FIO00-J. Do not operate on files in shared directories

Multiuser systems allow multiple users with different privileges to share a file system. Each user in such an environment must be able to determine which files are shared and which are private, and each user must be able to enforce these decisions.

Unfortunately, a wide variety of file system vulnerabilities can be exploited by an attacker to gain access to files for which they lack sufficient privileges, particularly when operating on files that reside in shared directories in which multiple users may create, move, or delete files. Privilege escalation is also possible when these programs run with elevated privileges. A number of file system properties and capabilities can be exploited by an attacker, including *file links*, *device files*, and *shared file access*. To prevent vulnerabilities, a program must operate only on files in *secure directories*.

A directory is secure with respect to a particular user if only the user and the system administrator are allowed to create, move, or delete files inside the directory. Furthermore, each parent directory must itself be a secure directory up to and including the root directory. On most systems, home or user directories are secure by default and only shared directories are insecure.

File Links

Similar to shared files, file links can be swapped out and may not always point to the intended location. As a result, file links in shared directories are untrusted and should not be operated on (see FIO15-J. Do not operate on untrusted file links).

Device Files

File names on many operating systems may be used to access device files. Device files are used to access hardware and peripherals. Reserved MS-DOS device names include `AUX`, `CON`, `PRN`, `COM1`, and `LPT1`. Character special files and block special files are POSIX device files that direct operations on the files to the appropriate device drivers.

Performing operations on device files intended only for ordinary character or binary files can result in crashes and denial-of-service (DoS) 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 POSIX can be a security risk when an attacker can access them in an unauthorized way. For instance, if malicious programs can read or write to the `/dev/kmem` device, they may be able to alter their own priority, user ID, or other attributes of their process or they may simply crash the system. Similarly, access to disk devices, tape devices, network devices, and terminals being used by other processes can also lead to problems [Garfinkel 1996].

On Linux, it is possible to lock certain applications by attempting to read or write data on devices rather than files. Consider the following device path names:

```
/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.

Shared File Access

On many systems, files can be simultaneously accessed by concurrent processes. Exclusive access grants unrestricted file access to the locking process while denying access to all other processes, eliminating the potential for a race condition on the locked region. The `java.nio.channels.FileLock` class may be used for file locking. According to the Java API, Class FileLock [API 2014], documentation,

> *A file lock is either* exclusive *or* shared. *A shared lock prevents other concurrently running programs from acquiring an overlapping exclusive lock but does allow them to acquire overlapping shared locks. An exclusive lock prevents other programs from acquiring an overlapping lock of either type. Once it is released, a lock has no further effect on the locks that may be acquired by other programs.*

Shared locks support concurrent read access from multiple processes; *exclusive locks* support exclusive write access. File locks provide protection across processes, but they do not provide protection from multiple threads within a single process. Both shared locks and exclusive locks eliminate the potential for a cross-process race condition on the locked region. Exclusive locks provide mutual exclusion; shared locks prevent alteration of the state of the locked file region (one of the required properties for a data race).

The Java API [API 2014] documentation states that "whether or not a lock actually prevents another program from accessing the content of the locked region is system-dependent and consequently unspecified."

Microsoft Windows uses a mandatory file-locking mechanism that prevents processes from accessing a locked file region.

Linux implements both mandatory locks and advisory locks. Advisory locks are not enforced by the operating system, which diminishes their value from a security perspective. Unfortunately, the mandatory file lock in Linux is generally impractical for the following reasons:

- Mandatory locking is supported only by certain network file systems.
- File systems must be mounted with support for mandatory locking, which is disabled by default.
- Locking relies on the group ID bit, which can be turned off by another process (thereby defeating the lock).

- The lock is implicitly dropped if the holding process closes any descriptor of the file.

Noncompliant Code Example

In this noncompliant code example, an attacker could specify the name of a locked device or a first in, first out (FIFO) file, causing the program to hang when opening the file:

```
String file = /* Provided by user */;
InputStream in = null;
try {
  in = new FileInputStream(file);
  // ...
} finally {
  try {
    if (in !=null) { in.close();}
  } catch (IOException x) {
    // Handle error
  }
}
```

Noncompliant Code Example

This noncompliant code example uses the *try-with-resources* statement (introduced in Java SE 7) to open the file. The `try`-with-resources statement guarantees the file's successful closure if an exception is thrown, but this code is subject to the same vulnerabilities as the previous example.

