FIO51-J. Identify files using multiple file attributes

Many file-related security vulnerabilities result from a program accessing an unintended file object. This often happens 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 reevaluated each time the file name is used in an operation. This reevaluation can introduce a time-of-check, time-of-use (TOCTOU) race condition into an application. Objects of type `java.io.File` and of type `java.nio.file.Path` are bound to underlying file objects by the operating system only when the file is accessed.

The `java.io.File` constructors and the `java.io.File` methods `renameTo()` and `delete()` rely solely on file names for file identification. The same holds for the `java.nio.file.Path.get()` methods for creating `Path` objects and the `move` and `delete` methods of `java.nio.file.Files`. Use all of these methods with caution.

Fortunately, files can often be identified by other attributes in addition to the file name—for example, by comparing file creation time or modification times. 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 was previously opened.

File identification is less crucial for applications that 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 FIO00-J. Do not operate on files in shared directories).

Noncompliant Code Example

In this noncompliant code example, the file identified by the string `filename` is opened, processed, closed, and then reopened for reading:

```
public void processFile(String filename){
  // Identify a file by its path
  Path file1 = Paths.get(filename);

  // Open the file for writing
  try (BufferedWriter bw = new BufferedWriter(new
       OutputStreamWriter(Files.newOutputStream(file1)))) {
    // Write to file...
  } catch (IOException e) {
    // Handle error
  }

  // Close the file

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

  // Reopen the file for reading
  Path file2 = Paths.get(filename);

  try (BufferedReader br = new BufferedReader(new
       InputStreamReader(Files.newInputStream(file2)))) {
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // Handle error
  }
}
```

Because the binding between the file name and the underlying file object is reevaluated when the `BufferedReader` is created, this code cannot guarantee that the file opened for reading is the same file that was previously opened for writing. An attacker might have replace the original file (with a symbolic link, for example) between the first call to `close()` and the subsequent creation of the `BufferedReader`.

Noncompliant Code Example (`Files.isSameFile()`)

In this noncompliant code example, the programmer attempts to ensure that the file opened for reading is the same as the file previously opened for writing by calling the method `Files.isSameFile()`:

```
public void processFile(String filename){
  // Identify a file by its path
  Path file1 = Paths.get(filename);

  // Open the file for writing
  try(BufferedWriter bw = new BufferedWriter(new
      OutputStreamWriter(Files.newOutputStream(file1)))) {
    // Write to file
  } catch (IOException e) {
    // Handle error
  }

  // ...
  // Reopen the file for reading
  Path file2 = Paths.get(filename);
  if (!Files.isSameFile(file1, file2)) {
    // File was tampered with, handle error
  }

  try(BufferedReader br = new BufferedReader(new
      InputStreamReader(Files.newInputStream(file2)))) {
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // Handle error
  }
}
```

Unfortunately, the Java API lacks any guarantee that the method `isSameFile()` actually checks whether the files are the same file. The Java 7 API for `isSameFile()` [API 2011] says:

> *If both `Path` objects are equal then this method returns `true` without checking if the file exists.*

That is, `isSameFile()` may simply check that the paths to the two files are the same and cannot detect if the file at that path had been replaced by a different file between the two open operations.

Compliant Solution (Multiple Attributes)

This compliant solution checks the creation and last-modified times of the files to increase the likelihood that the file opened for reading is the same file that was written:

```
public void processFile(String filename) throws IOException{
  // Identify a file by its path
  Path file1 = Paths.get(filename);
  BasicFileAttributes attr1 =
    Files.readAttributes(file1, BasicFileAttributes.class);
  FileTime creation1 = attr1.creationTime();
  FileTime modified1 = attr1.lastModifiedTime();

  // Open the file for writing
  try (BufferedWriter bw = new BufferedWriter(new
       OutputStreamWriter(Files.newOutputStream(file1)))) {
    // Write to file...
  } catch (IOException e) {
    // Handle error
  }

  // Reopen the file for reading
  Path file2 = Paths.get(filename);
  BasicFileAttributes attr2 =
    Files.readAttributes(file2, BasicFileAttributes.class);
  FileTime creation2 = attr2.creationTime();
  FileTime modified2 = attr2.lastModifiedTime();
  if ( (!creation1.equals(creation2)) ||
       (!modified1.equals(modified2)) ) {
    // File was tampered with, handle error
  }

  try(BufferedReader br = new BufferedReader(new
      InputStreamReader(Files.newInputStream(file2)))){
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // Handle error
  }
}
```

Although this solution is reasonably secure, a determined attacker could create a symbolic link with the same creation and last-modified times as the original file. Also, a TOCTOU race condition occurs between the time the file's attributes are first read and the time the file is first opened. Likewise, another TOCTOU condition occurs the second time the attributes are read and the file is reopened.

