Fortify Developer Workbook

Nov 6, 2012

565075

Report Overview

Report Summary

On Nov 6, 2012, a source code review was performed over the ConfApp code base. 103 files, 2,322 LOC (Executable) were scanned. A total of 146 issues were uncovered during the analysis. This report provides a comprehensive description of all the types of issues found in this project. Specific examples and source code are provided for each issue type.

Issues by Fortify Priority Order	
High (1 Hidden)	88
Low (13 Hidden)	46
Critical (1 Suppressed)	12

Issue Summary

Overall number of results

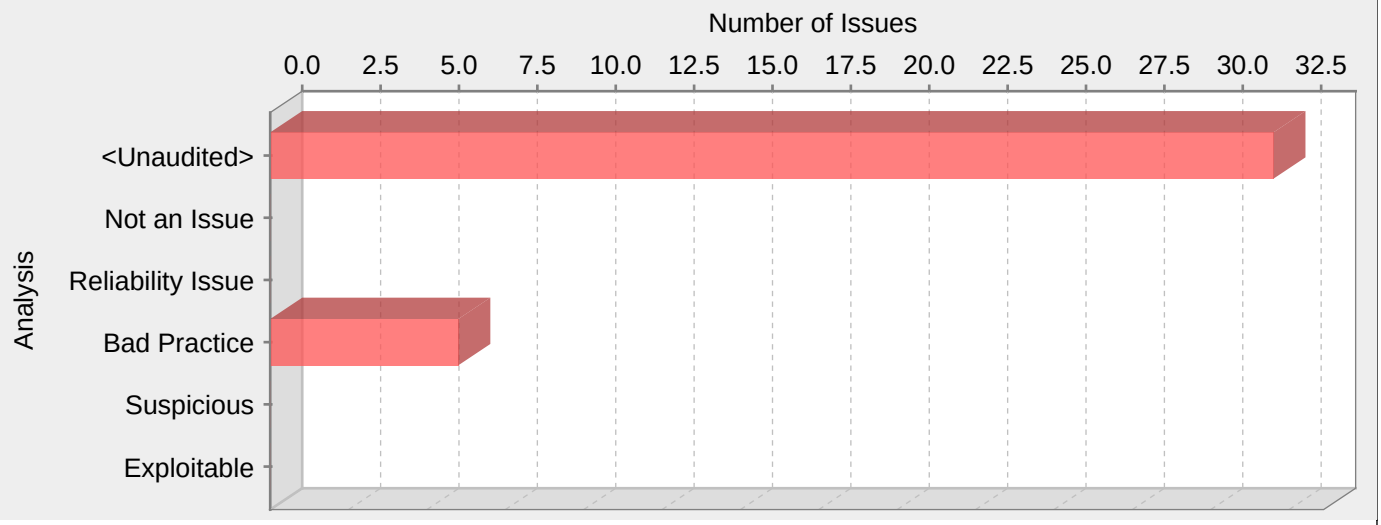
The scan found 146 issues.

Issues by Category	
Android Bad Practices: Missing Component Permission	38
Android Bad Practices: Missing exported Attribute	38
Privacy Violation (1 Suppressed)	11
System Information Leak	10
SQL Injection	9
Poor Style: Value Never Read (7 Hidden)	7
Poor Error Handling: Empty Catch Block	6
Access Control: Database	4
Poor Logging Practice: Use of a System Output Stream	4
Denial of Service	3
Redundant Null Check (3 Hidden)	3
Privilege Management: Android Data Storage	2
Privilege Management: Android Network	2
Privilege Management: Unnecessary Permission	2
Cross-Site Scripting: Persistent	1
Dead Code: Expression is Always true (1 Hidden)	1
Dead Code: Unused Method (1 Hidden)	1
Log Forging	1
Poor Error Handling: Overly Broad Catch	1
Poor Style: Non-final Public Static Field (1 Hidden)	1
Unreleased Resource: Streams (1 Hidden)	1

Results Outline

Vulnerability Examples by Category

Category: Android Bad Practices: Missing Component Permission (38 Issues)



Abstract:

The program does not explicitly assign an access permission to a component.

Explanation:

Any application can access public components that are not explicitly assigned an access permission in their manifest definition.

Example 1: Below is an example of a broadcast receiver declared without an explicit access permission.

<receiver android:name=".BroadcastReceiver"/>

Recommendations:

Components without explicit access permissions should be exceptions. Instead, developers should protect components from misuse by malicious applications by explicitly defining access permissions in the manifest file.

Example 2: Below is the declaration of the broadcast receiver from above re-written with explicitly assigned access permission.

<receiver android:name=".BroadcastReceiver" android:permission="receiver.permission.ACCESS_RECEIVER"/>

AndroidManifest.xml, line 23 (Android Bad Practices: Missing Component Permission)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 23 of AndroidManifest.xml the program does not explicitly assign an access permission to a component.		
Sink:	AndroidManifest.xml:23 null()		

```
21          <activity android:name=".MenuItems.Bio"></activity>
22          <activity android:name=".MenuItems.Favorites"></activity>
23          <activity android:name=".MenuItems.Notes"></activity>
24          <activity android:name=".MenuItems.Schedule"></activity>
25          <activity android:name=".MenuItems.ScheduleDetails"></activity>
```

AndroidManifest.xml, line 14 (Android Bad Practices: Missing Component Permission)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 14 of AndroidManifest.xml the program does not explicitly assign an access permission to a component.		
Sink:	AndroidManifest.xml:14 null()		

```
12          android:theme="@android:style/Theme.NoTitleBar" android:name="ConfApp">
13          <activity
14              android:label="@string/title_activity_main" android:name="MainActivity">
15              <intent-filter>
16                  <action android:name="android.intent.action.MAIN" />
```

Analysis: Bad Practice

AndroidManifest.xml, line 24 (Android Bad Practices: Missing Component Permission)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 24 of AndroidManifest.xml the program does not explicitly assign an access permission to a component.		
Sink:	AndroidManifest.xml:24 null()		

```
22          <activity android:name=".MenuItems.Favorites"></activity>
23          <activity android:name=".MenuItems.Notes"></activity>
24          <activity android:name=".MenuItems.Schedule"></activity>
25          <activity android:name=".MenuItems.ScheduleDetails"></activity>
26          <activity android:name=".MenuItems.Map"></activity>
```

AndroidManifest.xml, line 21 (Android Bad Practices: Missing Component Permission)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 21 of AndroidManifest.xml the program does not explicitly assign an access permission to a component.		
Sink:	AndroidManifest.xml:21 null()		

```
19              </intent-filter>
20          </activity>
21          <activity android:name=".MenuItems.Bio"></activity>
22          <activity android:name=".MenuItems.Favorites"></activity>
23          <activity android:name=".MenuItems.Notes"></activity>
```

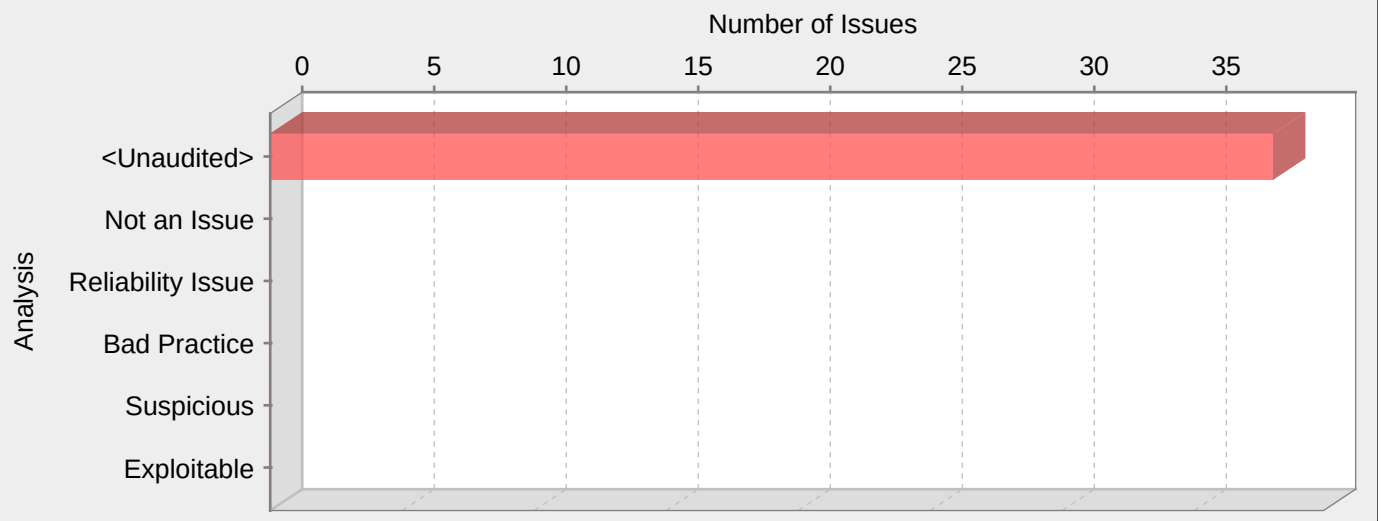
Analysis: Bad Practice

AndroidManifest.xml, line 22 (Android Bad Practices: Missing Component Permission)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 22 of AndroidManifest.xml the program does not explicitly assign an access permission to a component.		
Sink:	AndroidManifest.xml:22 null()		

```
20          </activity>
21          <activity android:name=".MenuItems.Bio"></activity>
22          <activity android:name=".MenuItems.Favorites"></activity>
23          <activity android:name=".MenuItems.Notes"></activity>
24          <activity android:name=".MenuItems.Schedule"></activity>
```

Analysis: Bad Practice

Category: Android Bad Practices: Missing exported Attribute (38 Issues)



Abstract:

The program does not explicitly set the exported attribute on a component.

Explanation:

Some components should always be private by not allowing other applications to access them. The release of v0.9r1 Android SDK introduced the notion of private components. There are two ways to decide whether the component is private. One way is to rely on the framework that follows certain inference rules. Another is to explicitly define the exported attribute on the component.

Example 1: Below is an example of an activity declared without explicitly setting the exported attribute.

<activity android:name="AndroidActivity"/>

Recommendations:

It is always a good idea to declare private components by explicitly setting the exported attribute because it avoids ambiguity.

Example 2: Below is the declaration of the activity from above re-written with the exported attribute explicitly set.

<activity android:name="AndroidActivity" android:exported="false"/>

AndroidManifest.xml, line 21 (Android Bad Practices: Missing exported Attribute)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 21 of AndroidManifest.xml the program does not explicitly set the exported attribute on a component.		

Sink: AndroidManifest.xml:21 null()

```
19                  </intent-filter>
20              </activity>
21              <activity android:name=".MenuItems.Bio"></activity>
22              <activity android:name=".MenuItems.Favorites"></activity>
23              <activity android:name=".MenuItems.Notes"></activity>
```

AndroidManifest.xml, line 14 (Android Bad Practices: Missing exported Attribute)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 14 of AndroidManifest.xml the program does not explicitly set the exported attribute on a component.		

Sink: AndroidManifest.xml:14 null()

```
12              android:theme="@android:style/Theme.NoTitleBar" android:name="ConfApp">
13              <activity
14                  android:label="@string/title_activity_main" android:name="MainActivity">
15                  <intent-filter>
16                      <action android:name="android.intent.action.MAIN" />
```

AndroidManifest.xml, line 22 (Android Bad Practices: Missing exported Attribute)

Fortify Priority:	High	**Folder**	High

Kingdom:	Security Features
Abstract:	On line 22 of AndroidManifest.xml the program does not explicitly set the exported attribute on a component.
Sink:	AndroidManifest.xml:22 null()

```
20              </activity>
21              <activity android:name=".MenuItems.Bio"></activity>
22              <activity android:name=".MenuItems.Favorites"></activity>
23              <activity android:name=".MenuItems.Notes"></activity>
24              <activity android:name=".MenuItems.Schedule"></activity>
```

AndroidManifest.xml, line 23 (Android Bad Practices: Missing exported Attribute)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 23 of AndroidManifest.xml the program does not explicitly set the exported attribute on a component.		
Sink:	AndroidManifest.xml:23 null()		

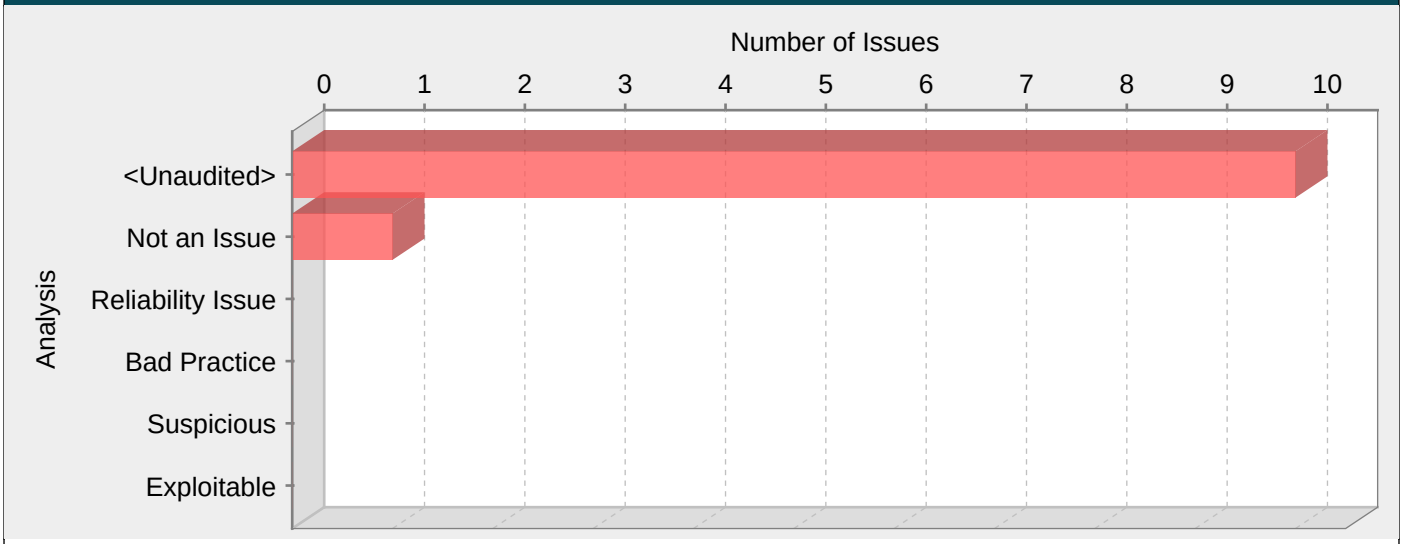
```
21              <activity android:name=".MenuItems.Bio"></activity>
22              <activity android:name=".MenuItems.Favorites"></activity>
23              <activity android:name=".MenuItems.Notes"></activity>
24              <activity android:name=".MenuItems.Schedule"></activity>
25              <activity android:name=".MenuItems.ScheduleDetails"></activity>
```

AndroidManifest.xml, line 24 (Android Bad Practices: Missing exported Attribute)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	On line 24 of AndroidManifest.xml the program does not explicitly set the exported attribute on a component.		
Sink:	AndroidManifest.xml:24 null()		

```
22              <activity android:name=".MenuItems.Favorites"></activity>
23              <activity android:name=".MenuItems.Notes"></activity>
24              <activity android:name=".MenuItems.Schedule"></activity>
25              <activity android:name=".MenuItems.ScheduleDetails"></activity>
26              <activity android:name=".MenuItems.Map"></activity>
```

Category: Privacy Violation (11 Issues: 1 Suppressed)



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: 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.

