Fortify Security Report

Sep 28, 2017

ChandlerAustin

Executive Summary

Issues Overview

On Sep 28, 2017, a source code review was performed over the benchmarks code base. 674 files, 12,108 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 2503 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order

Low (2191 Hidden)	2191
Critical	171
High (141 Hidden)	141

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/task5/benchmarks
Number of Files: 674
Lines of Code: 12108
Build Label: <No Build Label>

Scan Information

Scan time: 02:08
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:
org.mitre.scap.cpe.CPEName.main
org.mitre.scap.xccdf.XCCDFInterpreter.main
stonesoup.FTPServer.main
stonesoup.FileClient.main
stonesoup.FileServer.main
stonesoup.ImageConverter.main
stonesoup.Main.main
stonesoup.Palindrome.main
stonesoup.SolitareEncryptionAlgorithm.main

File System:
java.io.FileInputStream.FileInputStream

Private Information:
null.null.null
java.util.Properties.getProperty

Java Properties:
java.lang.System.getProperties

java.lang.System.getProperty
java.util.Properties.load

Standard Input Stream:
null.null.null

Stream:
java.io.BufferedInputStream.read
java.io.BufferedReader.read
java.io.FileInputStream.read
java.io.FilterInputStream.read
java.io.InputStream.read
java.io.InputStreamReader.read
java.io.Reader.read

System Information:
null.null.null
java.awt.HeadlessException.getMessage
java.lang.Throwable.getLocalizedMessage
java.lang.Throwable.getMessage
java.net.URISyntaxException.getMessage
org.apache.xmlbeans.xml.stream.XMLStreamException.getMessage
org.mitre.scap.xccdf.CircularReferenceException.getMessage
org.mitre.scap.xccdf.ProfileNotFoundException.getMessage
org.mitre.scap.xccdf.check.complexcheck.ComplexCheckException.getMessage
org.xml.sax.SAXException.getMessage

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

Audit guide not enabled

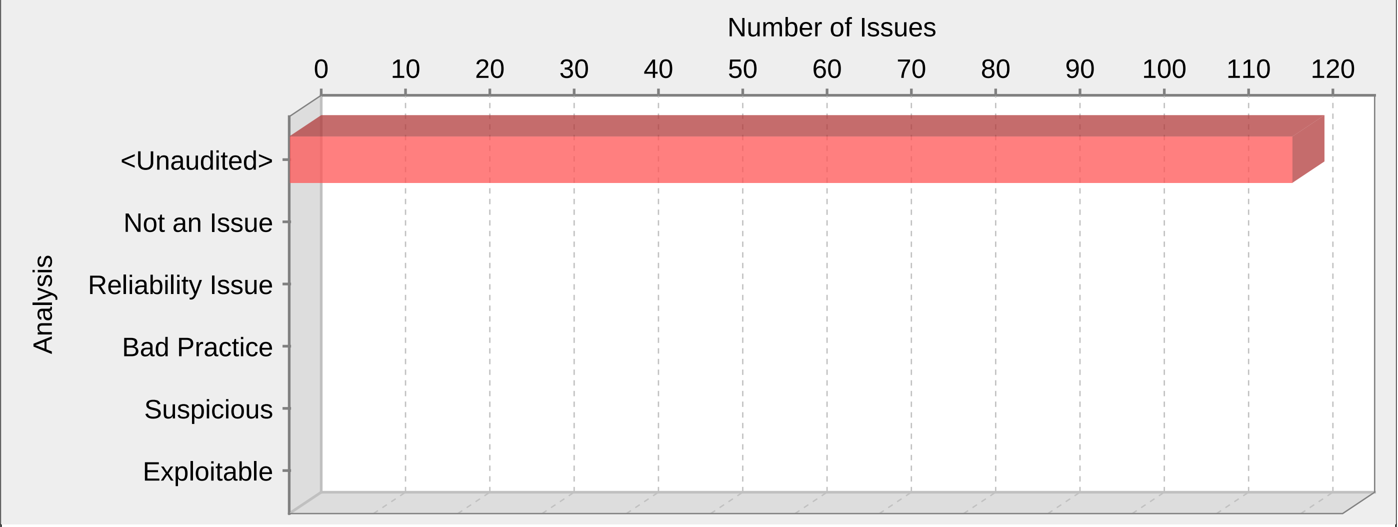
Results Outline

Overall number of results

The scan found 2503 issues.

Vulnerability Examples by Category

Category: Privacy Violation (119 Issues)



Abstract:

Mishandling private information, such as customer passwords or social security numbers, can compromise user privacy and is often illegal.

Explanation:

Privacy violations occur when:

1. Private user information enters the program.

2. The data is written to an external location, such as the console, file system, or network.

Example 1: The following code contains a logging statement that tracks the contents of records added to a database by storing them in a log file. Among other values that are stored, the getPassword() function returns the user-supplied plaintext password associated with the account.

```
pass = getPassword();
...
dbmsLog.println(id+":"+pass+":"+type+":"+tstamp);
```

The code in the example above logs a plaintext password to the filesystem. Although many developers trust the filesystem as a safe storage location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Privacy is one of the biggest concerns in the mobile world for a couple of reasons. One of them is a much higher chance of device loss. The other has to do with inter-process communication between mobile applications. 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 is why application authors need to be careful about what information they include in messages addressed to other applications running on the device. Sensitive information should never be part of inter-process communication between mobile applications.

