Fortify Security Report

1/29/23

Administrator

Executive Summary

Issues Overview

On Jan 29, 2023, a source code review was performed over the eightball code base. 1 files, 7 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 5 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	3
High	2

Recommendations and Conclusions

The Issues Category section provides Fortify recommendations for addressing issues at a generic level. The recommendations for specific fixes can be extrapolated from those generic recommendations by the development group.

Project Summary

Code Base Summary

Code location: C:/Program Files/Fortify/Fortify_SCA_and_Apps_22.1.0/Samples/basic/eightball
Number of Files: 1
Lines of Code: 7
Build Label: <No Build Label>

Scan Information

Scan time: 00:23
SCA Engine version: 22.1.0.0166
Machine Name: SCA
Username running scan: Administrator

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Command Line Arguments:
null.EightBall.main

File System:
java.io.FileReader.FileReader
java.io.FileReader.FileReader

Stream:
java.io.Reader.read

Filter Set Summary

Current Enabled Filter Set:
Security Auditor View

Filter Set Details:

Folder Filters:
If [fortify priority order] contains critical Then set folder to Critical
If [fortify priority order] contains high Then set folder to High

If [fortify priority order] contains medium Then set folder to Medium
If [fortify priority order] contains low Then set folder to Low

Audit Guide Summary

Audit guide not enabled

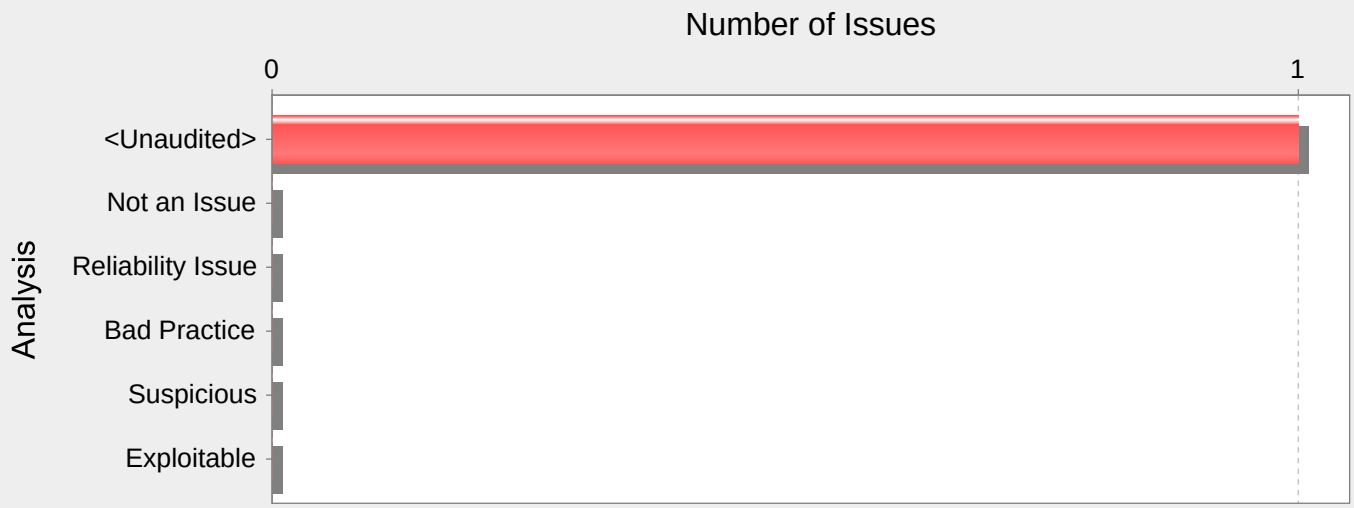
Results Outline

Overall number of results

The scan found 5 issues.

Vulnerability Examples by Category

Category: Path Manipulation (1 Issues)



Abstract:

Attackers can control the file system path argument to FileReader() at EightBall.java line 12, which allows them to access or modify otherwise protected files.

Explanation:

Path manipulation errors occur when the following two conditions are met:

1. An attacker can specify a path used in an operation on the file system.

2. By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program might give the attacker the ability to overwrite the specified file or run with a configuration controlled by the attacker.