```
String filename = /* Provided by user */;
Path path = new File(filename).toPath();
try (InputStream in = Files.newInputStream(path)) {
  // Read file
} catch (IOException x) {
  // Handle error
}
```

Noncompliant Code Example (`isRegularFile()`)

This noncompliant code example first checks that the file is a regular file (using the NIO.2 API) before opening it:

```
String filename = /* Provided by user */
Path path = new File(filename).toPath();
try {
  BasicFileAttributes attr =
      Files.readAttributes(path, BasicFileAttributes.class);

  // Check
  if (!attr.isRegularFile()) {
    System.out.println("Not a regular file");
    return;
  }
  // Other necessary checks

  // Use
  try (InputStream in = Files.newInputStream(path)) {
    // Read file
  }
} catch (IOException x) {
  // Handle error
}
```

This test can still be circumvented by a symbolic link. By default, the `readAttributes()` method follows symbolic links and reads the file attributes of the final target of the link. The result is that the program may reference a file other than the one intended.

Noncompliant Code Example (`NOFOLLOW_LINKS`)

This noncompliant code example checks the file by calling the `readAttributes()` method with the `NOFOLLOW_LINKS` link option to prevent the method from following symbolic links. This approach allows the detection of symbolic links because the `isRegularFile()` check is carried out on the symbolic link file and not on the final target of the link.

```
String filename = /* Provided by user */;
Path path = new File(filename).toPath();
try {
  BasicFileAttributes attr = Files.readAttributes(
      path, BasicFileAttributes.class, LinkOption.NOFOLLOW_LINKS);

  // Check
  if (!attr.isRegularFile()) {
    System.out.println("Not a regular file");
    return;
  }
  // Other necessary checks

  // Use
  try (InputStream in = Files.newInputStream(path)) {
    // Read file
  };
} catch (IOException x) {
  // Handle error
}
```

This code is still vulnerable to a time-of-check, time-of-use (TOCTOU) race condition. For example, an attacker can replace the regular file with a file link or device file after the code has completed its checks but before it opens the file.

Noncompliant Code Example (POSIX: Check-Use-Check)

This noncompliant code example performs the necessary checks and then opens the file. After opening the file, it performs a second check to make sure that the file has not been moved and that the file opened is the same file that was checked. This approach reduces the chance that an attacker has changed the file between checking and then opening the file. In both checks, the file's `fileKey` attribute is examined. The `fileKey` attribute serves as a unique key for identifying files and is more reliable than the path name as on indicator of the file's identity.

The SE 7 Documentation [J2SE 2011] describes the `fileKey` attribute:

> *Returns an object that uniquely identifies the given file, or null if a file key is not available. On some platforms or file systems it is possible to use an identifier, or a combination of identifiers to uniquely identify a file. Such identifiers are important for operations such as file tree traversal in file systems that support symbolic links or file systems that allow a file to be an entry in more than one directory. On UNIX file systems, for example, the device ID and inode are commonly used for such purposes.*
>
> *The file key returned by this method can only be guaranteed to be unique if the file system and files remain static. Whether a file system re-uses identifiers after a file is deleted is implementation dependent and consequently unspecified.*
>
> *File keys returned by this method can be compared for equality and are suitable for use in collections. If the file system and files remain static, and two files are the same with non-null file keys, then their file keys are equal.*

As noted in the documentation, `FileKey` cannot be used if it is not available. The `fileKey()` method returns `null` on Windows. Consequently, this solution is available only on systems such as POSIX in which `fileKey()` does not return `null`.