Example 2: The code below reads username and password for a given site from an Android WebView store and broadcasts them to all the registered receivers.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
String[] credentials = view.getHttpAuthUsernamePassword(host, realm);
String username = credentials[0];
String password = credentials[1];
Intent i = new Intent();
i.setAction("SEND_CREDENTIALS");
i.putExtra("username", username);
i.putExtra("password", password);
view.getContext().sendBroadcast(i);
}
});
...
```

There are several problems with this example. First of all, by default, WebView credentials are stored in plaintext and are not hashed. So if a user has a rooted device (or uses an emulator), she is able to read stored passwords for given sites. Second, plaintext credentials are broadcast to all the registered receivers, which means that any receiver registered to listen to intents with the SEND_CREDENTIALS action will receive the message. The broadcast is not even protected with a permission to limit the number of recipients, even though in this case, we do not recommend using permissions as a fix.

Private data can enter a program in a variety of ways:

- Directly from the user in the form of a password or personal information

- Accessed from a database or other data store by the application

- Indirectly from a partner or other third party

Typically, in the context of the mobile world, this private information would include (along with passwords, SSNs and other general personal information):

- Location

- Cell phone number

- Serial numbers and device IDs

- Network Operator information

- Voicemail information

Sometimes data that is not labeled as private can have a privacy implication in a different context. For example, student identification numbers are usually not considered private because there is no explicit and publicly-available mapping to an individual student's personal information. However, if a school generates identification numbers based on student social security numbers, then the identification numbers should be considered private.

Security and privacy concerns often seem to compete with each other. From a security perspective, you should record all important operations so that any anomalous activity can later be identified. However, when private data is involved, this practice can create risk.

Although there are many ways in which private data can be handled unsafely, a common risk stems from misplaced trust. Programmers often trust the operating environment in which a program runs, and therefore believe that it is acceptable to store private information on the file system, in the registry, or in other locally-controlled resources. However, even if access to certain resources is restricted, this does not guarantee that the individuals who do have access can be trusted. For example, in 2004, an unscrupulous employee at AOL sold approximately 92 million private customer e-mail addresses to a spammer marketing an offshore gambling web site [1].

In response to such high-profile exploits, the collection and management of private data is becoming increasingly regulated. Depending on its location, the type of business it conducts, and the nature of any private data it handles, an organization may be required to comply with one or more of the following federal and state regulations:

- Safe Harbor Privacy Framework [3]

- Gramm-Leach Bliley Act (GLBA) [4]

- Health Insurance Portability and Accountability Act (HIPAA) [5]

- California SB-1386 [6]

Despite these regulations, privacy violations continue to occur with alarming frequency.

This finding is from rules or research contributed by Will Enck. http://www.enck.org/

Recommendations:

When security and privacy demands clash, privacy should usually be given the higher priority. To accomplish this and still maintain required security information, cleanse any private information before it exits the program.

To enforce good privacy management, develop and strictly adhere to internal privacy guidelines. The guidelines should specifically describe how an application should handle private data. If your organization is regulated by federal or state law, ensure that your privacy guidelines are sufficiently strenuous to meet the legal requirements. Even if your organization is not regulated, you must protect private information or risk losing customer confidence.

The best policy with respect to private data is to minimize its exposure. Applications, processes, and employees should not be granted access to any private data unless the access is required for the tasks that they are to perform. Just as the principle of least privilege dictates that no operation should be performed with more than the necessary privileges, access to private data should be restricted to the smallest possible group.

For mobile applications, make sure they never communicate any sensitive data to other applications running on the device. When private data needs to be stored, it should always be encrypted. For Android, as well as any other platform that uses SQLite database, a good option is SQLCipher -- an extension to SQLite database that provides transparent 256-bit AES encryption of database files. Thus, credentials can be stored in an encrypted database.

Example 3: The code below demonstrates how to integrate SQLCipher into an Android application after downloading the necessary binaries, and store credentials into the database file.

```
import net.sqlcipher.database.SQLiteDatabase;
...
SQLiteDatabase.loadLibs(this);
File dbFile = getDatabasePath("credentials.db");
dbFile.mkdirs();
dbFile.delete();
SQLiteDatabase db = SQLiteDatabase.openOrCreateDatabase(dbFile, "credentials", null);
db.execSQL("create table credentials(u, p)");
db.execSQL("insert into credentials(u, p) values(?, ?)", new Object[]{username, password});
...
```

Note that references to android.database.sqlite.SQLiteDatabase are substituted with those of net.sqlcipher.database.SQLiteDatabase.

To enable encryption on the WebView store, WebKit has to be re-compiled with the sqlcipher.so library.

Example 4: The code below reads username and password for a given site from an Android WebView store and instead of broadcasting them to all the registered receivers, it only broadcasts internally so that the broadcast can only be seen by other parts of the same app.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
String[] credentials = view.getHttpAuthUsernamePassword(host, realm);
String username = credentials[0];
String password = credentials[1];
Intent i = new Intent();
i.setAction("SEND_CREDENTIALS");
i.putExtra("username", username);
i.putExtra("password", password);
LocalBroadcastManager.getInstance(view.getContext()).sendBroadcast(i);
}
});
...
```

Tips:

1. As part of any thorough audit for privacy violations, ensure that custom rules have been written to identify all sources of private or otherwise sensitive information entering the program. Most sources of private data cannot be identified automatically. Without custom rules, your check for privacy violations is likely to be substantially incomplete.

2. The Fortify Java Annotations FortifyPassword, FortifyNotPassword, FortifyPrivate and FortifyNotPrivate can be used to indicate which fields and variables represent passwords and private data.

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.

4. Fortify RTA adds protection against this category.

FileClient.java, line 173 (Privacy Violation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		

Abstract: The method sendRequest() in FileClient.java mishandles confidential information, which can compromise user privacy and is often illegal.

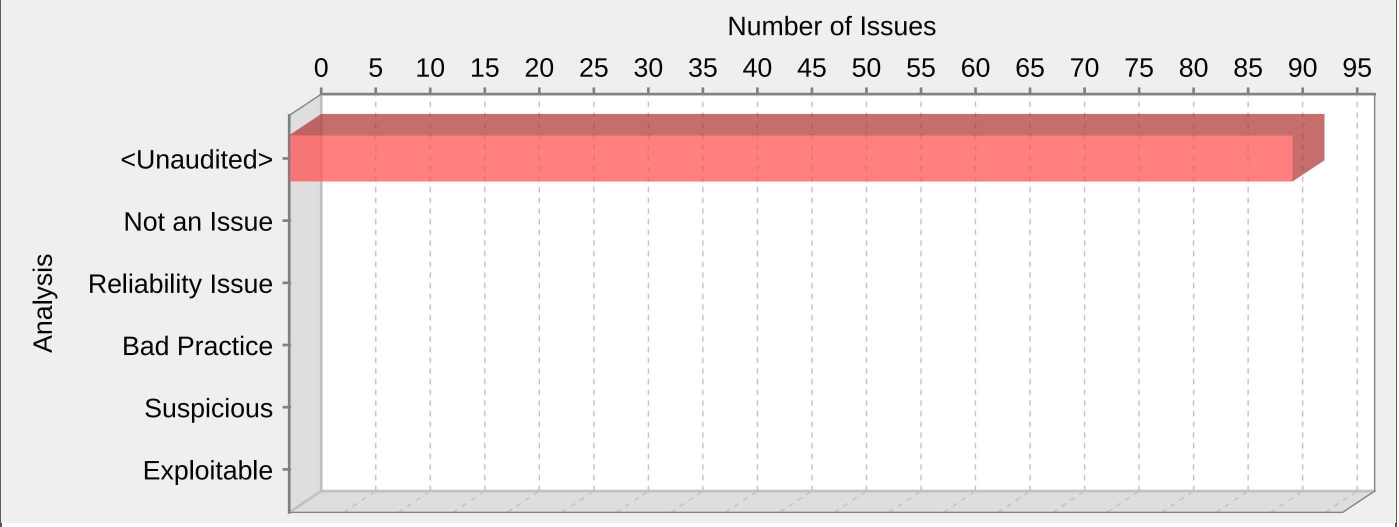
Source: FileClient.java:173 Read passwordLength()

```
171       byte[] passwordBytes = password.toLowerCase().getBytes(Charset.forName("US-ASCII"));
172       byte[] passwordLength = this.encodeLength(passwordBytes.length);
173       connection.getOutputStream().write(passwordLength);
174       connection.getOutputStream().write(passwordBytes);
175
```