Example 1: The following code uses input from an HTTP request to create a file name. The programmer has not considered the possibility that an attacker could provide a file name such as "../../tomcat/conf/server.xml", which causes the application to delete one of its own configuration files.

```
String rName = request.getParameter("reportName");
File rFile = new File("/usr/local/apfr/reports/" + rName);
...
rFile.delete();
```

Example 2: The following code uses input from a configuration file to determine which file to open and echo back to the user. If the program runs with adequate privileges and malicious users can change the configuration file, they can use the program to read any file on the system that ends with the extension .txt.

```
fis = new FileInputStream(cfg.getProperty("sub")+".txt");
amt = fis.read(arr);
out.println(arr);
```

Some think that in the mobile environment, classic vulnerabilities, such as path manipulation, do not make sense -- why would the user attack themself? However, keep in mind that the essence of mobile platforms is applications that are downloaded from various sources and run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which necessitates expanding the attack surface of mobile applications to include inter-process communication.

Example 3: The following code adapts Example 1 to the Android platform.

```
...
String rName = this.getIntent().getExtras().getString("reportName");
```

```
File rFile = getBaseContext().getFileStreamPath(rName);
...
rFile.delete();
...
```

Recommendations:

The best way to prevent path manipulation is with a level of indirection: create a list of legitimate values from which the user must select. With this approach, the user-provided input is never used directly to specify the resource name.

In some situations this approach is impractical because the set of legitimate resource names is too large or too hard to maintain. Programmers often resort to implementing a deny list in these situations. A deny list is used to selectively reject or escape potentially dangerous characters before using the input. However, any such list of unsafe characters is likely to be incomplete and will almost certainly become out of date. A better approach is to create a list of characters that are permitted to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program performs custom input validation to your satisfaction, use the Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

2. Implementation of an effective deny list is notoriously difficult. One should be skeptical if validation logic requires implementing a deny list. Consider different types of input encoding and different sets of metacharacters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the deny list can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

EightBall.java, line 12 (Path Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to FileReader() at EightBall.java line 12, which allows them to access or modify otherwise protected files.		

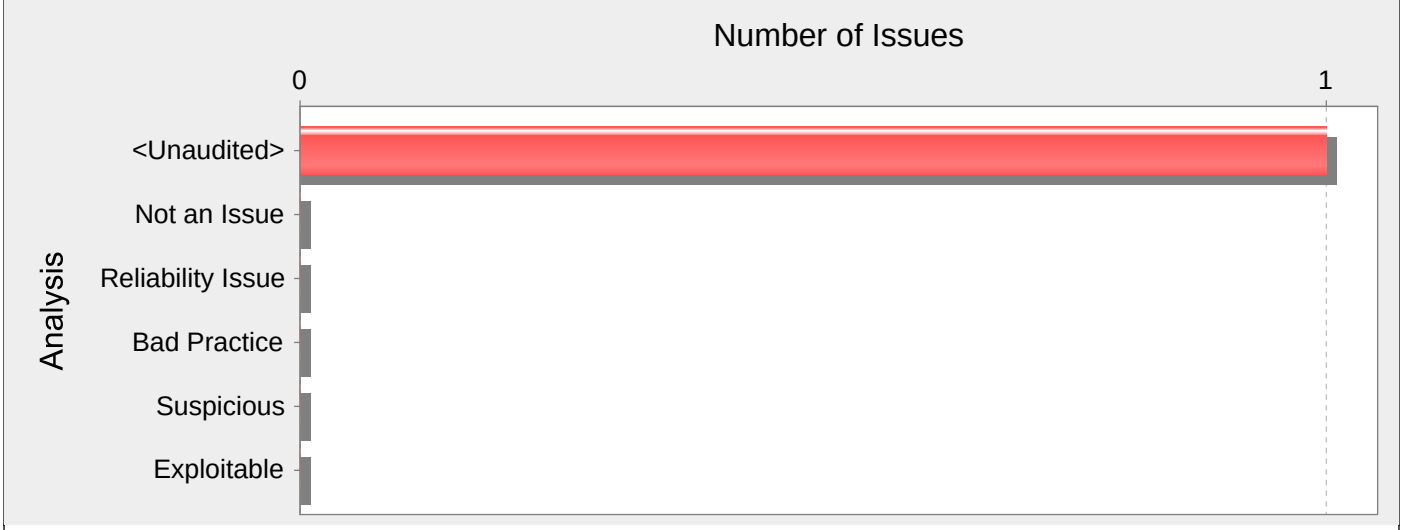
Source: EightBall.java:4 main(0)

```
2
3   public class EightBall {
4    public static void main(String args[]) throws Exception {
5     char[] buffer = new char[1024];
6     String filename = args[0];
```

