# Fortify Security Report

5/21/25

Demo User

# Executive Summary

## Issues Overview

On May 21, 2025, a source code review was performed over the IWA-Java code base. 767 files, 72,183 LOC (Executable) were scanned and reviewed for defects that could lead to potential security vulnerabilities. A total of 150 reviewed findings were uncovered during the analysis.

| Issues by Fortify Priority Order | |
|---|---|
| Medium | 60 |
| Low | 36 |
| High | 29 |
| Critical | 25 |

## 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: /home/kevinalee/repos/IWA-Java
Number of Files: 767
Lines of Code: 72183
Build Label: SNAPSHOT

## Scan Information

Scan time: 01:41
SCA Engine version: 25.2.0.0116
Machine Name: GBKLEE02
Username running scan: kevinalee

## Results Certification

Results Certification Valid

Details:

Results Signature:

SCA Analysis Results has Valid signature

Rules Signature:

There were no custom rules used in this scan

## Attack Surface

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

Environment Variables:
java.lang.System.getenv

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

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

Java Properties:
java.lang.System.getProperty

java.lang.System.getProperty

Serialized Data:
null.null.null
java.io.ObjectInputStream.readObject

System Information:
null.null.null
null.null.lambda
null.null.resolve
java.lang.System.getProperty
java.lang.System.getProperty
java.lang.Throwable.getLocalizedMessage
java.lang.Throwable.getMessage

Web:
null.~JS_Generic.val

## 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

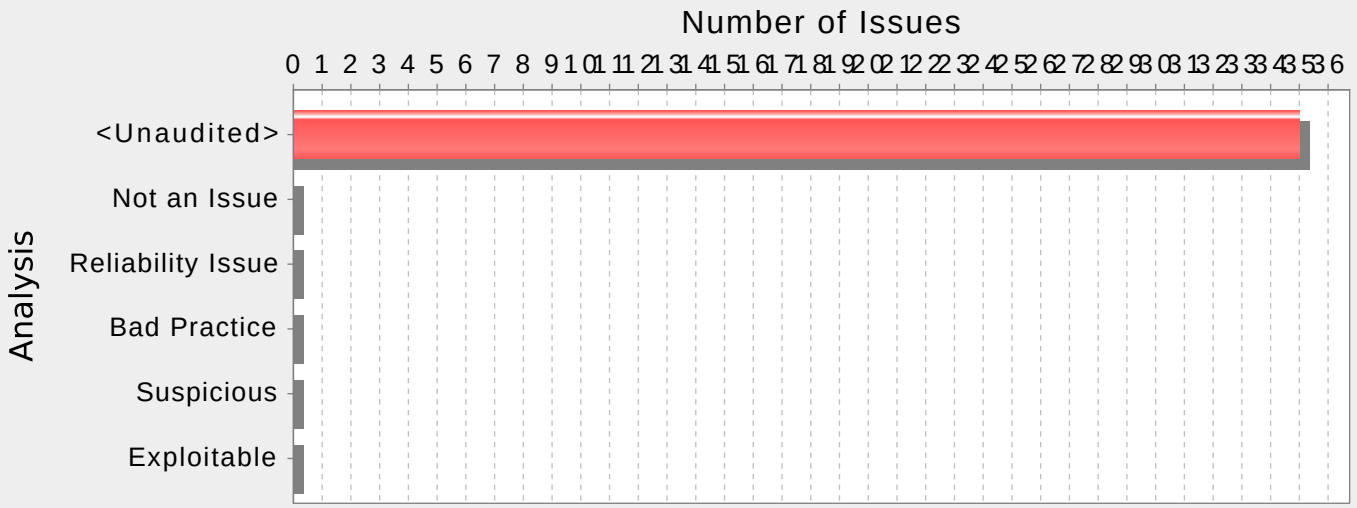
# Results Outline

## Overall number of results

The scan found 150 issues.

## Vulnerability Examples by Category

### Category: Password Management: Hardcoded Password (35 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 1: 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.