Sink: FileClient.java:173 java.io.OutputStream.write()

```
171       byte[] passwordBytes = password.toLowerCase().getBytes(Charset.forName("US-ASCII"));
172       byte[] passwordLength = this.encodeLength(passwordBytes.length);
173       connection.getOutputStream().write(passwordLength);
174       connection.getOutputStream().write(passwordBytes);
175
```

Category: Path Manipulation (92 Issues: 40 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.

FileServer.java, line 408 (Path Manipulation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the filesystem path argument to File() at FileServer.java line 408, which allows them to access or modify otherwise protected files.		

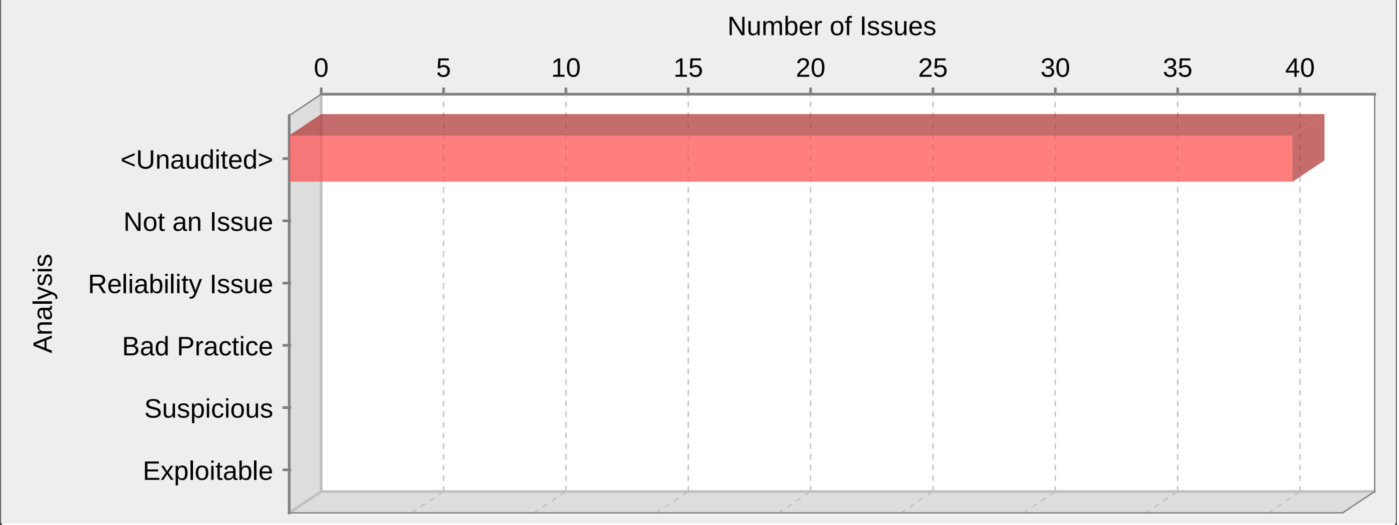
Source: FileServer.java:481 java.io.InputStream.read()

```
479
480         try {
481          while ((readSize = stream.read(buffer, 0, readWant)) != -1) {
482           totalRead += readSize;
483           readWant = length - totalRead;
```

Sink: FileServer.java:408 java.io.File.File()

```
406         //passed security, perform the lookup
407         //first make sure the file exists
408         File file = new File(this.fileRoot + targetUsername.toLowerCase() + File.separator +
targetFilename);
409         if (!file.exists() || !file.isFile() || !file.canRead()) {
410          System.out.printf("Requested file '%s' does not exist.\n", file.getAbsolutePath());
```

Category: Unreleased Resource: Sockets (41 Issues: 41 Hidden)



Abstract:

The program can potentially fail to release a socket.

Explanation:

The program can potentially fail to release a socket.

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, but 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 1: The following method never closes the socket it opens. In a busy environment, this can result in the JVM using up all of its sockets.

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException
{
Socket sock = new Socket(host, port);
BufferedReader reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
while ((String socketData = reader.readLine()) != null) {
System.out.println(socketData);
}
}
```

Example 2: Under normal conditions, the following fix properly closes the socket and any associated streams. But if an exception occurs while reading the input or writing the data to screen, the socket object will not be closed. If this happens often enough, the system will run out of sockets and not be able to handle any further connections.

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException
{
Socket sock = new Socket(host, port);
BufferedReader reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));
while ((String socketData = reader.readLine()) != null) {
System.out.println(socketData);
}
sock.close();
}
```

Recommendations:

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

```
private void echoSocket(String host, int port) throws UnknownHostException, SocketException, IOException
{
```

```
Socket sock;
BufferedReader reader;

try {
sock = new Socket(host, port);
reader = new BufferedReader(new InputStreamReader(sock.getInputStream()));

while ((String socketData = reader.readLine()) != null) {
System.out.println(socketData);
}
}
finally {
safeClose(sock);
}
}

public static void safeClose(Socket s) {
if (s != null && !s.isClosed()) {
try {
s.close();
} catch (IOException e) {
log(e);
}
}
}
```

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

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

Tips:

1. Closing a socket also closes any streams obtained via getInputStream and getOutputStream. Conversely, closing any of the socket's streams also closes the entire socket. When in doubt, it is always safer to close both explicitly.