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

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 in fact 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.

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.

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 Struts 2 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 Fortify 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.

BioDetails.java, line 40 (Privacy Violation) [Suppressed]

Fortify Priority:	Critical	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method initialize() in BioDetails.java mishandles confidential information, which can compromise user privacy and is often illegal.		

Source: Local.java:447 android.database.sqlite.SQLiteDatabase.rawQuery()

```
445   if(name!=null){
446    String statement = "SELECT * FROM Bio WHERE Name=?";
447    Cursor cursor = ourDatabase.rawQuery(statement, new String[]{name});
448    for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
449     bioItem = new BioItem(cursor);
```

Sink: BioDetails.java:40 android.widget.TextView.setText()

```
38          title.setText(bioItem.title);
39         if(name!=null)
40          name.setText(bioItem.name);
41         if(image!=null){
42        String blah = bioItem.image;
```

TouchImageView.java, line 360 (Privacy Violation)

Fortify Priority:	Critical	**Folder**	Low
Kingdom:	Security Features		
Abstract:	The method doWork() in TouchImageView.java mishandles confidential information, which can compromise user privacy and is often illegal.		

Source: Local.java:555 android.database.sqlite.SQLiteDatabase.rawQuery()

```
553   public  Exhibitor getExhibitor(BoothLocation boothLocation){
554    String statement = "SELECT * FROM Exhibitor WHERE boothname=?";
555    Cursor cursor = ourDatabase.rawQuery(statement, new
String[]{""+boothLocation.boothName});
556    Exhibitor exhibitor = null;
557    for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
```

Sink: TouchImageView.java:360 android.widget.Toast.setText()

```
358                     if(toast==null)
359                      toast = toast.makeText(context, exhibitor.company, Toast.LENGTH_SHORT);
360                     toast.setText(exhibitor.company);
361                     toast.show();
362                     break;
```

BioDetails.java, line 38 (Privacy Violation)

Fortify Priority:	Critical	Folder	Low
Kingdom:	Security Features		
Abstract:	The method initialize() in BioDetails.java mishandles confidential information, which can compromise user privacy and is often illegal.		
Source:	Local.java:447 android.database.sqlite.SQLiteDatabase.rawQuery()		

```
445   if(name!=null){
446    String statement = "SELECT * FROM Bio WHERE Name=?";
447    Cursor cursor = ourDatabase.rawQuery(statement, new String[]{name});
448    for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
449     bioItem = new BioItem(cursor);
```

Sink: BioDetails.java:38 android.widget.TextView.setText()

```
36
37          if(title!=null)
38           title.setText(bioItem.title);
39          if(name!=null)
40           name.setText(bioItem.name);
```

Analysis: Not an Issue

NoteDetails.java, line 34 (Privacy Violation)

Fortify Priority:	Critical	Folder	Low
Kingdom:	Security Features		
Abstract:	The method initialize() in NoteDetails.java mishandles confidential information, which can compromise user privacy and is often illegal.		
Source:	Local.java:349 android.database.sqlite.SQLiteDatabase.rawQuery()		

```
347   public  ScheduleItem2 getSchedule2Item(ScheduleItem2 scheduleItem){
348    String statement = "SELECT * FROM ScheduleItem2 WHERE pk_id="+scheduleItem.pk_id + "
ORDER BY DTStart";
349    Cursor cursor = ourDatabase.rawQuery(statement, null);
350    ScheduleItem2 result = null;
351    for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
```

Sink: NoteDetails.java:34 android.widget.TextView.setText()

```
32          notes = (EditText)dialog.findViewById(R.id.note_details_notes);
33          if(notes!=null)
34           notes.setText(scheduleItem.notes);
35         }
36        }
```

ScheduleItemDetails.java, line 27 (Privacy Violation)

Fortify Priority:	Critical	Folder	Low
Kingdom:	Security Features		
Abstract:	The method initialize() in ScheduleItemDetails.java mishandles confidential information, which can compromise user privacy and is often illegal.		
Source:	Local.java:349 android.database.sqlite.SQLiteDatabase.rawQuery()		

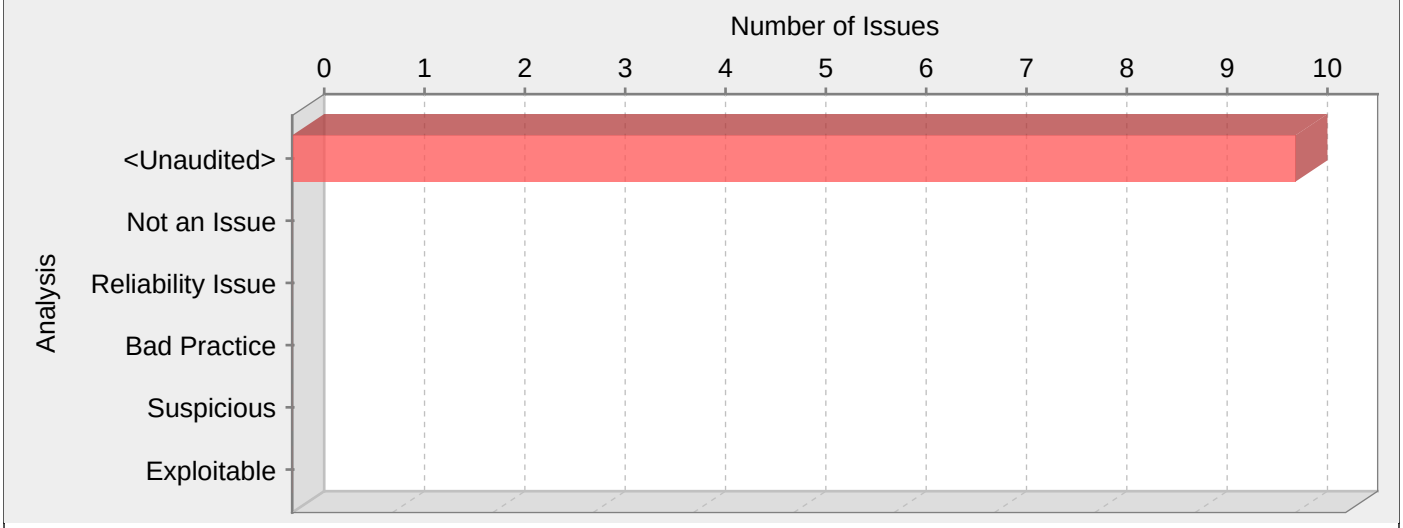
```
347   public  ScheduleItem2 getSchedule2Item(ScheduleItem2 scheduleItem){
348    String statement = "SELECT * FROM ScheduleItem2 WHERE pk_id="+scheduleItem.pk_id + "
ORDER BY DTStart";
349    Cursor cursor = ourDatabase.rawQuery(statement, null);
350    ScheduleItem2 result = null;
351    for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
```

Sink: ScheduleItemDetails.java:27 android.widget.TextView.setText()

```
25           TextView description =
(TextView)dialog.findViewById(R.id.details_schedule_description);
26           if(description!=null){
```

```
27                          description.setText(scheduleItem.description);
28                        }
29                      }
```

Category: System Information Leak (10 Issues)



Abstract:

Revealing system data or debugging information helps an adversary learn about the system and form a plan of attack.

Explanation:

An information leak occurs when system data or debugging information leaves the program through an output stream or logging function.

Example: The following code prints an exception to the standard error stream:

```
try {
...
} catch (Exception e) {
e.printStackTrace();
}
```

Depending upon the system configuration, this information can be dumped to a console, written to a log file, or exposed to a remote user. In some cases the error message tells the attacker precisely what sort of an attack the system is vulnerable to. For example, a database error message can reveal that the application is vulnerable to a SQL injection attack. Other error messages can reveal more oblique clues about the system. In the example above, the search path could imply information about the type of operating system, the applications installed on the system, and the amount of care that the administrators have put into configuring the program.

Recommendations:

Write error messages with security in mind. In production environments, turn off detailed error information in favor of brief messages. Restrict the generation and storage of detailed output that can help administrators and programmers diagnose problems. Be careful, debugging traces can sometimes appear in non-obvious places (embedded in comments in the HTML for an error page, for example).

Even brief error messages that do not reveal stack traces or database dumps can potentially aid an attacker. For example, an "Access Denied" message can reveal that a file or user exists on the system.

Tips:

1. Do not rely on wrapper scripts, corporate IT policy, or quick-thinking system administrators to prevent system information leaks. Write software that is secure on its own.

2. This category of vulnerability does not apply to all types of programs. For example, if your application executes on a client machine where system information is already available to an attacker, or if you print system information only to a trusted log file, you can use AuditGuide to filter out this category.

3. Fortify RTA adds protection against this category.

BioDetails.java, line 75 (System Information Leak)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	The function getBio() in BioDetails.java might reveal system data or debugging information by calling printStackTrace() on line 75. The information revealed by printStackTrace() could help an adversary form a plan of attack.		
Sink:	BioDetails.java:75 printStackTrace()		

```
73                    return convertStreamToString(activity.getAssets().open("text_"+bioPath));
74                   } catch (IOException e) {
75                    e.printStackTrace();
76                   }
77                   return "";
```

NewsItem.java, line 56 (System Information Leak)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The function setWithJSON() in NewsItem.java might reveal system data or debugging information by calling printStackTrace() on line 56. The information revealed by printStackTrace() could help an adversary form a plan of attack.		

Sink: NewsItem.java:56 printStackTrace()

```
54                    this.pubDate = news.getString("PubDate");
55                   } catch (JSONException e) {
56                    e.printStackTrace();
57                   }
58                  }
```

BoothLocation.java, line 100 (System Information Leak)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The function fromJson() in BoothLocation.java might reveal system data or debugging information by calling printStackTrace() on line 100. The information revealed by printStackTrace() could help an adversary form a plan of attack.		

Sink: BoothLocation.java:100 printStackTrace()

```
98                   } catch (JSONException e) {
99                    // TODO Auto-generated catch block
100                   e.printStackTrace();
101                  }
102                 }
```

BioDetails.java, line 67 (System Information Leak)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The function getResId() in BioDetails.java might reveal system data or debugging information by calling printStackTrace() on line 67. The information revealed by printStackTrace() could help an adversary form a plan of attack.		

Sink: BioDetails.java:67 printStackTrace()

```
65                      return idField.getInt(idField);
66                  } catch (Exception e) {
67                      e.printStackTrace();
68                      return -1;
69                  }
```

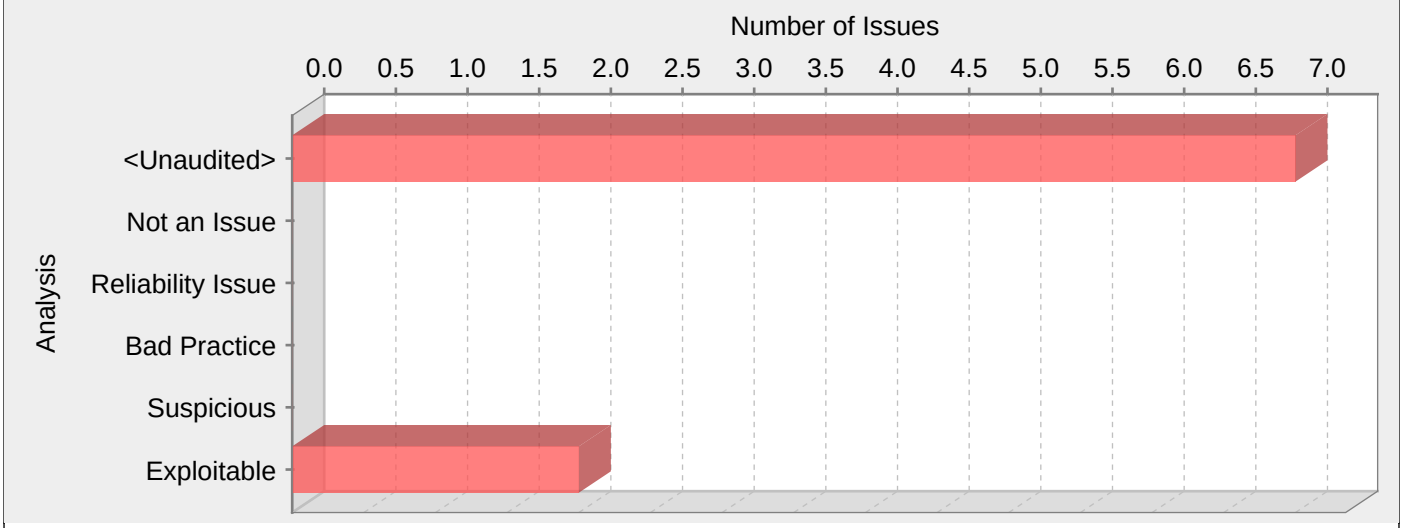
BioItem.java, line 49 (System Information Leak)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	The function setWithJSON() in BioItem.java might reveal system data or debugging information by calling printStackTrace() on line 49. The information revealed by printStackTrace() could help an adversary form a plan of attack.		

