Fortify Security Report

23 Feb 2022

Demo User

Executive Summary

Issues Overview

On 23 Feb 2022, a source code review was performed over the FortifyWebAppDemo code base. 8 files, 101 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 23 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	14
High	7
Critical	2

Recommendations and Conclusions

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

Project Summary

Code Base Summary

Code location: C:/Users/klee/source/repos/FortifyWebAppDemo/src
Number of Files: 8
Lines of Code: 101
Build Label: SNAPSHOT

Scan Information

Scan time: 00:15
SCA Engine version: 21.2.2.0004
Machine Name: GBklee01
Username running scan: klee

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

Attack Surface

Attack Surface:
Command Line Arguments:
com.microfocus.app.WebApp.main

Environment Variables:
java.lang.System.getenv

Private Information:
null.null.null
java.lang.System.getenv

System Information:
null.null.null

Filter Set Summary

Current Enabled Filter Set:
Security Auditor View

Filter Set Details:

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

Audit Guide Summary

Audit guide not enabled

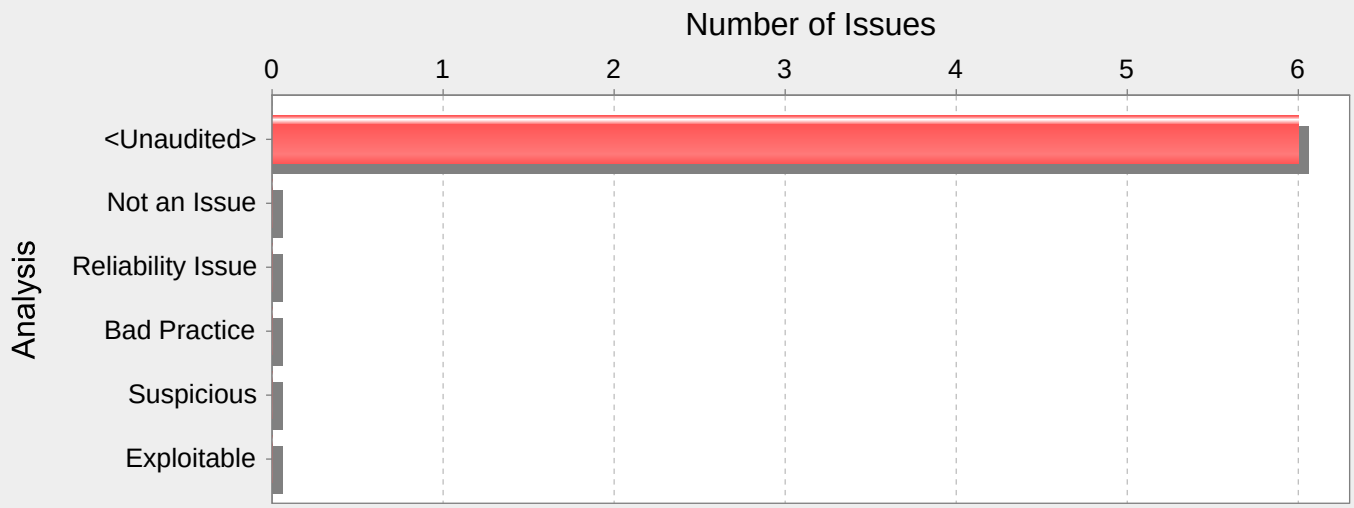
Results Outline

Overall number of results

The scan found 23 issues.

Vulnerability Examples by Category

Category: Unreleased Resource: Database (6 Issues)



Abstract:

The function getProducts() in DbAccess.java sometimes fails to release a database resource allocated by getConnection() on line 22.