FileClient.java, line 153 (Unreleased Resource: Sockets) [Hidden]

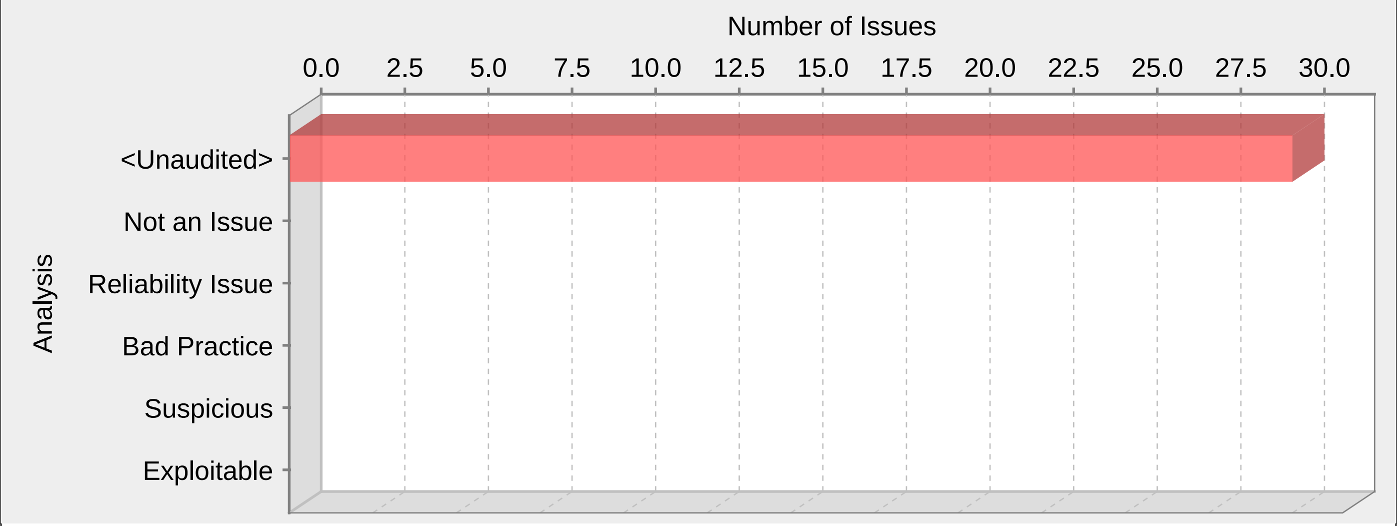
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The function downloadFile() in FileClient.java sometimes fails to release a socket allocated by <a href="location://tc_java_190_v01/src/stonesoup/FileClient.java###336###0###0">createSocketAndConnect()</a> on line 153.

Sink: FileClient.java:153 connection = createSocketAndConnect(...)

```
151
152      private void downloadFile(InetAddress address, int port, String username, String password, String filename, String fileOwner, boolean useSSL) throws Exception {
153        Socket connection = this.createSocketAndConnect(address, port, useSSL);
154
155        this.sendRequest(connection, username, password, filename, fileOwner);
```

Category: Unreleased Resource: Streams (30 Issues: 30 Hidden)



Abstract:

The program can potentially fail to release a system resource.

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, but 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.

FileServer.java, line 275 (Unreleased Resource: Streams) [Hidden]

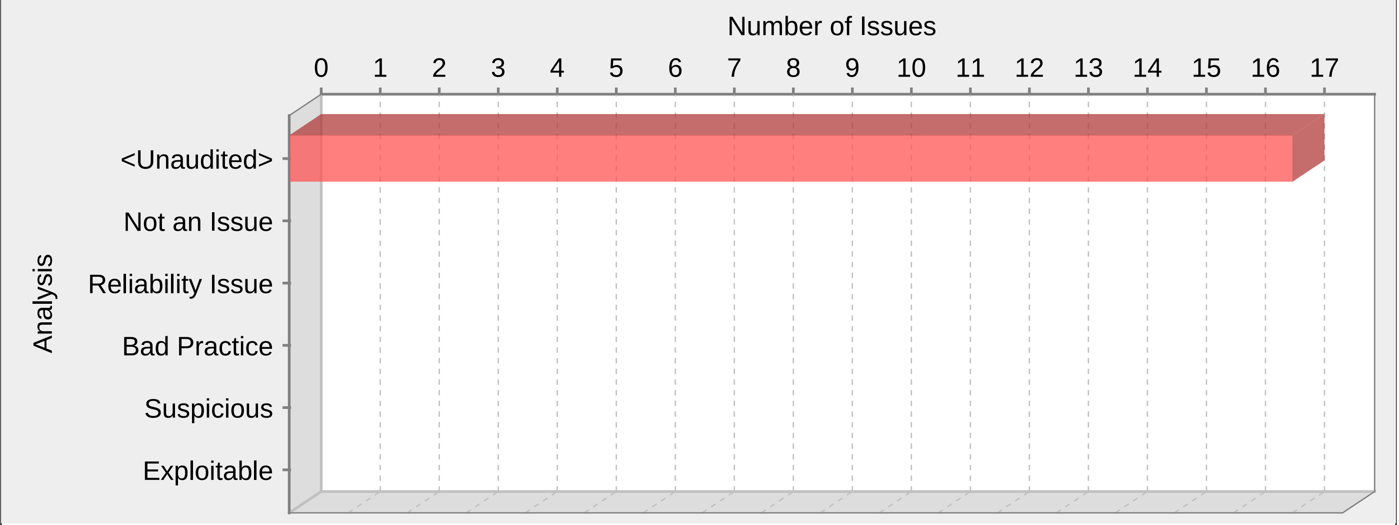
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The function readConfiguration() in FileServer.java sometimes fails to release a system resource allocated by FileInputStream() on line 275.

Sink: FileServer.java:275 fstream = new FileInputStream(...)

```
273    FileInputStream fstream = null;
274    try {
275     fstream = new FileInputStream(file);
276    } catch (FileNotFoundException e) {
277     System.out.printf("Configuration file '%s' is not accessible.\n", filename);
```

Category: Often Misused: Authentication (17 Issues: 17 Hidden)



Abstract:

Attackers can spoof DNS entries. Do not rely on DNS names for security.

Explanation:

Many DNS servers are susceptible to spoofing attacks, so you should assume that your software will someday run in an environment with a compromised DNS server. If attackers are allowed to make DNS updates (sometimes called DNS cache poisoning), they can route your network traffic through their machines or make it appear as if their IP addresses are part of your domain. Do not base the security of your system on DNS names.

