FIO05-C. Identify files using multiple file attributes

Files can often be identified by attributes other than the file name, such as by comparing file ownership or creation time. Information about a file that has been created and closed can be stored and then used to validate the identity of the file when it is reopened.

Comparing multiple attributes of the file increases the likelihood that the reopened file is the same file that had been previously operated on.

File identification is less of an issue if applications maintain their files in secure directories, where they can be accessed only by the owner of the file and (possibly) by a system administrator. (See FIO15-C. Ensure that file operations are performed in a secure directory.)

Noncompliant Code Example (Reopen)

This noncompliant code example opens a file for writing, closes it, opens the same named file for reading, and then closes it again. The logic relies solely on the file name to identify the file.

```
char *file_name;

/* Initialize file_name */

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

/*... Write to file ...*/

fclose(fd);
fd = NULL;

/*
 * A race condition here allows for an attacker
 * to switch out the file for another.
 */

/* ... */

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

/*... Read from file ...*/

fclose(fd);
fd = NULL;
```

There is no guarantee that the file opened for reading is the same file that is opened for writing. An attacker can replace the original file (for example, with a symbolic link) between the first `fclose()` and the second `fopen()`.

Compliant Solution (POSIX) (Device/I-node)

Reopening a file stream should generally be avoided. However, it may sometimes be necessary in long-running applications to avoid depleting available file descriptors.

This compliant solution uses a "check, use, check" pattern to ensure that the file opened for reading is the same file that was opened for writing. In this solution, the file is opened for writing using the `open()` function. If the file is successfully opened, the `fstat()` function is used to read information about the file into the `orig_st` structure. When the file is reopened for reading, information about the file is read into the `new_st` structure, and the `st_dev` and `st_ino` fields in `orig_st` and `new_st` are compared to improve identification.

```
struct stat orig_st;
struct stat new_st;
char *file_name;

/* Initialize file_name */

int fd = open(file_name, O_WRONLY);
if (fd == -1) {
  /* Handle error */
}

/*... Write to file ...*/

if (fstat(fd, &orig_st) == -1) {
  /* Handle error */
}
close(fd);
fd = -1;

/* ... */

fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* Handle error */
}

if (fstat(fd, &new_st) == -1) {
  /* Handle error */
}

if ((orig_st.st_dev != new_st.st_dev) ||
    (orig_st.st_ino != new_st.st_ino)) {
  /* File was tampered with! */
}

/*... Read from file ...*/

close(fd);
fd = -1;
```

This solution enables the program to recognize if an attacker has switched files between the first `close()` and the second `open()`. The program does not recognize whether the file has been modified in place, however.

Alternatively, the same solution can be implemented using the C `fopen()` function to open the file and the POSIX `fileno()` function to convert the `FILE` object pointer to a file descriptor.

The structure members `st_mode`, `st_ino`, `st_dev`, `st_uid`, `st_gid`, `st_atime`, `st_ctime`, and `st_mtime` all should have meaningful values for all file types on POSIX-compliant systems. The `st_ino` field contains the file serial number. The `st_dev` field identifies the device containing the file. The `st_ino` and `st_dev` fields, taken together, uniquely identify the file. The `st_dev` value is not necessarily consistent across reboots or system crashes, however, so this field may not be useful for file identification if a system crash or reboot may have occurred before you attempt to reopen a file.

Call the `fstat()` function on a file that is already opened instead of calling `stat()` on a file name followed by `open()`. Doing so ensures that the file for which the information is being collected is the same file that is already opened. See FIO01-C. Be careful using functions that use file names for identification for more information on avoiding race conditions resulting from the use of file names for identification.

It may also be necessary to call `open()` with `O_NONBLOCK`, as per FIO32-C. Do not perform operations on devices that are only appropriate for files, to ensure that the program does not hang when trying to open special files.

