CWE-732: Incorrect Permission Assignment for Critical Resource

Weakness ID: 732
Vulnerability Mapping: ALLOWED (with careful review of mapping notes)
Abstraction: Class

View customized information: Conceptual | Operational | Mapping Friendly | Complete | Custom

Description

The product specifies permissions for a security-critical resource in a way that allows that resource to be read or modified by unintended actors.

Extended Description

When a resource is given a permission setting that provides access to a wider range of actors than required, it could lead to the exposure of sensitive information, or the modification of that resource by unintended parties. This is especially dangerous when the resource is related to program configuration, execution, or sensitive user data. For example, consider a misconfigured storage account for the cloud that can be read or written by a public or anonymous user.

Common Consequences

Scope	Impact	Likelihood
Confidentiality	**Technical Impact:** *Read Application Data; Read Files or Directories* An attacker may be able to read sensitive information from the associated resource, such as credentials or configuration information stored in a file.	
Access Control	**Technical Impact:** *Gain Privileges or Assume Identity* An attacker may be able to modify critical properties of the associated resource to gain privileges, such as replacing a world-writable executable with a Trojan horse.	
Integrity Other	**Technical Impact:** *Modify Application Data; Other* An attacker may be able to destroy or corrupt critical data in the associated resource, such as deletion of records from a database.	

Potential Mitigations

Phase: Implementation

When using a critical resource such as a configuration file, check to see if the resource has insecure permissions (such as being modifiable by any regular user) [REF-62], and generate an error or even exit the software if there is a possibility that the resource could have been modified by an unauthorized party.

Phase: Architecture and Design

Divide the software into anonymous, normal, privileged, and administrative areas. Reduce the attack surface by carefully defining distinct user groups, privileges, and/or roles. Map these against data, functionality, and the related resources. Then set the permissions accordingly. This will allow you to maintain more fine-grained control over your resources. [REF-207]

Effectiveness: Moderate

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

Phases: Architecture and Design; 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

Note: 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.

Phases: Implementation; Installation

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

Effectiveness: High

Phase: System Configuration

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

Effectiveness: High

Phase: Documentation

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

Phase: Installation

Do not assume that a system administrator will manually change the configuration to the settings that are recommended in the software's manual.

Phases: Operation; System Configuration

Strategy: Environment Hardening

Ensure that the software runs properly under the United States Government Configuration Baseline (USGCB) [REF-199] or an equivalent hardening configuration guide, which many organizations use to limit the attack surface and potential risk of deployed software.

Phases: Implementation; System Configuration; 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.

Relationships

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

Nature	Type	ID	Name
ChildOf	C	668	Exposure of Resource to Wrong Sphere
ChildOf	C	285	Improper Authorization
ParentOf	B	276	Incorrect Default Permissions
ParentOf	V	277	Insecure Inherited Permissions
ParentOf	V	278	Insecure Preserved Inherited Permissions
ParentOf	V	279	Incorrect Execution-Assigned Permissions
ParentOf	B	281	Improper Preservation of Permissions
ParentOf	B	766	Critical Data Element Declared Public
ParentOf	V	1004	Sensitive Cookie Without 'HttpOnly' Flag

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

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

Modes Of Introduction

Phase	Note
Architecture and Design	
Implementation	REALIZATION: This weakness is caused during implementation of an architectural security tactic. The developer might make certain assumptions about the environment in which the product operates - e.g., that the software is running on a single-user system, or the software is only accessible to trusted administrators. When the software is running in a different environment, the permissions become a problem.
Installation	The developer may set loose permissions in order to minimize problems when the user first runs the program, then create documentation stating that permissions should be tightened. Since system administrators and users do not always read the documentation, this can result in insecure permissions being left unchanged.
Operation	

Applicable Platforms

Languages

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

Technologies

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

Class: Cloud Computing *(Often Prevalent)*

Likelihood Of Exploit

High

Demonstrative Examples

Example 1

The following code sets the umask of the process to 0 before creating a file and writing "Hello world" into the file.