Sink: EightBall.java:12 java.io.FileReader.FileReader()

```
10     System.out.println("Invalid input.");
11    }
12    new FileReader(filename).read(buffer);
13    System.out.println(buffer);
14   }
```

Category: Unreleased Resource: Streams (1 Issues)



Abstract:

The function main() in EightBall.java sometimes fails to release a system resource allocated by FileReader() on line 12.

Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

- Confusion over which part of the program is responsible for releasing the resource.

Most unreleased resource issues result in general software reliability problems. However, if an attacker can intentionally trigger a resource leak, the attacker may be able to launch a denial of service attack by depleting the resource pool.

Example: The following method never closes the file handle it opens. The finalize() method for FileInputStream eventually calls close(), but there is no guarantee as to how long it will take before the finalize() method will be invoked. In a busy environment, this can result in the JVM using up all of its file handles.

```
private void processFile(String fName) throws FileNotFoundException, IOException {
FileInputStream fis = new FileInputStream(fName);
int sz;
byte[] byteArray = new byte[BLOCK_SIZE];
while ((sz = fis.read(byteArray)) != -1) {
processBytes(byteArray, sz);
}
}
```

Recommendations:

1. Never rely on finalize() to reclaim resources. In order for an object's finalize() method to be invoked, the garbage collector must determine that the object is eligible for garbage collection. Because the garbage collector is not required to run unless the JVM is low on memory, there is no guarantee that an object's finalize() method will be invoked in an expedient fashion. When the garbage collector finally does run, it may cause a large number of resources to be reclaimed in a short period of time, which can lead to "bursty" performance and lower overall system throughput. This effect becomes more pronounced as the load on the system increases.

Finally, if it is possible for a resource reclamation operation to hang (if it requires communicating over a network to a database, for example), then the thread that is executing the finalize() method will hang.

2. Release resources in a finally block. The code for the Example should be rewritten as follows:

```
public void processFile(String fName) throws FileNotFoundException, IOException {
FileInputStream fis;
try {
fis = new FileInputStream(fName);
int sz;
byte[] byteArray = new byte[BLOCK_SIZE];
while ((sz = fis.read(byteArray)) != -1) {
processBytes(byteArray, sz);
```

```
}
}
finally {
if (fis != null) {
safeClose(fis);
}
}
}
public static void safeClose(FileInputStream fis) {
if (fis != null) {
try {
fis.close();
} catch (IOException e) {
log(e);
}
}
}
```

This solution uses a helper function to log the exceptions that might occur when trying to close the stream. Presumably this helper function will be reused whenever a stream needs to be closed.

Also, the processFile method does not initialize the fis object to null. Instead, it checks to ensure that fis is not null before calling safeClose(). Without the null check, the Java compiler reports that fis might not be initialized. This choice takes advantage of Java's ability to detect uninitialized variables. If fis is initialized to null in a more complex method, cases in which fis is used without being initialized will not be detected by the compiler.

EightBall.java, line 12 (Unreleased Resource: Streams)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function main() in EightBall.java sometimes fails to release a system resource allocated by FileReader() on line 12.		

Sink: EightBall.java:12 new FileReader(...)

```
10    System.out.println("Invalid input.");
11   }
12   new FileReader(filename).read(buffer);
13   System.out.println(buffer);
14  }
```

Issue Count by Category

Issues by Category	
Path Manipulation	2
J2EE Bad Practices: Leftover Debug Code	1
Unchecked Return Value	1
Unreleased Resource: Streams	1

Issue Breakdown by Analysis

Issues by Analysis