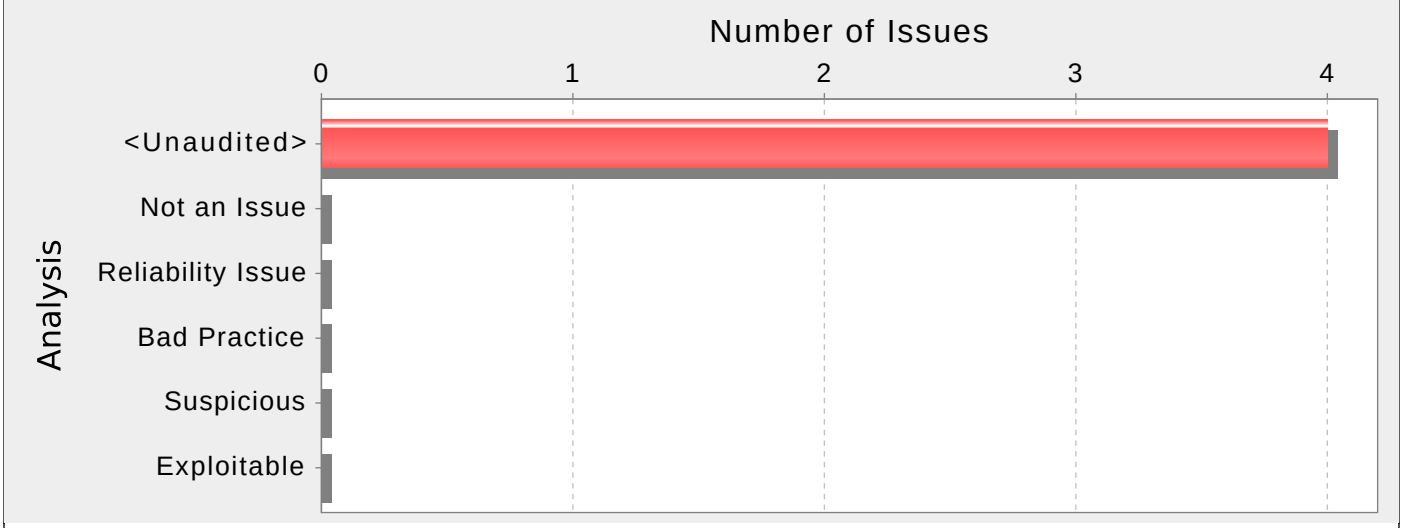
### application-dev.yml, line 37 (Password Management: Hardcoded Password)

| | | | |
|---|---|---|---|
| **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:37 ConfigPair()

```
35          url: jdbc:h2:mem:iwa_dev
36          username: sa
37          password: password
38          initialization-mode: always
39        jpa:
```

## Category: Privacy Violation (4 Issues)




### Abstract:

The method createUser() in ApiUserController.java mishandles confidential information, which can compromise user privacy and is often illegal.

### Explanation:

Privacy violations occur when:

1. Private user information enters the program.

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

Example 1: The following code contains a logging statement that tracks the records added to a database by storing the contents in a log file.

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

The code in Example 1 logs a plain text password to the file system. Although many developers trust the file system as a safe storage location for data, it should not be trusted implicitly, particularly when privacy is a concern.

Privacy is one of the biggest concerns in the mobile world for a couple of reasons. One of them is a much higher chance of device loss. The other has to do with inter-process communication between mobile applications. 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. Sensitive information should never be part of inter-process communication between mobile applications.

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

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

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

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

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

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

- Indirectly from a partner or other third party

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

- Location

- Cell phone number

- Serial numbers and device IDs

- Network Operator information

- Voicemail information

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

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

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

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

- Safe Harbor Privacy Framework [3]

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

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

- California SB-1386 [6]

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

## Recommendations:

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

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

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

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

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

```
import net.sqlcipher.database.SQLiteDatabase;
```

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

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

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

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

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

### Tips:

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

2. You can use the Fortify Java Annotations FortifyPassword, FortifyNotPassword, FortifyPrivate, and FortifyNotPrivate to indicate which fields and variables represent passwords and private data.

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.

## ApiUserController.java, line 123 (Privacy Violation)

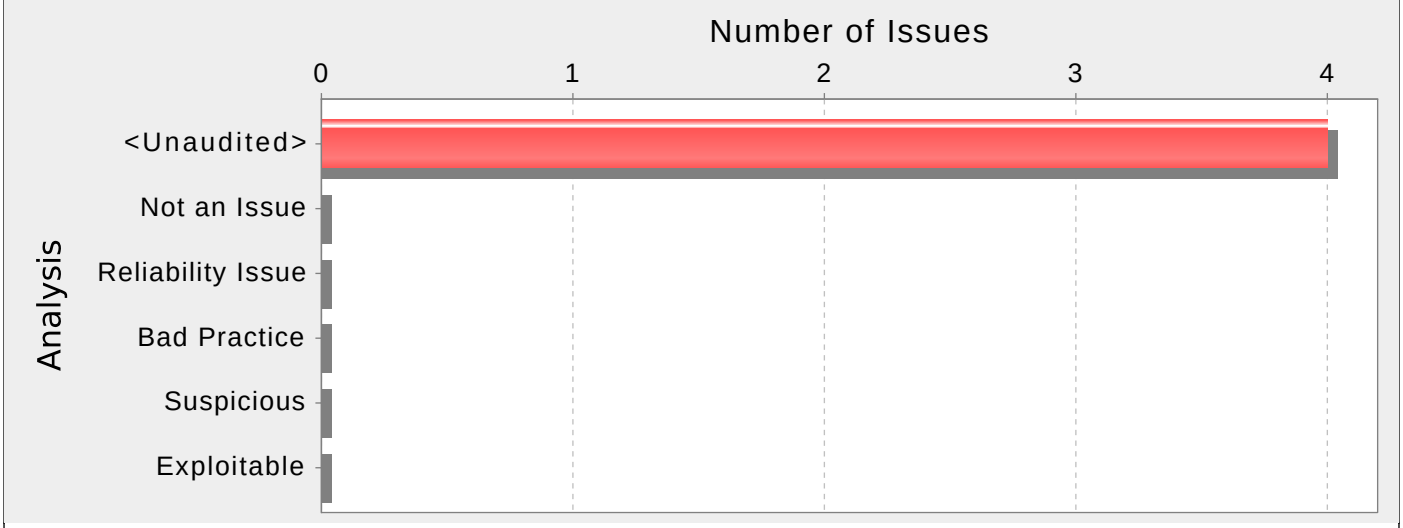
| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Security Features | | |
| **Abstract:** | The method createUser() in ApiUserController.java mishandles confidential information, which can compromise user privacy and is often illegal. | | |
| **Source:** | User.java:322 Read this.password() | | |

```
320                     @Override
321                     public String toString() {
322                         return "User [id=" + id + ", username=" + username + ", password=****** +
password + ", confirmPassword=******
323                                 + confirmPassword + ", secret=" + secret + ", firstName=" + firstName
+ ", lastName=" + lastName
324                                 + ", email=" + email + ", phone=" + phone + ", address=" + address +
", city=" + city + ", state="
```