Example Language: **C** *(bad code)*

```c
#define OUTFILE "hello.out"

umask(0);
FILE *out;
/* Ignore link following (CWE-59) for brevity */

out = fopen(OUTFILE, "w");
if (out) {
  fprintf(out, "hello world!\n");
  fclose(out);
}
```

After running this program on a UNIX system, running the "ls -l" command might return the following output:

(result)

```
-rw-rw-rw- 1 username 13 Nov 24 17:58 hello.out
```

The "rw-rw-rw-" string indicates that the owner, group, and world (all users) can read the file and write to it.

Example 2

This code creates a home directory for a new user, and makes that user the owner of the directory. If the new directory cannot be owned by the user, the directory is deleted.

Example Language: **PHP** *(bad code)*

```php
function createUserDir($username){
  $path = '/home/'.$username;
  if(!mkdir($path)){
    return false;
  }
  if(!chown($path,$username)){
    rmdir($path);
    return false;
  }
  return true;
}
```

Because the optional "mode" argument is omitted from the call to mkdir(), the directory is created with the default permissions 0777. Simply setting the new user as the owner of the directory does not explicitly change the permissions of the directory, leaving it with the default. This default allows any user to read and write to the directory, allowing an attack on the user's files. The code also fails to change the owner group of the directory, which may result in access by unexpected groups.

This code may also be vulnerable to Path Traversal (CWE-22) attacks if an attacker supplies a non alphanumeric username.

Example 3

The following code snippet might be used as a monitor to periodically record whether a web site is alive. To ensure that the file can always be modified, the code uses chmod() to make the file world-writable.

Example Language: **Perl** *(bad code)*

```perl
$fileName = "secretFile.out";

if (-e $fileName) {
  chmod 0777, $fileName;
}

my $outFH;
if (! open($outFH, ">>$fileName")) {
  ExitError("Couldn't append to $fileName: $!");
}
my $dateString = FormatCurrentTime();
my $status = IsHostAlive("cwe.mitre.org");
print $outFH "$dateString cwe status: $status!\n";
close($outFH);
```

The first time the program runs, it might create a new file that inherits the permissions from its environment. A file listing might look like:

(result)

```
-rw-r--r-- 1 username 13 Nov 24 17:58 secretFile.out
```

This listing might occur when the user has a default umask of 022, which is a common setting. Depending on the nature of the file, the user might not have intended to make it readable by everyone on the system.

The next time the program runs, however - and all subsequent executions - the chmod will set the file's permissions so that the owner, group, and world (all users) can read the file and write to it:

(result)

```
-rw-rw-rw- 1 username 13 Nov 24 17:58 secretFile.out
```

Perhaps the programmer tried to do this because a different process uses different permissions that might prevent the file from being updated.

Example 4

This program creates and reads from an admin file to determine privilege information.

If the admin file doesn't exist, the program will create one. In order to create the file, the program must have write privileges to write to the file. After the file is created, the permissions need to be changed to read only.

Example Language: **Go** *(bad code)*

```go
const adminFile = "/etc/admin-users"
func createAdminFileIfNotExists() error {
  file, err := os.Create(adminFile)
  if err != nil {
    return err
  }
  return nil
}

func changeModeOfAdminFile() error {
  fileMode := os.FileMode(0440)
  if err := os.Chmod(adminFile, fileMode); err != nil {
    return err
  }
  return nil
}
```

os.Create will create a file with 0666 permissions before umask if the specified file does not exist. A typical umask of 0022 would result in the file having 0644 permissions. That is, the file would have world-writable and world-readable permissions.

In this scenario, it is advised to use the more customizable method of os.OpenFile with the os.O_WRONLY and os.O_CREATE flags specifying 0640 permissions to create the admin file.

This is because on a typical system where the umask is 0022, the perm 0640 applied in os.OpenFile will result in a file of 0620 where only the owner and group can write.

Example 5

The following command recursively sets world-readable permissions for a directory and all of its children:

Example Language: **Shell** *(bad code)*

```shell
chmod -R ugo+r DIRNAME
```

If this command is run from a program, the person calling the program might not expect that all the files under the directory will be world-readable. If the directory is expected to contain private data, this could become a security problem.

