANS used this feedback to make several significant improvements to the 2011 Top 25, although it retains the same spirit and goals as last year's effort.

The 2011 version followed a similar process as 2010 for nominating potential entries and collecting votes, except this year, CWSS 0.8 was used, and voters had to evaluate likelihood of exploit in addition to the prevalence and importance factors that were used in 2010. More details are in Appendix C.

Similar to the process in 2010, people were asked to nominate potential weaknesses for this year's list. A list of 41 nominees was drawn up. During the voting phase, votes evaluated each weakness based on its prevalence, importance, and likelihood of exploit. A customization of the Common Weakness Scoring System (CWSS) was used to convert each vote into a CWSS score for the weakness. The scores for each weakness were averaged together in order to determine the final rankings. (Note that more detailed data on the scoring distribution is forthcoming.)

Some entries from the 2010 list were replaced with entries that were at a different level of abstraction. The entries on the 2011 Top 25 have a more consistent level of abstraction than the previous lists.

Changes between 2010 and 2011

This table summarizes the most important changes of the Top 25 between 2010 and 2011.

2010	2011
CWE-285	Replaced with CWE-862 and CWE-863, which are more specific.
CWE-98	Replaced with CWE-829, which is more general (CWE-98 only applies to PHP applications.)
New Entries	New entries onto the Top 25 this year (excluding CWE-829, CWE-862, and CWE-863, as mentioned previously): CWE-250, CWE-676, CWE-134, and CWE-759.

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

The 2011 version of the Top 25 list was generated using a process similar to that of 2010. Respondents from e-mail requests, and participants from previous years, were asked to nominate potential weaknesses for this year's list. A list of 41 nominees was drawn up from these nominees (coincidentally the same number as in 2010.) During the voting phase, voters were surveyed to evaluate each weakness based on its prevalence, importance, and likelihood of exploit. Unlike the 2010 voting, there were no restrictions on how many "Critical" or "Widespread" votes could be assigned.

There were 28 voters, representing software developers, scanning tool vendors, security consultants, government representatives, and university professors. Representation was international.

Then, CWSS 0.8 was used to evaluate each voter's assessment of a nominee, filling in the appropriate weights for prevalence, importance, and likelihood of exploit; the remaining 15 factors were all assigned ""Not Applicable" values, which reduces the impact of those factors on the final score. Due to how the CWSS formula is constructed, the use of "Not Applicable" values required a one-step normalization of a raw score to produce a final score that fell within the range of 0 and 100. In the CWSS 0.8 formula, with the three active factors, the final score is most affected by importance, then prevalence, then likelihood of exploit.

For each nominated entry, all of its scores were collected and averaged together to produce the final rankings. (Note that more detailed data on the scoring distribution is forthcoming.)

Appendix D: Comparison to OWASP Top Ten 2010

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

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

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

OWASP Top Ten 2010	2011 Top 25
A1 - Injection	CWE-89, CWE-78
A2 - Cross Site Scripting (XSS)	CWE-79
A3 - Broken Authentication and Session Management	CWE-306, CWE-307, CWE-798
A4 - Insecure Direct	

Object References	CWE-862, CWE-863, CWE-22, CWE-434, CWE-829
A5 - Cross Site Request Forgery (CSRF)	CWE-352
A6 - Security Misconfiguration	CWE-250, CWE-732
A7 - Insecure Cryptographic Storage	CWE-327, CWE-311, CWE-759
A8 - Failure to Restrict URL Access	CWE-862, CWE-863
A9 - Insufficient Transport Layer Protection	CWE-311
A10 - Unvalidated Redirects and Forwards	CWE-601
(not in 2010 OWASP Top Ten)	The following CWE entries are not directly covered by the OWASP Top Ten 2010: CWE-120, CWE-134, CWE-807, CWE-676, CWE-131, CWE-190.

Appendix E: Other Resources for the Top 25

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

- Frequently Asked Questions (FAQ)
- List of contributors
- On the Cusp - list of weaknesses that almost made it
- CWE View for the 2011 Top 25
- Change log for earlier draft versions
- Top 25 Documents & Podcasts

Changes to This Document

Version	Date	Description
1.0.3	September 13, 2011	Updated OWASP Top Ten mapping from RC1 to official 2010 version. Updated content to match new CWE version 2.1.
1.0.2	June 29, 2011	Updated OWASP Top Ten mapping. Fixed broken link.
1.0.1	June 27, 2011	Updated guidance, appendices, monster mitigations. Fixed some typos. Updated monster mitigations.
1.0	June 27, 2011	Initial version