This compliant solution may not work in some cases. For instance, a long-running service might choose to occasionally reopen a log file to add log messages but leave the file closed so that the log file may be periodically rotated. In this case, the i-node number would change, and this solution would no longer apply.

Compliant Solution (POSIX) (Open Only Once)

A simpler solution is to not reopen the file. In this code example, the file is opened once for both writing and reading. Once writing is complete, the `fseek()` function resets the file pointer to the beginning of the file, and its contents are read back. (See void FIO07-C. Prefer fseek() to rewind().)

Because the file is not reopened, the possibility of an attacker tampering with the file between the writes and subsequent reads is eliminated.

```
char *file_name;
FILE *fd;

/* Initialize file_name */

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

/*... Write to file ...*/

/* Go to beginning of file */
fseek(fd, 0, SEEK_SET);

/*... Read from file ...*/

fclose(fd);
fd = NULL;
```

Be sure to use `fflush()` after writing data to the file, in accordance with FIO39-C. Do not alternately input and output from a stream without an intervening flush or positioning call.

Noncompliant Code Example (Owner)

In this noncompliant code example, the programmer's intent is to open a file for reading, but only if the user running the process owns the specified file. This requirement is more restrictive than that imposed by the operating system, which requires only that the effective user have permissions to read the file. The code, however, relies solely on the file name to identify the file.

```
char *file_name;
FILE *fd;

/* Initialize file_name */

fd = fopen(file_name, "r+");
if (fd == NULL) {
  /* Handle error */
}

/* Read user's file */

fclose(fd);
fd = NULL;
```

If this code is run with superuser privileges, for example, as part of a `setuid-root` program, an attacker can exploit this program to read files for which the real user normally lacks sufficient privileges, including files not owned by the user.

Compliant Solution (POSIX) (Owner)

In this compliant solution, the file is opened using the `open()` function. If the file is successfully opened, the `fstat()` function is used to read information about the file into the `stat` structure. This information is compared with existing information about the real user (obtained by the `getuid()` and `getgid()` functions).

```
struct stat st;
char *file_name;

/* Initialize file_name */

int fd = open(file_name, O_RDONLY);
if (fd == -1) {
  /* Handle error */
}

if ((fstat(fd, &st) == -1) ||
   (st.st_uid != getuid()) ||
   (st.st_gid != getgid())) {
  /* File does not belong to user */
}

/*... Read from file ...*/

close(fd);
fd = -1;
```

By matching the file owner's user and group IDs to the process's real user and group IDs, this program now successfully restricts access to files owned by the real user of the program. This solution can be used to verify that the owner of the file is the one the program expects, reducing opportunities for attackers to replace configuration files with malicious ones, for example.

Alternatively, the same solution can be implemented using the C `fopen()` function to open the file and the POSIX `fileno()` function to convert the `FILE` object pointer to a file descriptor.

Risk Assessment

Many file-related vulnerabilities are exploited to cause a program to access an unintended file. Proper file identification is necessary to prevent exploitation.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
FIO05-C	Medium	Probable	Medium	P8	L2

Automated Detection

Tool	Version	Checker	Description
Compass /ROSE			Could report possible violations of this rule merely by reporting any `open()` or `fopen()` call that did not have a subsequent call to `fstat()`
LDRA tool suite	9.7.1	**44 S**	Enhanced Enforcement

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	VOID FIO05-CPP. Identify files using multiple file attributes
ISO/IEC TR 24772:2013	Path Traversal [EWR]
MITRE CWE	CWE-37, Path issue—Slash absolute path CWE-38, Path Issue—Backslash absolute path CWE-39, Path Issue—Drive letter or Windows volume CWE-62, UNIX hard link CWE-64, Windows shortcut following (.LNK) CWE-65, Windows hard link

Bibliography

[Drepper 2006]	Section 2.2.1 "Identification when Opening"

[IEEE Std 1003.1:2013]	System Interfaces: `open` System Interfaces: `fstat`
[Seacord 2013]	Chapter 8, "File I/O"