Example: The following code uses a DNS lookup to determine whether an inbound request is from a trusted host. If an attacker can poison the DNS cache, they can gain trusted status.

```
String ip = request.getRemoteAddr();
InetAddress addr = InetAddress.getByName(ip);
if (addr.getCanonicalHostName().endsWith("trustme.com")) {
trusted = true;
}
```

IP addresses are more reliable than DNS names, but they can also be spoofed. Attackers can easily forge the source IP address of the packets they send, but response packets will return to the forged IP address. To see the response packets, the attacker has to sniff the traffic between the victim machine and the forged IP address. In order to accomplish the required sniffing, attackers typically attempt to locate themselves on the same subnet as the victim machine. Attackers may be able to circumvent this requirement by using source routing, but source routing is disabled across much of the Internet today. In summary, IP address verification can be a useful part of an authentication scheme, but it should not be the single factor required for authentication.

Recommendations:

You can increase confidence in a domain name lookup if you check to make sure that the host's forward and backward DNS entries match. Attackers will not be able to spoof both the forward and the reverse DNS entries without controlling the nameservers for the target domain. This is not a foolproof approach however: attackers may be able to convince the domain registrar to turn over the domain to a malicious nameserver. Basing authentication on DNS entries is simply a risky proposition.

While no authentication mechanism is foolproof, there are better alternatives than host-based authentication. Password systems offer decent security, but are susceptible to bad password choices, insecure password transmission, and bad password management. A cryptographic scheme like SSL is worth considering, but such schemes are often so complex that they bring with them the risk of significant implementation errors, and key material can always be stolen. In many situations, multi-factor authentication including a physical token offers the most security available at a reasonable price.

Tips:

1. Check how the DNS information is being used. In addition to considering whether or not the program's authentication mechanisms can be defeated, consider how DNS spoofing can be used in a social engineering attack. For example, if attackers can make it appear that a posting came from an internal machine, can they gain credibility?

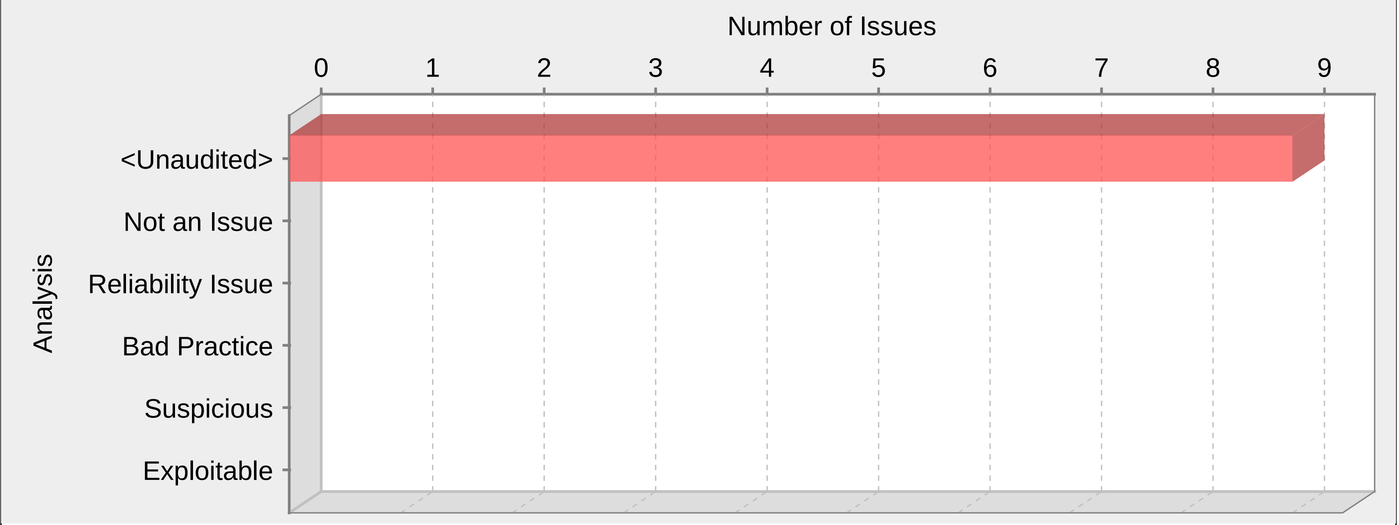
FileClient.java, line 84 (Often Misused: Authentication) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	API Abuse		
Abstract:	The information returned by the call to getByName() is not trustworthy. Attackers can spoof DNS entries. Do not rely on DNS for security.		
Sink:	FileClient.java:84 getByName()		

82

```
try {
 result = InetAddress.getByName(address);
} catch (UnknownHostException e) {
 System.out.printf("Invalid or unknown bind address '%s'.\n", address);
```

Category: Null Dereference (9 Issues: 9 Hidden)



Abstract:

The program can potentially dereference a null pointer, thereby causing a null pointer exception.

Explanation:

Null pointer exceptions usually occur when one or more of the programmer's assumptions is violated. A dereference-after-store error occurs when a program explicitly sets an object to null and dereferences it later. This error is often the result of a programmer initializing a variable to null when it is declared.

Most null pointer issues result in general software reliability problems, but if attackers can intentionally trigger a null pointer dereference, they can use the resulting exception to bypass security logic or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example: In the following code, the programmer explicitly sets the variable foo to null. Later, the programmer dereferences foo before checking the object for a null value.

```
Foo foo = null;
...
foo.setBar(val);
...
}
```

Recommendations:

Implement careful checks before dereferencing objects that might be null. When possible, abstract null checks into wrappers around code that manipulates resources to ensure that they are applied in all cases and to minimize the places where mistakes can occur.

Authenticator.java, line 73 (Null Dereference) [Hidden]

