Fortify Security Report

Sep 28, 2017

ChandlerAustin

Executive Summary

Issues Overview

On Sep 28, 2017, a source code review was performed over the task4 code base. 1 files, 4 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 2 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order

High (1 Hidden)	1
Low (1 Hidden)	1

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:/Users/ChandlerAustin/Documents/lab3/task4

Number of Files: 1

Lines of Code: 4

Build Label: <No Build Label>

Scan Information

Scan time: 00:13

SCA Engine version: 6.21.0007

Machine Name: IALAB04

Username running scan: ChandlerAustin

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.ResourceInjection.main

Filter Set Summary

Current Enabled Filter Set:

Quick 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

Visibility Filters:

If impact is not in range [2.5, 5.0] Then hide issue

If likelihood is not in range (1.0, 5.0] Then hide issue

Audit Guide Summary

File System Inputs

Hide issues involving file system inputs.
Depending on your system, inputs from files may or may not come from trusted users. AuditGuide can hide issues that are based on data coming from the file system if it is trusted.
Enable if you trust file system inputs.

Filters:
If taint contains file_system Then hide issue
If taint contains constantfile Then hide issue
If taint contains stream Then hide issue
If category is file access race condition Then hide issueTaint from Command-Line Arguments

Hide issues involving taint from command-line arguments.
Depending on your system, inputs from command-line arguments may or may not come from trusted users. AuditGuide can hide issues that are based on data coming from command-line arguments if they are trusted.
Enable if you trust command-line arguments.

Filters:
If taint contains args Then hide issueProperty File Inputs

Hide inputs from properties files.
Depending on your system, inputs from properties files may or may not come from trusted users. AuditGuide can hide issues that are based on data coming from properties files if they are trusted.
Enable if you trust inputs from properties files.

Filters:
If taint contains property Then hide issueEnvironment Variable Inputs

Hide issues involving environment variable inputs.
Depending on your system, inputs from environment variables may or may not come from trusted users. AuditGuide can hide issues that are based on data coming from environment variables if they are trusted.
Enable if you trust environment variable inputs.

Filters:
If taint contains environment Then hide issueJ2EE Bad Practices

Hide warnings about J2EE bad practices.
Depending on whether your application is a J2EE application, J2EE bad practice warnings may or may not apply. AuditGuide can hide J2EE bad practice warnings.
Enable if J2EE bad practice warnings do not apply to your application because it is not a J2EE application.

Filters:
If category contains j2ee Then hide issue
If category is race condition: static database connection Then hide issue

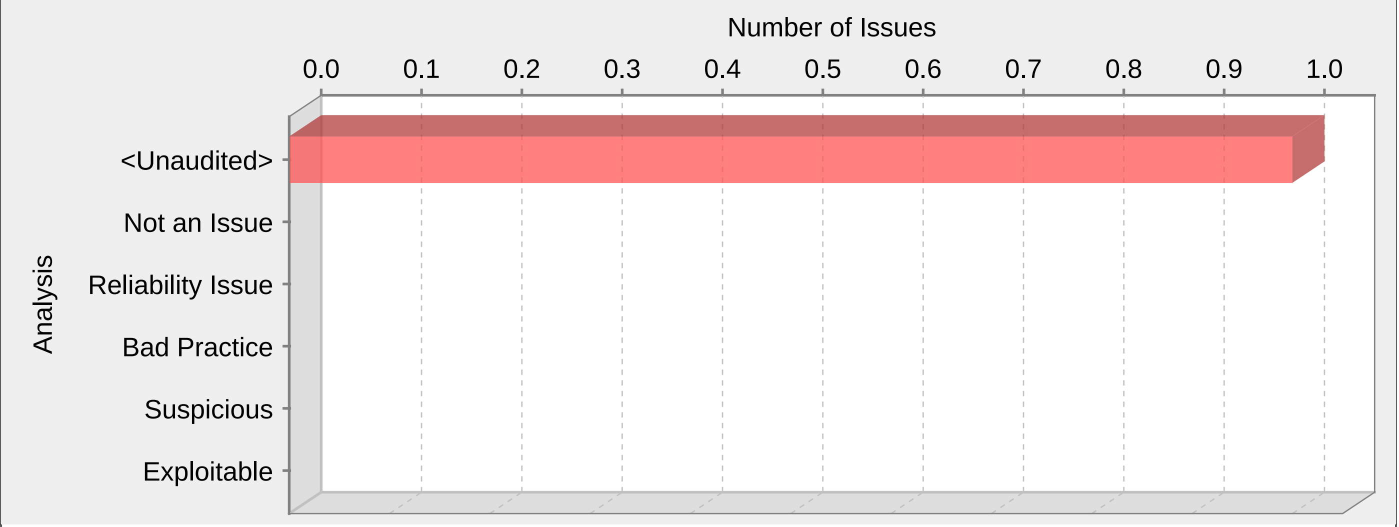
Results Outline

Overall number of results

The scan found 2 issues.

Vulnerability Examples by Category

Category: Path Manipulation (1 Issues: 1 Hidden)



Abstract:

Allowing user input to control paths used in filesystem operations could enable an attacker to access or modify otherwise protected system resources.

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 filesystem.

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

For example, the program may 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 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 world, 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 resource names that a user is allowed to specify, and only allow the user to select from the list. With this approach the input provided by the user 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 keep track of. Programmers often resort to blacklisting in these situations. Blacklisting selectively rejects or escapes 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 whitelist of characters that are allowed to appear in the resource name and accept input composed exclusively of characters in the approved set.

