Fortify Security Report

4/12/23

Demo User

Executive Summary

Issues Overview

On 12 Apr 2023, a source code review was performed over the IWAPharmacyDirect code base. 252 files, 4,515 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 353 reviewed findings were uncovered during the analysis.

Issues by Fortify Priority Order	
Low	249
High	53
Critical	41
Medium	10

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/IWAPharmacyDirect
Number of Files: 252
Lines of Code: 4515
Build Label: SNAPSHOT

Scan Information

Scan time: 01:13
SCA Engine version: 22.2.0.0130
Machine Name: GBklee02
Username running scan: klee

Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

rules/sca-custom-rules.xml has Valid signature

Attack Surface

Attack Surface:
Command Line Arguments:
com.microfocus.example.Application.main

Environment Variables:
null.null.null
java.lang.System.getenv

File System:
java.io.FileReader.FileReader
java.io.FileReader.FileReader
java.util.zip.ZipFile.entries

Private Information:
null.null.null
com.microfocus.example.utils.EmailUtils.setEmailPassword
java.lang.System.getenv

Java Properties:
null.null.null

java.lang.System.getProperty

System Information:
null.null.null
null.null.lambda
null.null.resolve
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.Throwable.getLocalizedMessage
java.lang.Throwable.getMessage
org.springframework.web.bind.MissingServletRequestParameterException.getParameterName

Web:
null.~JS_Generic.val
org.springframework.web.servlet.LocaleResolver.resolveLocale

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

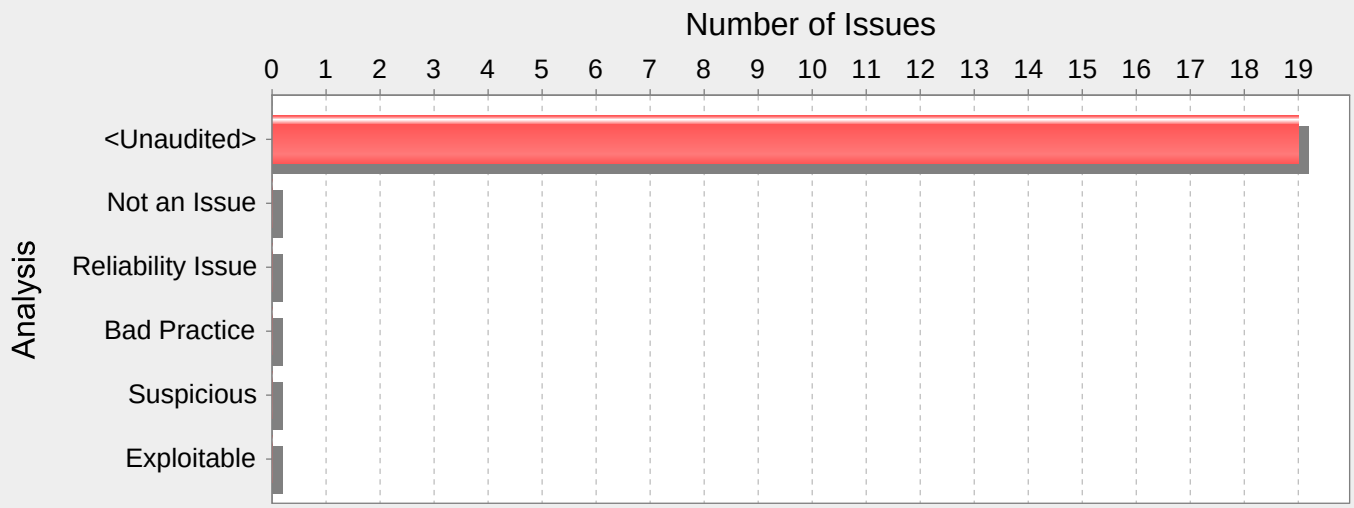
Results Outline

Overall number of results

The scan found 353 issues.

Vulnerability Examples by Category

Category: Path Manipulation (19 Issues)



Abstract:

Attackers can control the file system path argument to get() at FileSystemStorageService.java line 37, which allows them to access or modify otherwise protected files.

Explanation:

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

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

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

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

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

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

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

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

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

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

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

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

Recommendations:

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

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

Tips:

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

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

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

FileSystemStorageService.java, line 37 (Path Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	Attackers can control the file system path argument to get() at FileSystemStorageService.java line 37, which allows them to access or modify otherwise protected files.		

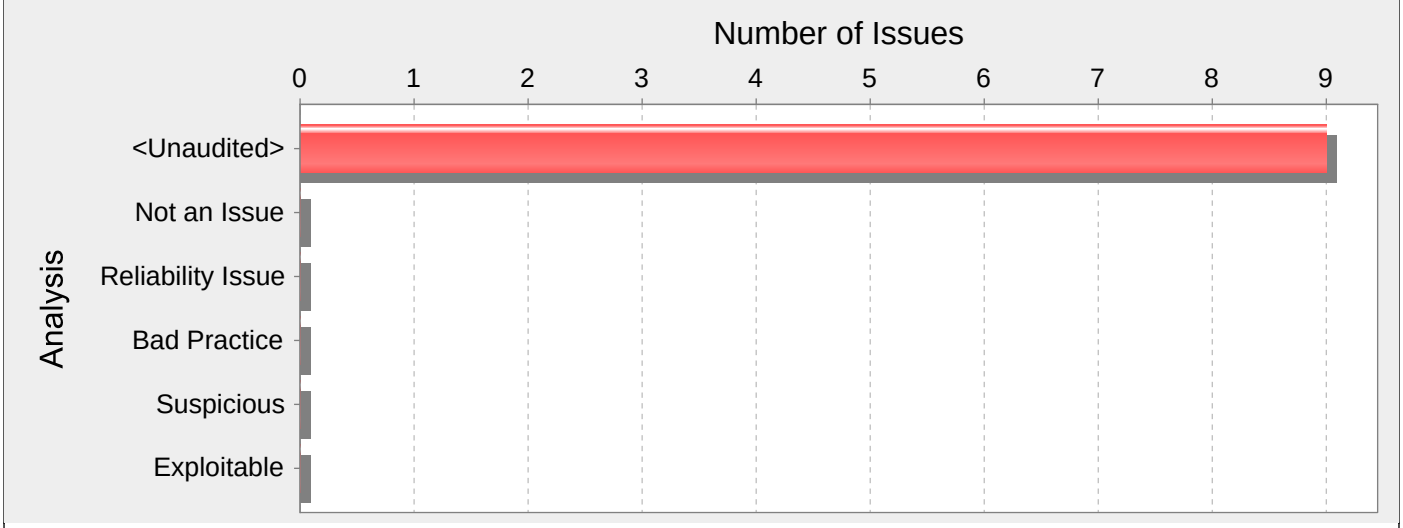
Source: StorageProperties.java:16 Read this.location()

```
14
15                      public String getLocation() {
16                          return location;
17                      }
```

Sink: FileSystemStorageService.java:37 java.nio.file.Paths.get()

```
35                      @Autowired
36                      public FileSystemStorageService(StorageProperties properties) {
37                          this.rootLocation = Paths.get(properties.getLocation());
38                          if (!Files.exists(this.rootLocation)) {
39                              log.debug("Creating storage service directory: " +
rootLocation.toString());
```

Category: Password Management: Hardcoded Password (9 Issues)



Abstract:

Hardcoded passwords can compromise system security in a way that is difficult to remedy.

Explanation:

Never hardcode passwords. Not only does it expose the password to all of the project's developers, it also makes fixing the problem extremely difficult. After the code is in production, a program patch is probably the only way to change the password. If the account the password protects is compromised, the system owners must choose between security and availability.

Example: The following YAML uses a hardcoded password:

...
credential_settings:
username: scott
password: tiger
...

This configuration may be valid, but anyone who has access to the configuration will have access to the password. After the program is released, changing the default user account "scott" with a password of "tiger" is difficult. Anyone with access to this information can use it to break into the system.

Recommendations:

Never hardcode password. Passwords 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.

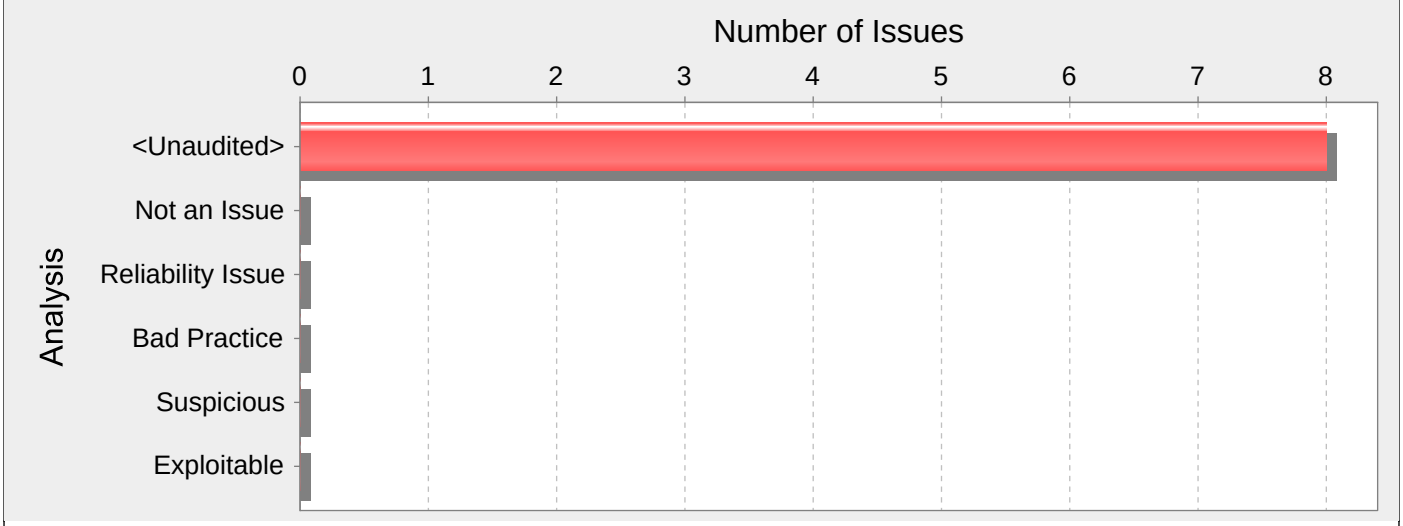
application-dev.yml, line 35 (Password Management: Hardcoded Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded passwords can compromise system security in a way that is difficult to remedy.		

Sink: application-dev.yml:35 ConfigPair()

```
33      driver-class-name: org.h2.Driver
34      url: jdbc:h2:mem:iwa_dev
35      username: sa
36      password: password
37      initialization-mode: always
```

Category: Access Control: Database (8 Issues)



Abstract:

Without proper access control, the method saveReviewFromAdminReviewForm() in ProductService.java can execute a SQL statement on line 242 that contains an attacker-controlled primary key, thereby allowing the attacker to access 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 might 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.

Some think that in the mobile world, classic web application vulnerabilities, such as database access control errors, 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 2: The following code adapts Example 1 to the Android platform.

```
...
String id = this.getIntent().getExtras().getString("invoiceID");
String query = "SELECT * FROM invoices WHERE id = ?";
SQLiteDatabase db = this.openOrCreateDatabase("DB", MODE_PRIVATE, null);
Cursor c = db.rawQuery(query, new Object[]{id});
...
```

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.

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 3: The following code implements the same functionality as Example 1 but imposes an additional constraint to verify that the invoice belongs to the currently authenticated user.

```
...
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();
...
```

And here is an Android equivalent:

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

ProductService.java, line 242 (Access Control: Database)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Without proper access control, the method saveReviewFromAdminReviewForm() in ProductService.java can execute a SQL statement on line 242 that contains an attacker-controlled primary key, thereby allowing the attacker to access unauthorized records.		

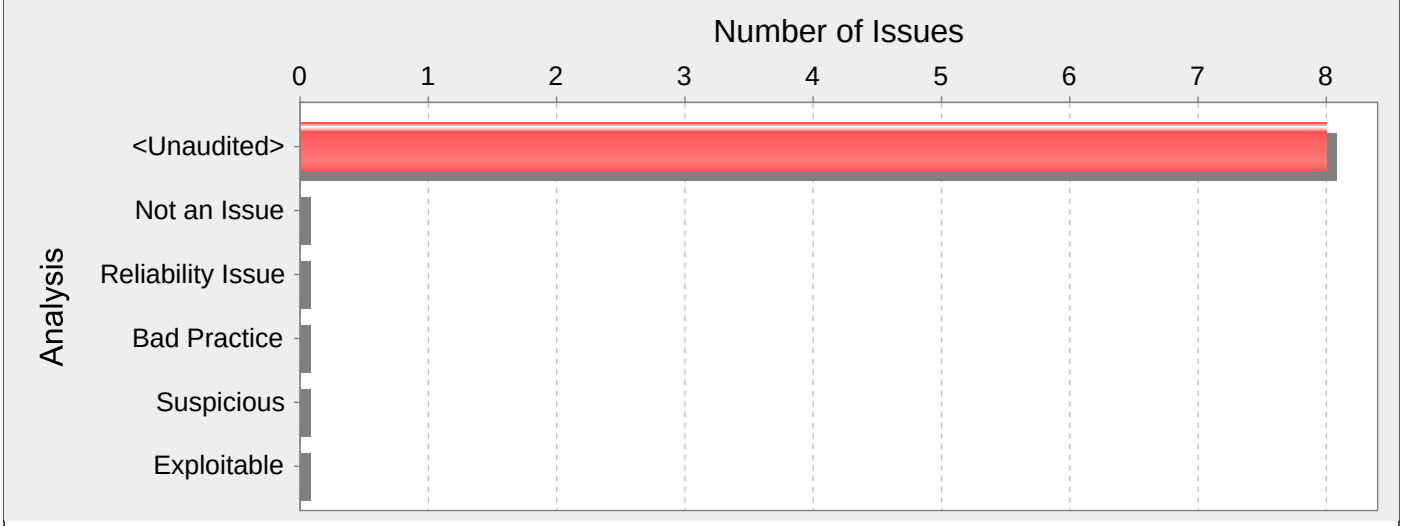
Source: AdminReviewController.java:101 saveReview(0)

```
99
100                     @PostMapping("/{id}/save")
101                     public String saveReview(@Valid @ModelAttribute("adminReviewForm") AdminReviewForm
adminReviewForm,
102                                              BindingResult bindingResult, Model model,
103                                              RedirectAttributes redirectAttributes,
```

Sink: ProductService.java:242 org.springframework.data.repository.CrudRepository.findById()

```
240
241                     public Review saveReviewFromAdminReviewForm(AdminReviewForm adminReviewForm)
throws ReviewNotFoundException {
242                         Optional<Review> optionalReview =
reviewRepository.findById(adminReviewForm.getId());
243                         if (optionalReview.isPresent()) {
244                             Review rtmp = optionalReview.get();
```

Category: SQL Injection (8 Issues)



Abstract:

On line 55 of ProductRepository.java, the method findAll() 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.