Sink: BioItem.java:49 printStackTrace()

```
47                    image = bioItem.getString("image");
48                   } catch (JSONException e) {
49                    e.printStackTrace();
50                   }
51                  }
```

Category: SQL Injection (9 Issues)



Abstract:

Constructing a dynamic SQL statement with input coming from an untrusted source could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

Explanation:

SQL injection errors occur when:

1. Data enters a program from an untrusted source.

In this case HP Fortify Static Code Analyzer could not determine that the source of the data is trusted.

2. The data is used to dynamically construct a SQL query.

Example 1: The following code dynamically constructs and executes a SQL query that searches for items matching a specified name. The query restricts the items displayed to those where the owner matches the user name of the currently-authenticated user.

```
...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ itemName + "'";
ResultSet rs = stmt.execute(query);
...
```

The query that this code intends to execute follows:

```
SELECT * FROM items
WHERE owner = <userName>
AND itemname = <itemName>;
```

However, because the query is constructed dynamically by concatenating a constant base query string and a user input string, the query only behaves correctly if itemName does not contain a single-quote character. If an attacker with the user name wiley enters the string "name' OR 'a'='a" for itemName, then the query becomes the following:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name' OR 'a'='a';
```

The addition of the OR 'a'='a' condition causes the where clause to always evaluate to true, so the query becomes logically equivalent to the much simpler query:

```
SELECT * FROM items;
```

This simplification of the query allows the attacker to bypass the requirement that the query only return items owned by the authenticated user; the query now returns all entries stored in the items table, regardless of their specified owner.

Example 2: This example examines the effects of a different malicious value passed to the query constructed and executed in Example 1. If an attacker with the user name wiley enters the string "name'; DELETE FROM items; --" for itemName, then the query becomes the following two queries:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

--'
```

Many database servers, including Microsoft(R) SQL Server 2000, allow multiple SQL statements separated by semicolons to be executed at once. While this attack string results in an error on Oracle and other database servers that do not allow the batch-execution of statements separated by semicolons, on databases that do allow batch execution, this type of attack allows the attacker to execute arbitrary commands against the database.

Notice the trailing pair of hyphens (--), which specifies to most database servers that the remainder of the statement is to be treated as a comment and not executed [4]. In this case the comment character serves to remove the trailing single-quote left over from the modified query. On a database where comments are not allowed to be used in this way, the general attack could still be made effective using a trick similar to the one used in Example 1. If an attacker enters the string "name'); DELETE FROM items; SELECT * FROM items WHERE 'a'='a", the following three valid statements will be created:

```
SELECT * FROM items
WHERE owner = 'wiley'
AND itemname = 'name';

DELETE FROM items;

SELECT * FROM items WHERE 'a'='a';
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from a whitelist of safe values or identify and escape a blacklist of potentially malicious values. Whitelisting can be a very effective means of enforcing strict input validation rules, but parameterized SQL statements require less maintenance and can offer more guarantees with respect to security. As is almost always the case, blacklisting is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers can:

- Target fields that are not quoted

- Find ways to bypass the need for certain escaped meta-characters

- Use stored procedures to hide the injected meta-characters

Manually escaping characters in input to SQL queries can help, but it will not make your application secure from SQL injection attacks.

Another solution commonly proposed for dealing with SQL injection attacks is to use stored procedures. Although stored procedures prevent some types of SQL injection attacks, they fail to protect against many others. Stored procedures typically help prevent SQL injection attacks by limiting the types of statements that can be passed to their parameters. However, there are many ways around the limitations and many interesting statements that can still be passed to stored procedures. Again, stored procedures can prevent some exploits, but they will not make your application secure against SQL injection attacks.

Recommendations:

The root cause of a SQL injection vulnerability is the ability of an attacker to change context in the SQL query, causing a value that the programmer intended to be interpreted as data to be interpreted as a command instead. When a SQL query is constructed, the programmer knows what should be interpreted as part of the command and what should be interpreted as data. Parameterized SQL statements can enforce this behavior by disallowing data-directed context changes and preventing nearly all SQL injection attacks. Parameterized SQL statements are constructed using strings of regular SQL, but when user-supplied data needs to be included, they create bind parameters, which are placeholders for data that is subsequently inserted. Bind parameters allow the program to explicitly specify to the database what should be treated as a command and what should be treated as data. When the program is ready to execute a statement, it specifies to the database the runtime values to use for the value of each of the bind parameters, without the risk of the data being interpreted as commands.

The previous example can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

```
...
String userName = ctx.getAuthenticatedUserName();
String itemName = request.getParameter("itemName");
String query =
"SELECT * FROM items WHERE itemname=? AND owner=?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setString(1, itemName);
```

```
stmt.setString(2, userName);
ResultSet results = stmt.execute();
...
```

More complicated scenarios, often found in report generation code, require that user input affect the command structure of the SQL statement, such as the addition of dynamic constraints in the WHERE clause. Do not use this requirement to justify concatenating user input into query strings. Prevent SQL injection attacks where user input must affect statement command structure with a level of indirection: create a set of legitimate strings that correspond to different elements you might include in a SQL statement. When constructing a statement, use input from the user to select from this set of application-controlled values.

Tips:

1. A common mistake is to use parameterized SQL statements that are constructed by concatenating user-controlled strings. Of course, this defeats the purpose of using parameterized SQL statements. If you are not certain that the strings used to form statements are constants controlled by the application, do not assume that they are safe because they are not being executed directly as SQL strings. Thoroughly investigate all uses of user-controlled strings in SQL statements and verify that none can be used to modify the meaning of the query.

2. Data is untrustworthy if it originates from public non-final string fields of a class. These types of fields may be modified by an unknown source.

3. Fortify RTA adds protection against this category.

Local.java, line 54 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 54 of Local.java, the method excStatement() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

Sink: Local.java:54 execSQL()

```
52    }
53    private void excStatement(SQLiteDatabase db,String statement){
54     db.execSQL(statement);
55    }
56    private void excStatements(SQLiteDatabase db,String[] statements){
```

Local.java, line 362 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 362 of Local.java, the method getSchedule2Now() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

Sink: Local.java:362 rawQuery()

```
360
361   String statement = "SELECT * FROM ScheduleItem2 WHERE DTEnd > '"+currentDateandTime+"'
AND DTStart < '"+currentDateandTime+"' ORDER BY DTStart";
362   Cursor cursor = ourDatabase.rawQuery(statement, null);
363   ArrayList<ScheduleItem2> scheduleItems = new ArrayList<ScheduleItem2>();
364   for (boolean hasItem = cursor.moveToFirst(); hasItem; hasItem = cursor.moveToNext()) {
```

Local.java, line 349 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 349 of Local.java, the method getSchedule2Item() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		

Sink: Local.java:349 rawQuery()

```
347   public  ScheduleItem2 getSchedule2Item(ScheduleItem2 scheduleItem){
348   String statement = "SELECT * FROM ScheduleItem2 WHERE pk_id="+scheduleItem.pk_id + "
ORDER BY DTStart";
349   Cursor cursor = ourDatabase.rawQuery(statement, null);
350   ScheduleItem2 result = null;
351   for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
```

Analysis: Exploitable

Comments:	*565075 2012-11-06 11:06 AM* Use bind vars

Local.java, line 337 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 337 of Local.java, the method getFavoriteTimeSlots() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		
Sink:	Local.java:337 rawQuery()		

```
335   ArrayList<ScheduleItem2> scheduleItems = new ArrayList<ScheduleItem2>();
336   if(dayQ.containsKey(day)){
337    Cursor cursor = ourDatabase.rawQuery(dayQ.get(day), null);
338    for (boolean hasItem = cursor.moveToFirst(); hasItem; hasItem = cursor.moveToNext()) {
339     ScheduleItem2 aAct = new ScheduleItem2(cursor);
```

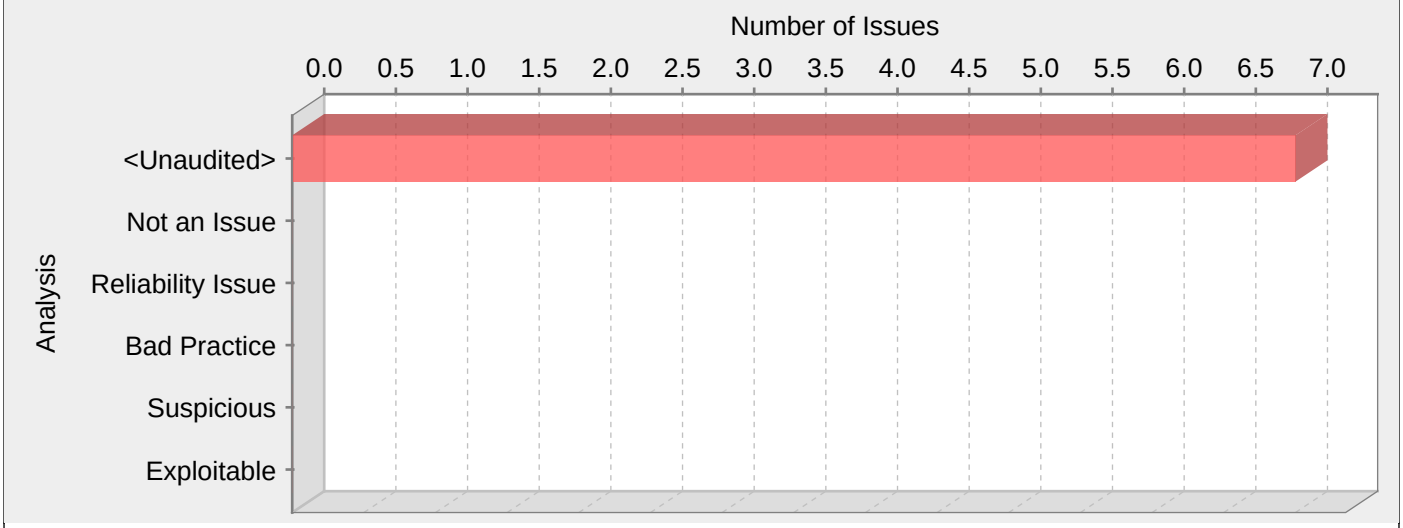
Analysis:	Exploitable

Local.java, line 58 (SQL Injection)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	On line 58 of Local.java, the method excStatements() invokes a SQL query built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		
Sink:	Local.java:58 execSQL()		

```
56   private void excStatements(SQLiteDatabase db,String[] statements){
57    for(String statement : statements){
58     db.execSQL(statement);
59    }
60   }
```

Category: Poor Style: Value Never Read (7 Issues: 7 Hidden)



Abstract:
The variable's value is assigned but never used, making it a dead store.

Explanation:
This variable's value is not used. After the assignment, the variable is either assigned another value or goes out of scope.

Example: The following code excerpt assigns to the variable r and then overwrites the value without using it.

```
r = getName();
r = getNewBuffer(buf);
```

Recommendations:
Remove unnecessary assignments in order to make the code easier to understand and maintain.

NoteDetails.java, line 46 (Poor Style: Value Never Read) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method hide() in NoteDetails.java never uses the value it assigns to the variable worked on line 46.		

Sink: NoteDetails.java:46 VariableAccess: worked()

```
44    if(myView!=null){
45     if(notes!=null){
46      worked = confApp.local.adjustNotes(scheduleItem, notes.getText().toString());
47     }
48     if(myView!=null)
```

ScheduleItem2.java, line 152 (Poor Style: Value Never Read) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method formatYearMonthDay() in ScheduleItem2.java never uses the value it assigns to the variable m on line 152.		

Sink: ScheduleItem2.java:152 VariableAccess: m()

```
150
151         int h = Integer.parseInt(hour);
152         int m = Integer.parseInt(minute);
153
154         h = h-4;
```

ScheduleItem2.java, line 117 (Poor Style: Value Never Read) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method formatTime() in ScheduleItem2.java never uses the value it assigns to the variable m on line 117.		

Sink:	ScheduleItem2.java:117 VariableAccess: m()

```
115
116                     int h = Integer.parseInt(hour);
117                     int m = Integer.parseInt(minute);
118
119                     h = h-4;
```

ComboDialog.java, line 102 (Poor Style: Value Never Read) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method onClick() in ComboDialog.java never uses the value it assigns to the variable worked on line 102.		
Sink:	ComboDialog.java:102 VariableAccess: worked()		

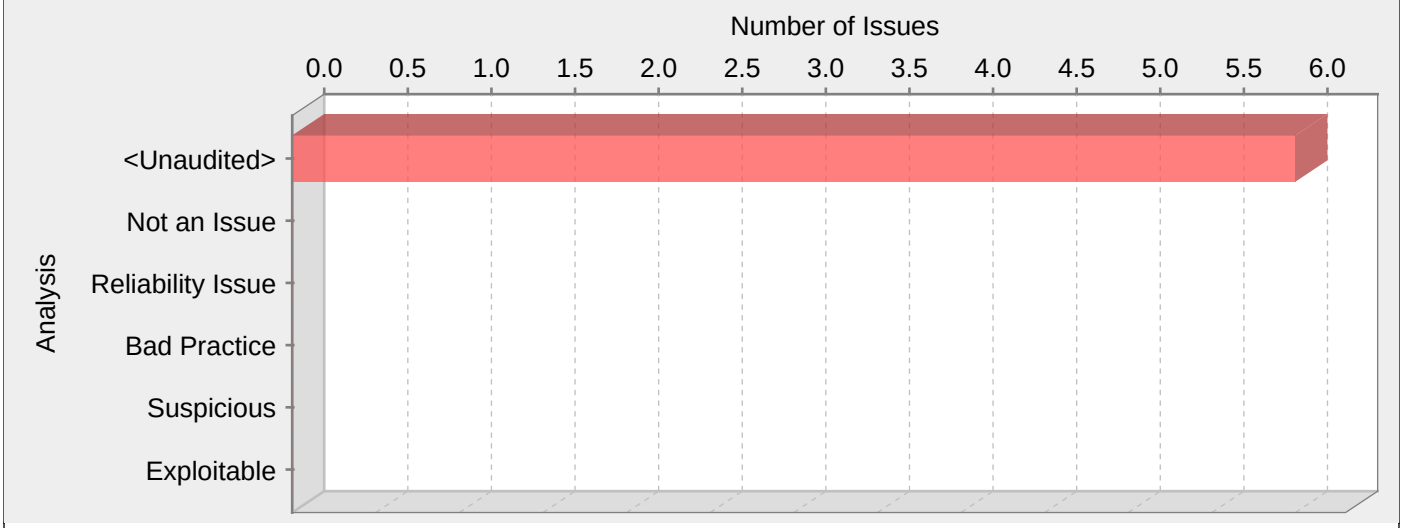
```
100                      favorites.setOnClickListener(new OnClickListener(){
101                    public void onClick(View v) {
102                     boolean worked = confApp.local.adjustFavorites(scheduleItem, !scheduleItem.favorited);
103                     scheduleItem.favorited = !scheduleItem.favorited;
104                     adjustFavorited(favorites);
```

ScheduleItem2.java, line 159 (Poor Style: Value Never Read) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method formatYearMonthDay() in ScheduleItem2.java never uses the value it assigns to the variable h on line 159.		
Sink:	ScheduleItem2.java:159 VariableAccess: h()		

```
157                      a = " pm";
158                     if(h>12)
159                      h = h-12;
160
161
```

Category: Poor Error Handling: Empty Catch Block (6 Issues)



Abstract:

Ignoring an exception can cause the program to overlook unexpected states and conditions.

Explanation:

Just about every serious attack on a software system begins with the violation of a programmer's assumptions. After the attack, the programmer's assumptions seem flimsy and poorly founded, but before an attack many programmers would defend their assumptions well past the end of their lunch break.

Two dubious assumptions that are easy to spot in code are "this method call can never fail" and "it doesn't matter if this call fails". When a programmer ignores an exception, they implicitly state that they are operating under one of these assumptions.

Example 1: The following code excerpt ignores a rarely-thrown exception from doExchange().