Example 6

The following Azure command updates the settings for a storage account:

Example Language: **Shell** *(bad code)*

```shell
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 code)*

```shell
az storage account update --name <storage-account> --resource-group <resource-group> --allow-blob-public-access false
```

Example 7

The following Google Cloud Storage command gets the settings for a storage account named 'BUCKET_NAME':

Example Language: **Shell** *(informative)*

```shell
gsutil iam get gs://BUCKET_NAME
```

Suppose the command returns the following result:

Example Language: **JSON** *(bad code)*

```json
{
  "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 code)*

```shell
gsutil iam ch -d allUsers gs://BUCKET_NAME
gsutil iam ch -d allAuthenticatedUsers gs://BUCKET_NAME
```

Observed Examples

Reference	Description
CVE-2022-29527	Go application for cloud management creates a world-writable sudoers file that allows local attackers to inject sudo rules and escalate privileges to root by winning a race condition.
CVE-2009-3482	Anti-virus product sets insecure "Everyone: Full Control" permissions for files under the "Program Files" folder, allowing attackers to replace executables with Trojan horses.
CVE-2009-3897	Product creates directories with 0777 permissions at installation, allowing users to gain privileges and access a socket used for authentication.
CVE-2009-3489	Photo editor installs a service with an insecure security descriptor, allowing users to stop or start the service, or execute commands as SYSTEM.
CVE-2020-15708	socket created with insecure permissions
CVE-2009-3289	Library function copies a file to a new target and uses the source file's permissions for the target, which is incorrect when the source file is a symbolic link, which typically has 0777 permissions.
CVE-2009-0115	Device driver uses world-writable permissions for a socket file, allowing attackers to inject arbitrary commands.
CVE-2009-1073	LDAP server stores a cleartext password in a world-readable file.
CVE-2009-0141	Terminal emulator creates TTY devices with world-writable permissions, allowing an attacker to write to the terminals of other users.
CVE-2008-0662	VPN product stores user credentials in a registry key with "Everyone: Full Control" permissions, allowing attackers to steal the credentials.
CVE-2008-0322	Driver installs its device interface with "Everyone: Write" permissions.
CVE-2009-3939	Driver installs a file with world-writable permissions.
CVE-2009-3611	Product changes permissions to 0777 before deleting a backup; the permissions stay insecure for subsequent backups.
CVE-2007-6033	Product creates a share with "Everyone: Full Control" permissions, allowing arbitrary program execution.
CVE-2007-5544	Product uses "Everyone: Full Control" permissions for memory-mapped files (shared memory) in inter-process communication, allowing attackers to tamper with a session.
CVE-2005-4868	Database product uses read/write permissions for everyone for its shared memory, allowing theft of credentials.
CVE-2004-1714	Security product uses "Everyone: Full Control" permissions for its configuration files.
CVE-2001-0006	"Everyone: Full Control" permissions assigned to a mutex allows users to disable network connectivity.
CVE-2002-0969	Chain: database product contains buffer overflow that is only reachable through a .ini configuration file - which has "Everyone: Full Control" permissions.

Detection Methods

Automated Static Analysis

Automated static analysis may be effective in detecting permission problems for system resources such as files, directories, shared memory, device interfaces, etc. Automated techniques may be able to detect the use of library functions that modify permissions, then analyze function calls for arguments that contain potentially insecure values.

However, since the software's intended security policy might allow loose permissions for certain operations (such as publishing a file on a web server), automated static analysis may produce some false positives - i.e., warnings that do not have any security consequences or require any code changes.

When custom permissions models are used - such as defining who can read messages in a particular forum in a bulletin board system - these can be difficult to detect using automated static analysis. It may be possible to define custom signatures that identify any custom functions that implement the permission checks and assignments.

Automated Dynamic Analysis

Automated dynamic analysis may be effective in detecting permission problems for system resources such as files, directories, shared memory, device interfaces, etc.