ProductRepository.java, line 55 (SQL Injection)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	On line 55 of ProductRepository.java, the method findAll() 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: ProductService.java:100 Read this.pageSize()

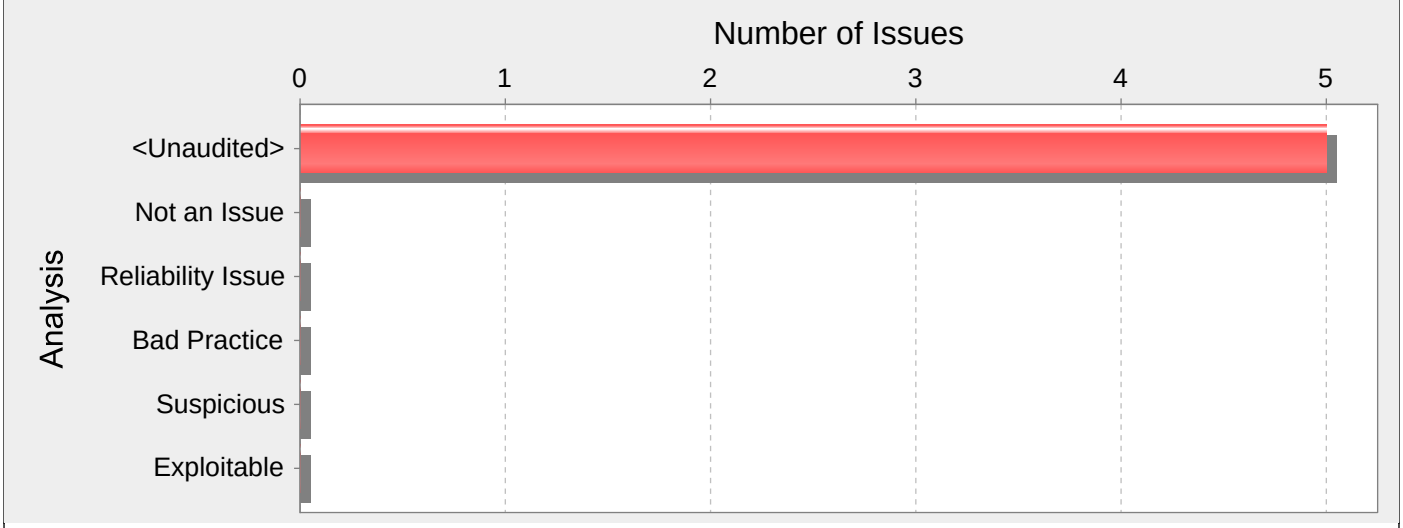
```
98                  return productRepository.findByKeywords(keywords, offset, pageSize);
99              } else {
100                 return productRepository.findAll(offset, pageSize);
101             }
```

```
102                 }
```

Sink: ProductRepository.java:55
org.springframework.jdbc.core.JdbcTemplate.query()

```
53                  String sqlQuery = "select * from products" +
54                          " LIMIT " + limit + " OFFSET " + offset;
55                  return jdbcTemplate.query(sqlQuery, new ProductMapper());
56                  }
```

Category: Insecure Randomness (5 Issues)



Abstract:

The random number generator implemented by nextInt() cannot withstand a cryptographic attack.

Explanation:

Insecure randomness errors occur when a function that can produce predictable values is used as a source of randomness in a security-sensitive context.

Computers are deterministic machines, and as such are unable to produce true randomness. Pseudorandom Number Generators (PRNGs) approximate randomness algorithmically, starting with a seed from which subsequent values are calculated.

There are two types of PRNGs: statistical and cryptographic. Statistical PRNGs provide useful statistical properties, but their output is highly predictable and form an easy to reproduce numeric stream that is unsuitable for use in cases where security depends on generated values being unpredictable. Cryptographic PRNGs address this problem by generating output that is more difficult to predict. For a value to be cryptographically secure, it must be impossible or highly improbable for an attacker to distinguish between the generated random value and a truly random value. In general, if a PRNG algorithm is not advertised as being cryptographically secure, then it is probably a statistical PRNG and should not be used in security-sensitive contexts, where its use can lead to serious vulnerabilities such as easy-to-guess temporary passwords, predictable cryptographic keys, session hijacking, and DNS spoofing.

Example: The following code uses a statistical PRNG to create a URL for a receipt that remains active for some period of time after a purchase.

```
String GenerateReceiptURL(String baseUrl) {
Random ranGen = new Random();
ranGen.setSeed((new Date()).getTime());
return (baseUrl + ranGen.nextInt(400000000) + ".html");
}
```

This code uses the Random.nextInt() function to generate "unique" identifiers for the receipt pages it generates. Since Random.nextInt() is a statistical PRNG, it is easy for an attacker to guess the strings it generates. Although the underlying design of the receipt system is also faulty, it would be more secure if it used a random number generator that did not produce predictable receipt identifiers, such as a cryptographic PRNG.

Recommendations:

When unpredictability is critical, as is the case with most security-sensitive uses of randomness, use a cryptographic PRNG. Regardless of the PRNG you choose, always use a value with sufficient entropy to seed the algorithm. (Do not use values such as the current time because it offers only negligible entropy.)

The Java language provides a cryptographic PRNG in java.security.SecureRandom. As is the case with other algorithm-based classes in java.security, SecureRandom provides an implementation-independent wrapper around a particular set of algorithms. When you request an instance of a SecureRandom object using SecureRandom.getInstance(), you can request a specific implementation of the algorithm. If the algorithm is available, then it is given as a SecureRandom object. If it is unavailable or if you do not specify a particular implementation, then you are given a SecureRandom implementation selected by the system.

Sun provides a single SecureRandom implementation with the Java distribution named SHA1PRNG, which Sun describes as computing:

"The SHA-1 hash over a true-random seed value concatenated with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA-1 output, only 64 bits are used [1]."

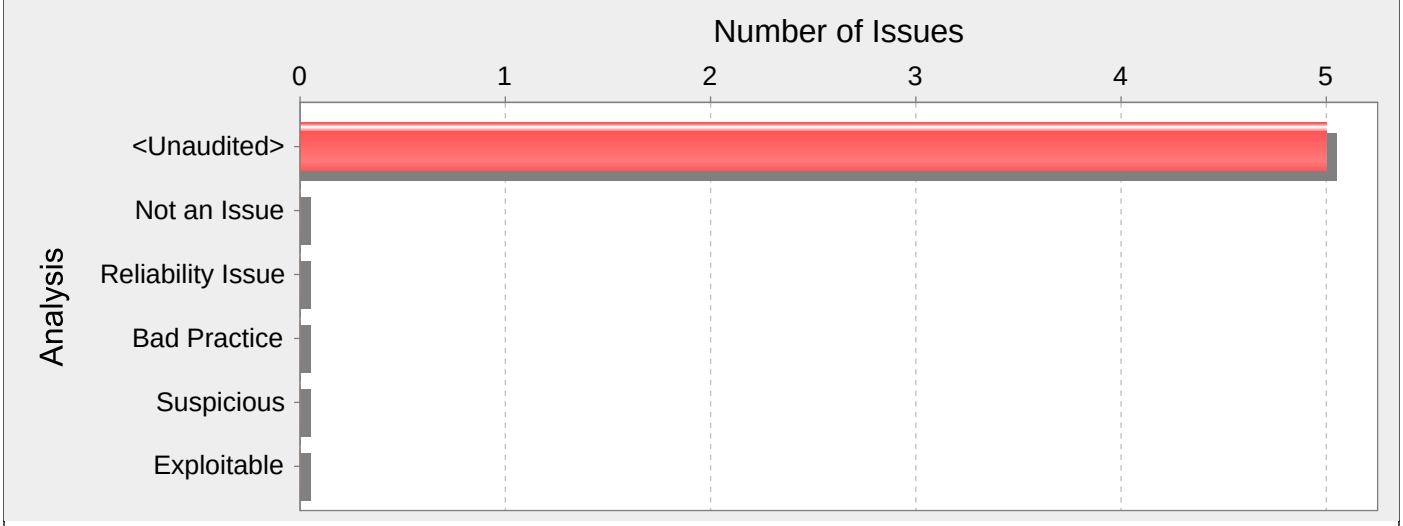
However, the specifics of the Sun implementation of the SHA1PRNG algorithm are poorly documented, and it is unclear what sources of entropy the implementation uses and therefore what amount of true randomness exists in its output. Although there is speculation on the Web about the Sun implementation, there is no evidence to contradict the claim that the algorithm is cryptographically strong and can be used safely in security-sensitive contexts.

ProductService.java, line 293 (Insecure Randomness)			
Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The random number generator implemented by nextInt() cannot withstand a cryptographic attack.		

Sink: ProductService.java:293 nextInt()

```
291         int low = 10;
292         int high = 100;
293         int result = r.nextInt(high-low) + low;
294         String formatted = String.format("%03d", result);
295         otmp.setOrderNum("OID-P100-"+formatted);
```

Category: JSON Injection (5 Issues)



Abstract:

On line 101 of UserUtils.java, the method registerUser() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.

Explanation:

JSON injection occurs when:

1. Data enters a program from an untrusted source.

2. The data is written to a JSON stream.

Applications typically use JSON to store data or send messages. When used to store data, JSON is often treated like cached data and may potentially contain sensitive information. When used to send messages, JSON is often used in conjunction with a RESTful service and can be used to transmit sensitive information such as authentication credentials.

The semantics of JSON documents and messages can be altered if an application constructs JSON from unvalidated input. In a relatively benign case, an attacker may be able to insert extraneous elements that cause an application to throw an exception while parsing a JSON document or request. In a more serious case, such as ones that involves JSON injection, an attacker may be able to insert extraneous elements that allow for the predictable manipulation of business critical values within a JSON document or request. In some cases, JSON injection can lead to cross-site scripting or dynamic code evaluation.

Example 1: The following Java code uses Jackson to serialize user account authentication information for non-privileged users (those with a role of "default" as opposed to privileged users with a role of "admin") from user-controlled input variables username and password to the JSON file located at ~/user_info.json:

```
...
JsonFactory jfactory = new JsonFactory();
JsonGenerator jGenerator = jfactory.createJsonGenerator(new File("~/user_info.json"), JsonEncoding.UTF8);
jGenerator.writeStartObject();
jGenerator.writeFieldName("username");
jGenerator.writeRawValue("\"" + username + "\"");
jGenerator.writeFieldName("password");
jGenerator.writeRawValue("\"" + password + "\"");
jGenerator.writeFieldName("role");
jGenerator.writeRawValue("\"default\"");
jGenerator.writeEndObject();
jGenerator.close();
```

Yet, because the JSON serialization is performed using JsonGenerator.writeRawValue(), the untrusted data in username and password will not be validated to escape JSON-related special characters. This allows a user to arbitrarily insert JSON keys, possibly changing the structure of the serialized JSON. In this example, if the non-privileged user mallory with password Evil123! were to append ","role":"admin to her username when entering it at the prompt that sets the value of the username variable, the resulting JSON saved to ~/user_info.json would be:

```
{
"username":"mallory",
```

"role":"admin",
"password":"Evil123!",
"role":"default"
}

If this serialized JSON file were then deserialized to an HashMap object with Jackson's JsonParser as so:

JsonParser jParser = jfactory.createJsonParser(new File("~/user_info.json"));

while (jParser.nextToken() != JsonToken.END_OBJECT) {

String fieldname = jParser.getCurrentName();

if ("username".equals(fieldname)) {
jParser.nextToken();
userInfo.put(fieldname, jParser.getText());
}

if ("password".equals(fieldname)) {
jParser.nextToken();
userInfo.put(fieldname, jParser.getText());
}

if ("role".equals(fieldname)) {
jParser.nextToken();
userInfo.put(fieldname, jParser.getText());
}

if (userInfo.size() == 3)
break;
}

jParser.close();

The resulting values for the username, password, and role keys in the HashMap object would be mallory, Evil123!, and admin respectively. Without further verification that the deserialized JSON values are valid, the application will incorrectly assign user mallory "admin" privileges.

Recommendations:

When writing user supplied data to JSON, follow these guidelines:

1. Do not create JSON attributes with names that are derived from user input.

2. Ensure that all serialization to JSON is performed using a safe serialization function that delimits untrusted data within single or double quotes and escapes any special characters.

Example 2: The following Java code implements the same functionality as that in Example 1, but instead uses JsonGenerator.writeString() rather than JsonGenerator.writeRawValue() to serialize the data, therefore ensuring that any untrusted data is properly delimited and escaped:

...

JsonFactory jfactory = new JsonFactory();

JsonGenerator jGenerator = jfactory.createJsonGenerator(new File("~/user_info.json"), JsonEncoding.UTF8);

jGenerator.writeStartObject();

jGenerator.writeFieldName("username");
jGenerator.writeString(username);

jGenerator.writeFieldName("password");
jGenerator.writeString(password);

jGenerator.writeFieldName("role");
jGenerator.writeString("default");

jGenerator.writeEndObject();

jGenerator.close();

UserUtils.java, line 101 (JSON Injection)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	On line 101 of UserUtils.java, the method registerUser() writes unvalidated input into JSON. This call could allow an attacker to inject arbitrary elements or attributes into the JSON entity.		

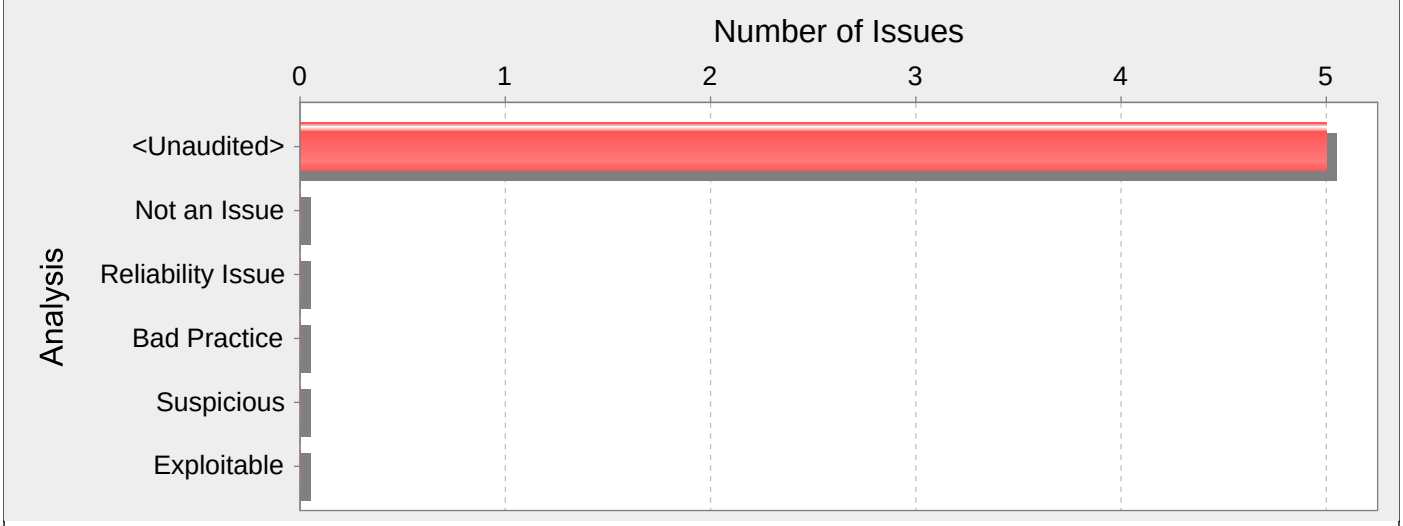
Source: UserUtils.java:81 java.io.FileReader.FileReader()

```
79          File dataFile = new File(getFilePath(NEWSLETTER_USER_FILE));
80          if (dataFile.exists()) {
81              jsonArray = (JSONArray) jsonParser.parse(new
FileReader(getFilePath(NEWSLETTER_USER_FILE)));
82          } else {
83              Boolean created = dataFile.createNewFile();
```

Sink: UserUtils.java:101 com.fasterxml.jackson.core.JsonGenerator.writeRawValue()

```
99                  JSONObject person = (JSONObject) jsonObject;
100                 jGenerator.writeFieldName("firstName");
101                 jGenerator.writeRawValue("\"" + (String) person.get("firstName") +
"\"");
102                 jGenerator.writeFieldName("lastName");
103                 jGenerator.writeRawValue("\"" + (String) person.get("lastName") +
"\"");
```

Category: XML External Entity Injection (5 Issues)



Abstract:

XML parser configured in UserController.java:600 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack.