```
try {
doExchange();
}
catch (RareException e) {
// this can never happen
}
```

If a RareException were to ever be thrown, the program would continue to execute as though nothing unusual had occurred. The program records no evidence indicating the special situation, potentially frustrating any later attempt to explain the program's behavior.

Recommendations:

At a minimum, log the fact that the exception was thrown so that it will be possible to come back later and make sense of the resulting program behavior. Better yet, abort the current operation. If the exception is being ignored because the caller cannot properly handle it but the context makes it inconvenient or impossible for the caller to declare that it throws the exception itself, consider throwing a RuntimeException or an Error, both of which are unchecked exceptions. As of JDK 1.4, RuntimeException has a constructor that makes it easy to wrap another exception.

Example 2: The code in Example 1 could be rewritten in the following way:

```
try {
doExchange();
}
catch (RareException e) {
throw RuntimeException("This can never happen", e);
}
```

Tips:

1. There are rare types of exceptions that can be discarded in some contexts. For instance, Thread.sleep() throws InterruptedException, and in many situations the program should behave the same way whether or not it was awoken prematurely.

```
try {
Thread.sleep(1000);
}
```

```
catch (InterruptedException e){
// The thread has been woken up prematurely, but its
// behavior should be the same either way.
}
```

Network.java, line 159 (Poor Error Handling: Empty Catch Block)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method getExhibitors() in Network.java ignores an exception on line 159, which could cause the program to overlook unexpected states and conditions.		

Sink: Network.java:159 CatchBlock()

```
157          }
158          in.close();
159        } catch (MalformedURLException e) {
160        } catch (IOException e) {
161        }
```

Network.java, line 49 (Poor Error Handling: Empty Catch Block)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method grabFeed() in Network.java ignores an exception on line 49, which could cause the program to overlook unexpected states and conditions.		

Sink: Network.java:49 CatchBlock()

```
47           }
48           in.close();
49         } catch (MalformedURLException e) {
50         } catch (IOException e) {
51           //Toast.makeText(activity, "Cannot connect to the internet!",
Toast.LENGTH_LONG).show();
```

Network.java, line 83 (Poor Error Handling: Empty Catch Block)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method getSchedule2() in Network.java ignores an exception on line 83, which could cause the program to overlook unexpected states and conditions.		

Sink: Network.java:83 CatchBlock()

```
81           }
82           in.close();
83         } catch (MalformedURLException e) {
84         } catch (IOException e) {
85           //Toast.makeText(activity, "Cannot connect to the internet!",
Toast.LENGTH_LONG).show();
```

Network.java, line 84 (Poor Error Handling: Empty Catch Block)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The method getSchedule2() in Network.java ignores an exception on line 84, which could cause the program to overlook unexpected states and conditions.		

Sink: Network.java:84 CatchBlock()

```
82           in.close();
83         } catch (MalformedURLException e) {
84         } catch (IOException e) {
85           //Toast.makeText(activity, "Cannot connect to the internet!",
Toast.LENGTH_LONG).show();
86         }
```

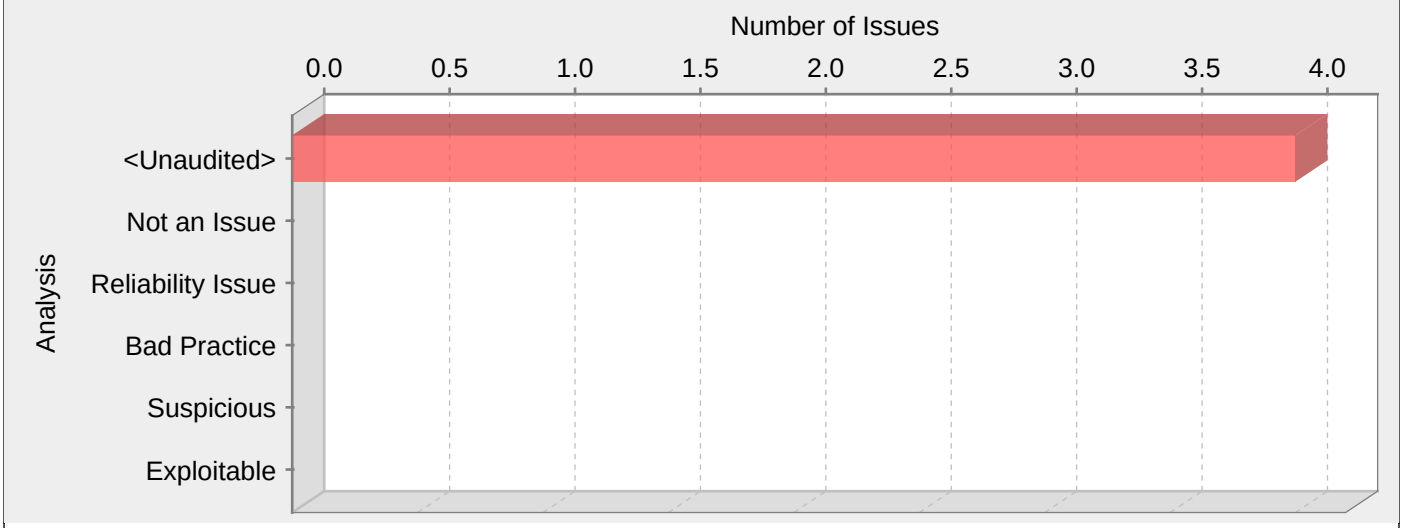
Network.java, line 50 (Poor Error Handling: Empty Catch Block)

Fortify Priority:	Low	**Folder**	Low

Kingdom:	Errors
Abstract:	The method grabFeed() in Network.java ignores an exception on line 50, which could cause the program to overlook unexpected states and conditions.
Sink:	Network.java:50 CatchBlock()

```
48                      in.close();
49                    } catch (MalformedURLException e) {
50                    } catch (IOException e) {
51                      //Toast.makeText(activity, "Cannot connect to the internet!",
Toast.LENGTH_LONG).show();
52                    }
```

Category: Access Control: Database (4 Issues)



Abstract:

Without proper access control, executing a SQL statement that contains a user-controlled primary key can allow an attacker to view unauthorized records.

Explanation:

Database access control errors occur when:

1. Data enters a program from an untrusted source.

2. The data is used to specify the value of a primary key in a SQL query.

Example 1: The following code uses a parameterized statement, which escapes metacharacters and prevents SQL injection vulnerabilities, to construct and execute a SQL query that searches for an invoice matching the specified identifier [1]. The identifier is selected from a list of all invoices associated with the current authenticated user.

```
...
id = Integer.decode(request.getParameter("invoiceID"));
String query = "SELECT * FROM invoices WHERE id = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setInt(1, id);
ResultSet results = stmt.execute();
...
```

The problem is that the developer has failed to consider all of the possible values of id. Although the interface generates a list of invoice identifiers that belong to the current user, an attacker can bypass this interface to request any desired invoice. Because the code in this example does not check to ensure that the user has permission to access the requested invoice, it will display any invoice, even if it does not belong to the current user.

A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the 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 Fortify 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.

Recommendations:

Rather than relying on the presentation layer to restrict values submitted by the user, access control should be handled by the application and database layers. Under no circumstances should a user be allowed to retrieve or modify a row in the database without the appropriate permissions. Every query that accesses the database should enforce this policy, which can often be accomplished by simply including the current authenticated username as part of the query.

Example 2: The following code implements the same functionality as Example 1 but imposes an additional constraint requiring that the current authenticated user have specific access to the invoice.

```
...
userName = ctx.getAuthenticatedUserName();
id = Integer.decode(request.getParameter("invoiceID"));
```

```
String query =
"SELECT * FROM invoices WHERE id = ? AND user = ?";
PreparedStatement stmt = conn.prepareStatement(query);
stmt.setInt(1, id);
stmt.setString(2, userName);
ResultSet results = stmt.execute();
...
```

Local.java, line 292 (Access Control: Database)

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

Abstract: Without proper access control, the method insertRow() in Local.java can execute a SQL statement on line 292 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

Source: Network.java:155 java.io.BufferedReader.readLine()

```
153         BufferedReader in = new BufferedReader(new InputStreamReader(is));
154         String str;
155         while ((str = in.readLine()) != null) {
156             result.add(str);
157         }
```

Sink: Local.java:292 android.database.sqlite.SQLiteDatabase.insert()

```
290   ContentValues contentValues = getContentValues(values);
291   if(contentValues!=null)
292    return ourDatabase.insert(tableName,null,contentValues);
293   else
294    return -1;
```

Local.java, line 555 (Access Control: Database)

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

Abstract: Without proper access control, the method getExhibitor() in Local.java can execute a SQL statement on line 555 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

Source: Local.java:539 android.database.sqlite.SQLiteDatabase.rawQuery()

```
537   int ya = y+150;
538   int yb = y-150;
539   Cursor cursor = ourDatabase.rawQuery(statement2, new
String[]{""+xb,""+xa,""+yb,""+ya});
540
541   ArrayList<BoothLocation> booths = new ArrayList<BoothLocation>();
```

Sink: Local.java:555 android.database.sqlite.SQLiteDatabase.rawQuery()

```
553  public  Exhibitor getExhibitor(BoothLocation boothLocation){
554   String statement = "SELECT * FROM Exhibitor WHERE boothname=?";
555   Cursor cursor = ourDatabase.rawQuery(statement, new
String[]{""+boothLocation.boothName});
556   Exhibitor exhibitor = null;
557   for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
```

Local.java, line 292 (Access Control: Database)

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

Abstract: Without proper access control, the method insertRow() in Local.java can execute a SQL statement on line 292 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

Source: Network.java:79 java.io.BufferedReader.readLine()

```
77          BufferedReader in = new BufferedReader(new InputStreamReader(urla.openStream()));
78          String str;
79          while ((str = in.readLine()) != null) {
80              result.add(str);
```

```
81                  }
```

Sink: Local.java:292 android.database.sqlite.SQLiteDatabase.insert()

```
290   ContentValues contentValues = getContentValues(values);
291   if(contentValues!=null)
292    return ourDatabase.insert(tableName,null,contentValues);
293   else
294    return -1;
```

Local.java, line 292 (Access Control: Database)

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

Abstract: Without proper access control, the method insertRow() in Local.java can execute a SQL statement on line 292 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.

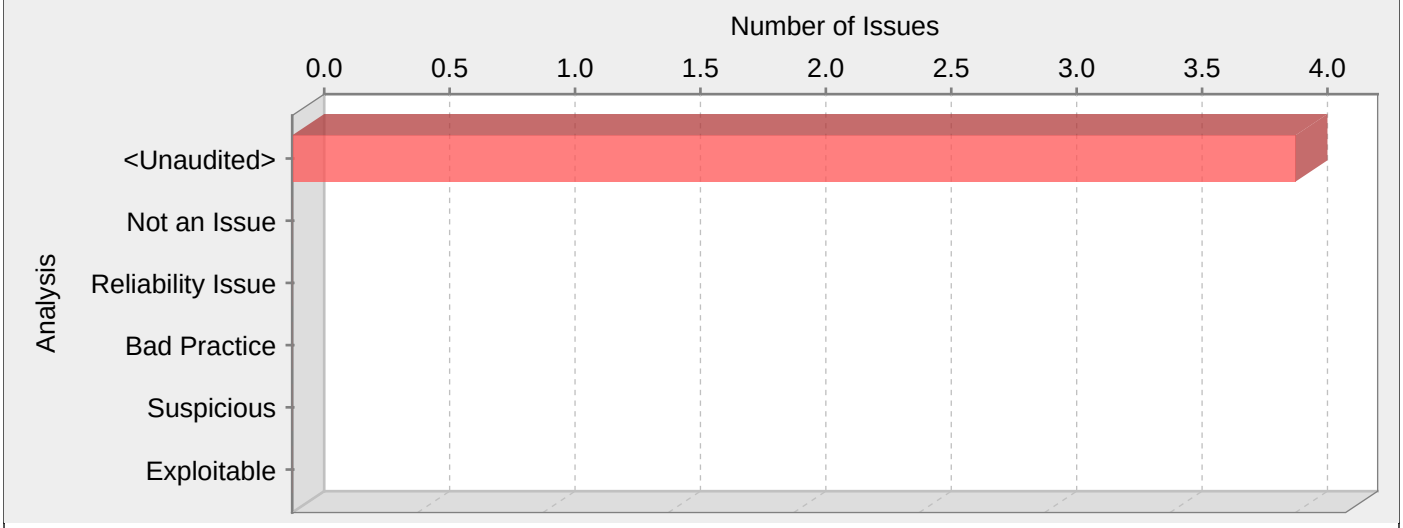
Source: Network.java:45 java.io.BufferedReader.readLine()

```
43          BufferedReader in = new BufferedReader(new InputStreamReader(urla.openStream()));
44          String str;
45          while ((str = in.readLine()) != null) {
46              result += str;
47          }
```

Sink: Local.java:292 android.database.sqlite.SQLiteDatabase.insert()