However, since the software's intended security policy might allow loose permissions for certain operations (such as publishing a file on a web server), automated dynamic analysis may produce some false positives - i.e., warnings that do not have any security consequences or require any code changes.

When custom permissions models are used - such as defining who can read messages in a particular forum in a bulletin board system - these can be difficult to detect using automated dynamic analysis. It may be possible to define custom signatures that identify any custom functions that implement the permission checks and assignments.

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.

Note: These may be more effective than strictly automated techniques. This is especially the case with weaknesses that are related to design and business rules.

Manual Static Analysis

Manual static analysis may be effective in detecting the use of custom permissions models and functions. The code could then be examined to identifying usage of the related functions. Then the human analyst could evaluate permission assignments in the context of the intended security model of the software.

Manual Dynamic Analysis

Manual dynamic analysis may be effective in detecting the use of custom permissions models and functions. The program could then be executed with a focus on exercising code paths that are related to the custom permissions. Then the human analyst could evaluate permission assignments in the context of the intended security model of the software.

Fuzzing

Fuzzing is not effective in detecting this weakness.

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 watch for library functions or system calls on OS resources such as files, directories, and shared memory. Examine the arguments to these calls to infer which permissions are being used.

Note: Note that this technique is only useful for permissions issues related to system resources. It is not likely to detect application-level business rules that are related to permissions, such as if a user of a blog system marks a post as "private," but the blog system inadvertently marks it as "public."

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

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

Effectiveness: SOAR Partial

Dynamic Analysis with Automated Results Interpretation

According to SOAR, the following detection techniques may be useful:

Cost effective for partial coverage:

- Host-based Vulnerability Scanners - Examine configuration for flaws, verifying that audit mechanisms work, ensure host configuration meets certain predefined criteria
- Web Application Scanner
- Web Services Scanner
- Database Scanners

Effectiveness: SOAR Partial

Dynamic Analysis with Manual Results Interpretation

According to SOAR, the following detection techniques may be useful:

Highly cost effective:

- Host Application Interface Scanner

Cost effective for partial coverage:

- Fuzz Tester
- Framework-based Fuzzer
- Automated Monitored Execution
- Forced Path Execution

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:

Cost effective for partial coverage:

- Context-configured Source Code Weakness Analyzer

Effectiveness: SOAR Partial

Automated Static Analysis

According to SOAR, the following detection techniques may be useful:

Cost effective for partial coverage:

- Configuration Checker

Effectiveness: SOAR Partial

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

Memberships

Nature	Type	ID	Name
MemberOf	C	743	CERT C Secure Coding Standard (2008) Chapter 10 - Input Output (FIO)
MemberOf	C	753	2009 Top 25 - Porous Defenses
MemberOf	C	803	2010 Top 25 - Porous Defenses
MemberOf	C	815	OWASP Top Ten 2010 Category A6 - Security Misconfiguration
MemberOf	C	857	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 14 - Input Output (FIO)
MemberOf	C	859	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 16 - Platform Security (SEC)
MemberOf	C	860	The CERT Oracle Secure Coding Standard for Java (2011) Chapter 17 - Runtime Environment (ENV)
MemberOf	C	866	2011 Top 25 - Porous Defenses
MemberOf	C	877	CERT C++ Secure Coding Section 09 - Input Output (FIO)
MemberOf	V	884	CWE Cross-section
MemberOf	C	946	SFP Secondary Cluster: Insecure Resource Permissions
MemberOf	C	1147	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 13. Input Output (FIO)
MemberOf	C	1149	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 15. Platform Security (SEC)
MemberOf	C	1150	SEI CERT Oracle Secure Coding Standard for Java - Guidelines 16. Runtime Environment (ENV)
MemberOf	V	1200	Weaknesses in the 2019 CWE Top 25 Most Dangerous Software Errors
MemberOf	C	1308	CISQ Quality Measures - Security
MemberOf	V	1337	Weaknesses in the 2021 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	V	1340	CISQ Data Protection Measures
MemberOf	V	1350	Weaknesses in the 2020 CWE Top 25 Most Dangerous Software Weaknesses
MemberOf	C	1396	Comprehensive Categorization: Access Control

Vulnerability Mapping Notes

Usage: ALLOWED-WITH-REVIEW *(this CWE ID could be used to map to real-world vulnerabilities in limited situations requiring careful review)*

Reason: Frequent Misuse

Rationale:

While the name itself indicates an assignment of permissions for resources, this is often misused for vulnerabilities in which "permissions" are not checked, which is an "authorization" weakness (CWE-285 or descendants) within CWE's model [REF-1287].