Explanation:

XML External Entities attacks benefit from an XML feature to build documents dynamically at the time of processing. An XML entity allows inclusion of data dynamically from a given resource. External entities allow an XML document to include data from an external URI. Unless configured to do otherwise, external entities force the XML parser to access the resource specified by the URI, e.g., a file on the local machine or on a remote system. This behavior exposes the application to XML External Entity (XXE) attacks, which can be used to perform denial of service of the local system, gain unauthorized access to files on the local machine, scan remote machines, and perform denial of service of remote systems.

The following XML document shows an example of an XXE attack.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///dev/random" >]><foo>&xxe;</foo>
```

This example could crash the server (on a UNIX system), if the XML parser attempts to substitute the entity with the contents of the /dev/random file.

Recommendations:

The XML unmarshaller should be configured securely so that it does not allow external entities as part of an incoming XML document.

To avoid XXE injection do not use unmarshal methods that process an XML source directly as java.io.File, java.io.Reader or java.io.InputStream. Parse the document with a securely configured parser and use an unmarshal method that takes the secure parser as the XML source as shown in the following example:

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
dbf.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
DocumentBuilder db = dbf.newDocumentBuilder();
Document document = db.parse(<XML Source>);
Model model = (Model) u.unmarshal(document);
```

Tips:

1. Fortify AppDefender adds protection against this category.

UserController.java, line 600 (XML External Entity Injection)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	XML parser configured in UserController.java:600 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack.		
Source:	StorageProperties.java:16 Read this.location()		

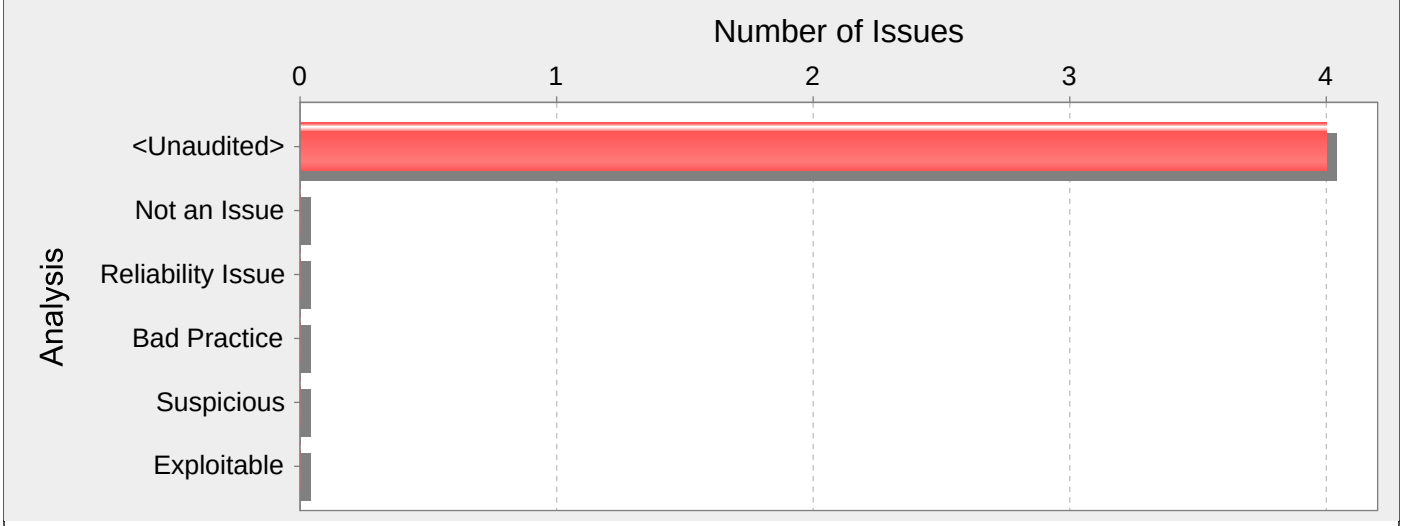
```
14
15                      public String getLocation() {
16                          return location;
```

```
17                      }
```

Sink: UserController.java:600 javax.xml.parsers.DocumentBuilder.parse()

```
598                     dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);
599                           DocumentBuilder db = dbf.newDocumentBuilder();
600                           Document doc = db.parse(fpath.toFile());
601                           try (ByteArrayOutputStream bytesOutStream = new ByteArrayOutputStream()) {
602                            writeXml(doc, bytesOutStream);
```

Category: Cross-Site Scripting: Persistent (4 Issues)



Abstract:

The method getOrdersByKeywords() in ApiOrderController.java sends unvalidated data to a web browser on line 105, which 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 data store, 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 validation.

The malicious content sent to the web browser often takes the form of a JavaScript segment, but can also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data such as 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 may 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 difficult to identify the threat and increases the possibility that the attack might 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 metacharacters or source code, then the code is 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 email 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.

Some think that in the mobile environment, classic web application vulnerabilities, such as cross-site scripting, 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 enables JavaScript in Android's WebView (by default, JavaScript is disabled) and loads a page based on the value received from an Android intent.

```
...
WebView webview = (WebView) findViewById(R.id.webview);
webview.getSettings().setJavaScriptEnabled(true);
String url = this.getIntent().getExtras().getString("url");
webview.loadUrl(url);
...
```

If the value of url starts with javascript:, JavaScript code that follows executes within the context of the web page inside WebView.

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 emailed 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 might include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Example 3, 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 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.

Recommendations:

The solution to prevent XSS is to ensure that validation occurs in the required places and that relevant properties are set to prevent vulnerabilities.

Because 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 all 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 might accept input through a shared data store or other trusted source, and that data store might 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 that the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create an allow list of safe characters that can appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alphanumeric 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 must be considered valid input after 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 to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. 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 which characters have special meaning, many web browsers try to correct common mistakes in HTML and might treat other characters as special in certain contexts. This is why we do not recommend the use of deny lists as a means to prevent XSS. 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 in double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed in single quotes, 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 greater than 127 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.

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

- Semicolons, parentheses, curly braces, and new line characters must be filtered out 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 might 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).

After 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 is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and might 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. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

Tips:

1. The Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also 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. Nonetheless, it is often still a good idea to validate everything read from the database.

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 AppDefender adds protection against this category.

ApiOrderController.java, line 105 (Cross-Site Scripting: Persistent)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method getOrdersByKeywords() in ApiOrderController.java sends unvalidated data to a web browser on line 105, which can result in the browser executing malicious code.		

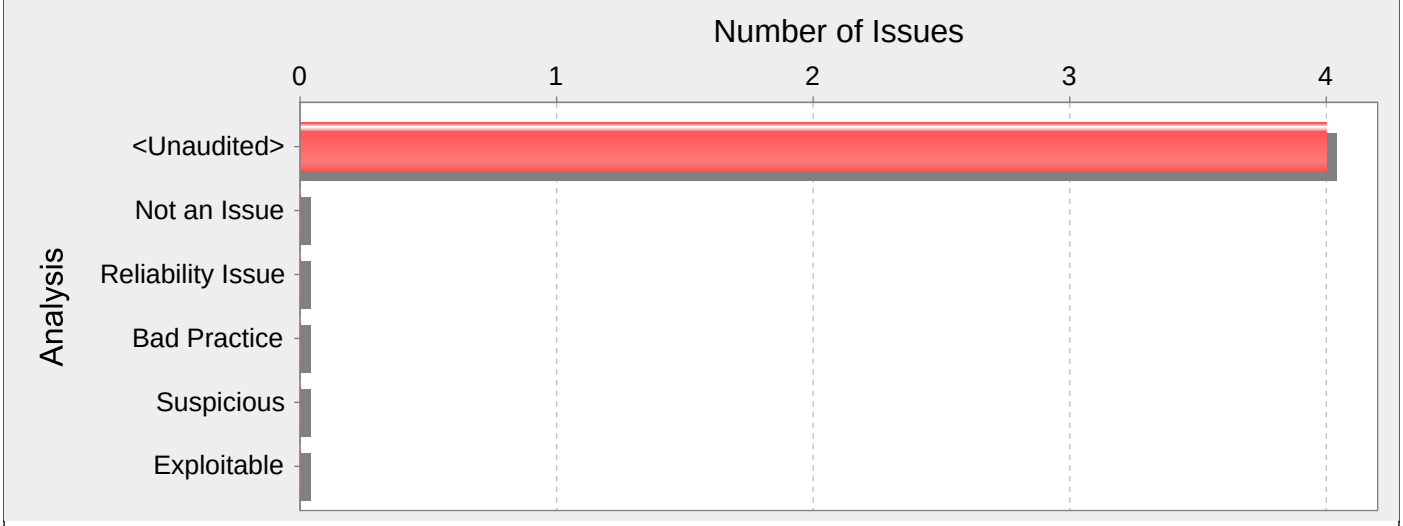
Source: ProductService.java:308
org.springframework.data.jpa.repository.JpaRepository.findAll()

```
306     }
307
308     public List<Order> getAllOrders() { return orderRepository.findAll(); }
309
310     public List<Order> getAllOrders(Integer offset, String keywords) {
```

Sink: ApiOrderController.java:105
org.springframework.http.ResponseEntity.BodyBuilder.body()

```
103                         productService.getAllOrders().stream()
104                                 .map(OrderResponse::new)
105                                 .collect(Collectors.toList()));
106             } else {
107                 String k = (keywords.orElse(""));
```

Category: Cross-Site Scripting: Reflected (4 Issues)



Abstract:

The method getKeywordsContent() in ProductController.java sends unvalidated data to a web browser on line 90, which 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 reflected XSS, the untrusted source is typically a web request, while in the case of persisted (also known as stored) XSS it is typically a database or other back-end data store.

2. The data is included in dynamic content that is sent to a web user without validation.

The malicious content sent to the web browser often takes the form of a JavaScript segment, but can also include HTML, Flash or any other type of code that the browser executes. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data such as 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 reads an employee ID, eid, from an HTTP request and displays it to the user.

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

The code in this example operates correctly if eid contains only standard alphanumeric text. If eid has a value that includes metacharacters or source code, then the code is 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 email 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.

Example 2: 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 %>
```

As in Example 1, this code functions correctly when the values of name are well-behaved, but it does nothing to prevent exploits if they are not. Again, 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 may 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 difficult to identify the threat and increases the possibility that the attack might 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.

Some think that in the mobile environment, classic web application vulnerabilities, such as cross-site scripting, 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 enables JavaScript in Android's WebView (by default, JavaScript is disabled) and loads a page based on the value received from an Android intent.

```
...
WebView webview = (WebView) findViewById(R.id.webview);
webview.getSettings().setJavaScriptEnabled(true);
String url = this.getIntent().getExtras().getString("url");
webview.loadUrl(url);
...
```

If the value of url starts with javascript:, JavaScript code that follows executes within the context of the web page inside WebView.

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, 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 emailed 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 might include session information, from the user's machine to the attacker or perform other nefarious activities.

- As in Example 2, 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 3, 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 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.

Recommendations:

The solution to prevent XSS is to ensure that validation occurs in the required places and that relevant properties are set to prevent vulnerabilities.

Because 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 all 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 might accept input through a shared data store or other trusted source, and that data store might 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 that the best way to prevent XSS vulnerabilities is to validate everything that enters the application and leaves the application destined for the user.

The most secure approach to validation for XSS is to create an allow list of safe characters that can appear in HTTP content and accept input composed exclusively of characters in the approved set. For example, a valid username might only include alphanumeric 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 must be considered valid input after 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 to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. 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 which characters have special meaning, many web browsers try to correct common mistakes in HTML and might treat other characters as special in certain contexts. This is why we do not recommend the use of deny lists as a means to prevent XSS. 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 in double quotes, the double quotes are special because they mark the end of the attribute value.

- In attribute values enclosed in single quotes, 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 greater than 127 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.

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

- Semicolons, parentheses, curly braces, and new line characters must be filtered out 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 might 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).

After 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 is to remove special characters with filtering. However, filtering has the side effect of changing any visual representation of the filtered content and might 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. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

Tips:

1. The Fortify Secure Coding Rulepacks warn about SQL Injection and Access Control: Database issues when untrusted data is written to a database and also 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. Nonetheless, it is often still a good idea to validate everything read from the database.

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 AppDefender adds protection against this category.

ProductController.java, line 90 (Cross-Site Scripting: Reflected)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The method getKeywordsContent() in ProductController.java sends unvalidated data to a web browser on line 90, which can result in the browser executing malicious code.		

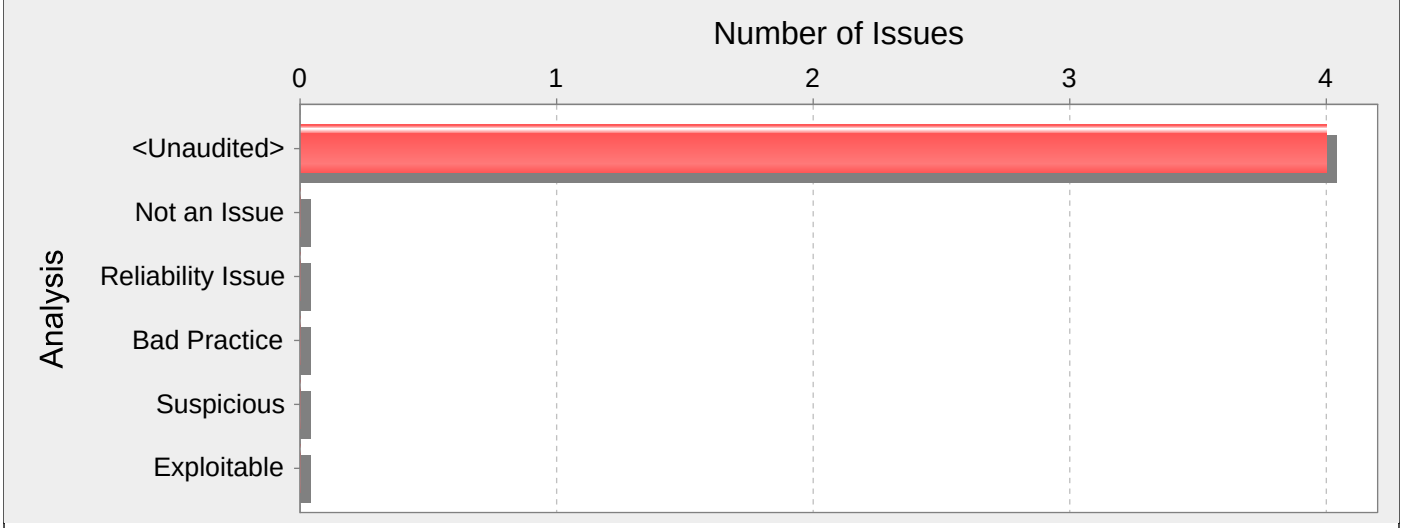
Source: ProductController.java:86 getKeywordsContent(0)

