FIO16-J. Canonicalize path names before validating them

According to the Java API [API 2006] for class `java.io.File`:

> *A pathname, whether abstract or in string form, may be either* absolute *or* relative. *An absolute pathname is complete in that no other information is required to locate the file that it denotes. A relative pathname, in contrast, must be interpreted in terms of information taken from some other pathname.*

Absolute or relative path names may contain file links such as symbolic (soft) links, hard links, shortcuts, shadows, aliases, and junctions. These file links must be fully resolved before any file validation operations are performed. For example, the final target of a symbolic link called `trace` might be the path name `/home/system/trace`. Path names may also contain special file names that make validation difficult:

1. "`.`" refers to the directory itself.
2. Inside a directory, the special file name "`..`" refers to the directory's parent directory.

In addition to these specific issues, a wide variety of operating system–specific and file system–specific naming conventions make validation difficult.

Canonicalizing file names makes it easier to validate a path name. More than one path name can refer to a single directory or file. Further, the textual representation of a path name may yield little or no information regarding the directory or file to which it refers. Consequently, all path names must be fully resolved or *canonicalized* before validation.

Validation may be necessary, for example, when attempting to restrict user access to files within a particular directory or to otherwise make security decisions based on the name of a file name or path name. Frequently, these restrictions can be circumvented by an attacker by exploiting a *directory traversal* or *path equivalence* vulnerability. A *directory traversal* vulnerability allows an I/O operation to escape a specified operating directory. A *path equivalence* vulnerability occurs when an attacker provides a different but equivalent name for a resource to bypass security checks.

Canonicalization contains an inherent race window between the time the program obtains the canonical path name and the time it opens the file. While the canonical path name is being validated, the file system may have been modified and the canonical path name may no longer reference the original valid file. Fortunately, this race condition can be easily mitigated. The canonical path name can be used to determine whether the referenced file name is in a secure directory (see FIO00-J. Do not operate on files in shared directories for more information). If the referenced file is in a secure directory, then, by definition, an attacker cannot tamper with it and cannot exploit the race condition.

This recommendation is a specific instance of IDS01-J. Normalize strings before validating them.

Noncompliant Code Example

This noncompliant code example allows the user to specify the path of an image file to open. By prepending `/img/` to the directory, this code enforces a policy that only files in this directory should be opened. The program also uses the `isInSecureDir()` method defined in FIO00-J. Do not operate on files in shared directories.

However, the user can still specify a file outside the intended directory by entering an argument that contains `../` sequences. An attacker can also create a link in the `/img` directory that refers to a directory or file outside of that directory. The path name of the link might appear to reside in the `/img` directory and consequently pass validation, but the operation will actually be performed on the final target of the link, which can reside outside the intended directory.

```
File file = new File("/img/" + args[0]);
if (!isInSecureDir(file)) {
  throw new IllegalArgumentException();
}
FileOutputStream fis = new FileOutputStream(file);
// ...
```

Noncompliant Code Example (`getCanonicalPath()`)

This noncompliant code example attempts to mitigate the issue by using the `File.getCanonicalPath()` method, introduced in Java 2, which fully resolves the argument and constructs a canonicalized path. Special file names such as dot dot (`..`) are also removed so that the input is reduced to a canonicalized form before validation is carried out. An attacker cannot use `../` sequences to break out of the specified directory when the `validate()` method is present. For example, the path `/img/../etc/passwd` resolves to `/etc/passwd`. The `getCanonicalPath()` method throws a security exception when used in applets because it reveals too much information about the host machine. The `getCanonicalFile()` method behaves like `getCanonicalPath()` but returns a new `File` object instead of a `String`.

Unfortunately, the canonicalization is performed after the validation, which renders the validation ineffective.

```
File file = new File("/img/" + args[0]);
if (!isInSecureDir(file)) {
  throw new IllegalArgumentException();
}
String canonicalPath = file.getCanonicalPath();
FileOutputStream fis = new FileOutputStream(canonicalPath);
// ...
```

Compliant Solution (`getCanonicalPath()`)

This compliant solution obtains the file name from the untrusted user input, canonicalizes it, and then validates it against a list of benign path names. It operates on the specified file only when validation succeeds, that is, only if the file is one of the two valid files `file1.txt` or `file2.txt` in `/img/java`.

```
File file = new File("/img/" + args[0]);
if (!isInSecureDir(file)) {
  throw new IllegalArgumentException();
}
String canonicalPath = file.getCanonicalPath();
if (!canonicalPath.equals("/img/java/file1.txt") &&
    !canonicalPath.equals("/img/java/file2.txt")) {
   // Invalid file; handle error
}

FileInputStream fis = new FileInputStream(f);
```

Compliant Solution (Security Manager)

A comprehensive way to handle this issue is to grant the application the permissions to operate only on files present within the intended directory—the `/img` directory in this example. This compliant solution specifies the absolute path of the program in its security policy file and grants `java.io.FilePermission` with target `/img/java` and the read action.
This solution requires that the `/img` directory is a secure directory, as described in FIO00-J. Do not operate on files in shared directories.

```
// All files in /img/java can be read
grant codeBase "file:/home/programpath/" {
  permission java.io.FilePermission "/img/java", "read";
};
```

Risk Assessment

Using path names from untrusted sources without first canonicalizing them and then validating them can result in directory traversal and path equivalence vulnerabilities.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
FIO16-J	Medium	Unlikely	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
The Checker Framework	2.1.3	**Tainting Checker**	Trust and security errors (see Chapter 8)
Coverity	7.5	**BAD_EQ PATH_MANIPULATION**	Implemented
Fortify	1.0	**Path_Manipulation**	Implemented

Related Vulnerabilities

CVE-2005-0789 describes a directory traversal vulnerability in LimeWire 3.9.6 through 4.6.0 that allows remote attackers to read arbitrary files via a `..` (dot dot) in a magnet request.

CVE-2008-5518 describes multiple directory traversal vulnerabilities in the web administration console in Apache Geronimo Application Server 2.1 through 2.1.3 on Windows that allow remote attackers to upload files to arbitrary directories.

Related Guidelines

SEI CERT C Coding Standard	FIO02-C. Canonicalize path names originating from tainted sources
SEI CERT C++ Coding Standard	VOID FIO02-CPP. Canonicalize path names originating from untrusted sources
ISO/IEC TR 24772:2013	Path Traversal [EWR]
MITRE CWE	CWE-171, Cleansing, Canonicalization, and Comparison Errors CWE-647, Use of Non-canonical URL Paths for Authorization Decisions

Android Implementation Details

This rule is applicable in principle to Android. Please refer to the Android-specific instance of this rule: DRD08-J. Always canonicalize a URL received by a content provider.

Bibliography

[API 2014]	Method `getCanonicalPath()`
[Harold 1999]