```
290   ContentValues contentValues = getContentValues(values);
291   if(contentValues!=null)
292    return ourDatabase.insert(tableName,null,contentValues);
293   else
294    return -1;
```

Category: Poor Logging Practice: Use of a System Output Stream (4 Issues)



Abstract:

Using System.out or System.err rather than a dedicated logging facility makes it difficult to monitor the behavior of the program.

Explanation:

Example 1: The first Java program that a developer learns to write often looks like this:

```
public class MyClass
public static void main(String[] args) {
System.out.println("hello world");
}
}
```

While most programmers go on to learn many nuances and subtleties about Java, a surprising number hang on to this first lesson and never give up on writing messages to standard output using System.out.println().

The problem is that writing directly to standard output or standard error is often used as an unstructured form of logging. Structured logging facilities provide features like logging levels, uniform formatting, a logger identifier, timestamps, and, perhaps most critically, the ability to direct the log messages to the right place. When the use of system output streams is jumbled together with the code that uses loggers properly, the result is often a well-kept log that is missing critical information.

Developers widely accept the need for structured logging, but many continue to use system output streams in their "pre-production" development. If the code you are reviewing is past the initial phases of development, use of System.out or System.err may indicate an oversight in the move to a structured logging system.

Recommendations:

Use a Java logging facility rather than System.out or System.err.

Example 2: For example, the "hello world" program above can be re-written using log4j like this:

```
import org.apache.log4j.Logger;
import org.apache.log4j.BasicConfigurator;

public class MyClass {
private final static Logger logger =
Logger.getLogger(MyClass.class);

public static void main(String[] args) {
BasicConfigurator.configure();
logger.info("hello world");
}
}
```

Network.java, line 170 (Poor Logging Practice: Use of a System Output Stream)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Using println() rather than a dedicated logging facility makes it difficult to monitor the behavior of the program.		

Sink:	Network.java:170 FunctionCall: println()

```
168                 long worked = confApp.local.insertRow("Exhibitor", exhibitor.hashed());
169                 if(worked > 0)
170                  System.out.println("Worked : "+worked);
171                 else
172                  System.out.println("BLARG : ");
```

Network.java, line 173 (Poor Logging Practice: Use of a System Output Stream)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	Using print() rather than a dedicated logging facility makes it difficult to monitor the behavior of the program.		
Sink:	Network.java:173 FunctionCall: print()		

```
171                 else
172                  System.out.println("BLARG : ");
173                 System.out.print("blan");
174                }
175                ConfApp.confApp.local.endTransaction();
```

MainActivity.java, line 108 (Poor Logging Practice: Use of a System Output Stream)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	Using println() rather than a dedicated logging facility makes it difficult to monitor the behavior of the program.		
Sink:	MainActivity.java:108 FunctionCall: println()		

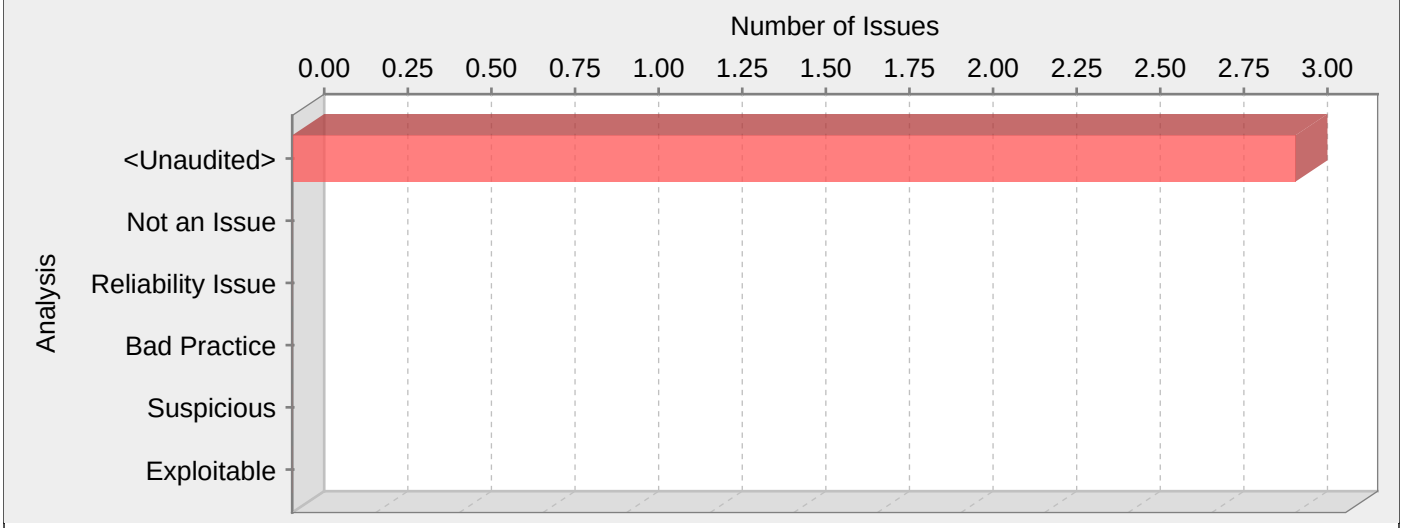
```
106             }
107             Log.wtf("HERE", output);
108             System.out.println(output);
109             writer.flush();
110             writer.close();
```

Network.java, line 172 (Poor Logging Practice: Use of a System Output Stream)

Fortify Priority:	Low	Folder	Low
Kingdom:	Encapsulation		
Abstract:	Using println() rather than a dedicated logging facility makes it difficult to monitor the behavior of the program.		
Sink:	Network.java:172 FunctionCall: println()		

```
170                  System.out.println("Worked : "+worked);
171                 else
172                  System.out.println("BLARG : ");
173                 System.out.print("blan");
174                }
```

Category: Denial of Service (3 Issues)



Abstract:

An attacker could cause the program to crash or otherwise become unavailable to legitimate users.

Explanation:

Attackers may be able to deny service to legitimate users by flooding the application with requests, but flooding attacks can often be defused at the network layer. More problematic are bugs that allow an attacker to overload the application using a small number of requests. Such bugs allow the attacker to specify the quantity of system resources their requests will consume or the duration for which they will use them.

Example 1: The following code allows a user to specify the amount of time for which a thread will sleep. By specifying a large number, an attacker can tie up the thread indefinitely. With a small number of requests, the attacker can deplete the application's thread pool.

```
int usrSleepTime = Integer.parseInt(usrInput);
Thread.sleep(usrSleepTime);
```

Example 2: The following code reads a String from a zip file. Because it uses the readLine() method, it will read an unbounded amount of input. An attacker can take advantage of this code to cause an OutOfMemoryException or to consume a large amount of memory so that the program spends more time performing garbage collection or runs out of memory during some subsequent operation.

```
InputStream zipInput = zipFile.getInputStream(zipEntry);
Reader zipReader = new InputStreamReader(zipInput);
BufferedReader br = new BufferedReader(zipReader);
String line = br.readLine();
```

Recommendations:

Validate user input to ensure that it will not cause inappropriate resource utilization.

Example 1 Revisited: The following code allows a user to specify the amount of time for which a thread will sleep, but only if the value is within reasonable bounds.

```
int usrSleepTime = Integer.parseInt(usrInput);
if (usrSleepTime >= SLEEP_MIN &&
usrSleepTime <= SLEEP_MAX) {
Thread.sleep(usrSleepTime);
} else {
throw new Exception("Invalid sleep duration");
}
}
```

Example 2 Revisited: The following code reads a String from a zip file. The maximum string length it will read is MAX_STR_LEN characters.

```
InputStream zipInput = zipFile.getInputStream(zipEntry);
Reader zipReader = new InputStreamReader(zipInput);
BufferedReader br = new BufferedReader(zipReader);
StringBuffer sb = new StringBuffer();
```

```
int intC;
while ((intC = br.read()) != -1) {
char c = (char) intC;
if (c == '\n') {
break;
}
if (sb.length() >= MAX_STR_LEN) {
throw new Exception("input too long");
}
sb.append(c);
}
String line = sb.toString();
```

Tips:

1. Denial of service can happen even if the quantity of system resources that will be consumed or the duration for which they will be used is not controlled by an attacker, or at least not directly. Instead, a programmer might choose unsafe constant values for specifying these parameters. The HP Fortify Secure Coding Rulepacks will report such cases as potential Denial of Services vulnerabilities.

Network.java, line 155 (Denial of Service)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to readLine() at Network.java line 155 might allow an attacker to crash the program or otherwise make it unavailable to legitimate users.		

Sink: Network.java:155 readLine()

```
153         BufferedReader in = new BufferedReader(new InputStreamReader(is));
154         String str;
155         while ((str = in.readLine()) != null) {
156             result.add(str);
157         }
```

Network.java, line 45 (Denial of Service)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to readLine() at Network.java line 45 might allow an attacker to crash the program or otherwise make it unavailable to legitimate users.		

Sink: Network.java:45 readLine()

```
43          BufferedReader in = new BufferedReader(new InputStreamReader(urla.openStream()));
44          String str;
45          while ((str = in.readLine()) != null) {
46              result += str;
47          }
```

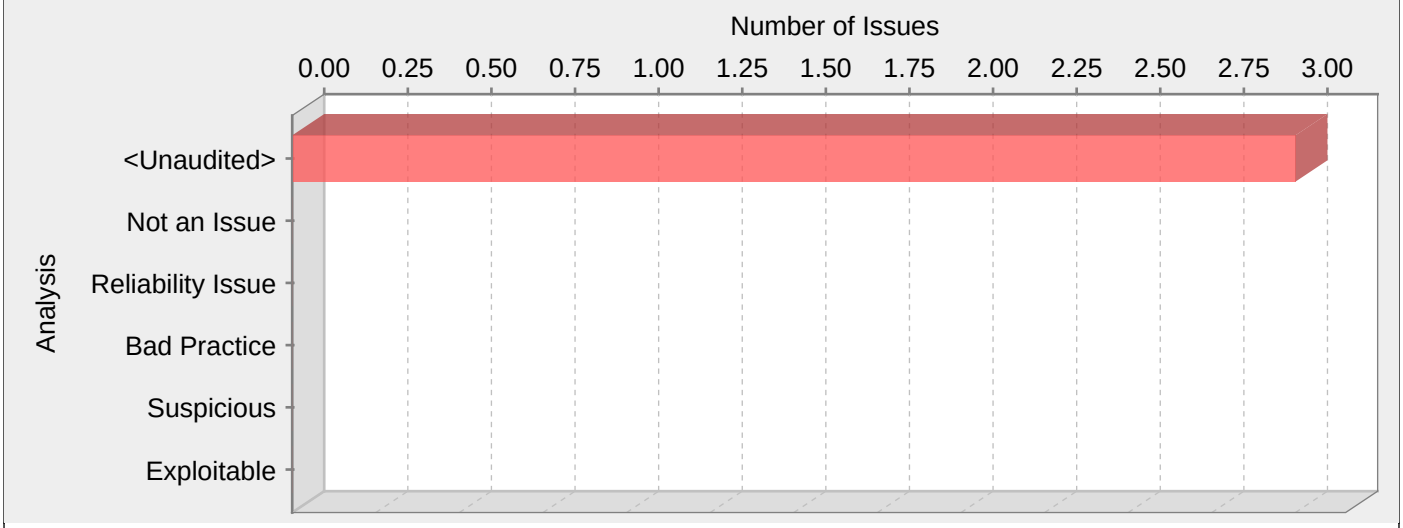
Network.java, line 79 (Denial of Service)

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Input Validation and Representation		
Abstract:	The call to readLine() at Network.java line 79 might allow an attacker to crash the program or otherwise make it unavailable to legitimate users.		

Sink: Network.java:79 readLine()

```
77          BufferedReader in = new BufferedReader(new InputStreamReader(urla.openStream()));
78          String str;
79          while ((str = in.readLine()) != null) {
80              result.add(str);
81          }
```

Category: Redundant Null Check (3 Issues: 3 Hidden)



Abstract:

The program can 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. Specifically, dereference-after-check errors occur when a program makes an explicit check for null, but proceeds to dereference the object when it is known to be null. Errors of this type are often the result of a typo or programmer oversight.

Most null pointer issues result in general software reliability problems, but if attackers can intentionally cause the program to dereference a null pointer, they can use the resulting exception to mount a denial of service attack or to cause the application to reveal debugging information that will be valuable in planning subsequent attacks.

Example 1: In the following code, the programmer confirms that the variable foo is null and subsequently dereferences it erroneously. If foo is null when it is checked in the if statement, then a null dereference will occur, thereby causing a null pointer exception.

```
if (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.

ComboDialog.java, line 87 (Redundant Null Check) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method initialize() in ComboDialog.java can crash the program by dereferencing a null pointer on line 87.		
Sink:	ComboDialog.java:87 Dereferenced : details()		

```
85          });
86          }
87          if(details.getVisibility()!=View.GONE)
88           details.performClick();
89          else if(speaker.getVisibility()!=View.GONE)
```

ComboDialog.java, line 89 (Redundant Null Check) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method initialize() in ComboDialog.java can crash the program by dereferencing a null pointer on line 89.		
Sink:	ComboDialog.java:89 Dereferenced : speaker()		

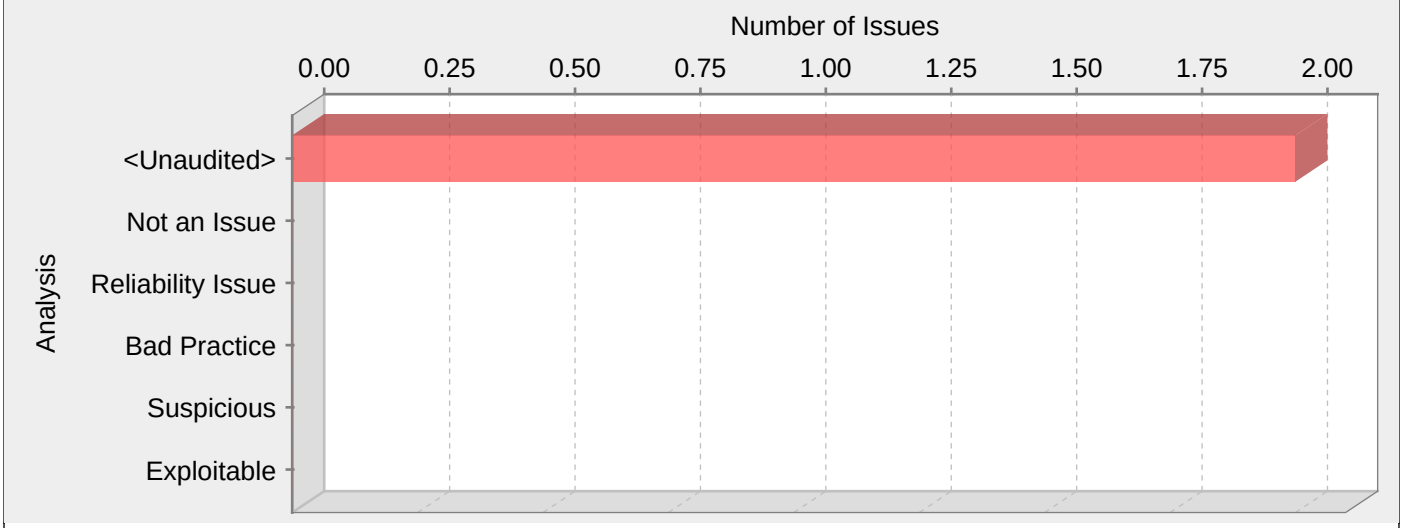
```
87          if(details.getVisibility()!=View.GONE)
```

```
88                          details.performClick();
89                        else if(speaker.getVisibility()!=View.GONE)
90                          speaker.performClick();
91                        else
```

ComboDialog.java, line 92 (Redundant Null Check) [Hidden]			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method initialize() in ComboDialog.java can crash the program by dereferencing a null pointer on line 92.		
Sink:	ComboDialog.java:92 Dereferenced : notes()		

```
90                          speaker.performClick();
91                        else
92                          notes.performClick();
93                    TextView title = (TextView)dialog.findViewById(R.id.dialog_schedule_details_title);
94                    if(title!=null)
```

Category: Privilege Management: Android Data Storage (2 Issues)



Abstract:

The program requests permission to write data to Android's external storage.