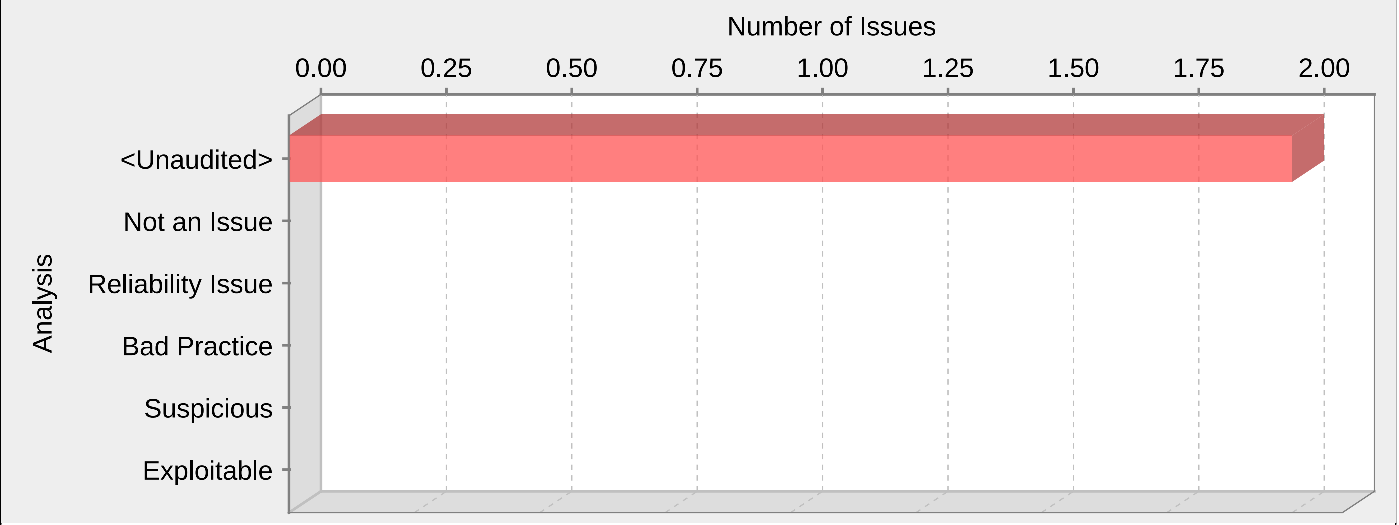
Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		

Abstract: The method hashPassword() in Authenticator.java can crash the program by dereferencing a null pointer on line 73.

Sink: Authenticator.java:73 Dereferenced : md()

```
71          byte[] passwordBytes = password.getBytes(Charset.forName("US-ASCII"));
72
73          md.update(passwordBytes);
74          byte[] passwordHash = md.digest();
75          md = null;
```

Category: Command Injection (2 Issues: 2 Hidden)



Abstract:

Executing commands from an untrusted source or in an untrusted environment can cause an application to execute malicious commands on behalf of an attacker.

Explanation:

Command injection vulnerabilities take two forms:

- An attacker can change the command that the program executes: the attacker explicitly controls what the command is.

- An attacker can change the environment in which the command executes: the attacker implicitly controls what the command means.

In this case we are primarily concerned with the first scenario, the possibility that an attacker may be able to control the command that is executed. Command injection vulnerabilities of this type occur when:

1. Data enters the application from an untrusted source.

2. The data is used as or as part of a string representing a command that is executed by the application.

3. By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

Example 1: The following code from a system utility uses the system property APPHOME to determine the directory in which it is installed and then executes an initialization script based on a relative path from the specified directory.

```
...
String home = System.getProperty("APPHOME");
String cmd = home + INITCMD;
java.lang.Runtime.getRuntime().exec(cmd);
...
```

The code in Example 1 allows an attacker to execute arbitrary commands with the elevated privilege of the application by modifying the system property APPHOME to point to a different path containing a malicious version of INITCMD. Because the program does not validate the value read from the environment, if an attacker can control the value of the system property APPHOME, then they can fool the application into running malicious code and take control of the system.

Example 2: The following code is from an administrative web application designed to allow users to kick off a backup of an Oracle database using a batch-file wrapper around the rman utility and then run a cleanup.bat script to delete some temporary files. The script rmanDB.bat accepts a single command line parameter, which specifies the type of backup to perform. Because access to the database is restricted, the application runs the backup as a privileged user.

```
...
String btype = request.getParameter("backuptype");
String cmd = new String("cmd.exe /K
\"c:\\util\\rmanDB.bat "+btype+"&&c:\\util\\cleanup.bat\"")
System.Runtime.getRuntime().exec(cmd);
...
```

The problem here is that the program does not do any validation on the backuptype parameter read from the user. Typically the Runtime.exec() function will not execute multiple commands, but in this case the program first runs the cmd.exe shell in order to run multiple commands with a single call to Runtime.exec(). Once the shell is invoked, it will allow for the execution of multiple commands separated by two ampersands. If an attacker passes a string of the form "&& del c:\\dbms\\*.*", then the application will execute this command along with the others specified by the program. Because of the nature of the application, it runs with the privileges necessary to interact with the database, which means whatever command the attacker injects will run with those privileges as well.

Example 3: The following code is from a web application that allows users access to an interface through which they can update their password on the system. Part of the process for updating passwords in certain network environments is to run a make command in the /var/yp directory, the code for which is shown below.

```
...
System.Runtime.getRuntime().exec("make");
...
```

The problem here is that the program does not specify an absolute path for make and fails to clean its environment prior to executing the call to Runtime.exec(). If an attacker can modify the $PATH variable to point to a malicious binary called make and cause the program to be executed in their environment, then the malicious binary will be loaded instead of the one intended. Because of the nature of the application, it runs with the privileges necessary to perform system operations, which means the attacker's make will now be run with these privileges, possibly giving the attacker complete control of the system.

Some think that in the mobile world, classic vulnerabilities, such as command injection, do not make sense -- why would a user attack him or herself? 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 4: The following code reads commands to be executed from an Android intent.

```
...
String[] cmds = this.getIntent().getStringArrayExtra("commands");
Process p = Runtime.getRuntime().exec("su");
DataOutputStream os = new DataOutputStream(p.getOutputStream());
for (String cmd : cmds) {
os.writeBytes(cmd+"\n");
}
os.writeBytes("exit\n");
os.flush();
...
```

On a rooted device, a malicious application can force a victim application to execute arbitrary commands with super user privileges.

This category was derived from the Cigital Java Rulepack. http://www.cigital.com/

Recommendations:

Do not allow users to have direct control over the commands executed by the program. In cases where user input must affect the command to be run, use the input only to make a selection from a predetermined set of safe commands. If the input appears to be malicious, the value passed to the command execution function should either default to some safe selection from this set or the program should decline to execute any command at all.

In cases where user input must be used as an argument to a command executed by the program, this approach often becomes impractical because the set of legitimate argument values is too large or too hard to keep track of. Developers often fall back on blacklisting in these situations. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. Any list of unsafe characters is likely to be incomplete and will be heavily dependent on the system where the commands are executed. A better approach is to create a whitelist of characters that are allowed to appear in the input and accept input composed exclusively of characters in the approved set.

An attacker can indirectly control commands executed by a program by modifying the environment in which they are executed. The environment should not be trusted and precautions should be taken to prevent an attacker from using some manipulation of the environment to perform an attack. Whenever possible, commands should be controlled by the application and executed using an absolute path. In cases where the path is not known at compile time, such as for cross-platform applications, an absolute path should be constructed from trusted values during execution. Command values and paths read from configuration files or the environment should be sanity-checked against a set of invariants that define valid values.

