FIO32-C. Do not perform operations on devices that are only appropriate for files

File names on many operating systems, including Windows and UNIX, may be used to access *special files*, which are actually devices. Reserved Microsoft Windows device names include `AUX`, `CON`, `PRN`, `COM1`, and `LPT1` or paths using the `\\.\` device namespace. Device files on UNIX systems are used to apply access rights and to direct operations on the files to the appropriate device drivers.

Performing operations on device files that are intended for ordinary character or binary files can result in crashes and denial-of-service attacks. For example, when Windows attempts to interpret the device name as a file resource, it performs an invalid resource access that usually results in a crash [Howard 2002].

Device files in UNIX can be a security risk when an attacker can access them in an unauthorized way. For example, if attackers can read or write to the `/dev/kmem` device, they may be able to alter the priority, UID, or other attributes of their process or simply crash the system. Similarly, access to disk devices, tape devices, network devices, and terminals being used by other processes can lead to problems [Garfinkel 1996].

On Linux, it is possible to lock certain applications by attempting to open devices rather than files. Consider the following example:

```
/dev/mouse
/dev/console
/dev/tty0
/dev/zero
```

A Web browser that failed to check for these devices would allow an attacker to create a website with image tags such as `<IMG src="file:///dev/mouse">` that would lock the user's mouse [Howard 2002].

Noncompliant Code Example

In this noncompliant code example, the user can specify a locked device or a FIFO (first-in, first-out) file name, which can cause the program to hang on the call to `fopen()`:

```
#include <stdio.h>

void func(const char *file_name) {
  FILE *file;
  if ((file = fopen(file_name, "wb")) == NULL) {
    /* Handle error */
  }

  /* Operate on the file */

  if (fclose(file) == EOF) {
    /* Handle error */
  }
}
```

Compliant Solution (POSIX)

POSIX defines the `O_NONBLOCK` flag to `open()`, which ensures that delayed operations on a file do not hang the program [IEEE Std 1003.1:2013].

> *When opening a FIFO with `O_RDONLY` or `O_WRONLY` set:*
>
> - *If `O_NONBLOCK` is set, an `open()` for reading-only returns without delay. An `open()` for writing-only returns an error if no process currently has the file open for reading.*
> - *If `O_NONBLOCK` is clear, an `open()` for reading-only blocks the calling thread until a thread opens the file for writing. An `open()` for writing-only blocks the calling thread until a thread opens the file for reading.*
>
> *When opening a block special or character special file that supports nonblocking opens:*
>
> - *If `O_NONBLOCK` is set, the `open()` function returns without blocking for the device to be ready or available; subsequent behavior is device-specific.*
> - *If `O_NONBLOCK` is clear, the `open()` function blocks the calling thread until the device is ready or available before returning.*
>
> *Otherwise, the behavior of `O_NONBLOCK` is unspecified.*

Once the file is open, programmers can use the POSIX `lstat()` and `fstat()` functions to obtain information about a file and the `S_ISREG()` macro to determine if the file is a regular file.

Because the behavior of `O_NONBLOCK` on subsequent calls to `read()` or `write()` is unspecified, it is advisable to disable the flag after it has been determined that the file in question is not a special device.

When available (Linux 2.1.126+, FreeBSD, Solaris 10, POSIX.1-2008), the `O_NOFOLLOW` flag should also be used. (See POS01-C. Check for the existence of links when dealing with files.) When `O_NOFOLLOW` is not available, symbolic link checks should use the method from POS35-C. Avoid race conditions while checking for the existence of a symbolic link.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#ifdef O_NOFOLLOW
  #define OPEN_FLAGS O_NOFOLLOW | O_NONBLOCK
#else
  #define OPEN_FLAGS O_NONBLOCK
#endif

void func(const char *file_name) {
  struct stat orig_st;
  struct stat open_st;
  int fd;
  int flags;

  if ((lstat(file_name, &orig_st) != 0) ||
      (!S_ISREG(orig_st.st_mode))) {
    /* Handle error */
  }

  /* Race window */

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

  if (fstat(fd, &open_st) != 0) {
    /* Handle error */
  }

  if ((orig_st.st_mode != open_st.st_mode) ||
      (orig_st.st_ino  != open_st.st_ino) ||
      (orig_st.st_dev  != open_st.st_dev)) {
    /* The file was tampered with */
  }

  /*
   * Optional: drop the O_NONBLOCK now that we are sure
   * this is a good file.
   */
  if ((flags = fcntl(fd, F_GETFL)) == -1) {
    /* Handle error */
  }