Explanation:

Files written to external storage are readable and writeable by arbitrary programs and users. Programs must never write sensitive information, for instance personally identifiable information, to external storage. When you connect the Android device via USB to a PC or other device it enables USB mass storage mode. Any file written to external storage can be read and modified in this mode. In addition, files in external storage will remain there even after the application that wrote them is uninstalled, further increasing the risk that any sensitive information stored in them will be compromised.

Example 1:The <uses-permission .../%gt; element of AndroidManifest.xml includes the dangerous attribute.

<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

Recommendations:

Do not write sensitive information or data that must later be trusted to external storage. Instead, write to an program-specific location, such as a SQLite database (provided by the Android platform). Any databases you create will be accessible by name to any class in the program, but not outside the program.

Example 2. Create a new SQLite database by creating a subclass of SQLiteOpenHelper and override the OnCreate() method.

```
public class MyDbOpenHelper extends SQLiteOpenHelper {

private static final int DATABASE_VERSION = 2;
private static final String DICTIONARY_TABLE_NAME = "dictionary";
private static final String DICTIONARY_TABLE_CREATE =
"CREATE TABLE " + DICTIONARY_TABLE_NAME + " (" +
KEY_WORD + " TEXT, " +
KEY_DEFINITION + " TEXT);";

DictionaryOpenHelper(Context context) {
super(context, DATABASE_NAME, null, DATABASE_VERSION);
}

@Override
public void onCreate(SQLiteDatabase db) {
db.execSQL(DICTIONARY_TABLE_CREATE);
}
}
```

Another option is to write to the device's internal storage. By default, files saved to the internal storage are private to the program that writes them and are inaccessible to other programs and to the user's direct means. When the user uninstalls a program, files stored on internal storage are also removed, diminishing the chance that something important will be left around.

Example 3: The following code creates and writes a private file to the device's internal storage. The declaration Context.MODE_PRIVATE creates the file (or replaces a file of the same name) and makes it private to current program.

```
String FILENAME = "hello_file";
String string = "hello world!";
```

```
FileOutputStream fos = openFileOutput(FILENAME, Context.MODE_PRIVATE);
fos.write(string.getBytes());
fos.close();
```

AndroidManifest.xml, line 8 (Privilege Management: Android Data Storage)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The program requests permission to write data to Android's external storage on line 8 of AndroidManifest.xml.		

Sink: AndroidManifest.xml:8 null()

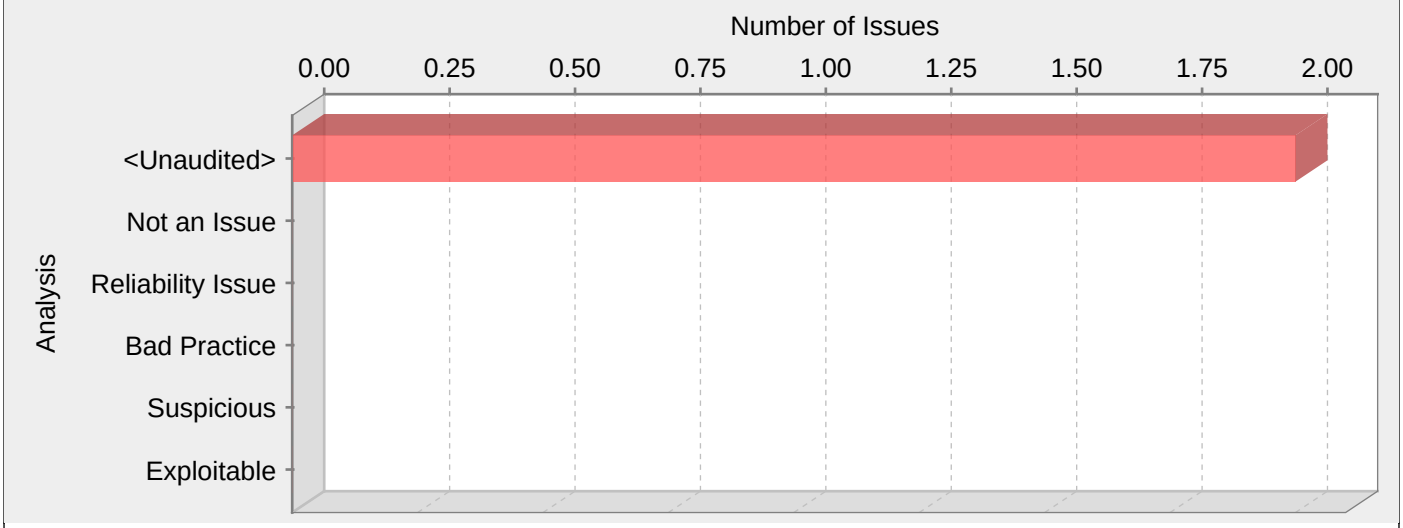
```
6      <uses-sdk android:minSdkVersion="8" />
7      <uses-permission android:name="android.permission.INTERNET" />
8      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
9      <application
10         android:icon="@drawable/app_icon"
```

AndroidManifest.xml, line 8 (Privilege Management: Android Data Storage)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The program requests permission to write data to Android's external storage on line 8 of AndroidManifest.xml.		

Sink: AndroidManifest.xml:8 null()

```
6      <uses-sdk android:minSdkVersion="8" />
7      <uses-permission android:name="android.permission.INTERNET" />
8      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
9      <application
10         android:icon="@drawable/app_icon"
```

Category: Privilege Management: Android Network (2 Issues)



Abstract:

The program requests permission to make a network connection.

Explanation:

Granting this permission will allow the software to open network sockets. This permission would give the program control over the device that can negatively impact the user. Because this type of permission introduces potential risk, the system may not automatically grant it to the requestor.

Example 1: The <uses-permission .../> element of the AndroidManifest.xml below includes a network permission attribute.

<uses-permission android:name="android.permission.INTERNET"/>

Recommendations:

Do not request this privilege without consideration. If this permission is not required for the program, expect that the user will deny installation.

AndroidManifest.xml, line 7 (Privilege Management: Android Network)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The program requests permission to make a network connection on line 7 of AndroidManifest.xml.		
Sink:	AndroidManifest.xml:7 null()		

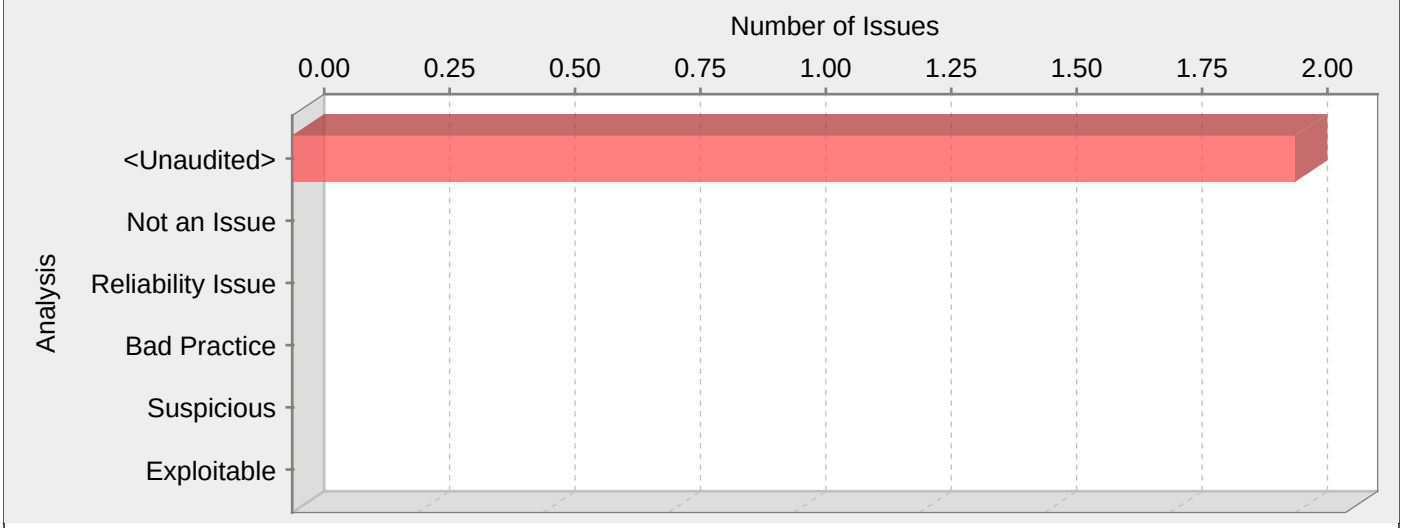
```
5      android:versionName="1.5" >
6      <uses-sdk android:minSdkVersion="8" />
7      <uses-permission android:name="android.permission.INTERNET" />
8      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
9      <application
```

AndroidManifest.xml, line 7 (Privilege Management: Android Network)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The program requests permission to make a network connection on line 7 of AndroidManifest.xml.		
Sink:	AndroidManifest.xml:7 null()		

```
5      android:versionName="1.5" >
6      <uses-sdk android:minSdkVersion="8" />
7      <uses-permission android:name="android.permission.INTERNET" />
8      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
9      <application
```

Category: Privilege Management: Unnecessary Permission (2 Issues)



Abstract:

The application fails to adhere to the principle of least privilege, which greatly amplifies the risk posed by other vulnerabilities.

Explanation:

An application should only have the minimum permissions required for its proper execution. Extra permissions might deter users from installing the application. This permission might be unnecessary for this program.

Recommendations:

Consider whether the application requires the requested permission in order to function properly. If it does not, you should remove the permission from the AndroidManifest.xml file. Do no over-permission the application by requesting more permissions than it really needs. This can lead to other malicious applications installed on the device taking advantage of such over-permissioned applications to adversely impact the user experience and the stored data. Additionally, extra permissions may unnecessarily discourage customers from installing your application.

AndroidManifest.xml, line 8 (Privilege Management: Unnecessary Permission)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The application fails to adhere to the principle of least privilege, which greatly amplifies the risk posed by other vulnerabilities.		

Sink: AndroidManifest.xml:8 null()

```
6     <uses-sdk android:minSdkVersion="8" />
7     <uses-permission android:name="android.permission.INTERNET" />
8     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
9     <application
10        android:icon="@drawable/app_icon"
```

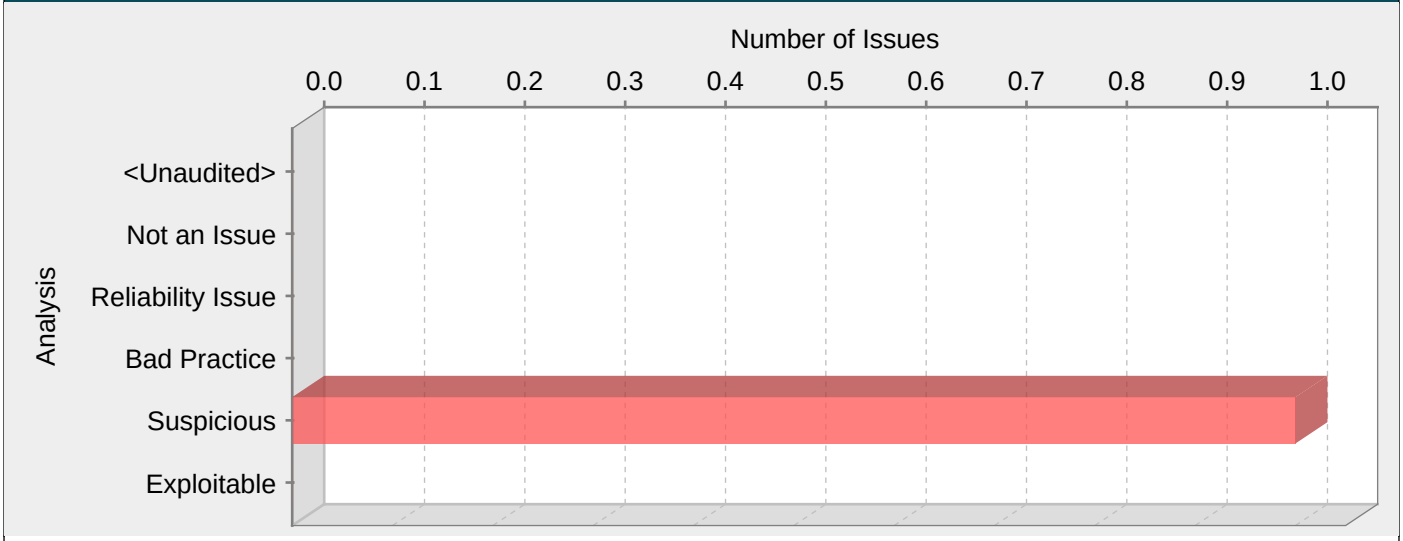
AndroidManifest.xml, line 8 (Privilege Management: Unnecessary Permission)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The application fails to adhere to the principle of least privilege, which greatly amplifies the risk posed by other vulnerabilities.		