```
84      @GetMapping("/xss")
85      @ResponseBody
86      public ResponseEntity<String> getKeywordsContent(@Param("keywords") String
keywords) {
87
88          String retContent = "Product search using: " + keywords;
```

Sink: ProductController.java:90 org.springframework.http.ResponseEntity.BodyBuilder.body()

```
88          String retContent = "Product search using: " + keywords;
89
90              return ResponseEntity.ok().body(retContent);
91          }
```

Category: Header Manipulation (4 Issues)



Abstract:

The method downloadFile() in ProductController.java includes unvalidated data in an HTTP response header on line 170. This enables attacks such as cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.

Explanation:

Header Manipulation vulnerabilities occur when:

1. Data enters a web application through an untrusted source, most frequently an HTTP request.

2. The data is included in an HTTP response header sent to a web user without being validated.

As with many software security vulnerabilities, Header Manipulation is a means to an end, not an end in itself. At its root, the vulnerability is straightforward: an attacker passes malicious data to a vulnerable application, and the application includes the data in an HTTP response header.

One of the most common Header Manipulation attacks is HTTP Response Splitting. To mount a successful HTTP Response Splitting exploit, the application must allow input that contains CR (carriage return, also given by %0d or \r) and LF (line feed, also given by %0a or \n)characters into the header. These characters not only give attackers control of the remaining headers and body of the response the application intends to send, but also allows them to create additional responses entirely under their control.

Many of today's modern application servers will prevent the injection of malicious characters into HTTP headers. For example, recent versions of Apache Tomcat will throw an IllegalArgumentException if you attempt to set a header with prohibited characters. If your application server prevents setting headers with new line characters, then your application is not vulnerable to HTTP Response Splitting. However, solely filtering for new line characters can leave an application vulnerable to Cookie Manipulation or Open Redirects, so care must still be taken when setting HTTP headers with user input.

Example: The following code segment reads the name of the author of a weblog entry, author, from an HTTP request and sets it in a cookie header of an HTTP response.

```
String author = request.getParameter(AUTHOR_PARAM);
...
Cookie cookie = new Cookie("author", author);
cookie.setMaxAge(cookieExpiration);
response.addCookie(cookie);
```

Assuming a string consisting of standard alphanumeric characters, such as "Jane Smith", is submitted in the request the HTTP response including this cookie might take the following form:

```
HTTP/1.1 200 OK
...
Set-Cookie: author=Jane Smith
...
```

However, because the value of the cookie is formed of unvalidated user input the response will only maintain this form if the value submitted for AUTHOR_PARAM does not contain any CR and LF characters. If an attacker submits a malicious string, such as "Wiley Hacker\r\nHTTP/1.1 200 OK\r\n...", then the HTTP response would be split into two responses of the following form:

```
HTTP/1.1 200 OK
```

...
Set-Cookie: author=Wiley Hacker

HTTP/1.1 200 OK

...

Clearly, the second response is completely controlled by the attacker and can be constructed with any header and body content desired. The ability of attacker to construct arbitrary HTTP responses permits a variety of resulting attacks, including: cross-user defacement, web and browser cache poisoning, cross-site scripting, and page hijacking.

Cross-User Defacement: An attacker will be able to make a single request to a vulnerable server that will cause the server to create two responses, the second of which may be misinterpreted as a response to a different request, possibly one made by another user sharing the same TCP connection with the server. This can be accomplished by convincing the user to submit the malicious request themselves, or remotely in situations where the attacker and the user share a common TCP connection to the server, such as a shared proxy server. In the best case, an attacker may leverage this ability to convince users that the application has been hacked, causing users to lose confidence in the security of the application. In the worst case, an attacker may provide specially crafted content designed to mimic the behavior of the application but redirect private information, such as account numbers and passwords, back to the attacker.

Cache Poisoning: The impact of a maliciously constructed response can be magnified if it is cached either by a web cache used by multiple users or even the browser cache of a single user. If a response is cached in a shared web cache, such as those commonly found in proxy servers, then all users of that cache will continue receive the malicious content until the cache entry is purged. Similarly, if the response is cached in the browser of an individual user, then that user will continue to receive the malicious content until the cache entry is purged, although only the user of the local browser instance will be affected.

Cross-Site Scripting: Once attackers have control of the responses sent by an application, they have a choice of a variety of malicious content to provide users. Cross-site scripting is common form of attack where malicious JavaScript or other code included in a response is executed in the user's browser. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data such as 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. The most common and dangerous attack vector against users of a vulnerable application uses JavaScript to transmit session and authentication information back to the attacker who can then take complete control of the victim's account.

Page Hijacking: In addition to using a vulnerable application to send malicious content to a user, the same root vulnerability can also be leveraged to redirect sensitive content generated by the server and intended for the user to the attacker instead. By submitting a request that results in two responses, the intended response from the server and the response generated by the attacker, an attacker may cause an intermediate node, such as a shared proxy server, to misdirect a response generated by the server for the user to the attacker. Because the request made by the attacker generates two responses, the first is interpreted as a response to the attacker's request, while the second remains in limbo. When the user makes a legitimate request through the same TCP connection, the attacker's request is already waiting and is interpreted as a response to the victim's request. The attacker then sends a second request to the server, to which the proxy server responds with the server generated request intended for the victim, thereby compromising any sensitive information in the headers or body of the response intended for the victim.

Cookie Manipulation: When combined with attacks like Cross-Site Request Forgery, attackers may change, add to, or even overwrite a legitimate user's cookies.

Open Redirect: Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

Recommendations:

The solution to prevent Header Manipulation is to ensure that input validation occurs in the required places and checks for the correct properties.

Since Header Manipulation 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 responses dynamically, this method is prone to errors of omission (missing validation). An effective way to mitigate this risk is to also perform input validation for Header Manipulation.

Web applications must validate all input to prevent other vulnerabilities, such as SQL injection, so augmenting an application's existing input validation mechanism to include checks for Header Manipulation is generally relatively easy. Despite its value, input validation for Header Manipulation does not take the place of rigorous output validation. An application might accept input through a shared data store or other trusted source, and that data store might 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 that the best way to prevent Header Manipulation vulnerabilities is to validate everything that enters the application or leaves the application destined for the user.

The most secure approach to validation for Header Manipulation is to create an allow list of safe characters that can appear in HTTP response headers and accept input composed exclusively of characters in the approved set. For example, a valid name might only include alphanumeric characters or an account number might only include digits 0-9.

A more flexible, but less secure approach is to implement a deny list, which selectively rejects or escapes potentially dangerous characters before using the input. To form such a list, you first need to understand the set of characters that hold special meaning in HTTP response headers. Although the CR and LF characters are at the heart of an HTTP response splitting attack, other characters, such as ':' (colon) and '=' (equal), have special meaning in response headers as well.

After you identify the correct points in an application to perform validation for Header Manipulation attacks and what special characters the validation should consider, the next challenge is to identify how your validation handles special characters. The application should reject any input destined to be included in HTTP response headers that contains special characters, particularly CR and LF, as invalid.

Many application servers attempt to limit an application's exposure to HTTP response splitting vulnerabilities by providing implementations for the functions responsible for setting HTTP headers and cookies that perform validation for the characters essential to an HTTP response splitting attack. Do not rely on the server running your application to make it secure. For any developed application, there are no guarantees about which application servers it will run on during its lifetime. As standards and known exploits evolve, there are no guarantees that application servers will continue to stay in sync.

Tips:

1. Many HttpServletRequest implementations return a URL-encoded string from getHeader(), will not cause a HTTP response splitting issue unless it is decoded first because the CR and LF characters will not carry a meta-meaning in their encoded form. However, this behavior is not specified in the J2EE standard and varies by implementation. Furthermore, even encoded user input returned from getHeader() can lead to other vulnerabilities, including open redirects and other HTTP header tampering.

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.

ProductController.java, line 170 (Header Manipulation)

Fortify Priority:	High	**Folder**	High
Kingdom:	Input Validation and Representation		
Abstract:	The method downloadFile() in ProductController.java includes unvalidated data in an HTTP response header on line 170. This enables attacks such as cache-poisoning, cross-site scripting, cross-user defacement, page hijacking, cookie manipulation or open redirect.		

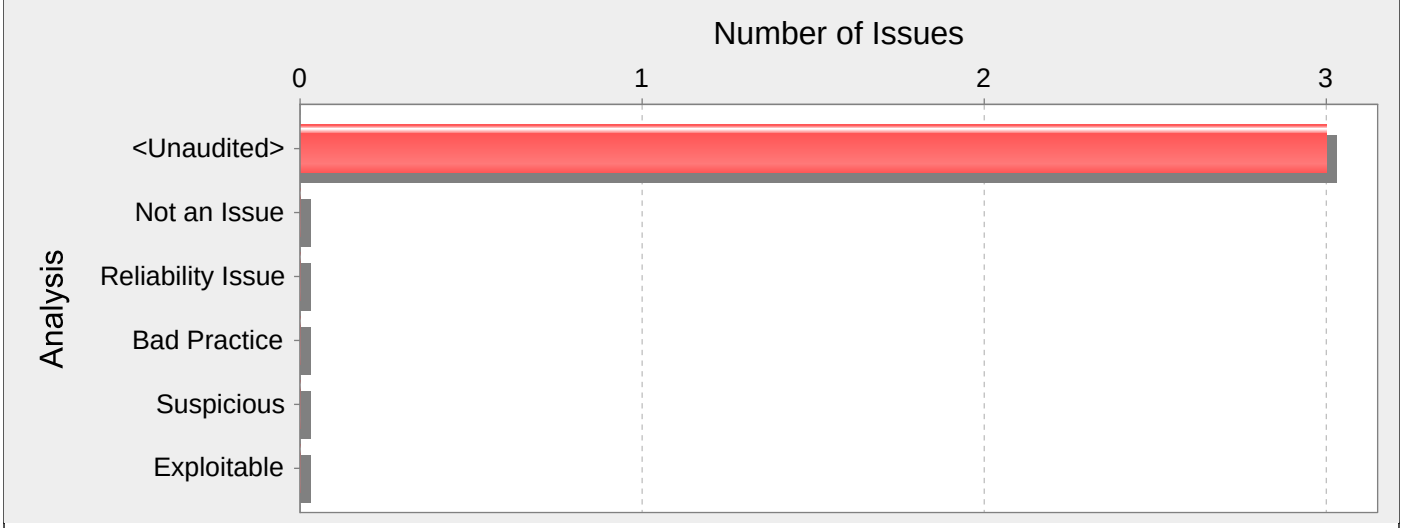
Source: ProductController.java:136 downloadFile(1)

```
134     @GetMapping("/{id}/download/{fileName:.+}")
135     public ResponseEntity<Resource> downloadFile(@PathVariable(value = "id") UUID
productId,
136                                                  @PathVariable String fileName,
HttpServletRequest request) {
137         Resource resource;
138         File dataDir;
```

Sink: ProductController.java:170 org.springframework.http.ResponseEntity.HeadersBuilder.header()

```
168
169         return ResponseEntity.ok().contentType(MediaType.parseMediaType(contentType))
170                 .header(HttpHeaders.CONTENT_DISPOSITION, "attachment; filename=\"" +
resource.getFilename() + "\"")
171                 .body(resource);
172     }
```

Category: System Information Leak: External (3 Issues)



Abstract:

The function handle() in ApiAccessDeniedHandler.java reveals system data or debug information by calling println() on line 66. The information revealed by println() could help an adversary form a plan of attack.

Explanation:

An external information leak occurs when system data or debug information leaves the program to a remote machine via a socket or network connection. External leaks can help an attacker by revealing specific data about operating systems, full pathnames, the existence of usernames, or locations of configuration files, and are more serious than internal information leaks, which are more difficult for an attacker to access.

Example 1: The following code leaks Exception information in the HTTP response:

```
protected void doPost (HttpServletRequest req, HttpServletResponse res) throws IOException {
...
PrintWriter out = res.getWriter();
try {
...
} catch (Exception e) {
out.println(e.getMessage());
}
}
```

This information can be exposed to a remote user. In some cases, the error message provides the attacker with the precise type of attack to which the system is vulnerable. 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 Example 1, the leaked information 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.

Information leaks are also a concern in a mobile computing environment. With mobile platforms, applications are downloaded from various sources and are run alongside each other on the same device. The likelihood of running a piece of malware next to a banking application is high, which is why application authors need to be careful about what information they include in messages addressed to other applications running on the device.

Example 2: The following code broadcasts the stack trace of a caught exception to all the registered Android receivers.

```
...
try {
...
} catch (Exception e) {
String exception = Log.getStackTraceString(e);
Intent i = new Intent();
i.setAction("SEND_EXCEPTION");
i.putExtra("exception", exception);
view.getContext().sendBroadcast(i);
}
...
```

This is another scenario specific to the mobile environment. Most mobile devices now implement a Near-Field Communication (NFC) protocol for quickly sharing information between devices using radio communication. It works by bringing devices in close proximity or having the devices touch each other. Even though the communication range of NFC is limited to just a few centimeters, eavesdropping, data modification and various other types of attacks are possible, because NFC alone does not ensure secure communication.

Example 3: The Android platform provides support for NFC. The following code creates a message that gets pushed to the other device within range.