Other checks can sometimes be performed to detect if these sources may have been tampered with. For example, if a configuration file is world-writable, the program might refuse to run. In cases where information about the binary to be executed is known in advance, the program may perform checks to verify the identity of the binary. If a binary should always be owned by a particular user or have a particular set of access permissions assigned to it, these properties can be verified programmatically before the binary is executed.

Although it may be impossible to completely protect a program from an imaginative attacker bent on controlling the commands the program executes, be sure to apply the principle of least privilege wherever the program executes an external command: do not hold privileges that are not essential to the execution of the command.

Tips:

1. 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.

JavaJarExecutor.java, line 164 (Command Injection) [Hidden]

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

Abstract: The method execute() in JavaJarExecutor.java calls ProcessBuilder() with a command built from untrusted data. This call can cause the program to execute malicious commands on behalf of an attacker.

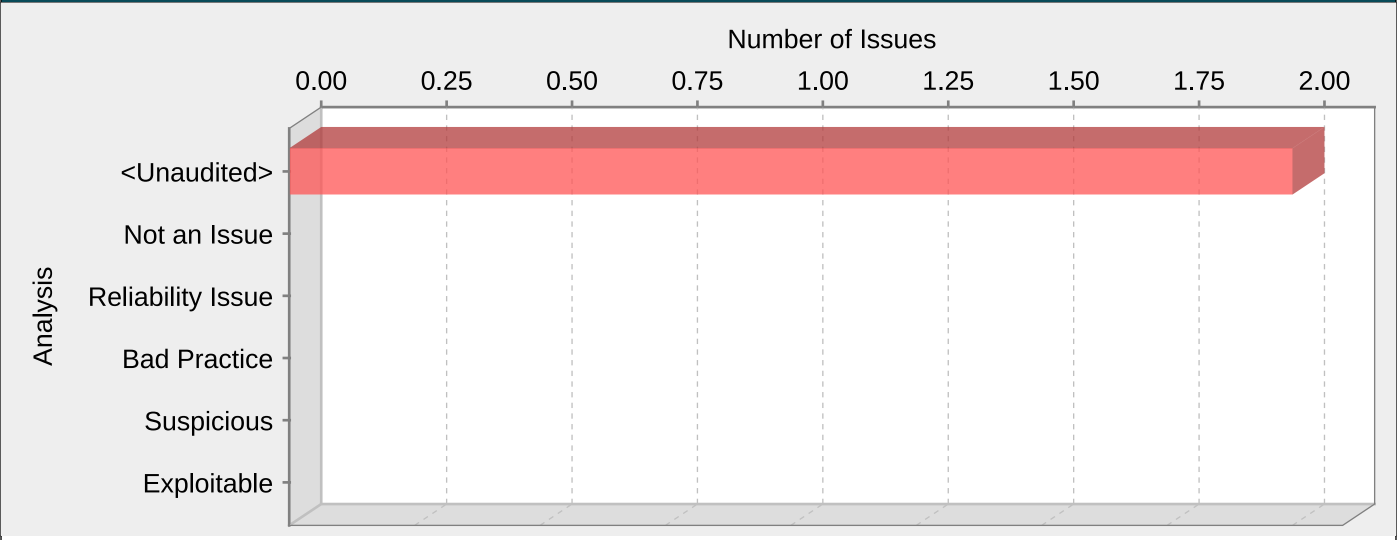
Source: JavaJarExecutor.java:52 java.lang.System.getProperty()

```
50
51    public class JavaJarExecutor {
52      private static final String JAVA_HOME           = System.getProperty( "java.home" );
53      private static final String JAVA_BIN            = JAVA_HOME + File.separator + "bin"
      + File.separator + "java";
54      private static final String JAVA_CLASSPATH_ARG  = "-cp";
```

Sink: JavaJarExecutor.java:164 java.lang.ProcessBuilder.ProcessBuilder()

```
162     public Process execute() throws IOException {
163       List<String>    cmdList = this.buildExecString();
164       ProcessBuilder  pb      = new ProcessBuilder( cmdList );
165
166       if( this.workingDirectory != null ){
```

Category: Poor Error Handling: Return Inside Finally (2 Issues: 2 Hidden)



Abstract:

Returning from inside a finally block will cause exceptions to be lost.

Explanation:

A return statement inside a finally block will cause any exception that might be thrown in the try block to be discarded.

Example 1: In the following code excerpt, the MagicException thrown by the second call to doMagic with true passed to it will never be delivered to the caller. The return statement inside the finally block will cause the exception to be discarded.

```
public class MagicTrick {

public static class MagicException extends Exception { }

public static void main(String[] args) {

System.out.println("Watch as this magical code makes an " +
"exception disappear before your very eyes!");

System.out.println("First, the kind of exception handling " +
"you're used to:");
try {
doMagic(false);
} catch (MagicException e) {
// An exception will be caught here
e.printStackTrace();
}

System.out.println("Now, the magic:");
try {
doMagic(true);
} catch (MagicException e) {
// No exception caught here, the finally block ate it
e.printStackTrace();
}
System.out.println("tada!");
}

public static void doMagic(boolean returnFromFinally)
throws MagicException {

try {
throw new MagicException();
}
finally {
if (returnFromFinally) {
return;
}
}
}

}
```

Recommendations:

Move the return statement outside the finally block. If a value from inside the finally block must be returned, simply assign it to a local variable and return this value after the finally block executes.

Tips:

1. HP Fortify Static Code Analyzer will detect explicit throws as well as calls to methods that throw uncaught exceptions in finally blocks.

ISBNChecker.java, line 83 (Poor Error Handling: Return Inside Finally) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Errors		

Abstract: The method isValidISBN() in ISBNChecker.java returns from inside a finally block on line 83, which will cause exceptions to be lost.