Tips:

1. If the program is performing input validation, satisfy yourself that the validation is correct, and use the HP Fortify Custom Rules Editor to create a cleanse rule for the validation routine.

2. Since implementing a blacklist that is effective on its own is notoriously difficult, if validation logic relies on blacklisting, one should be skeptical. Consider different types of input encoding and different sets of meta-characters that might have special meaning when interpreted by different operating systems, databases, or other resources. Determine whether or not the blacklist can be updated easily, correctly, and completely if these requirements ever change.

3. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Spring MVC are among them. To highlight the unvalidated sources of input, the HP Fortify Secure Coding Rulepacks dynamically re-prioritize the issues reported by HP Fortify Static Code Analyzer 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 HP Fortify user with the auditing process, the HP 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.

ResourceInjection.java, line 9 (Path Manipulation) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		

Abstract: Attackers can control the filesystem path argument to File() at ResourceInjection.java line 9, which allows them to access or modify otherwise protected files.

Source: ResourceInjection.java:6 main(0)

```
4
5    public class ResourceInjection {
6        public static void main(String args[]) {
7            ArrayList<String> a1 = new ArrayList<String>();
8            a1.add(args[0]);
```

Sink: ResourceInjection.java:9 java.io.File.File()

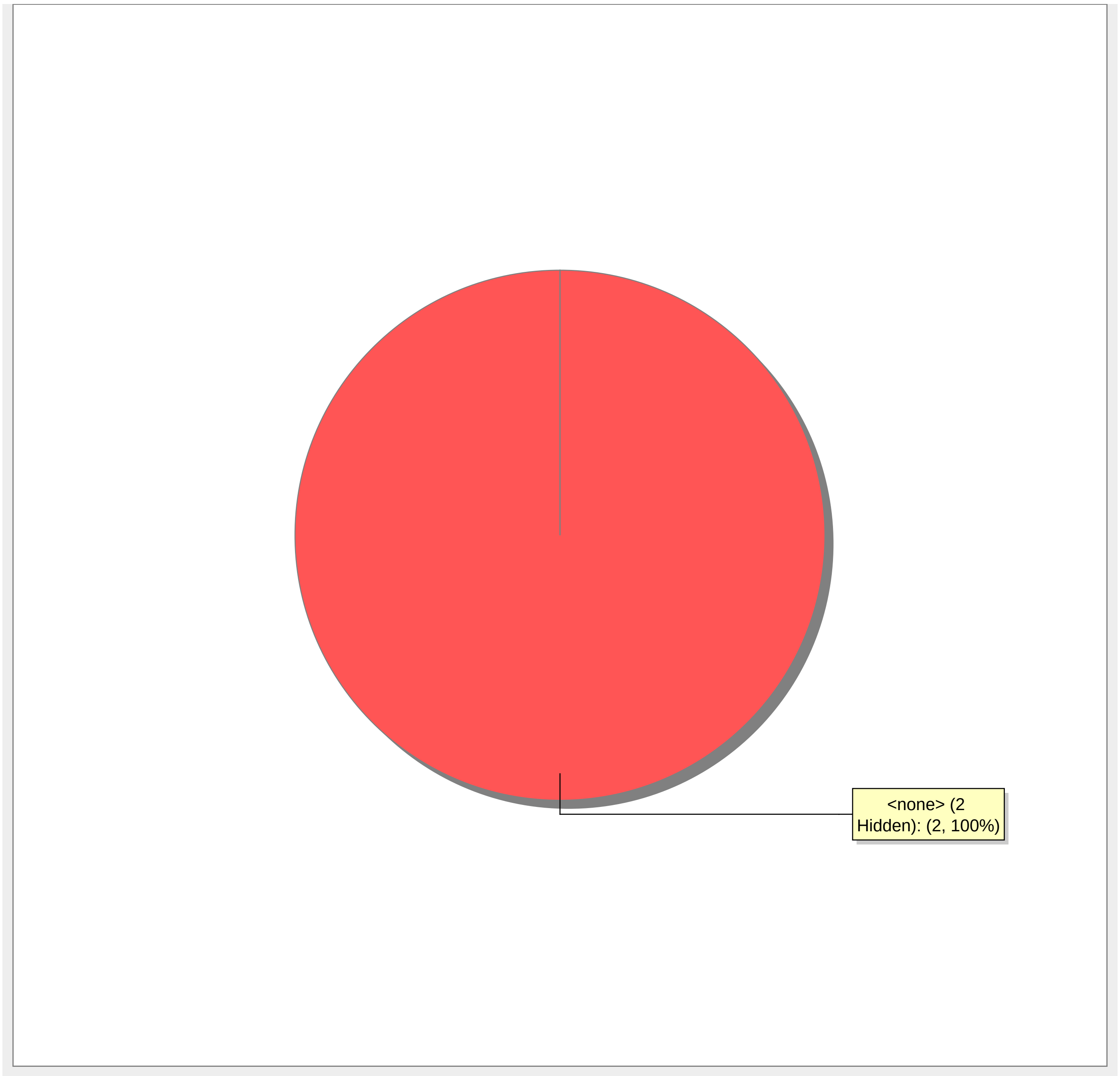
```
7            ArrayList<String> a1 = new ArrayList<String>();
8            a1.add(args[0]);
9            File f1 = new File(a1.get(0));
10       }
11   }
```

Issue Count by Category

Issues by Category	
J2EE Bad Practices: Leftover Debug Code (1 Hidden)	1
Path Manipulation (1 Hidden)	1

Issue Breakdown by Analysis

Issues by Analysis



● <none> (2 Hidden)