```
...
public static final String TAG = "NfcActivity";
private static final String DATA_SPLITTER = "__:DATA:__";
private static final String MIME_TYPE = "application/my.applications.mimetype";
...
TelephonyManager tm = (TelephonyManager)Context.getSystemService(Context.TELEPHONY_SERVICE);
String VERSION = tm.getDeviceSoftwareVersion();
...
NfcAdapter nfcAdapter = NfcAdapter.getDefaultAdapter(this);
if (nfcAdapter == null)
return;
String text = TAG + DATA_SPLITTER + VERSION;
NdefRecord record = new NdefRecord(NdefRecord.TNF_MIME_MEDIA,
MIME_TYPE.getBytes(), new byte[0], text.getBytes());
NdefRecord[] records = { record };
NdefMessage msg = new NdefMessage(records);
nfcAdapter.setNdefPushMessage(msg, this);
...
```

An NFC Data Exchange Format (NDEF) message contains typed data, a URI, or a custom application payload. If the message contains information about the application, such as its name, MIME type, or device software version, this information could be leaked to an eavesdropper.

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. Debug 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. Because of this, never send information to a resource directly outside the program.

Example 4: The following code broadcasts the stack trace of a caught exception within your application only, so that it cannot be leaked to other apps on the system. Additionally, this technique is more efficient than globally broadcasting through the system.

```
...
try {
...
} catch (Exception e) {
String exception = Log.getStackTraceString(e);
Intent i = new Intent();
i.setAction("SEND_EXCEPTION");
i.putExtra("exception", exception);
LocalBroadcastManager.getInstance(view.getContext()).sendBroadcast(i);
}
...
```

If you are concerned about leaking system data via NFC on an Android device, you could do one of the following three things. Do not include system data in the messages pushed to other devices in range, encrypt the payload of the message, or establish a secure communication channel at a higher layer.

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 Audit Guide to filter out this category from your scan results.

ApiAccessDeniedHandler.java, line 66 (System Information Leak: External)

Fortify Priority:	High	**Folder**	High
Kingdom:	Encapsulation		
Abstract:	The function handle() in ApiAccessDeniedHandler.java reveals system data or debug information by calling println() on line 66. The information revealed by println() could help an adversary form a plan of attack.		

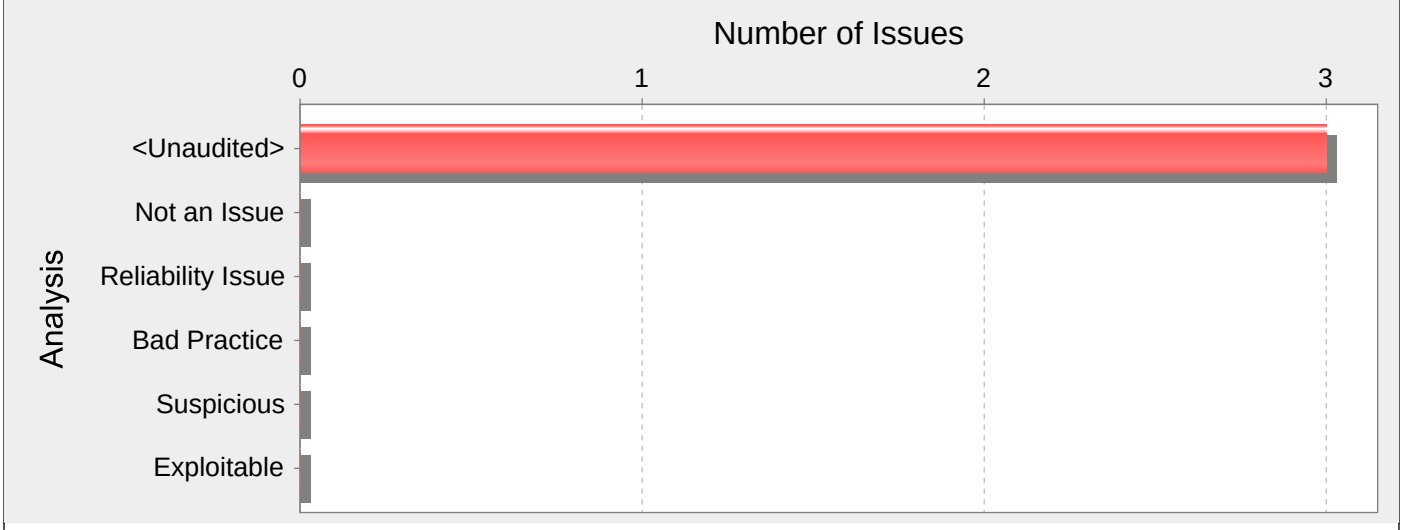
Source: ApiAccessDeniedHandler.java:53 java.lang.Throwable.getLocalizedMessage()

```
51              response.setStatus(HttpServletResponse.SC_FORBIDDEN);
52              ArrayList<String> errors = new ArrayList<>();
53              errors.add(ex.getLocalizedMessage());
54              ApiStatusResponse apiStatusResponse = new ApiStatusResponse
55                      .ApiResponseBuilder()
```

Sink: ApiAccessDeniedHandler.java:66 java.io.PrintWriter.println()

```
64              String jsonString = mapper.writeValueAsString(apiError.getBody());
65              PrintWriter writer = response.getWriter();
66              writer.println(jsonString);
67          }
```

Category: Weak Encryption (3 Issues)



Abstract:

The call to SecretKeySpec() at EncryptedPasswordUtils.java line 41 uses a weak encryption algorithm that cannot guarantee the confidentiality of sensitive data.

Explanation:

Antiquated encryption algorithms such as DES no longer provide sufficient protection for use with sensitive data. Encryption algorithms rely on key size as one of the primary mechanisms to ensure cryptographic strength. Cryptographic strength is often measured by the time and computational power needed to generate a valid key. Advances in computing power have made it possible to obtain small encryption keys in a reasonable amount of time. For example, the 56-bit key used in DES posed a significant computational hurdle in the 1970s when the algorithm was first developed, but today DES can be cracked in less than a day using commonly available equipment.

Recommendations:

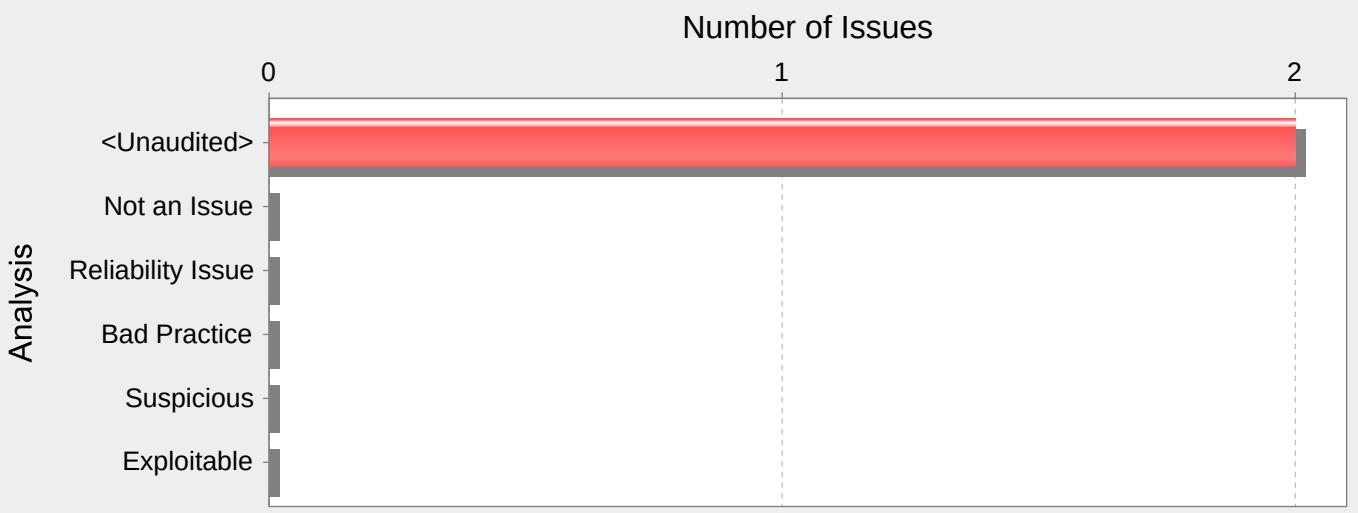
Use strong encryption algorithms with large key sizes to protect sensitive data. A strong alternative to DES is AES (Advanced Encryption Standard, formerly Rijndael). Before selecting an algorithm, first determine if your organization has standardized on a specific algorithm and implementation.

EncryptedPasswordUtils.java, line 41 (Weak Encryption)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The call to SecretKeySpec() at EncryptedPasswordUtils.java line 41 uses a weak encryption algorithm that cannot guarantee the confidentiality of sensitive data.		
Sink:	EncryptedPasswordUtils.java:41 SecretKeySpec()		

```
39
40    private static final byte[] iv = { 22, 33, 11, 44, 55, 99, 66, 77 };
41    private static final SecretKey keySpec = new SecretKeySpec(iv, "DES");
42
43       public static String encryptPassword(String password) {
```

Category: Dockerfile Misconfiguration: Default User Privilege (2 Issues)



Abstract:

The Dockerfile does not specify a USER, so it defaults to running with a root user.

Explanation:

When a Dockerfile does not specify a USER, Docker containers run with super user privileges by default. These super user privileges are propagated to the code running inside the container, which is usually more permission than necessary. Running the Docker container with super user privileges broadens the attack surface which might enable attackers to perform more serious forms of exploitation.

Recommendations:

It is good practice to run your containers as a non-root user when possible.

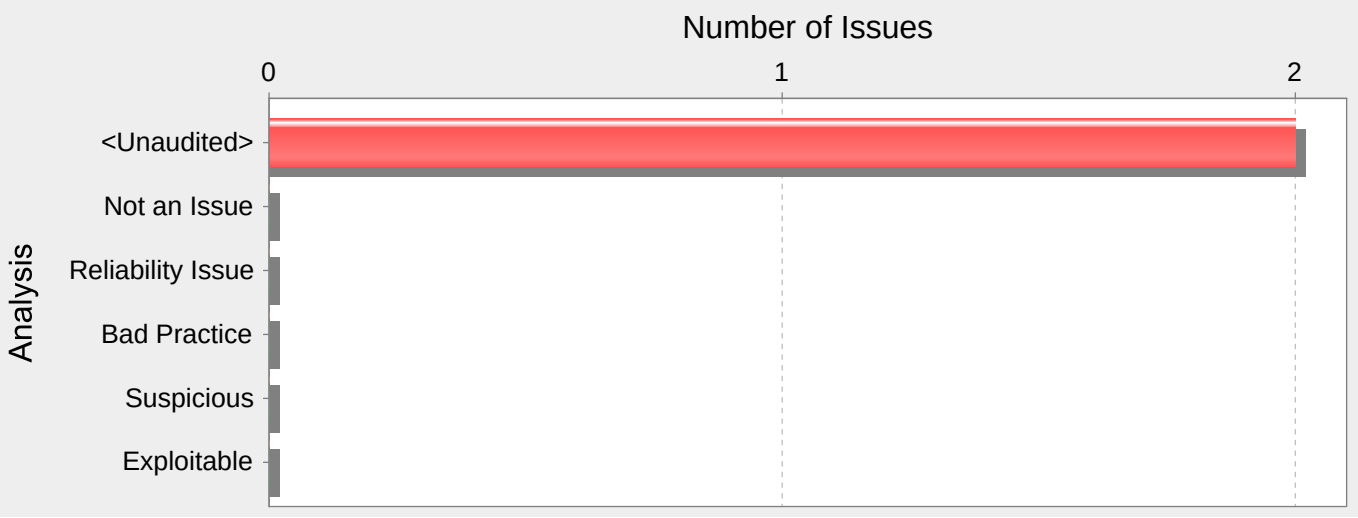
To modify a docker container to use a non-root user, the Dockerfile needs to specify a different user, such as:

```
RUN useradd myLowPrivilegeUser
USER myLowPrivilegeUser
```

Dockerfile, line 1 (Dockerfile Misconfiguration: Default User Privilege)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	The Dockerfile does not specify a USER, so it defaults to running with a root user.		
Sink:	Dockerfile:1 FROM()		
-1	`FROM openjdk:8-jdk-slim`		
0			
1	`LABEL maintainer="kevin.lee@microfocus.com"`		

Category: HTML5: Missing Content Security Policy (2 Issues)



Abstract:

Content Security Policy (CSP) is not configured.

Explanation:

Content Security Policy (CSP) is a declarative security header that enables developers to dictate which domains the site is allowed to load content from or initiate connections to when rendered in the web browser. It provides an additional layer of security from critical vulnerabilities such as cross-site scripting, clickjacking, cross-origin access and the like, on top of input validation and checking an allow list in code.

Spring Security and other frameworks do not add Content Security Policy headers by default. The web application author must declare the security policy/policies to enforce or monitor for the protected resources to benefit from this additional layer of security.

Recommendations:

Configure a Content Security Policy to mitigate possible injection vulnerabilities.

Example: The following code sets a Content Security Policy in a Spring Security protected application:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
...
String policy = getCSPolicy();
http.headers().contentSecurityPolicy(policy);
...
}
```

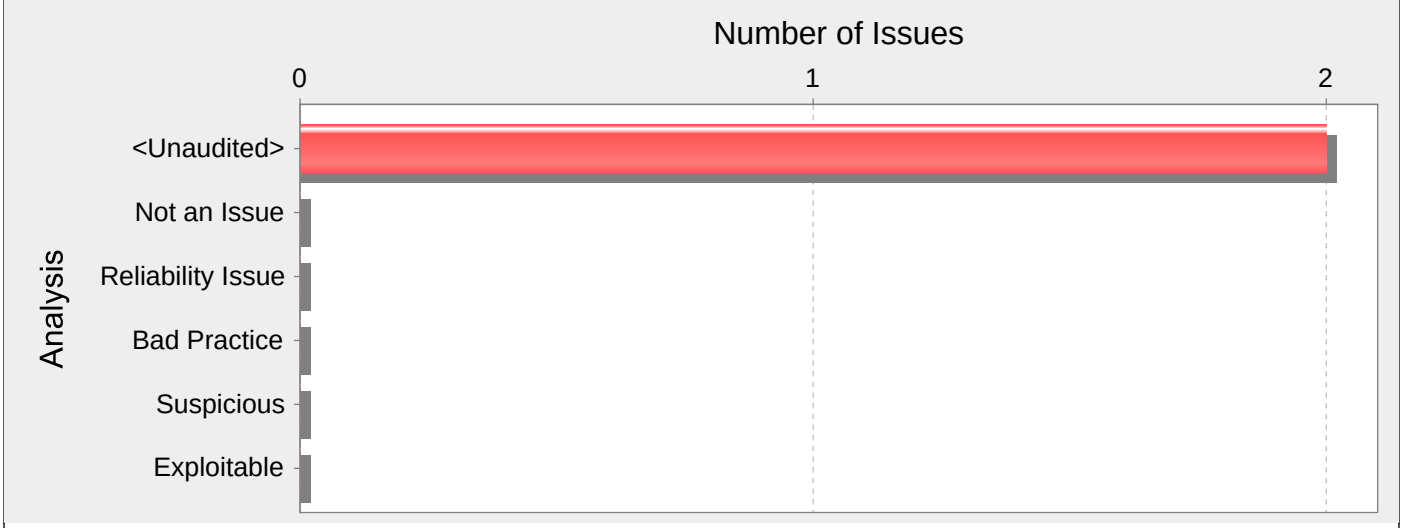
Content Security Policy is not intended to solve all content injection vulnerabilities. Instead, you can leverage CSP to help reduce the harm caused by content injection attacks. Use regular defensive coding,above, current such as input validation and output encoding.

