POS01-C. Check for the existence of links when dealing with files

Many common operating systems such as Windows and UNIX support file links, including hard links, symbolic (soft) links, and virtual drives. Hard links can be created in UNIX with the `ln` command or in Windows operating systems by calling the `CreateHardLink()` function. Symbolic links can be created in UNIX using the `ln -s` command or in Windows by using directory junctions in NTFS or the Linkd.exe (Win 2K resource kit) or "junction" freeware. Virtual drives can also be created in Windows using the `subst` command.

File links can create security issues for programs that fail to consider the possibility that the file being opened may actually be a link to a different file. This is especially dangerous when the vulnerable program is running with elevated privileges.

Frequently, there is no need to check for the existence of symbolic links because this problem can be solved using other techniques. When opening an existing file, for example, the simplest solution is often to drop privileges to the privileges of the user. This solution permits the use of links while preventing access to files for which the user of the application is not privileged.

When creating new files, it may be possible to use functions that create a new file only where a file does not already exist. This prevents the application from overwriting an existing file during file creation. (See FIO03-C. Do not make assumptions about fopen() and file creation.)

In rare cases, it is necessary to check for the existence of symbolic or hard links to ensure that a program is reading from an intended file and not a different file in another directory. In these cases, avoid creating a race condition when checking for the existence of symbolic links. (See POS35-C. Avoid race conditions while checking for the existence of a symbolic link.)

Noncompliant Code Example

This noncompliant code example opens the file specified by the string `file_name` for read/write access and then writes user-supplied data to the file:

```
char *file_name = /* something */;
char *userbuf = /* something */;
unsigned int userlen = /* length of userbuf string */;

int fd = open(file_name, O_RDWR);
if (fd == -1) {
   /* handle error */
}
write(fd, userbuf, userlen);
```

If the process is running with elevated privileges, an attacker can exploit this code, for example, by replacing the file with a symbolic link to the `/etc/passwd` authentication file. The attacker can then overwrite data stored in the password file to create a new root account with no password. As a result, this attack can be used to gain root privileges on a vulnerable system.

Compliant Solution (Linux 2.1.126+, FreeBSD, Solaris 10, POSIX.1-2008 `O_NOFOLLOW`)

Some systems provide the `O_NOFOLLOW` flag to help mitigate this problem. The flag is required by the POSIX.1-2008 standard and so will become more portable over time [Open Group 2008]. If the flag is set and the supplied `file_name` is a symbolic link, then the open will fail.

```
char *file_name = /* something */;
char *userbuf = /* something */;
unsigned int userlen = /* length of userbuf string */;

int fd = open(file_name, O_RDWR | O_NOFOLLOW);
if (fd == -1) {
  /* handle error */
}
write(fd, userbuf, userlen);
```

Note that this compliant solution does not check for hard links.

Compliant Solution (`lstat-fopen-fstat`)

This compliant solution uses the `lstat-fopen-fstat` idiom illustrated in FIO05-C. Identify files using multiple file attributes:

```
char *file_name = /* some value */;

struct stat orig_st;
if (lstat( file_name, &orig_st) != 0) {
  /* handle error */
}

if (!S_ISREG( orig_st.st_mode)) {
  /* file is irregular or symlink */
}

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

struct stat new_st;
if (fstat(fd, &new_st) != 0) {
  /* handle error */
}

if (orig_st.st_dev != new_st.st_dev ||
    orig_st.st_ino != new_st.st_ino) {
  /* file was tampered with during race window */
}

/* ... file is good, operate on fd ... */
```

This code is still subject to a time-of-check, time-of-use (TOCTOU) race condition, but before doing any operation on the file, it verifies that the file opened is the same file as was previously checked (by checking the file's device and i-node.) As a result, the code will recognize if an attacker has tampered with the file during the race window and can operate accordingly.

Note that this code does not check for hard links.

Hard Links

Hard links are problematic because if a file has multiple hard links, it is impossible to distinguish the original link from one that might have been created by a malicious attacker.

One way to deal with hard links is simply to disallow opening of any file with two or more hard links. The following code snippet, when inserted into the previous example, will identify if a file has multiple hard links:

```
if (orig_st.st_nlink > 1) {
  /* file has multiple hard links */
}
```

Because a hard link may not be created if the link and the linked-to file are on different devices, many platforms place system-critical files on a different device from the one where user-editable files are kept. For instance, the `/` directory, which contains critical system files like `/etc/passwd`, would live on one hard drive, while the `/home` directory, which contains user-editable files, would reside on a separate hard drive. This prevents users, for example, from creating hard links to `/etc/passwd`.

Risk Assessment

Failing to check for the existence of links can result in a critical system file being overwritten, leading to data integrity violations.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
POS01-C	medium	likely	high	**P6**	**L2**

Automated Detection

Tool	Version	Checker	Description
Compass /ROSE			Could report possible violations of this rule by flagging calls to `open()` that do not have an `O_NOFOLLOW` flag and that are not preceded by a call to `lstat()` and succeeded by a call to `fstat`

Related Vulnerabilities

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

Related Guidelines

MITRE CWE	CWE-59, Failure to resolve links before file access (aka "link following") CWE-362, Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition') CWE-367, Time-of-check, time-of-use (TOCTOU) race condition

Bibliography

[Austin Group 2008]	
[Open Group 2004]	`open()`
[Open Group 2008]	
[Seacord 2013]	Chapter 8, "File I/O"