```
String filename = /* Provided by user */;
Path path = new File(filename).toPath();
try {
  BasicFileAttributes attr = Files.readAttributes(
      path, BasicFileAttributes.class, LinkOption.NOFOLLOW_LINKS);
  Object fileKey = attr.fileKey();

  // Check
  if (!attr.isRegularFile()) {
    System.out.println("Not a regular file");
    return;
  }
  // Other necessary checks

  // Use
  try (InputStream in = Files.newInputStream(path)) {

    // Check
    BasicFileAttributes attr2 = Files.readAttributes(
        path, BasicFileAttributes.class, LinkOption.NOFOLLOW_LINKS
    );
    Object fileKey2 = attr2.fileKey();
    if (!fileKey.equals(fileKey2)) {
      System.out.println("File has been tampered with");
    }

    // Read file
  };
} catch (IOException x) {
  // Handle error
}
```

Although this code goes to great lengths to prevent an attacker from successfully tricking it into opening the wrong file, it still has several vulnerabilities:

- The TOCTOU race condition still exists between the first check and open. During this race window, an attacker can replace the regular file with a symbolic link or other nonregular file. The second check detects this race condition but does not eliminate it.
- An attacker could subvert this code by letting the check operate on a regular file, substituting the nonregular file for the open, and then resubstituting the regular file to circumvent the second check. This vulnerability exists because Java lacks a mechanism to obtain file attributes from a file by any means other than the file name, and the binding of the file name to a file object is reasserted every time the file name is used in an operation. Consequently, an attacker can still swap a file for a nefarious file, such as a symbolic link.
- A system with hard links allows an attacker to construct a malicious file that is a hard link to a protected file. Hard links cannot be reliably detected by a program and can foil canonicalization attempts, which are prescribed by FIO16-J. Canonicalize path names before validating them.

Compliant Solution (POSIX: Secure Directory)

Because of the potential for race conditions and the inherent accessibility of shared directories, files must be operated on only in secure directories. Because programs may run with reduced privileges and lack the facilities to construct a secure directory, a program may need to throw an exception if it determines that a given path name is not in a secure directory.

Following is a POSIX-specific implementation of an `isInSecureDir()` method. This method ensures that the supplied file and all directories above it are owned by either the user or the system administrator, that each directory lacks write access for any other users, and that directories above the given file may not be deleted or renamed by any users other than the system administrator.

```
public static boolean isInSecureDir(Path file) {
  return isInSecureDir(file, null);
}
public static boolean isInSecureDir(Path file, UserPrincipal user) {
   return isInSecureDir(file, user, 5);
}

/**
 * Indicates whether file lives in a secure directory relative
 * to the program's user
 * @param file Path to test
 * @param user User to test. If null, defaults to current user
 * @param symlinkDepth Number of symbolic links allowed
 * @return true if file's directory is secure.
 */
public static boolean isInSecureDir(Path file, UserPrincipal user,
```

```
                                    int symlinkDepth) {
  if (!file.isAbsolute()) {
    file = file.toAbsolutePath();
  } if (symlinkDepth <=0) {
      // Too many levels of symbolic links
      return false;
    }

  // Get UserPrincipal for specified user and superuser
  FileSystem fileSystem =
      Paths.get(file.getRoot().toString()).getFileSystem();
  UserPrincipalLookupService upls =
      fileSystem.getUserPrincipalLookupService();
  UserPrincipal root = null;
  try {
    root = upls.lookupPrincipalByName("root");
    if (user == null) {
      user = upls.lookupPrincipalByName(System.getProperty("user.name"));
    }
    if (root == null || user == null) {
      return false;
    }
  } catch (IOException x) {
    return false;
  }

  // If any parent dirs (from root on down) are not secure,
  // dir is not secure
  for (int i = 1; i <= file.getNameCount(); i++) {
    Path partialPath = Paths.get(file.getRoot().toString(),
                                 file.subpath(0, i).toString());

    try {
      if (Files.isSymbolicLink(partialPath)) {
        if (!isInSecureDir(Files.readSymbolicLink(partialPath),)) {
                           user, symlinkDepth - 1)
          // Symbolic link, linked-to dir not secure
          return false;
        }
      } else {
        UserPrincipal owner = Files.getOwner(partialPath);
        if (!user.equals(owner) && !root.equals(owner)) {
          // dir owned by someone else, not secure
          return false;
        }
        PosixFileAttributes attr =
            Files.readAttributes(partialPath, PosixFileAttributes.class);
        Set<PosixFilePermission> perms = attr.permissions();
        if (perms.contains(PosixFilePermission.GROUP_WRITE) ||
            perms.contains(PosixFilePermission.OTHERS_WRITE)) {
          // Someone else can write files, not secure
          return false;
        }
      }
    } catch (IOException x) {
      return false;
    }
  }