Explanation:

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

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

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

Example: 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 the Example 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:

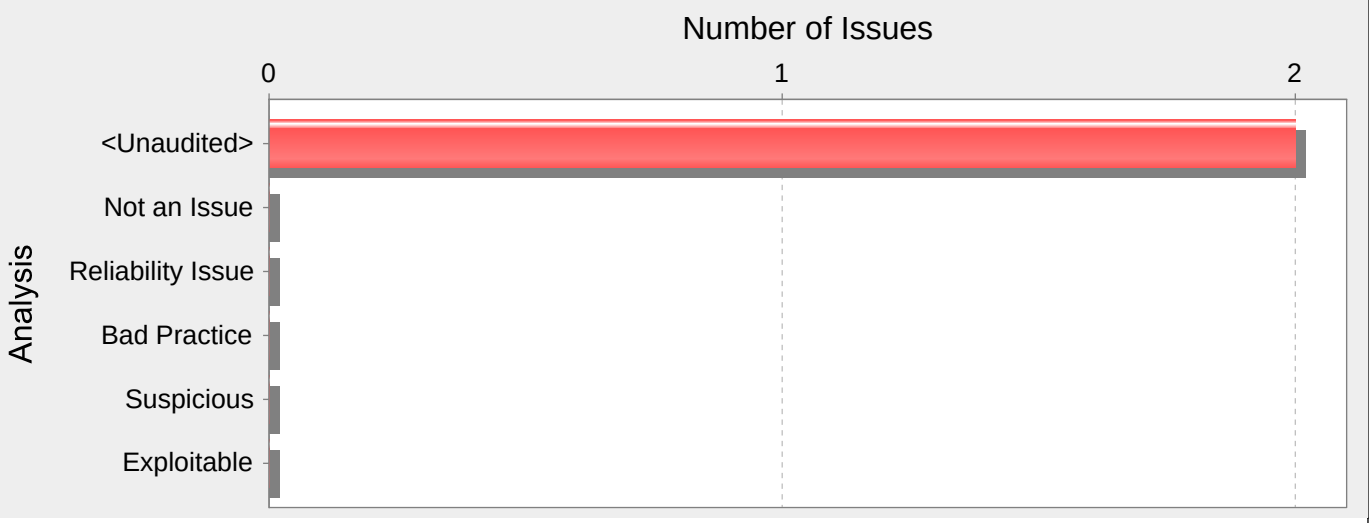
1. Be aware that closing a database connection may or may not automatically free other resources associated with the connection object. If the application uses connection pooling, it is best to explicitly close the other resources after the connection is closed. If the application is not using connection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

DbAccess.java, line 22 (Unreleased Resource: Database)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function getProducts() in DbAccess.java sometimes fails to release a database resource allocated by getConnection() on line 22.		
Sink:	DbAccess.java:22 conn = getConnection()		

```
20
21                  public ArrayList<Product> getProducts(String query) throws Exception {
22                          Connection conn = getConnection();
23
24                   initDbIfNeed(conn);
```

Category: Password Management: Hardcoded Password (2 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is not easy to remedy.

Explanation:

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. After the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system must choose between security and availability.

Example 1: The following code uses a hardcoded password to connect to a database:

```
...
DriverManager.getConnection(url, "scott", "tiger");
...
```

This code will run successfully, but anyone who has access to it will have access to the password. After the program ships, there is likely no way to change the database user "scott" with a password of "tiger" unless the program is patched. An employee with access to this information can use it to break into the system. Even worse, if attackers have access to the bytecode for the application they can use the javap -c command to access the disassembled code, which will contain the values of the passwords used. The result of this operation might look something like the following for Example 1:

```
javap -c ConnMngr.class
22: ldc   #36; //String jdbc:mysql://ixne.com/rxsql
24: ldc   #38; //String scott
26: ldc   #17; //String tiger
```

In the mobile environment, password management is especially important given that there is such a high chance of device loss.