Comments:

Closely analyze the specific mistake that is allowing the resource to be exposed, and perform a CWE mapping for that mistake.

Notes

Maintenance

The relationships between privileges, permissions, and actors (e.g. users and groups) need further refinement within the Research view. One complication is that these concepts apply to two different pillars, related to control of resources (CWE-664) and protection mechanism failures (CWE-693).

Taxonomy Mappings

Mapped Taxonomy Name	Node ID	Fit	Mapped Node Name
The CERT Oracle Secure Coding Standard for Java (2011)	FIO03-J		Create files with appropriate access permission
The CERT Oracle Secure Coding Standard for Java (2011)	SEC01-J		Do not allow tainted variables in privileged blocks
The CERT Oracle Secure Coding Standard for Java (2011)	ENV03-J		Do not grant dangerous combinations of permissions
CERT C Secure Coding	FIO06-C		Create files with appropriate access permissions

Related Attack Patterns

CAPEC-ID	Attack Pattern Name
CAPEC-1	Accessing Functionality Not Properly Constrained by ACLs
CAPEC-122	Privilege Abuse
CAPEC-127	Directory Indexing
CAPEC-17	Using Malicious Files
CAPEC-180	Exploiting Incorrectly Configured Access Control Security Levels
CAPEC-206	Signing Malicious Code
CAPEC-234	Hijacking a privileged process
CAPEC-60	Reusing Session IDs (aka Session Replay)
CAPEC-61	Session Fixation
CAPEC-62	Cross Site Request Forgery
CAPEC-642	Replace Binaries

References

[REF-62] Mark Dowd, John McDonald and Justin Schuh. "The Art of Software Security Assessment". Chapter 9, "File Permissions." Page 495. 1st Edition. Addison Wesley. 2006.

[REF-207] John Viega and Gary McGraw. "Building Secure Software: How to Avoid Security Problems the Right Way". Chapter 8, "Access Control." Page 194. 1st Edition. Addison-Wesley. 2002.

[REF-594] Jason Lam. "Top 25 Series - Rank 21 - Incorrect Permission Assignment for Critical Response". SANS Software Security Institute. 2010-03-24. <http://software-security.sans.org/blog/2010/03/24/top-25-series-rank-21-incorrect-permission-assignment-for-critical-response>.

[REF-199] NIST. "United States Government Configuration Baseline (USGCB)". <https://csrc.nist.gov/Projects/United-States-Government-Configuration-Baseline>. *URL validated: 2023-03-28*.

[REF-1287] MITRE. "Supplemental Details - 2022 CWE Top 25". Details of Problematic Mappings. 2022-06-28. <https://cwe.mitre.org/top25/archive/2022/2022_cwe_top25_supplemental.html#problematicMappingDetails>.

[REF-1307] Center for Internet Security. "CIS Microsoft Azure Foundations Benchmark version 1.5.0". Section 3.7. 2022-08-16. <https://www.cisecurity.org/benchmark/azure>. *URL validated: 2023-01-19*.

[REF-1327] Center for Internet Security. "CIS Google Cloud Computing Platform Benchmark version 1.3.0". Section 5.1. 2022-03-31. <https://www.cisecurity.org/benchmark/google_cloud_computing_platform>. *URL validated: 2023-04-24*.

Content History

Submissions

Submission Date	Submitter	Organization
2008-09-08 *(CWE 1.0, 2008-09-09)*	CWE Content Team new weakness-focused entry for Research view.	MITRE

Modifications

Previous Entry Names

Page Last Updated: July 16, 2024