Sink: ISBNChecker.java:83 ReturnStatement()

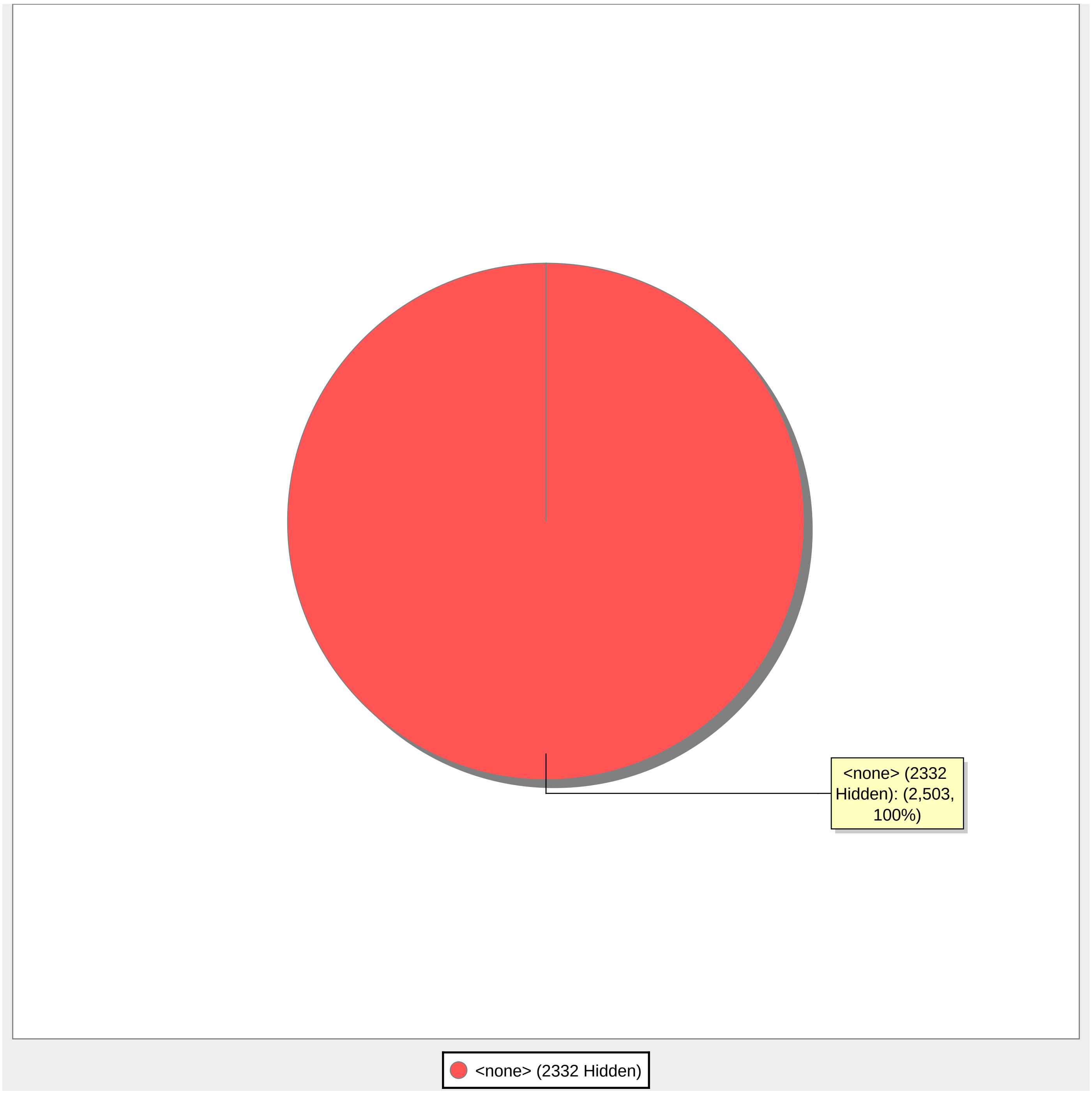
```
81                  // Validate check digit
82                  if (isbnInts[12] == (10 - ((isbnInts[0]+3*isbnInts[1]+isbnInts[2]+3*isbnInts[3]+isbnInts[4]+3*isbnInts[5]+isbnInts[6]+3*isbnInts[7]+isbnInts[8]+3*isbnInts[9]+isbnInts[10]+3*isbnInts[11]) % 10)) % 10) {
83                      return true;
84                  } else {
85                      return false;
```

Issue Count by Category

Issues by Category	
Poor Logging Practice: Use of a System Output Stream (1109 Hidden)	1109
J2EE Bad Practices: JVM Termination (331 Hidden)	331
Poor Error Handling: Overly Broad Throws (145 Hidden)	145
Privacy Violation	119
Dead Code: Unused Method (104 Hidden)	104
Path Manipulation (40 Hidden)	92
J2EE Bad Practices: Leftover Debug Code (68 Hidden)	68
System Information Leak: Internal (63 Hidden)	63
Denial of Service (58 Hidden)	58
Unreleased Resource: Sockets (41 Hidden)	41
J2EE Bad Practices: Sockets (37 Hidden)	37
Poor Error Handling: Empty Catch Block (33 Hidden)	33
System Information Leak (33 Hidden)	33
Poor Error Handling: Overly Broad Catch (31 Hidden)	31
Unreleased Resource: Streams (30 Hidden)	30
J2EE Bad Practices: Threads (29 Hidden)	29
Password Management: Password in Comment (28 Hidden)	28
Code Correctness: Class Does Not Implement equals (26 Hidden)	26
Poor Style: Value Never Read (20 Hidden)	20
Often Misused: Authentication (17 Hidden)	17
Poor Error Handling: Throw Inside Finally (12 Hidden)	12
Null Dereference (9 Hidden)	9
Dead Code: Expression is Always true (7 Hidden)	7
Password Management: Null Password (7 Hidden)	7
Poor Logging Practice: Logger Not Declared Static Final (7 Hidden)	7
Weak Cryptographic Hash (7 Hidden)	7
Code Correctness: Class Does Not Implement Cloneable (6 Hidden)	6
Missing Check against Null (6 Hidden)	6
Object Model Violation: Erroneous clone() Method (6 Hidden)	6
Command Injection (3 Hidden)	3
Denial of Service: Parse Double (3 Hidden)	3
Code Correctness: Misleading Method Signature (2 Hidden)	2
Dead Code: Expression is Always false (2 Hidden)	2
Object Model Violation: Just one of equals() and hashCode() Defined (2 Hidden)	2
Poor Error Handling: Return Inside Finally (2 Hidden)	2
Poor Style: Non-final Public Static Field (2 Hidden)	2
Unchecked Return Value (2 Hidden)	2
Poor Error Handling: Program Catches NullPointerException (1 Hidden)	1
Poor Style: Redundant Initialization (1 Hidden)	1
Race Condition: Format Flaw (1 Hidden)	1
Resource Injection (1 Hidden)	1

Issue Breakdown by Analysis

Issues by Analysis


<none> (2332 Hidden): (2,503, 100%)
● <none> (2332 Hidden)