Example 2: The following code uses hardcoded username and password to setup authentication for viewing protected pages with Android's WebView.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
handler.proceed("guest", "allow");
}
});
...
```

Similar to Example 1, this code will run successfully, but anyone who has access to it will have access to the password.

Recommendations:

Passwords should never be hardcoded and should generally be obfuscated and managed in an external source. Storing passwords in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the password. At the very least, hash passwords before storing them.

Some third-party products claim the ability to securely manage passwords. For example, WebSphere Application Server 4.x uses a simple XOR encryption algorithm for obfuscating values, but be skeptical about such facilities. WebSphere and other application servers offer outdated and relatively weak encryption mechanisms that are insufficient for security-sensitive environments. Today, the best option for a secure generic solution is to create a proprietary mechanism yourself.

For Android, as well as any other platform that uses SQLite database, SQLCipher is a good alternative. SQLCipher is an extension to the SQLite database that provides transparent 256-bit AES encryption of database files. Thus, credentials can be stored in an encrypted database.

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

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

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

To enable encryption on the WebView store, you must recompile WebKit with the sqlcipher.so library.

Tips:

1. You can use the Fortify Java Annotations FortifyPassword and FortifyNotPassword to indicate which fields and variables represent passwords.

2. To identify null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Fortify Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named fields and variables.

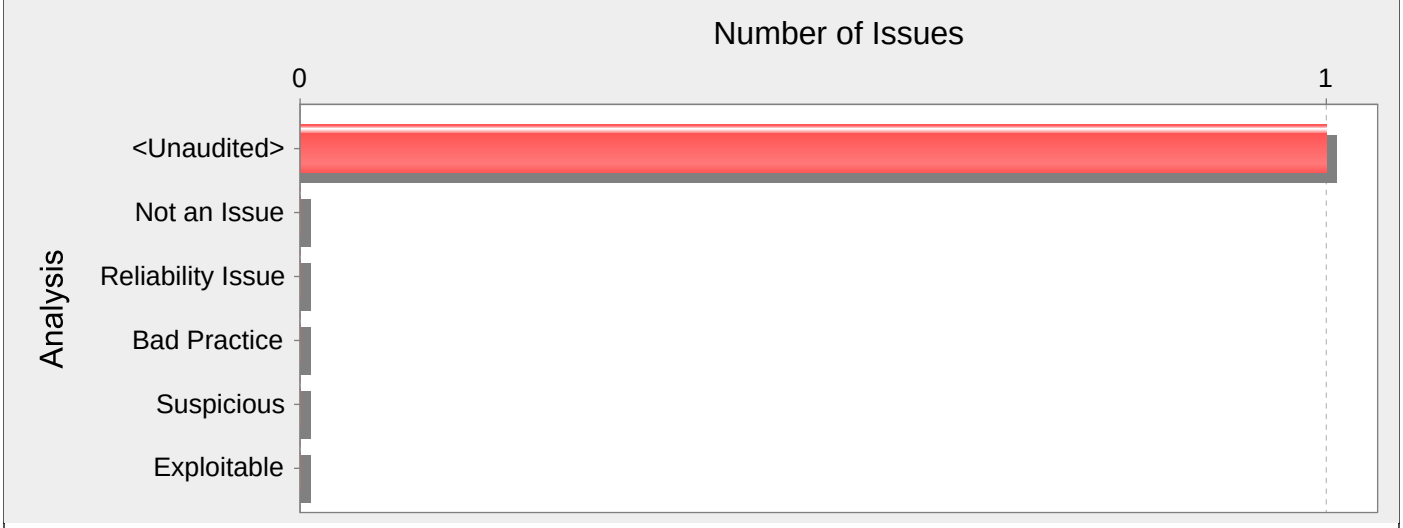
DbAccess.java, line 19 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that is not easy to remedy.		

Sink: DbAccess.java:19 FieldAccess: DEFAULT_PASSWORD()

```
17  private final static String DEFAULT_DB = "products";
18  private final static String DEFAULT_USER = "mysql";
19  private final static String DEFAULT_PASSWORD =******
20
21  public ArrayList<Product> getProducts(String query) throws Exception {
```

Category: SQL Injection (1 Issues)



Abstract:

On line 36 of DbAccess.java, the method getProducts() invokes a SQL query built with input that comes from an untrusted source. This call 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.

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 intends to execute the following code:

```
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 must 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 shown 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';
```

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

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

```
...
PasswordAuthentication pa = authenticator.getPasswordAuthentication();
String userName = pa.getUserName();
String itemName = this.getIntent().getExtras().getString("itemName");
String query = "SELECT * FROM items WHERE owner = '"
+ userName + "' AND itemname = '"
+ itemName + "'";
SQLiteDatabase db = this.openOrCreateDatabase("DB", MODE_PRIVATE, null);
Cursor c = db.rawQuery(query, null);
...
```

One traditional approach to preventing SQL injection attacks is to handle them as an input validation problem and either accept only characters from an allow list of safe values or identify and escape a list of potentially malicious values (deny list). Checking an allow list 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, implementing a deny list is riddled with loopholes that make it ineffective at preventing SQL injection attacks. For example, attackers may:

- Target fields that are not quoted
- Find ways to bypass the need for certain escaped metacharacters
- Use stored procedures to hide the injected metacharacters

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 where user-supplied data needs to be included, they include bind parameters, which are placeholders for data that is subsequently inserted. In other words, bind parameters allow the programmer 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 each of the bind parameters without the risk that the data will be interpreted as a modification to the command.

Example 1 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();
...

And here is an Android equivalent:

...
PasswordAuthentication pa = authenticator.getPasswordAuthentication();
String userName = pa.getUserName();
String itemName = this.getIntent().getExtras().getString("itemName");
String query = "SELECT * FROM items WHERE itemname=? AND owner=?";
SQLiteDatabase db = this.openOrCreateDatabase("DB", MODE_PRIVATE, null);
Cursor c = db.rawQuery(query, new Object[]{itemName, userName});
...

More complicated scenarios, often found in report generation code, require that user input affect the structure of the SQL statement, for instance by adding a dynamic constraint in the WHERE clause. Do not use this requirement to justify concatenating user input to create a query string. Prevent SQL injection attacks where user input must affect 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 parameterized 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. A number of modern web frameworks provide mechanisms to perform user input validation (including Struts and Spring MVC). To highlight the unvalidated sources of input, Fortify Secure Coding Rulepacks dynamically re-prioritize the issues Fortify Static Code Analyzer reports by lowering their probability of exploit and providing pointers to the supporting evidence whenever the framework validation mechanism is in use. We refer to this feature as Context-Sensitive Ranking. To further assist the Fortify user with the auditing process, the Fortify Software Security Research group makes available the Data Validation project template that groups the issues into folders based on the validation mechanism applied to their source of input.

3. Fortify AppDefender adds protection against this category.

DbAccess.java, line 36 (SQL Injection)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	On line 36 of DbAccess.java, the method getProducts() invokes a SQL query built with input that comes from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.		
Source:	products.jsp:68 javax.servlet.ServletRequest.getParameter()		

```
66  <%
67    DbAccess da=new DbAccess();
68    String keywords = request.getParameter("keywords");
69    if (keywords == null) keywords = "";
```

```
70                    ArrayList<Product> products = da.getProducts(keywords);
```

Sink: DbAccess.java:36 java.sql.Statement.executeQuery()

```
34
35                    String sql = "SELECT * FROM products WHERE Title LIKE '%" + query + "%' OR
                  Description LIKE '%" + query + "%'";
36                    ResultSet rs = stmt.executeQuery(sql);
37
38                ArrayList<Product> products= new ArrayList<Product>();
```

Issue Count by Category

Issues by Category	
Unreleased Resource: Database	6
Poor Error Handling: Overly Broad Throws	4
Hardcoded Domain in HTML	3
System Information Leak: HTML Comment in JSP	3
Password Management: Hardcoded Password	2
Cross-Site Request Forgery	1
J2EE Bad Practices: getConnection()	1
Password Management	1
Password Management: Password in Comment	1
SQL Injection	1

Issue Breakdown by Analysis

Issues by Analysis