  if (fcntl(fd, F_SETFL, flags & ~O_NONBLOCK) == -1) {
    /* Handle error */
  }

  /* Operate on the file */

  if (close(fd) == -1) {
    /* Handle error */
  }
}
```

This code contains an intractable TOCTOU (time-of-check, time-of-use) race condition under which an attacker can alter the file referenced by `file_name` following the call to `lstat()` but before the call to `open()`. The switch will be discovered after the file is opened, but opening the file cannot be prevented in the case where this action itself causes undesired behavior. (See FIO45-C. Avoid TOCTOU race conditions while accessing files for more information about TOCTOU race conditions.)

Essentially, an attacker can switch out a file for one of the file types shown in the following table with the specified effect.

File Types and Effects

Type	Note on Effect
Another regular file	The `fstat()` verification fails.
FIFO	Either `open()` returns `-1` and sets `errno` to `ENXIO`, or `open()` succeeds and the `fstat()` verification fails.
Symbolic link	`open()` returns `-1` if `O_NOFOLLOW` is available; otherwise, the `fstat()` verification fails.
Special device	Usually the `fstat()` verification fails on `st_mode`. This can still be a problem if the device is one for which just opening (or closing) it causes a side effect. If `st_mode` compares equal, then the device is one that, after opening, appears to be a regular file. It would then fail the `fstat()` verification on `st_dev` and `st_ino` (unless it happens to be the *same* file, as can happen with `/dev/fd/*` on Solaris, but this would not be a problem).

To be compliant with this rule and to prevent this TOCTOU race condition, `file_name` must refer to a file in a secure directory. (See FIO15-C. Ensure that file operations are performed in a secure directory.)

Noncompliant Code Example (Windows)

This noncompliant code example uses the `GetFileType()` function to attempt to prevent opening a special file:

```
#include <Windows.h>

void func(const TCHAR *file_name) {
  HANDLE hFile = CreateFile(
    file_name,
    GENERIC_READ | GENERIC_WRITE, 0,
    NULL, OPEN_EXISTING,
    FILE_ATTRIBUTE_NORMAL, NULL
  );
  if (hFile == INVALID_HANDLE_VALUE) {
    /* Handle error */
  } else if (GetFileType(hFile) != FILE_TYPE_DISK) {
    /* Handle error */
    CloseHandle(hFile);
  } else {
    /* Operate on the file */
    CloseHandle(hFile);
  }
}
```

Although tempting, the Win32 `GetFileType()` function is dangerous in this case. If the file name given identifies a named pipe that is currently blocking on a read request, the call to `GetFileType()` will block until the read request completes. This provides an effective attack vector for a denial-of-service attack on the application. Furthermore, the act of opening a file handle may cause side effects, such as line states being set to their default voltage when opening a serial device.

Compliant Solution (Windows)

Microsoft documents a list of reserved identifiers that represent devices and have a device namespace to be used specifically by devices [MSDN]. In this compliant solution, the `isReservedName()` function can be used to determine if a specified path refers to a device. Care must be taken to avoid a TOCTOU race condition when first testing a path name using the `isReservedName()` function and then later operating on that path name.

```
#include <ctype.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

static bool isReservedName(const char *path) {
  /* This list of reserved names comes from MSDN */
  static const char *reserved[] = {
    "nul", "con", "prn", "aux", "com1", "com2", "com3",
    "com4", "com5", "com6", "com7", "com8", "com9",
    "lpt1", "lpt2", "lpt3", "lpt4", "lpt5", "lpt6",
    "lpt7", "lpt8", "lpt9"
  };
  bool ret = false;

/*
 * First, check to see if this is a device namespace, which
 * always starts with \\.\, because device namespaces are not
 * valid file paths.
 */

  if (!path || 0 == strncmp(path, "\\\\.\\", 4)) {
    return true;
  }

  /* Compare against the list of ancient reserved names */
  for (size_t i = 0; !ret &&
       i < sizeof(reserved) / sizeof(*reserved); ++i) {
   /*
    * Because Windows uses a case-insensitive file system, operate on
    * a lowercase version of the given filename. Note: This ignores
    * globalization issues and assumes ASCII characters.
    */
    if (0 == _stricmp(path, reserved[i])) {
      ret = true;
    }
  }
  return ret;
}
```

Risk Assessment

Allowing operations that are appropriate only for regular files to be performed on devices can result in denial-of-service attacks or more serious exploits depending on the platform.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO32-C	Medium	Unlikely	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Compass/ROSE			Could detect some violations of this rule. This rule applies only to untrusted file name strings, and ROSE cannot tell which strings are trusted and which are not. The best heuristic is to note if there is any verification of the file name before or after the `fopen()` call. If there is any verification, then the file opening should be preceded by an `lstat()` call and succeeded by an `fstat()` call. Although that does not enforce the rule completely, it does indicate that the coder is aware of the `lstat-fopen-fstat` idiom
Parasoft C/C++test	10.4.2	**CERT_C-FIO32-a**	Protect against file name injection
Polyspace Bug Finder	R2019b	CERT C: Rule FIO32-C	Checks for inappropriate I/O operation on device files (rule partially covered)

PRQA QA-C	9.7	**4921, 4922, 4923**	Enforced by QAC
PRQA QA-C++	4.4	**4921, 4922, 4923**	

Related Vulnerabilities

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

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT C Secure Coding Standard	FIO05-C. Identify files using multiple file attributes	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C Secure Coding Standard	FIO15-C. Ensure that file operations are performed in a secure directory	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C Secure Coding Standard	POS01-C. Check for the existence of links when dealing with files	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C Secure Coding Standard	POS35-C. Avoid race conditions while checking for the existence of a symbolic link	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT Oracle Secure Coding Standard for Java	FIO00-J. Do not operate on files in shared directories	Prior to 2018-01-12: CERT: Unspecified Relationship

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-67 and FIO32-C

FIO32-C = Union(CWE-67, list) where list =

- Treating trusted device names like regular files in Windows.

- Treating device names (both trusted and untrusted) like regular files in POSIX

Bibliography

[Garfinkel 1996]	Section 5.6, "Device Files"
[Howard 2002]	Chapter 11, "Canonical Representation Issues"
[IEEE Std 1003.1:2013]	XSH, System Interfaces, `open`
[MSDN]	