Sink: AndroidManifest.xml:8 null()

```
6     <uses-sdk android:minSdkVersion="8" />
7     <uses-permission android:name="android.permission.INTERNET" />
8     <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE" />
9     <application
10        android:icon="@drawable/app_icon"
```

Category: Cross-Site Scripting: Persistent (1 Issues)



Abstract:

Sending unvalidated data to a web browser can result in the browser executing malicious code.

Explanation:

Cross-site scripting (XSS) vulnerabilities occur when:

1. Data enters a web application through an untrusted source. In the case of Persistent (also known as Stored) XSS, the untrusted source is typically a database or other back-end datastore, while in the case of Reflected XSS it is typically a web request.

2. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Example 1: The following JSP code segment queries a database for an employee with a given ID and prints the corresponding employee's name.

```
<%...
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select * from emp where id="+eid);
if (rs != null) {
rs.next();
String name = rs.getString("name");
}
%>
Employee Name: <%= name %>
```

This code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. This code can appear less dangerous because the value of name is read from a database, whose contents are apparently managed by the application. However, if the value of name originates from user-supplied data, then the database can be a conduit for malicious content. Without proper input validation on all data stored in the database, an attacker can execute malicious commands in the user's web browser. This type of exploit, known as Persistent (or Stored) XSS, is particularly insidious because the indirection caused by the data store makes it more difficult to identify the threat and increases the possibility that the attack will affect multiple users. XSS got its start in this form with web sites that offered a "guestbook" to visitors. Attackers would include JavaScript in their guestbook entries, and all subsequent visitors to the guestbook page would execute the malicious code.

Example 2: The following JSP code segment reads an employee ID, eid, from an HTTP request and displays it to the user.

```
<% String eid = request.getParameter("eid"); %>
...
Employee ID: <%= eid %>
```

As in Example 1, this code operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes meta-characters or source code, then the code will be executed by the web browser as it displays the HTTP response.

Initially this might not appear to be much of a vulnerability. After all, why would someone enter a URL that causes malicious code to run on their own computer? The real danger is that an attacker will create the malicious URL, then use e-mail or social engineering tricks to lure victims into visiting a link to the URL. When victims click the link, they unwittingly reflect the malicious content through the vulnerable web application back to their own computers. This mechanism of exploiting vulnerable web applications is known as Reflected XSS.

As the examples demonstrate, XSS vulnerabilities are caused by code that includes unvalidated data in an HTTP response. There are three vectors by which an XSS attack can reach a victim:

- As in Example 1, the application stores dangerous data in a database or other trusted data store. The dangerous data is subsequently read back into the application and included in dynamic content. Persistent XSS exploits occur when an attacker injects dangerous content into a data store that is later read and included in dynamic content. From an attacker's perspective, the optimal place to inject malicious content is in an area that is displayed to either many users or particularly interesting users. Interesting users typically have elevated privileges in the application or interact with sensitive data that is valuable to the attacker. If one of these users executes malicious content, the attacker may be able to perform privileged operations on behalf of the user or gain access to sensitive data belonging to the user.

- As in Example 2, data is read directly from the HTTP request and reflected back in the HTTP response. Reflected XSS exploits occur when an attacker causes a user to supply dangerous content to a vulnerable web application, which is then reflected back to the user and executed by the web browser. The most common mechanism for delivering malicious content is to include it as a parameter in a URL that is posted publicly or e-mailed directly to victims. URLs constructed in this manner constitute the core of many phishing schemes, whereby an attacker convinces victims to visit a URL that refers to a vulnerable site. After the site reflects the attacker's content back to the user, the content is executed and proceeds to transfer private information, such as cookies that may include session information, from the user's machine to the attacker or perform other nefarious activities.

- A source outside the application stores dangerous data in a database or other data store, and the dangerous data is subsequently read back into the application as trusted data and included in dynamic content.

A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 are among them. To highlight the unvalidated sources of input, the 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 Fortify 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.

Recommendations:

The solution to XSS is to ensure that validation occurs in the correct places and checks for the correct properties.

Since XSS vulnerabilities occur when an application includes malicious data in its output, one logical approach is to validate data immediately before it leaves the application. However, because web applications often have complex and intricate code for generating dynamic content, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for XSS.

Web applications must validate their input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for XSS is generally relatively easy. Despite its value, input validation for XSS does not take the place of rigorous output validation. An application may accept input through a shared data store or other trusted source, and that data store may accept input from a source that does not perform adequate input validation. Therefore, the application cannot implicitly rely on the safety of this or any other data. This means the best way to prevent XSS vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for XSS is to create a whitelist of safe characters that are allowed to appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alpha-numeric characters or a phone number might only include digits 0-9. However, this solution is often infeasible in web applications because many characters that have special meaning to the browser should still be considered valid input once they are encoded, such as a web design bulletin board that must accept HTML fragments from its users.

A more flexible, but less secure approach is known as blacklisting, which selectively rejects or escapes potentially dangerous characters before using the input. In order to form such a list, you first need to understand the set of characters that hold special meaning for web browsers. Although the HTML standard defines what characters have special meaning, many web browsers try to correct common mistakes in HTML and may treat other characters as special in certain contexts. The CERT(R) Coordination Center at the Software Engineering Institute at Carnegie Mellon University provides the following details about special characters in various contexts [1]:

In the content of a block-level element (in the middle of a paragraph of text):

- "<" is special because it introduces a tag.

- "&" is special because it introduces a character entity.

- ">" is special because some browsers treat it as special, on the assumption that the author of the page intended to include an opening "<", but omitted it in error.

The following principles apply to attribute values:

- In attribute values enclosed with double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed with single quote, the single quotes are special because they mark the end of the attribute value.

- In attribute values without any quotes, white-space characters, such as space and tab, are special.

- "&" is special when used with certain attributes, because it introduces a character entity.

In URLs, for example, a search engine might provide a link within the results page that the user can click to re-run the search. This can be implemented by encoding the search query inside the URL, which introduces additional special characters:

- Space, tab, and new line are special because they mark the end of the URL.

- "&" is special because it either introduces a character entity or separates CGI parameters.

- Non-ASCII characters (that is, everything above 128 in the ISO-8859-1 encoding) are not allowed in URLs, so they are considered to be special in this context.

- The "%" symbol must be filtered from input anywhere parameters encoded with HTTP escape sequences are decoded by server -side code. For example, "%" must be filtered if input such as "%68%65%6C%6C%6F" becomes "hello" when it appears on the web page in question.

Within the body of a <SCRIPT> </SCRIPT>:

- The semicolon, parenthesis, curly braces, and new line should be filtered in situations where text could be inserted directly into a pre-existing script tag.

Server-side scripts:

- Server-side scripts that convert any exclamation characters (!) in input to double-quote characters (") on output might require additional filtering.

Other possibilities:

- If an attacker submits a request in UTF-7, the special character '<' appears as '+ADw-' and may bypass filtering. If the output is included in a page that does not explicitly specify an encoding format, then some browsers try to intelligently identify the encoding based on the content (in this case, UTF-7).

Once you identify the correct points in an application to perform validation for XSS attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. If special characters are not considered valid input to the application, then you can reject any input that contains special characters as invalid. A second option in this situation is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and may be unacceptable in circumstances where the integrity of the input must be preserved for display.

If input containing special characters must be accepted and displayed accurately, validation must encode any special characters to remove their significance. A complete list of ISO 8859-1 encoded values for special characters is provided as part of the official HTML specification [2].

Many application servers attempt to limit an application's exposure to cross-site scripting vulnerabilities by providing implementations for the functions responsible for setting certain specific HTTP response content that perform validation for the characters essential to a cross-site scripting attack. Do not rely on the server running your application to make it secure. When an application is developed there are no guarantees about what application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will also stay in sync.

Tips:

1. The HP Fortify Secure Coding Rulepacks treat the database as a source of untrusted data, which can lead to XSS vulnerabilities. If the database is a trusted resource in your environment, use custom filters to filter out dataflow issues that include the DATABASE taint flag or originate from database sources.

2. Even though URL encoding untrusted data protects against many XSS attacks, some browsers (specifically, Internet Explorer 6 and 7 and possibly others) automatically decode content at certain locations within the Document Object Model (DOM) prior to passing it to the JavaScript interpreter. To reflect this danger, the rulepacks no longer treat URL encoding routines as sufficient to protect against Cross-Site Scripting. Data values that are URL encoded and subsequently output will cause Fortify to report Cross-Site Scripting: Poor Validation vulnerabilities.

3. Fortify RTA adds protection against this category.

TouchImageView.java, line 360 (Cross-Site Scripting: Persistent)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method doWork() in TouchImageView.java sends unvalidated data to a web browser on line 360, which can result in the browser executing malicious code.		
Source:	Local.java:555 android.database.sqlite.SQLiteDatabase.rawQuery()		

```
553    public  Exhibitor getExhibitor(BoothLocation boothLocation){
554      String statement = "SELECT * FROM Exhibitor WHERE boothname=?";
555      Cursor cursor = ourDatabase.rawQuery(statement, new
String[]{""+boothLocation.boothName});
556      Exhibitor exhibitor = null;
```

```
557                 for (boolean hasItem = cursor.moveToFirst(); hasItem;) {
```

Sink: TouchImageView.java:360 android.widget.Toast.setText()

```
358                     if(toast==null)
359                      toast = toast.makeText(context, exhibitor.company, Toast.LENGTH_SHORT);
360                     toast.setText(exhibitor.company);
361                     toast.show();
362                     break;
```

Analysis: Suspicious

Comments: *565075 2012-11-06 10:41 AM* Need to talk to security...

Category: Dead Code: Expression is Always true (1 Issues: 1 Hidden)

Abstract:

This expression (or part of it) will always evaluate to true.

Explanation:

This expression (or part of it) will always evaluate to true; the program could be rewritten in a simpler form. The nearby code may be present for debugging purposes, or it may not have been maintained along with the rest of the program. The expression may also be indicative of a bug earlier in the method.

Example 1: The following method never sets the variable secondCall after initializing it to true. (The variable firstCall is mistakenly used twice.) The result is that the expression firstCall || secondCall will always evaluate to true, so setUpForCall() will always be invoked.

```
public void setUpCalls() {
boolean firstCall = true;
boolean secondCall = true;

if (fCall < 0) {
cancelFCall();
firstCall = false;
}
if (sCall < 0) {
cancelSCall();
firstCall = false;
}

if (firstCall || secondCall) {
setUpForCall();
}
}
```

Example 2: The following method tries to check the variables firstCall and secondCall. (The variable firstCall is mistakenly set to true instead of being checked.) The result is that the expression first part of the expression firstCall = true && secondCall == true will always evaluate to true.

```
public void setUpCalls() {
boolean firstCall = false;
boolean secondCall = false;

if (fCall > 0) {
setUpFCall();
firstCall = true;
}
if (sCall > 0) {
setUpSCall();
secondCall = true;
```

```
}
if (firstCall = true && secondCall == true) {
setUpDualCall();
}
}
```

Recommendations:

In general, you should repair or remove unused code. It causes additional complexity and maintenance burden without contributing to the functionality of the program.

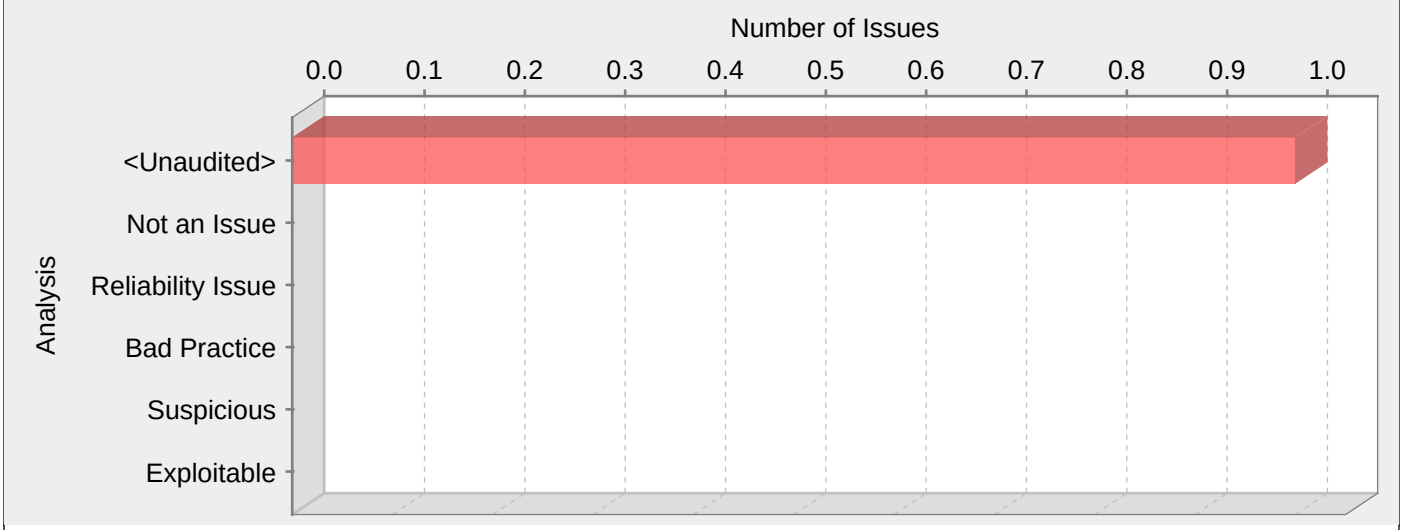
TouchImageView.java, line 73 (Dead Code: Expression is Always true) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The expression (or part of it) at TouchImageView.java line 73 will always evaluate to true.		

Sink: TouchImageView.java:73 IfStatement()

```
71                  super.setClickable(true);
72                  this.context = context;
73                if(conf == null)
74                  conf = (ConfApp)context.getApplicationContext();
75                  //this.setBackgroundResource(R.drawable.ballroom_1664x3324);
```

Category: Dead Code: Unused Method (1 Issues: 1 Hidden)



Abstract:

This method is not reachable from any method outside the class.

Explanation:

This method is never called or is only called from other dead code.