WebSecurityConfiguration.java, line 99 (HTML5: Missing Content Security Policy)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Encapsulation		
Abstract:	Content Security Policy (CSP) is not configured.		
Sink:	WebSecurityConfiguration.java:99 Function: configure()		

```
97
98          @Override
99          protected void configure(HttpSecurity httpSecurity) throws Exception {
100
101             /*http.cors().and().csrf().disable()
```

Category: Mass Assignment: Request Parameters Bound into Persisted Objects (2 Issues)



Abstract:

The class in Authority.java is both a database persistent entity and a dynamically bound request object. Allowing database persistent entities to be auto-populated by request parameters will let an attacker create unintended database records in association entities or update unintended fields in the entity object.

Explanation:

Persistent objects are bound to the underlying database and updated automatically by the persistence framework, such as Hibernate or JPA. Allowing these objects to be dynamically bound to the request by Spring MVC will let an attacker inject unexpected values into the database by providing additional request parameters.

Example 1: The Order, Customer, and Profile are Hibernate persisted classes.

```
public class Order {
String ordered;
List lineItems;
Customer cust;
...
}
public class Customer {
String customerId;
...
Profile p;
...
}
public class Profile {
String profileId;
String username;
String password;
...
}
```

OrderController is the Spring controller class handling the request:

```
@Controller
public class OrderController {
...
@RequestMapping("/updateOrder")
public String updateOrder(Order order) {
...
session.save(order);
}
}
```

Because command classes are automatically bound to the request, an attacker may use this vulnerability to update another user's password by adding the following request parameters to the request: "http://www.yourcorp.com/webApp/updateOrder?order.customer.profile.profileId=1234&order.customer.profile.password=urpowned"

Recommendations:

Do not use persistent entity objects as your request bound objects. Manually copy the attributes which you are interested in persisting from your request bound objects to your persistent entity objects. An alternative would be to explicitly define which attributes on the request bound object are settable via request parameters.

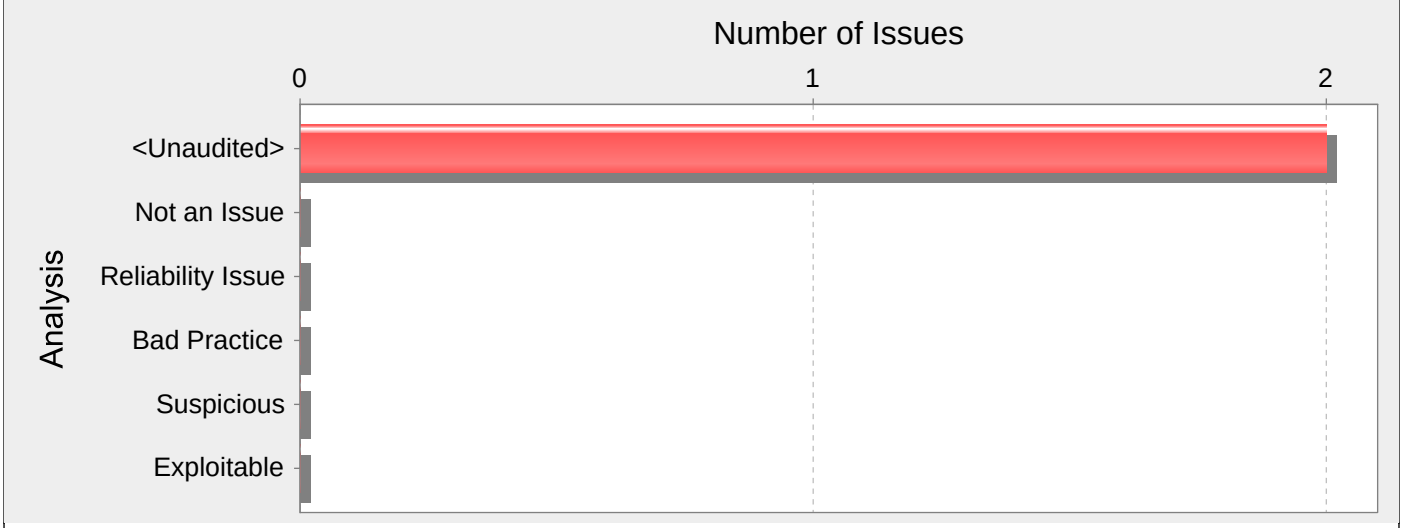
Authority.java, line 36 (Mass Assignment: Request Parameters Bound into Persisted Objects)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	API Abuse		
Abstract:	The class in Authority.java is both a database persistent entity and a dynamically bound request object. Allowing database persistent entities to be auto-populated by request parameters will let an attacker create unintended database records in association entities or update unintended fields in the entity object.		

Sink: Authority.java:36 Class: Authority()

```
34  @Table(name = "authorities")
35  @JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class, property = "id")
36  public class Authority {
37
38      private static final long serialVersionUID = 1L;
```

Category: Race Condition: Singleton Member Field (2 Issues)



Abstract:

The class ProductService is a singleton, so the member field pageSize is shared between users. The result is that one user could see another user's data.

Explanation:

Many Servlet developers do not understand that a Servlet is a singleton. There is only one instance of the Servlet, and that single instance is used and re-used to handle multiple requests that are processed simultaneously by different threads.

A common result of this misunderstanding is that developers use Servlet member fields in such a way that one user may inadvertently see another user's data. In other words, storing user data in Servlet member fields introduces a data access race condition.

Example 1: The following Servlet stores the value of a request parameter in a member field and then later echoes the parameter value to the response output stream.

```
public class GuestBook extends HttpServlet {

String name;

protected void doPost (HttpServletRequest req, HttpServletResponse res) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
```

While this code will work perfectly in a single-user environment, if two users access the Servlet at approximately the same time, it is possible for the two request handler threads to interleave in the following way:

Thread 1: assign "Dick" to name
Thread 2: assign "Jane" to name
Thread 1: print "Jane, thanks for visiting!"
Thread 2: print "Jane, thanks for visiting!"

Thereby showing the first user the second user's name.

Recommendations:

Do not use Servlet member fields for anything but constants. (i.e. make all member fields static final).

Developers are often tempted to use Servlet member fields for user data when they need to transport data from one region of code to another. If this is your aim, consider declaring a separate class and using the Servlet only to "wrap" this new class.

Example 2: The bug in Example 1 can be corrected in the following way:

```
public class GuestBook extends HttpServlet {

protected void doPost (HttpServletRequest req, HttpServletResponse res) {
GBRequestHandler handler = new GBRequestHandler();
handler.handle(req, res);
}
```

```
}
public class GBRequestHandler {
String name;
public void handle(HttpServletRequest req, HttpServletResponse res) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
```

Alternatively, a Servlet can utilize synchronized blocks to access servlet instance variables but using synchronized blocks may cause significant performance problems.

Please notice that wrapping the field access within a synchronized block will only prevent the issue if all read and write operations on that member are performed within the same synchronized block or method.

Example 3: Wrapping the Example 1 write operation (assignment) in a synchronized block will not fix the problem since the threads will have to get a lock to modify name field, but they will release the lock afterwards, allowing a second thread to change the value again. If, after changing the name value, the first thread resumes execution, the value printed will be the one assigned by the second thread:

```
public class GuestBook extends HttpServlet {
String name;
protected void doPost (HttpServletRequest req, HttpServletResponse res) {
synchronized(name) {
name = req.getParameter("name");
}
...
out.println(name + ", thanks for visiting!");
}
}
```

In order to fix the race condition, all the write and read operations on the shared member field should be run atomically within the same synchronized block:

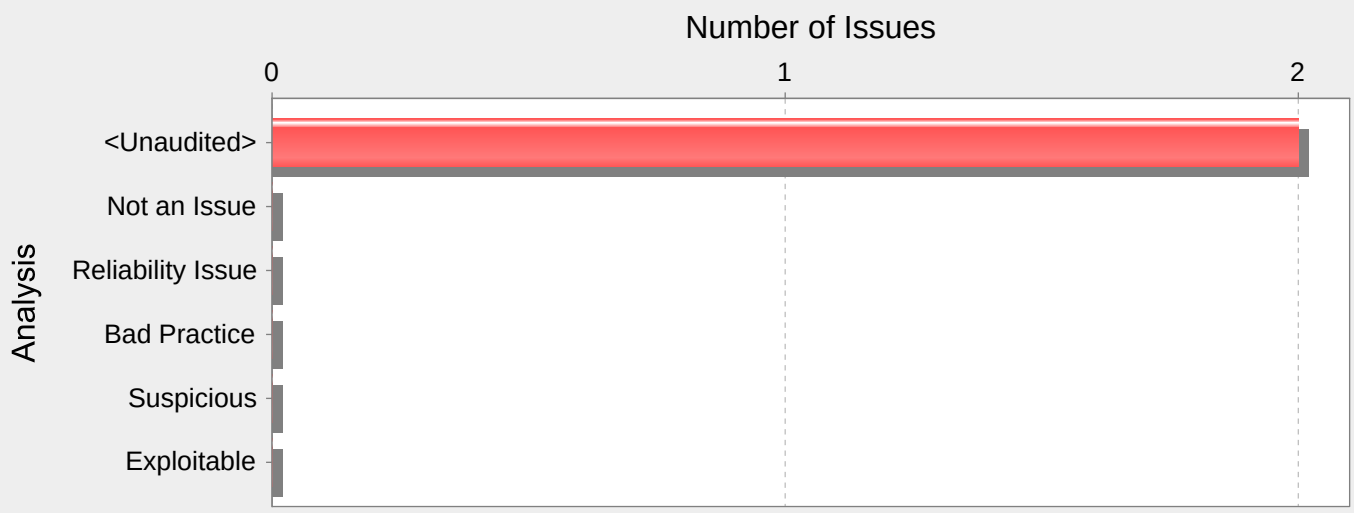
```
public class GuestBook extends HttpServlet {
String name;
protected void doPost (HttpServletRequest req, HttpServletResponse res) {
synchronized(name) {
name = req.getParameter("name");
...
out.println(name + ", thanks for visiting!");
}
}
}
```

ProductService.java, line 81 (Race Condition: Singleton Member Field)

Fortify Priority:	High	**Folder**	High
Kingdom:	Time and State		
Abstract:	The class ProductService is a singleton, so the member field pageSize is shared between users. The result is that one user could see another user's data.		
Sink:	ProductService.java:81 AssignmentStatement()		

```
79
80                      public void setPageSize(Integer pageSize) {
81                          this.pageSize = pageSize;
82                      }
```

Category: Weak Encryption: Insecure Mode of Operation (2 Issues)



Abstract:

The function encryptPassword() in EncryptedPasswordUtils.java uses a cryptographic encryption algorithm with an insecure mode of operation on line 46.

Explanation:

The mode of operation of a block cipher is an algorithm that describes how to repeatedly apply a cipher's single-block operation to securely transform amounts of data larger than a block. Some modes of operation include Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Counter (CTR).

ECB mode is inherently weak, as it produces the same ciphertext for identical blocks of plain text. CBC mode is vulnerable to padding oracle attacks. CTR mode is the superior choice because it does not have these weaknesses.

Example 1: The following code uses the AES cipher with ECB mode:

```
...
SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");
cipher.init(Cipher.ENCRYPT_MODE, key);
...
```

Recommendations:

Avoid using ECB and CBC modes of operation when encrypting data larger than a block. CBC mode is somewhat inefficient and poses a serious risk if used with SSL [1]. Instead, use CCM (Counter with CBC-MAC) mode or, if performance is a concern, GCM (Galois/Counter Mode) mode where they are available.

Example 2: The following code uses the AES cipher with GCM mode:

```
...
SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
Cipher cipher = Cipher.getInstance("AES/GCM/PKCS5Padding", "BC");
cipher.init(Cipher.ENCRYPT_MODE, key);
...
```

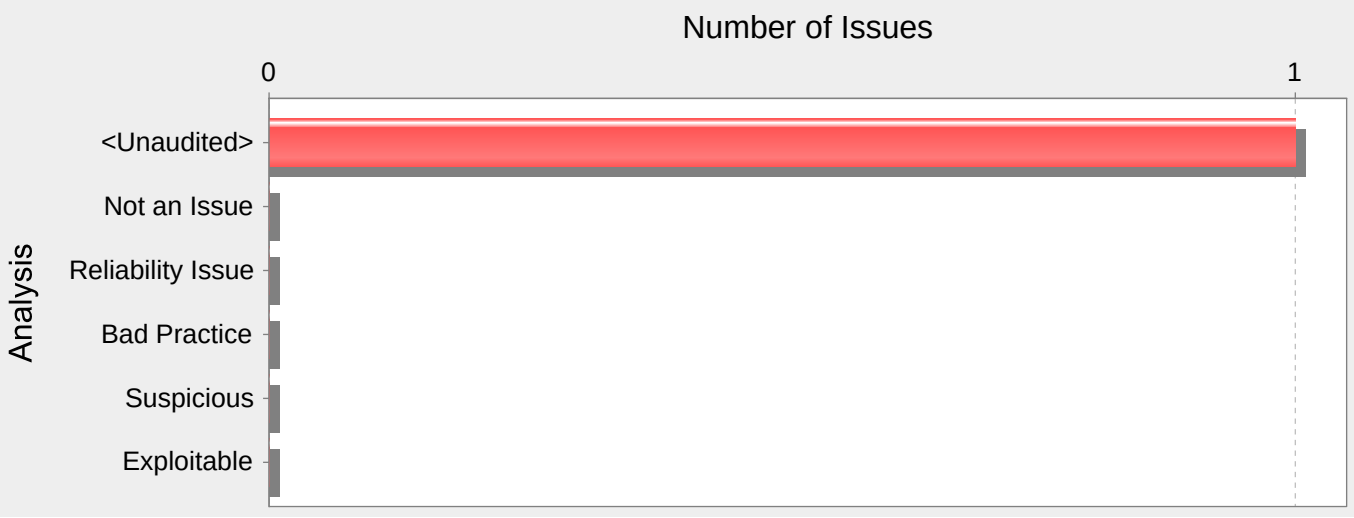
EncryptedPasswordUtils.java, line 46 (Weak Encryption: Insecure Mode of Operation)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The function encryptPassword() in EncryptedPasswordUtils.java uses a cryptographic encryption algorithm with an insecure mode of operation on line 46.		

Sink: EncryptedPasswordUtils.java:46 getInstance()

```
44          byte[] encrypted = null;
45          try {
46        Cipher desCipher = Cipher.getInstance("DES");
47        desCipher.init(Cipher.ENCRYPT_MODE, keySpec);
48        encrypted = desCipher.doFinal(password.getBytes());
```

Category: Insecure Randomness: Hardcoded Seed (1 Issues)



Abstract:

The function genId() in AdminUtils.java is passed a constant value for the seed. Functions that generate random or pseudorandom values, which are passed a seed, should not be called with a constant argument.