  return true;
}
```

When checking directories, it is important to traverse from the root directory to the leaf directory to avoid a dangerous race condition whereby an attacker who has privileges to at least one of the directories can rename and re-create a directory after the privilege verification of subdirectories but before the verification of the tampered directory.

If the path contains any symbolic links, this routine will recursively invoke itself on the linked-to directory and ensure it is also secure. A symlinked directory may be secure if both its source and linked-to directory are secure. The method checks every directory in the path, ensuring that every directory is owned by the current user or the system administrator and that all directories in the path prevent other users from creating, deleting, or renaming files.

On POSIX systems, disabling group and world write access to a directory prevents modification by anyone other than the owner of the directory and the system administrator.

Note that this method is effective only on file systems that are fully compatible with POSIX file access permissions; it may behave incorrectly for file systems with other permission mechanisms.

The following compliant solution uses the `isInSecureDir()` method to ensure that an attacker cannot tamper with the file to be opened and subsequently removed. Note that once the path name of a directory has been checked using `isInSecureDir()`, all further file operations on that directory must be performed using the same path. This compliant solution also performs the same checks performed by the previous examples, such as making sure the requested file is a regular file, and not a symbolic link, device file, or other special file.

```
String filename = /* Provided by user */;
Path path = new File(filename).toPath();
try {
  if (!isInSecureDir(path)) {
    System.out.println("File not in secure directory");
    return;
  }

  BasicFileAttributes attr = Files.readAttributes(
      path, BasicFileAttributes.class, LinkOption.NOFOLLOW_LINKS);

  // Check
  if (!attr.isRegularFile()) {
    System.out.println("Not a regular file");
    return;
  }
  // Other necessary checks

  try (InputStream in = Files.newInputStream(path)) {
    // Read file
  }
} catch (IOException x) {
  // Handle error
}
```

Programs with elevated privileges may need to write files to directories owned by unprivileged users. One example is a mail daemon that reads a mail message from one user and places it in a directory owned by another user. In such cases, the mail daemon should assume the privileges of a user when reading or writing files on behalf of that user, in which case all file access should occur in secure directories relative to that user. When a program with elevated privileges must write files on its own behalf, these files should be in secure directories relative to the privileges of the program (such as directories accessible only by the system administrator).

Exceptions

FIO00-J-EX0: Programs that operate on single-user systems or on systems that have no shared directories or no possibility of file system vulnerabilities do not need to ensure that files are maintained in secure directories before operating on them.

Risk Assessment

Performing operations on files in shared directories can result in DoS attacks. If the program has elevated privileges, privilege escalation exploits are possible.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO00-J	Medium	Unlikely	Medium	**P4**	**L3**

Related Guidelines

SEI CERT C Coding Standard	FIO32-C. Do not perform operations on devices that are only appropriate for files
MITRE CWE	CWE-67, Improper Handling of Windows Device Names

Android Implementation Details

On Android, the SD card (`/sdcard` or `/mnt/sdcard`) is shared among multiple applications, so sensitive files should not be stored on the SD card (see DRD00-J. Do not store sensitive information on external storage (SD card)).

Bibliography

[API 2014]	Class `FileLock` Methods `createTempFile` Method `delete` Method `deleteOnExit`
[Darwin 2004]	Section 11.5, "Creating a Transient File"
[Garfinkel 1996]	Section 5.6, "Device Files"
[Howard 2002]	Chapter 11, "Canonical Representation Issues"
[J2SE 2011]	"The `try`-with-resources Statement"
[JDK Bug 2015]	Bug JDK-4171239 Bug JDK-4405521 Bug JDK-4631820
[Open Group 2004]	`open()`
[Secunia 2008]	Secunia Advisory 20132