Example 1: In the following class, the method doWork() can never be called.

```
public class Dead {
private void doWork() {
System.out.println("doing work");
}
public static void main(String[] args) {
System.out.println("running Dead");
}
}
```

Example 2: In the following class, two private methods call each other, but since neither one is ever invoked from anywhere else, they are both dead code.

```
public class DoubleDead {
private void doTweedledee() {
doTweedledumb();
}
private void doTweedledumb() {
doTweedledee();
}
public static void main(String[] args) {
System.out.println("running DoubleDead");
}
}
```

(In this case it is a good thing that the methods are dead: invoking either one would cause an infinite loop.)

Recommendations:

A dead method may indicate a bug in dispatch code.

Example 3: If method is flagged as dead named getWitch() in a class that also contains the following dispatch method, it may be because of a copy-and-paste error. The 'w' case should return getWitch() not getMummy().

```
public ScaryThing getScaryThing(char st) {
switch(st) {
case 'm':
return getMummy();
case 'w':
```

```
return getMummy();
default:
return getBlob();
}
}
```

In general, you should repair or remove dead code. To repair dead code, execute the dead code directly or indirectly through a public method. Dead code causes additional complexity and maintenance burden without contributing to the functionality of the program.

Tips:

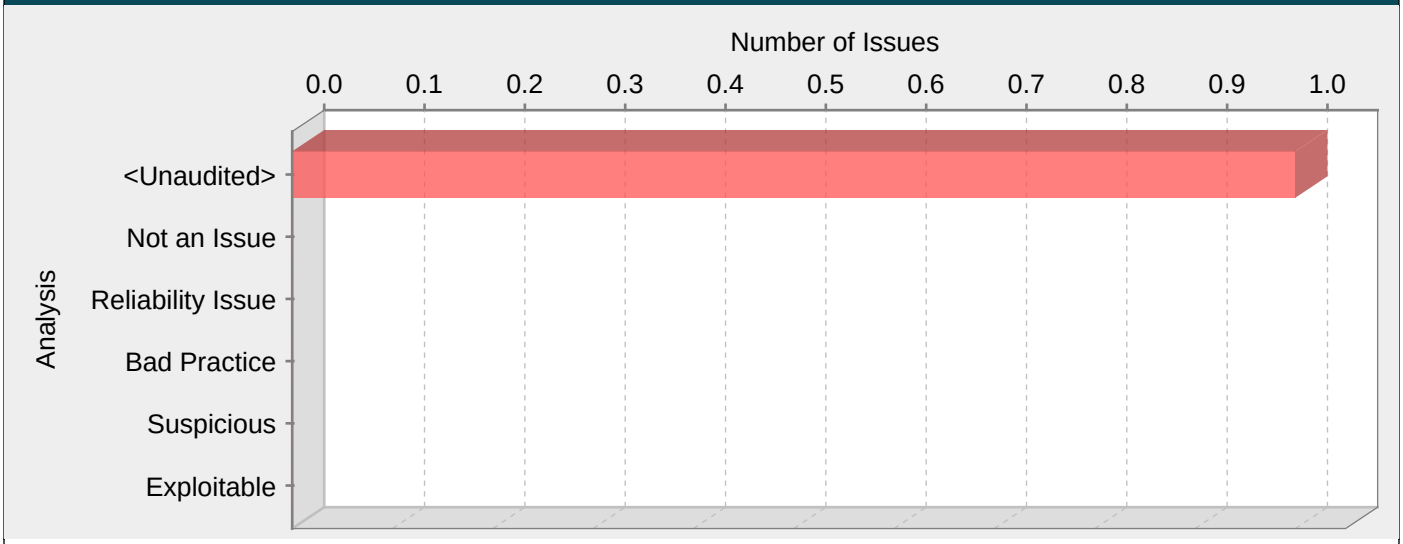
1. This issue may be a false positive if the program uses reflection to access private methods. (This is a non-standard practice. Private methods that are only invoked via reflection should be well documented.)

Local.java, line 56 (Dead Code: Unused Method) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Code Quality		
Abstract:	The method excStatements() in Local.java is not reachable from any method outside the class. It is dead code. Dead code is defined as code that is never directly or indirectly executed by a public method.		
Sink:	Local.java:56 Function: excStatements()		

```
54      db.execSQL(statement);
55    }
56    private void excStatements(SQLiteDatabase db,String[] statements){
57     for(String statement : statements){
58      db.execSQL(statement);
```

Category: Log Forging (1 Issues)



Abstract:

Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs.

Explanation:

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.

2. The data is written to an application or system log file.

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility [2].

Example: The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
String val = request.getParameter("val");
try {
int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
log.info("Failed to parse val = " + val);
}
```

If a user submits the string "twenty-one" for val, the following entry is logged:

INFO: Failed to parse val=twenty-one

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", the following entry is logged:

INFO: Failed to parse val=twenty-one

INFO: User logged out=badguy

Clearly, attackers can use this same mechanism to insert arbitrary log entries.

Recommendations:

Prevent log forging attacks with indirection: create a set of legitimate log entries that correspond to different events that must be logged and only log entries from this set. To capture dynamic content, such as users logging out of the system, always use server-controlled values rather than user-supplied data. This ensures that the input provided by the user is never used directly in a log entry.

In some situations this approach is impractical because the set of legitimate log entries is too large or complicated. In these situations, developers often fall back on blacklisting. Blacklisting selectively rejects or escapes potentially dangerous characters before using the input. However, a list of unsafe characters can quickly become incomplete or outdated. A better approach is to create a white list of characters that are allowed to appear in log entries and accept input composed exclusively of characters in the approved set. The most critical character in most log forging attacks is the '\n' (newline) character, which should never appear on a log entry white list.

Tips:

1. Many logging operations are created only for the purpose of debugging a program during development and testing. In our experience, debugging will be enabled, either accidentally or purposefully, in production at some point. Do not excuse log forging vulnerabilities simply because a programmer says "I don't have any plans to turn that on in production".

2. A number of modern web frameworks provide mechanisms for performing validation of user input. Struts and Struts 2 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 Fortify 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.

MainActivity.java, line 107 (Log Forging)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method outputToFile() in MainActivity.java writes unvalidated user input to the log on line 107. An attacker could take advantage of this behavior to forge log entries or inject malicious content into the log.		

Source: Local.java:499 android.database.sqlite.SQLiteDatabase.rawQuery()

```
497   ArrayList<BoothLocation> booths = new ArrayList<BoothLocation>();
498   String statement = "SELECT * FROM BoothLocation";
499   Cursor cursor = ourDatabase.rawQuery(statement, null);
500   for (boolean hasItem = cursor.moveToFirst(); hasItem; hasItem = cursor.moveToNext()) {
501    BoothLocation boothLoc = new BoothLocation(cursor);
```

Sink: MainActivity.java:107 android.util.Log.wtf()

```
105
106           }
107           Log.wtf("HERE", output);
108           System.out.println(output);
109           writer.flush();
```

Category: Poor Error Handling: Overly Broad Catch (1 Issues)

Abstract:

The catch block handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.

Explanation:

Multiple catch blocks can get ugly and repetitive, but "condensing" catch blocks by catching a high-level class like Exception can obscure exceptions that deserve special treatment or that should not be caught at this point in the program. Catching an overly broad exception essentially defeats the purpose of Java's typed exceptions, and can become particularly dangerous if the program grows and begins to throw new types of exceptions. The new exception types will not receive any attention.

Example: The following code excerpt handles three types of exceptions in an identical fashion.

```
try {
doExchange();
}
catch (IOException e) {
logger.error("doExchange failed", e);
}
catch (InvocationTargetException e) {
logger.error("doExchange failed", e);
}
catch (SQLException e) {
logger.error("doExchange failed", e);
}
```

At first blush, it may seem preferable to deal with these exceptions in a single catch block, as follows:

```
try {
doExchange();
}
catch (Exception e) {
logger.error("doExchange failed", e);
}
```

However, if doExchange() is modified to throw a new type of exception that should be handled in some different kind of way, the broad catch block will prevent the compiler from pointing out the situation. Further, the new catch block will now also handle exceptions derived from RuntimeException such as ClassCastException, and NullPointerException, which is not the programmer's intent.

Recommendations:

Do not catch broad exception classes like Exception, Throwable, Error, or <RuntimeException> except at the very top level of the program or thread.

Tips:

1. The HP Fortify Secure Coding Rulepacks will not flag an overly broad catch block if the catch block in question immediately throws a new exception.

BioDetails.java, line 66 (Poor Error Handling: Overly Broad Catch)			
Fortify Priority:	Low	**Folder**	Low
Kingdom:	Errors		
Abstract:	The catch block at BioDetails.java line 66 handles a broad swath of exceptions, potentially trapping dissimilar issues or problems that should not be dealt with at this point in the program.		
Sink:	BioDetails.java:66 CatchBlock()		

```
64                  Field idField = c.getDeclaredField(variableName);
65                  return idField.getInt(idField);
66              } catch (Exception e) {
67                  e.printStackTrace();
68                  return -1;
```

Category: Poor Style: Non-final Public Static Field (1 Issues: 1 Hidden)

Abstract:

Non-final public static fields can be changed by external classes.

Explanation:

Typically, you do not want to provide external classes direct access to your object's member fields since a public field can be changed by any external class. Good object oriented designed uses encapsulation to prevent implementation details, such as member fields, from being exposed to other classes. Further, if the system assumes that this field cannot be changed, then malicious code might be able to adversely change the behavior of the system.

'

Example 1: In the following code, the field ERROR_CODE is declared as public and static, but not final:

```
public class MyClass
{
public static int ERROR_CODE = 100;
//...
}
```

In this case, malicious code might be able to change this error code and cause the program to behave in an unexpected manner.

This category is from the Cigital Java Rulepack. http://www.cigital.com/securitypack/

Recommendations:

If you intend to expose a field as a constant value, the field should be declared as public static final, otherwise declare the field private.

Example 2:

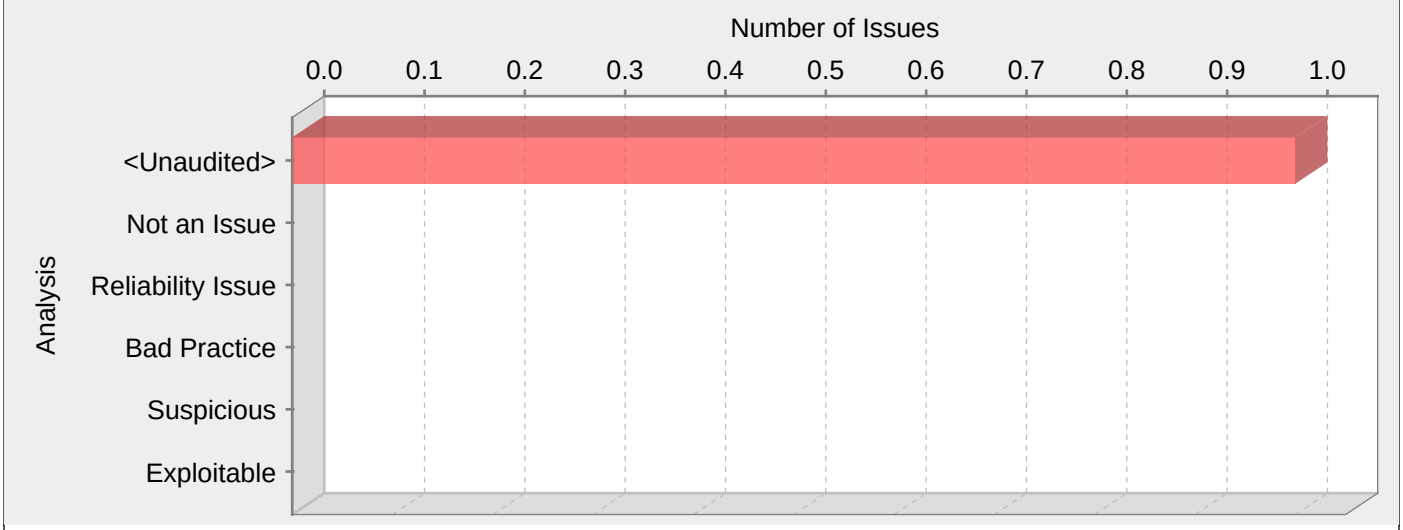
```
public class MyClass
{
public static final int ERROR_CODE = 123;
//...
}
```

ConfApp.java, line 18 (Poor Style: Non-final Public Static Field) [Hidden]

Fortify Priority:	Low	**Folder**	Low
Kingdom:	Encapsulation		
Abstract:	Non-final public static fields can be changed by external classes.		
Sink:	ConfApp.java:18 Field: confApp()		

```
16    public Local local;
17
18    public static ConfApp confApp;
19
20
```

Category: Unreleased Resource: Streams (1 Issues: 1 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 might be able to launch a denial of service attack by depleting the resource pool.

Example 1: 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);
}
}
```

Example 2: Under normal conditions, the following code executes a database query, processes the results returned by the database, and closes the allocated statement object. But if an exception occurs while executing the SQL or processing the results, the statement object will not be closed. If this happens often enough, the database will run out of available cursors and not be able to execute any more SQL queries.

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
harvestResults(rs);
stmt.close();
```

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 Example 2 should be rewritten as follows:

```
public void execCxnSql(Connection conn) {
Statement stmt;
try {
stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(CXN_SQL);
...
}
finally {
if (stmt != null) {
safeClose(stmt);
}
}
}
public static void safeClose(Statement stmt) {
if (stmt != null) {
try {
stmt.close();
} catch (SQLException e) {
log(e);
}
}
}
```

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

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

Tips:

1. Be aware that closing a database connection may or may not automatically free other resources associated with the connection object. If the application uses connection pooling, it is best to explicitly close the other resources after the connection is closed. If the application is not using connection pooling, the other resources are automatically closed when the database connection is closed. In such a case, this vulnerability is invalid.

MainActivity.java, line 92 (Unreleased Resource: Streams) [Hidden]

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function outputToFile() in MainActivity.java sometimes fails to release a system resource allocated by FileWriter() on line 92.		
Sink:	MainActivity.java:92 writer = new BufferedWriter(new java.io.FileWriter())		

```
90          // File fileWithinMyDir = new File(mydir, "myfile"); //Getting a file within
the dir.
91
92          BufferedWriter writer = new BufferedWriter(new FileWriter(file));
93
94          boolean first=true;
```