FIO01-C. Be careful using functions that use file names for identification

Many file-related security vulnerabilities result from a program accessing an unintended file object because file names are only loosely bound to underlying file objects. File names provide no information regarding the nature of the file object itself. Furthermore, the binding of a file name to a file object is reasserted every time the file name is used in an operation. File descriptors and `FILE` pointers are bound to underlying file objects by the operating system. (See FIO03-C. Do not make assumptions about fopen() and file creation.)

Accessing files via file descriptors or `FILE` pointers rather than file names provides a greater degree of certainty as to which object is actually acted upon. It is recommended that files be accessed through file descriptors or `FILE` pointers where possible.

The following C functions rely solely on file names for file identification:

- `remove()`
- `rename()`
- `fopen()`
- `freopen()`

Use these functions with caution. See FIO10-C. Take care when using the rename() function, and FIO08-C. Take care when calling remove() on an open file.

Noncompliant Code Example

In this noncompliant code example, the file identified by `file_name` is opened, processed, closed, and removed. However, it is possible that the file object identified by `file_name` in the call to `remove()` is not the same file object identified by `file_name` in the call to `fopen()`.

```
char *file_name;
FILE *f_ptr;

/* Initialize file_name */

f_ptr = fopen(file_name, "w");
if (f_ptr == NULL) {
  /* Handle error */
}

/*... Process file ...*/

if (fclose(f_ptr) != 0) {
  /* Handle error */
}

if (remove(file_name) != 0) {
  /* Handle error */
}
```

Compliant Solution

Not much can be done programmatically to ensure the file removed is the same file that was opened, processed, and closed except to make sure that the file is opened in a secure directory with privileges that would prevent the file from being manipulated by an untrusted user. (See FIO15-C. Ensure that file operations are performed in a secure directory.)

Noncompliant Code Example (POSIX)

In this noncompliant code example, the function `chmod()` is called to set the permissions of a file. However, it is not clear whether the file object referred to by `file_name` refers to the same object in the call to `fopen()` and in the call to `chmod()`.

```
char *file_name;
FILE *f_ptr;

/* Initialize file_name */

f_ptr = fopen(file_name, "w");
if (f_ptr == NULL)  {
  /* Handle error */
}

/* ... */

if (chmod(file_name, S_IRUSR) == -1) {
  /* Handle error */
}
```

Compliant Solution (POSIX)

This compliant solution uses the POSIX `fchmod()` and `open()` functions [IEEE Std 1003.1:2013]. Using these functions guarantees that the file opened is the same file that is operated on.

```
char *file_name;
int fd;

/* Initialize file_name */

fd = open(
  file_name,
  O_WRONLY | O_CREAT | O_EXCL,
  S_IRWXU
);
if (fd == -1) {
  /* Handle error */
}

/* ... */

if (fchmod(fd, S_IRUSR) == -1) {
  /* Handle error */
}
```

Mitigation Strategies (POSIX)

Many file-related race conditions can be eliminated by using

- `fchown()` rather than `chown()`
- `fstat()` rather than `stat()`
- `fchmod()` rather than `chmod()`

or simply by ensuring the security of the working directory per FIO15-C. Ensure that file operations are performed in a secure directory.

POSIX functions that have no file descriptor counterpart should be used with caution:

- `link()` and `unlink()`
- `mkdir()` and `rmdir()`
- `mount()` and `unmount()`
- `lstat()`
- `mknod()`
- `symlink()`
- `utime()`

Risk Assessment

Many file-related vulnerabilities, such as *time-of-check, time-of-use* (TOCTOU) race conditions, can be exploited to cause a program to access an unintended file. Using `FILE` pointers or file descriptors to identify files instead of file names reduces the chance of accessing an unintended file. Remediation costs are medium because, although insecure functions can be easily identified, simple drop-in replacements are not always available.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FIO01-C	Medium	Likely	Medium	**P12**	**L1**

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.0p0	**IO.RACE** **IO.TAINT.FNAME** **BADFUNC.TEMP.***	File System Race Condition Tainted Filename A collection of warning classes that report uses of library functions associated with temporary file vulnerabilities (including name issues).
Compass/ROSE			Can detect some violations of this recommendation. In particular, it warns when `chown()`, `stat()`, or `chmod()` are called on an open file
Coverity	6.5	**TOCTOU**	Fully implemented
Klocwork	2018	**SV.TOCTOU.FILE_ACCESS**	
LDRA tool suite	9.7.1	**592 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_C-FIO01-a** **CERT_C-FIO01-b**	Don't use chmod(), chown(), chgrp() Usage of functions prone to race is not allowed
Polyspace Bug Finder	R2018a	File access between time of check and use (TOCTOU)	File or folder might change state due to access race
PRQA QA-C	9.5	**5011**	Partially implemented

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard	VOID FIO01-CPP. Be careful using functions that use file names for identification
MITRE CWE	CWE-73, External control of file name or path CWE-367, Time-of-check, time-of-use race condition CWE-676, Use of potentially dangerous function

Bibliography

[Apple Secure Coding Guide]	"Avoiding Race Conditions and Insecure File Operations"
[Drepper 2006]	Section 2.2.1 "Identification when Opening"
[IEEE Std 1003.1:2013]	XSH, System Interfaces, `fchmod` XSH, System Interfaces, `open`
[Seacord 2013]	Chapter 8, "File I/O"

← ↑ →