Explanation:

Functions that generate random or pseudorandom values, which are passed a seed, should not be called with a constant argument. If a pseudorandom number generator (such as Random) is seeded with a specific value (using a function such as Random.setSeed()), the values returned by Random.nextInt() and similar methods which return or assign values are predictable for an attacker that can collect a number of PRNG outputs.

Example 1: The values produced by the Random object s are predictable from the Random object r.

```
Random r = new Random();
r.setSeed(12345);
int i = r.nextInt();
byte[] b = new byte[4];
r.nextBytes(b);

Random s = new Random();
s.setSeed(12345);
int j = s.nextInt();
byte[] c = new byte[4];
s.nextBytes(c);
```

In this example, pseudorandom number generators: r and s were identically seeded, so i == j, and corresponding values of arrays b[] and c[] are equal.

Recommendations:

Use a cryptographic PRNG seeded with hardware-based sources of randomness, such as ring oscillators, disk drive timing, thermal noise, or radioactive decay. Doing so makes the sequence of data produced by Random.nextInt() and similar methods much harder to predict than setting the seed to a constant.

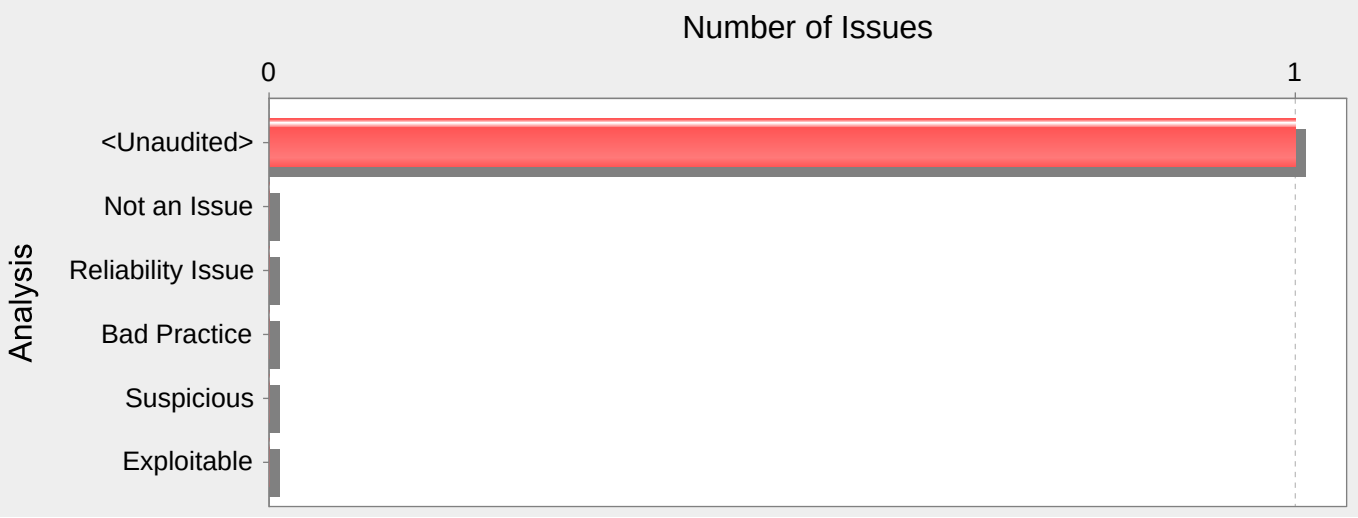
AdminUtils.java, line 114 (Insecure Randomness: Hardcoded Seed)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	The function genId() in AdminUtils.java is passed a constant value for the seed. Functions that generate random or pseudorandom values, which are passed a seed, should not be called with a constant argument.		

Sink: AdminUtils.java:114 setSeed()

```
112
113                 Random r=new Random();
114                 r.setSeed(12345);
115                 return r.nextInt();
116             }
```

Category: Insecure SSL: Server Identity Verification Disabled (1 Issues)



Abstract:

The connection established via send() in EmailUtils.java does not verify the server certificate when making an SSL connection. This leaves the application vulnerable to a man-in-the-middle attack.

Explanation:

In some libraries that use SSL connections, the server certificate is not verified by default. This is equivalent to trusting all certificates.

Example 1: This application does not explicitly verify the server certificate.

```
...
Email email = new SimpleEmail();
email.setHostName("smtp.servermail.com");
email.setSmtpPort(465);
email.setAuthenticator(new DefaultAuthenticator(username, password));
email.setSSLOnConnect(true);
email.setFrom("user@gmail.com");
email.setSubject("TestMail");
email.setMsg("This is a test mail ... :-)");
email.addTo("foo@bar.com");
email.send();
...
```

When trying to connect to smtp.mailserver.com:465, this application would readily accept a certificate issued to "hackedserver.com". The application would now potentially leak sensitive user information on a broken SSL connection to the hacked server.

Recommendations:

Do not forget server verification checks when making SSL connections. Depending on the library used, make sure to verify server identity and establish a secure SSL connection.

Example 2: This application does explicitly verify the server certificate.

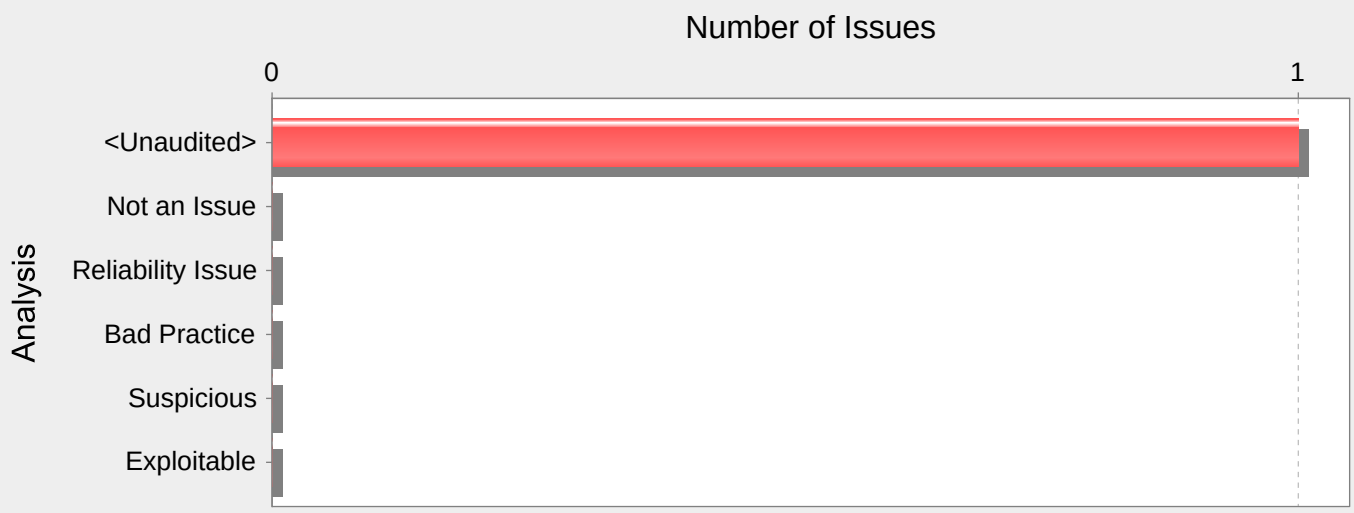
```
...
Email email = new SimpleEmail();
email.setHostName("smtp.servermail.com");
email.setSmtpPort(465);
email.setAuthenticator(new DefaultAuthenticator(username, password));
email.setSSLCheckServerIdentity(true);
email.setSSLOnConnect(true);
email.setFrom("user@gmail.com");
email.setSubject("TestMail");
email.setMsg("This is a test mail ... :-)");
email.addTo("foo@bar.com");
email.send();
```

...

EmailUtils.java, line 95 (Insecure SSL: Server Identity Verification Disabled)			
Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Security Features		
Abstract:	The connection established via send() in EmailUtils.java does not verify the server certificate when making an SSL connection. This leaves the application vulnerable to a man-in-the-middle attack.		
Sink:	EmailUtils.java:95 server.send() : Mail sent to unverified server()		

```
93              server.setMsg(request.getBody());
94              server.setBounceAddress(request.getBounce());
95              server.send();
96          }
97      }
```

Category: Key Management: Hardcoded Encryption Key (1 Issues)



Abstract:

Hardcoded encryption keys can compromise security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to hardcode an encryption key because it allows all of the project's developers to view the encryption key, and makes fixing the problem extremely difficult. After the code is in production, a software patch is required to change the encryption key. If the account that is protected by the encryption key is compromised, the owners of the system must choose between security and availability.

Example 1: The following code uses a hardcoded encryption key:

```
...
private static final String encryptionKey = "lakdsljkalkjlksdfkl";
byte[] keyBytes = encryptionKey.getBytes();
SecretKeySpec key = new SecretKeySpec(keyBytes, "AES");
Cipher encryptCipher = Cipher.getInstance("AES");
encryptCipher.init(Cipher.ENCRYPT_MODE, key);
...
```

Anyone with access to the code has access to the encryption key. After the application has shipped, there is no way to change the encryption key unless the program is patched. An employee with access to this information can use it to break into the system. If attackers had access to the executable for the application, they could extract the encryption key value.

Recommendations:

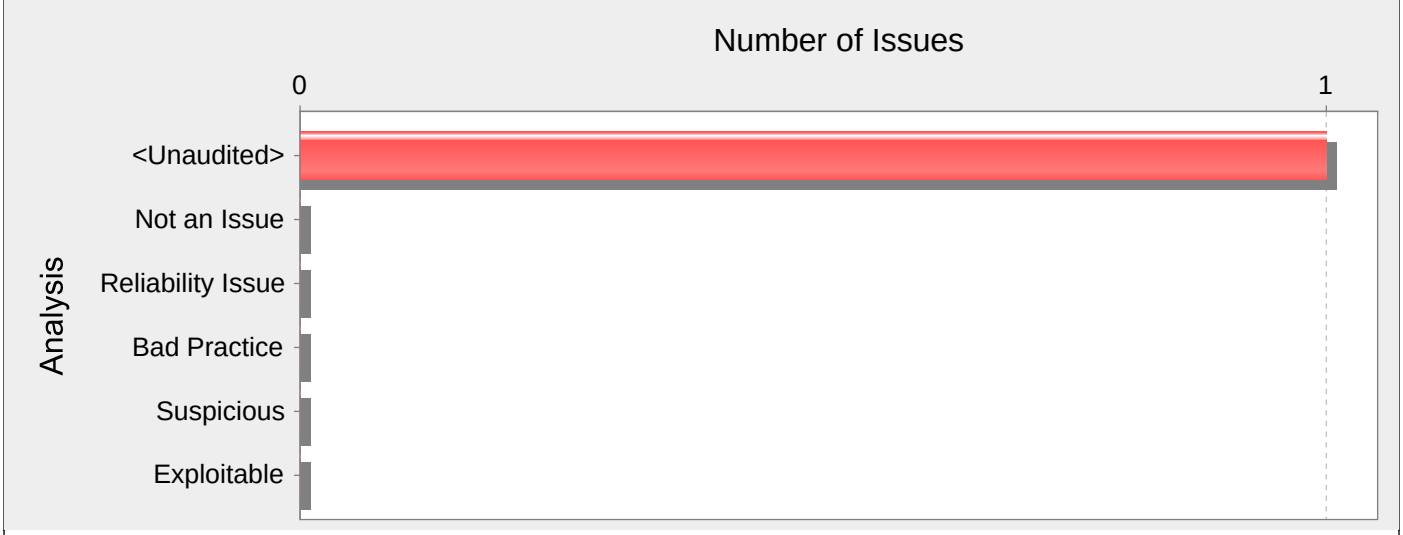
Encryption keys should never be hardcoded and should be obfuscated and managed in an external source. Storing encryption keys in plain text anywhere on the system allows anyone with sufficient permissions to read and potentially misuse the encryption key.

EncryptedPasswordUtils.java, line 41 (Key Management: Hardcoded Encryption Key)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Hardcoded encryption keys can compromise security in a way that cannot be easily remedied.		
Sink:	EncryptedPasswordUtils.java:41 FunctionCall: SecretKeySpec()		

```
39
40                  private static final byte[] iv = { 22, 33, 11, 44, 55, 99, 66, 77 };
41                  private static final SecretKey keySpec = new SecretKeySpec(iv, "DES");
42
43                      public static String encryptPassword(String password) {
```

Category: Open Redirect (1 Issues)



Abstract:

The file DefaultController.java passes unvalidated data to an HTTP redirect function on line 178. Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.

Explanation:

Redirects allow web applications to direct users to different pages within the same application or to external sites. Applications utilize redirects to aid in site navigation and, in some cases, to track how users exit the site. Open redirect vulnerabilities occur when a web application redirects clients to any arbitrary URL that can be controlled by an attacker.

Attackers might utilize open redirects to trick users into visiting a URL to a trusted site, but then redirecting them to a malicious site. By encoding the URL, an attacker can make it difficult for end-users to notice the malicious destination of the redirect, even when it is passed as a URL parameter to the trusted site. Open redirects are often abused as part of phishing scams to harvest sensitive end-user data.

Example 1: The following JSP code instructs the user's browser to open a URL parsed from the dest request parameter when a user clicks the link.

```
<%
...
String strDest = request.getParameter("dest");
pageContext.forward(strDest);
...
%>
```

If a victim received an email instructing them to follow a link to "http://trusted.example.com/ecommerce/redirect.asp?dest=www.wilyhacker.com", the user would likely click on the link believing they would be transferred to the trusted site. However, when the victim clicks the link, the code in Example 1 will redirect the browser to "http://www.wilyhacker.com".

Many users have been educated to always inspect URLs they receive in emails to make sure the link specifies a trusted site they know. However, if the attacker Hex encoded the destination url as follows:

"http://trusted.example.com/ecommerce/redirect.asp?dest=%77%69%6C%79%68%61%63%6B%65%72%2E%63%6F%6D"

then even a savvy end-user may be fooled into following the link.

Recommendations:

Unvalidated user input should not be allowed to control the destination URL in a redirect. Instead, use a level of indirection: create a list of legitimate URLs that users are allowed to specify and only allow users to select from the list. With this approach, input provided by users is never used directly to specify a URL for redirects.

Example 2: The following code references an array populated with valid URLs. The link the user clicks passes in the array index that corresponds to the desired URL.

```
<%
...
try {
int strDest = Integer.parseInt(request.getParameter("dest"));
if((strDest >= 0) && (strDest <= strURLArray.length -1 ))
{
```

```
strFinalURL = strURLArray[strDest];
pageContext.forward(strFinalURL);
}
}
catch (NumberFormatException nfe) {
// Handle exception
...
}
...
%>
```

In some situations this approach is impractical because the set of legitimate URLs is too large or too hard to keep track of. In such cases, use a similar approach to restrict the domains that users can be redirected to, which can at least prevent attackers from sending users to malicious external sites.