Compliant Solution (POSIX `fileKey` Attribute)

In environments that support the `fileKey` attribute, a more reliable approach is to check that the `fileKey` attributes of the two files are the same. The `fileKey` attribute is an object that "uniquely identifies the file" [API 2011], as shown in this compliant solution:

```
public void processFile(String filename) throws IOException{
  // Identify a file by its path
  Path file1 = Paths.get(filename);
  BasicFileAttributes attr1 =
    Files.readAttributes(file1, BasicFileAttributes.class);
  Object key1 = attr1.fileKey();
  // Open the file for writing
  try(BufferedWriter bw = new BufferedWriter(new
      OutputStreamWriter(Files.newOutputStream(file1)))) {
    // Write to file
  } catch (IOException e) {
    // Handle error
  }

  // Reopen the file for reading
  Path file2 = Paths.get(filename);
  BasicFileAttributes attr2 =
    Files.readAttributes(file2, BasicFileAttributes.class);
  Object key2 = attr2.fileKey();

  if ( !key1.equals(key2) ) {
    System.out.println("File tampered with");
    // File was tampered with, handle error
  }

  try(BufferedReader br = new BufferedReader(new
      InputStreamReader(Files.newInputStream(file2)))) {
    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // Handle error
  }
}
```

This approach will not work on all platforms. For example, on Windows 7 Enterprise Edition, all `fileKey` attributes are null.

The file key returned by the `fileKey()` method is guaranteed to be unique only if the file system and files remain static. A file system may reuse an identifier, for example, after a file is deleted. Like the previous compliant solution, there is a TOCTOU race window between the time the file's attributes are first read and the time the file is first opened. Another TOCTOU condition occurs the second time the attributes are read and the file is reopened.

Compliant Solution (`RandomAccessFile`)

A better approach is to avoid reopening a file. The following compliant solution demonstrates use of a `RandomAccessFile`, which can be opened for both reading and writing. Because the file is only closed automatically by the `try-with-resources` statement, no race condition can occur.

```
public void processFile(String filename) throws IOException{
  // Identify a file by its path
  try (RandomAccessFile file = new
       RandomAccessFile(filename, "rw")) {

    // Write to file...

    // Go back to beginning and read contents
    file.seek(0);
    string line;
    while (line=file.readLine()) != null {
      System.out.println(line);
    }
  }
}
```

Noncompliant Code Example (File Size)

This noncompliant code example tries to ensure that the file it opens contains exactly 1024 bytes:

```
static long goodSize = 1024;

public void doSomethingWithFile(String filename) {
  long size = new File( filename).length();
  if (size != goodSize) {
    System.out.println("File has wrong size!");
    return;
  }

  try (BufferedReader br = new BufferedReader(new
       InputStreamReader(new FileInputStream( filename)))) {
    // ... Work with file
  } catch (IOException e) {
    // Handle error
  }
}
```

This code is subject to a TOCTOU race condition between when the file size is checked and when the file is opened. If an attacker replaces a 1024-byte file with another file during this race window, he or she can cause this program to open any file, defeating the check.

Compliant Solution (File Size)

This compliant solution uses the `FileChannel.size()` method to obtain the file size. Because this method is applied to the `FileInputStream` only after the file has been opened, this solution eliminates the race window.

```
static long goodSize = 1024;

public void doSomethingWithFile(String filename) {
  try (FileInputStream in = new FileInputStream( filename);
    BufferedReader br = new BufferedReader(
                        new InputStreamReader(in))) {
    long size = in.getChannel().size();
    if (size != goodSize) {
      System.out.println("File has wrong size!");
      return;
    }

    String line;
    while ((line = br.readLine()) != null) {
      System.out.println(line);
    }
  } catch (IOException e) {
    // Handle error
  }
}
```

Applicability

Attackers frequently exploit file-related vulnerabilities to cause programs to access an unintended file. Proper file identification is necessary to prevent exploitation.

Bibliography

[API 2011]	Class `java.io.File` Interface `java.nio.file.Path` Class `java.nio.file.Files` Interface `java.nio.file.attribute.BasicFileAttributes`