Tips:

1. 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.

2. Fortify AppDefender adds protection against this category.

DefaultController.java, line 178 (Open Redirect)

Fortify Priority:	Critical	**Folder**	Critical
Kingdom:	Input Validation and Representation		
Abstract:	The file DefaultController.java passes unvalidated data to an HTTP redirect function on line 178. Allowing unvalidated input to control the URL used in a redirect can aid phishing attacks.		

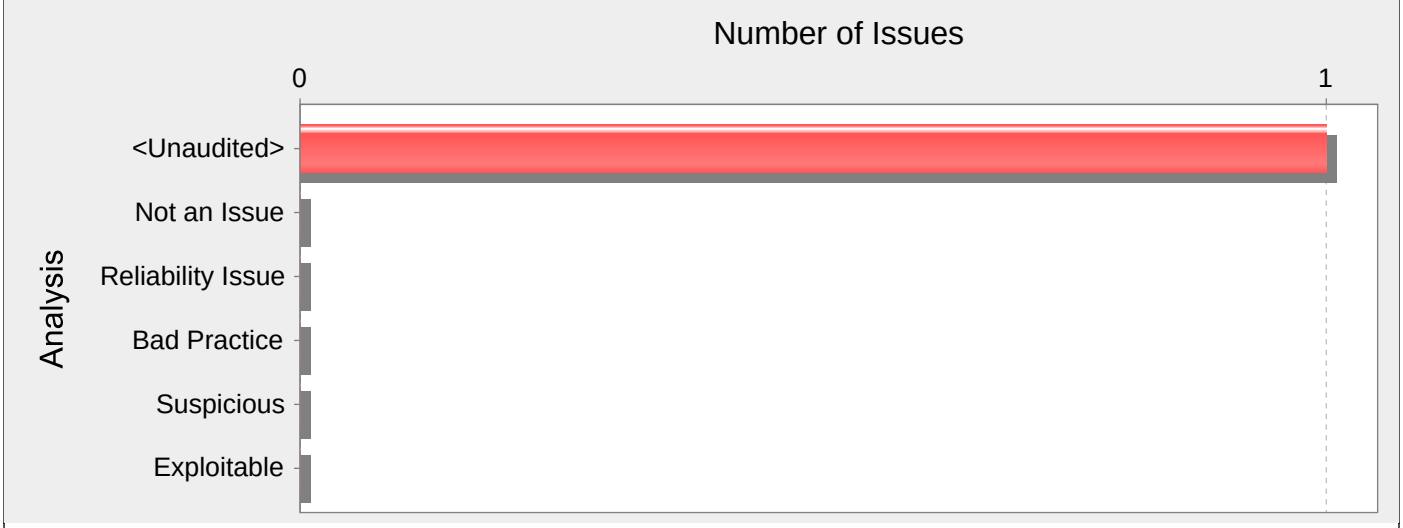
Source: DefaultController.java:99 javax.servlet.http.HttpServletRequest.getHeader()

```
97      public String login(HttpServletRequest request, Model model, Principal principal)
{
98          HttpSession session = request.getSession(false);
99          String referer = (String) request.getHeader("referer");
100         session.setAttribute("loginReferer", referer);
101         this.setModelDefaults(model, principal, "login");
```

Sink: DefaultController.java:178 Return()

```
176         String jwtToken = jwtUtils.generateAndSetSession(request, response,
authentication);
177         String targetUrl = CustomAuthenticationSuccessHandler.getTargetUrl(request,
response, authentication);
178         return "redirect:"+targetUrl;
179     }
```

Category: Password Management: Empty Password (1 Issues)



Abstract:

Empty passwords may compromise system security in a way that cannot be easily remedied.

Explanation:

It is never a good idea to assign an empty string to a password variable. If the empty password is used to successfully authenticate against another system, then the corresponding account's security is likely compromised because it accepts an empty password. If the empty password is merely a placeholder until a legitimate value can be assigned to the variable, then it can confuse anyone unfamiliar with the code and potentially cause problems on unexpected control flow paths.

Example 1: The following code attempts to connect to a database with an empty password.

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

If the code in Example 1 succeeds, it indicates that the database user account "scott" is configured with an empty password, which an attacker can easily guess. After the program ships, updating the account to use a non-empty password will require a code change.

Example 2: The following code initializes a password variable to an empty string, attempts to read a stored value for the password, and compares it against a user-supplied value.

```
...
String storedPassword = "";
String temp;
if ((temp = readPassword()) != null) {
storedPassword = temp;
}
if(storedPassword.equals(userPassword))
// Access protected resources
...
}
...
```

If readPassword() fails to retrieve the stored password due to a database error or another problem, then an attacker could trivially bypass the password check by providing an empty string for userPassword.

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

Example 3: The following code initializes username and password variables to empty strings, reads credentials from an Android WebView store if they have not been previously rejected by the server for the current request, and uses them to setup authentication for viewing protected pages.

```
...
webview.setWebViewClient(new WebViewClient() {
public void onReceivedHttpAuthRequest(WebView view,
HttpAuthHandler handler, String host, String realm) {
```

```
String username = "";
String password = "";
if (handler.useHttpAuthUsernamePassword()) {
String[] credentials = view.getHttpAuthUsernamePassword(host, realm);
username = credentials[0];
password = credentials[1];
}
handler.proceed(username, password);
}
});
...
```

Similar to Example 2, if useHttpAuthUsernamePassword() returns false, an attacker will be able to view protected pages by supplying an empty password.

Recommendations:

Always read stored password values from encrypted, external resources and assign password variables meaningful values. Ensure that sensitive resources are never protected with empty or null passwords.

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 4: 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.

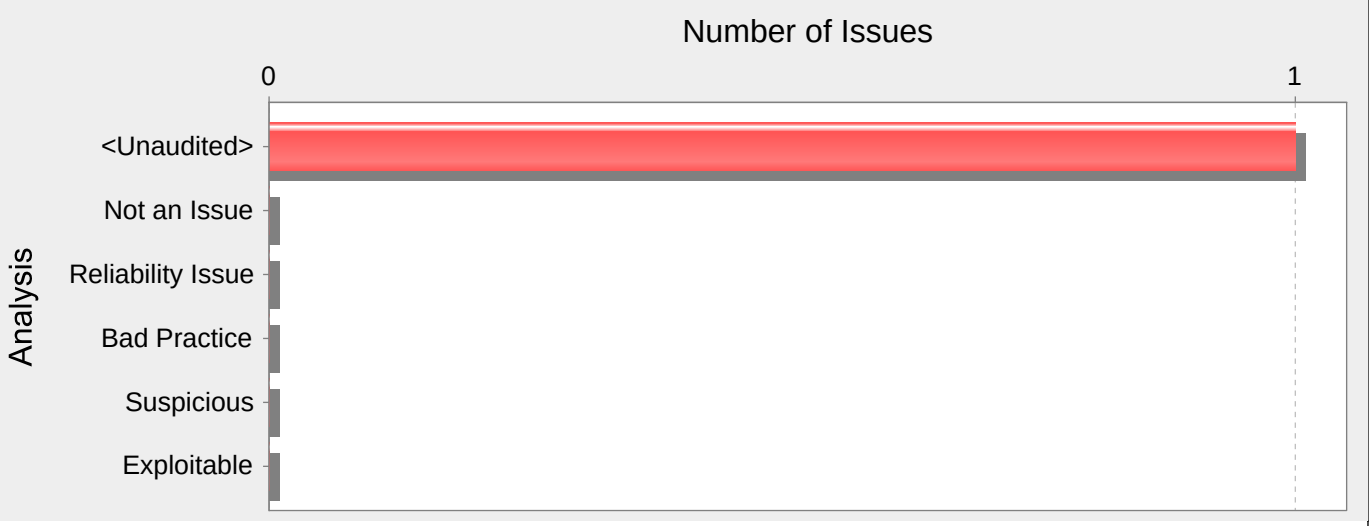
2. To identify null, empty, or hardcoded passwords, default rules only consider fields and variables that contain the word password. However, the Fortify Custom Rules Editor provides the Password Management wizard that makes it easy to create rules for detecting password management issues on custom-named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

EncryptedPasswordUtils.java, line 60 (Password Management: Empty Password)

Fortify Priority:	High	**Folder**	High
Kingdom:	Security Features		
Abstract:	Empty passwords may compromise system security in a way that cannot be easily remedied.		
Sink:	EncryptedPasswordUtils.java:60 VariableAccess: encPassword1()		

```
58      public static boolean matches(String password1, String password2) {
59        byte[] encrypted = null;
60        String encPassword1 = "";
61        try {
62      Cipher desCipher = Cipher.getInstance("DES");
```

Category: Password Management: Empty Password in Configuration File (1 Issues)



Abstract:

Using an empty string as a password is insecure.

Explanation:

It is never appropriate to use an empty string as a password. It is too easy to guess.

Recommendations:

Require that sufficiently hard-to-guess passwords protect all accounts and system resources. Consult the references to help establish appropriate password guidelines.

Tips:

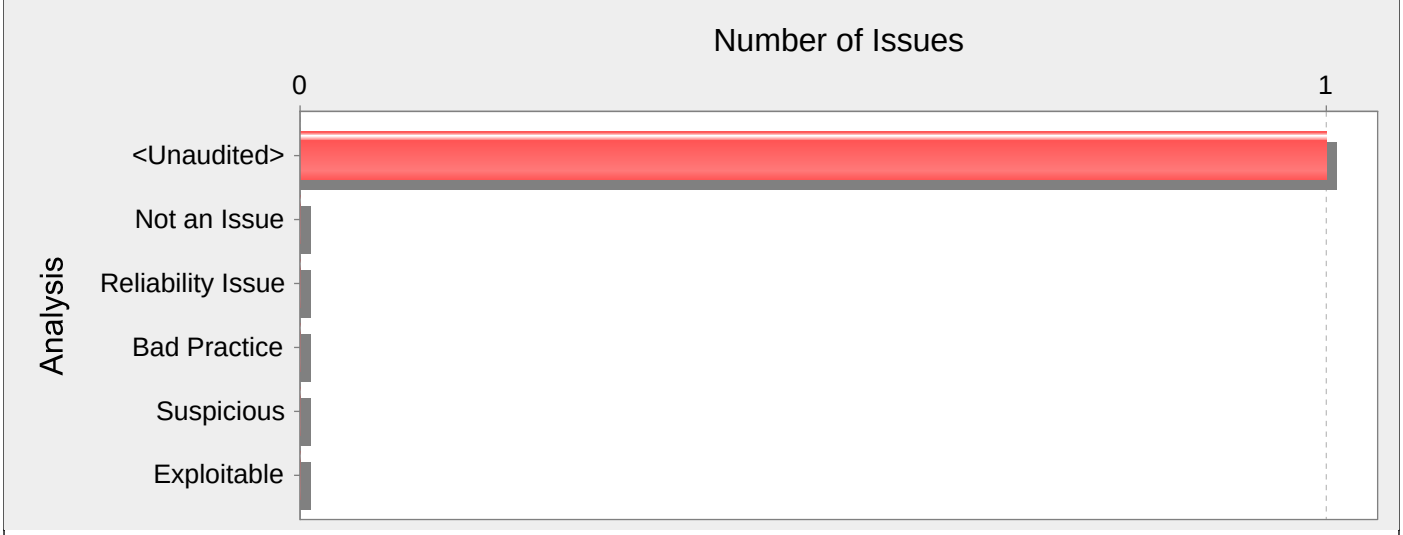
1. Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password.

application.yml, line 59 (Password Management: Empty Password in Configuration File)

Fortify Priority:	High	**Folder**	High
Kingdom:	Environment		
Abstract:	Using an empty string as a password is insecure.		
Sink:	application.yml:59 spring.mail.password()		

```
57      host: smtp.sendgrid.net
58      username: apikey
59      password: # Enter SendGrid API Password here
60      port: 587
61      test-connection: false
```

Category: Unreleased Resource: Streams (1 Issues)



Abstract:

The function registerUser() in UserUtils.java sometimes fails to release a system resource allocated by FileReader() on line 81.

Explanation:

The program can potentially fail to release a system resource.

Resource leaks have at least two common causes:

- Error conditions and other exceptional circumstances.

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

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

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

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

Recommendations:

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

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

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

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

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

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

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

UserUtils.java, line 81 (Unreleased Resource: Streams)

Fortify Priority:	High	**Folder**	High
Kingdom:	Code Quality		
Abstract:	The function registerUser() in UserUtils.java sometimes fails to release a system resource allocated by FileReader() on line 81.		

Sink: UserUtils.java:81 new FileReader(...)

```
79          File dataFile = new File(getFilePath(NEWSLETTER_USER_FILE));
80          if (dataFile.exists()) {
81              jsonArray = (JSONArray) jsonParser.parse(new
FileReader(getFilePath(NEWSLETTER_USER_FILE)));
82          } else {
83              Boolean created = dataFile.createNewFile();
```

Issue Count by Category

Issues by Category

Trust Boundary Violation	52
Access Control: Database	37
Log Forging (debug)	26
System Information Leak: Internal	25
System Information Leak: External	22
Path Manipulation	19
Session Puzzling: Spring	14
SQL Injection	14
Password Management: Password in Comment	13
System Information Leak	11
Dead Code: Unused Field	10
Log Forging	10
Password Management: Hardcoded Password	9
Cross-Site Request Forgery	7
Poor Style: Value Never Read	7
Insecure Randomness	5
JSON Injection	5
XML External Entity Injection	5
Cross-Site Scripting: Persistent	4
Cross-Site Scripting: Reflected	4
Header Manipulation	4
XML Entity Expansion Injection	4
Database Bad Practices: Use of Restricted Accounts	3
Poor Error Handling: Overly Broad Throws	3
Weak Encryption	3
Code Correctness: Byte Array to String Conversion	2
Dockerfile Misconfiguration: Default User Privilege	2
HTML5: Missing Content Security Policy	2
J2EE Bad Practices: Non-Serializable Object Stored in Session	2
Mass Assignment: Request Parameters Bound into Persisted Objects	2
Missing XML Validation	2
Often Misused: Boolean.getBoolean()	2
Often Misused: File Upload	2
Poor Error Handling: Overly Broad Catch	2
Race Condition: Singleton Member Field	2
Resource Injection	2
Weak Encryption: Insecure Mode of Operation	2
Dead Code: Unused Method	1
HTML5: Cross-Site Scripting Protection	1
Insecure Randomness: Hardcoded Seed	1
Insecure SSL: Server Identity Verification Disabled	1
Key Management: Hardcoded Encryption Key	1
Open Redirect	1
Password Management: Empty Password	1
Password Management: Empty Password in Configuration File	1
Poor Error Handling: Empty Catch Block	1
Privacy Violation	1
Spring Security Misconfiguration: Lack of Fallback Check	1

Spring Security Misconfiguration: Overly Permissive Firewall Policy	1
Unreleased Resource: Streams	1

Issue Breakdown by Analysis

Issues by Analysis