**Sink:** ApiUserController.java:123 org.slf4j.Logger.debug()

```
121                     @io.swagger.v3.oas.annotations.parameters.RequestBody(description = "")
@Valid @RequestBody User newUser) {
122                     //newUser.setId(new UUID()); // set to 0 for sequence id generation
123                     log.debug("API::Creating new user: " + newUser.toString());
124                     return new ResponseEntity<>(new UserResponse(userService.saveUser(newUser)),
HttpStatus.OK);
125                 }
```

## Category: Weak Encryption (4 Issues)




### Abstract:

The call to SecretKeySpec() at EncryptedPasswordUtils.java line 42 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:

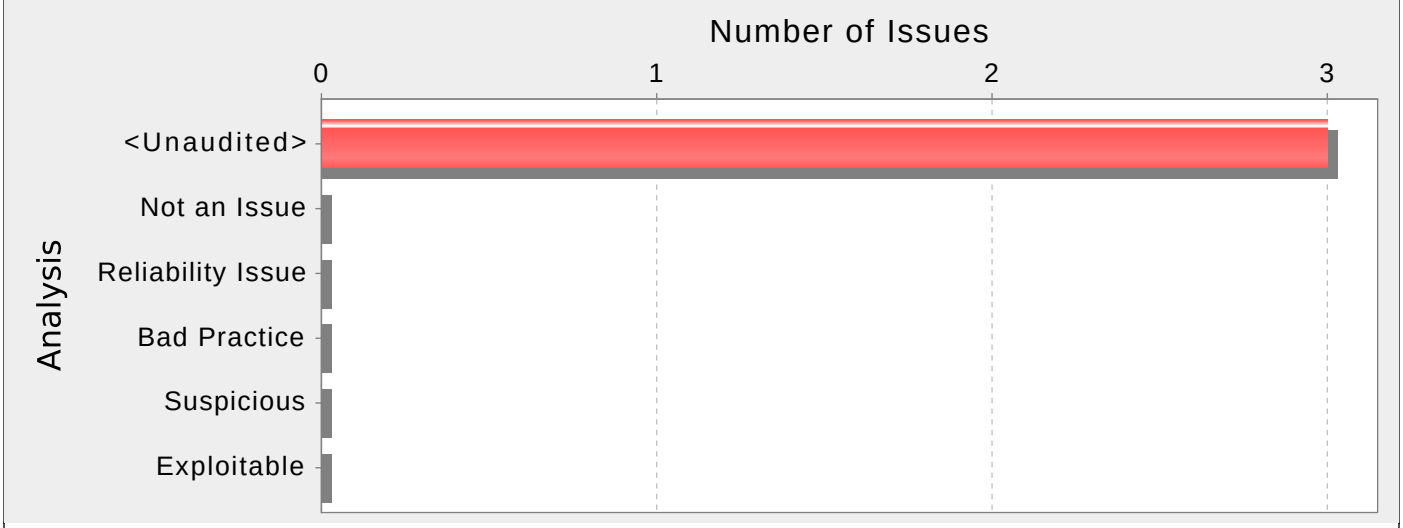
Use strong encryption algorithms with large key sizes to protect sensitive data. A strong alternative to DES is AES (Advanced Encryption Standard, formerly Rijndael). Before selecting an algorithm, first determine if your organization has standardized on a specific algorithm and implementation.

### EncryptedPasswordUtils.java, line 42 (Weak Encryption)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | Critical | **Folder** | Critical |
| **Kingdom:** | Security Features | | |
| **Abstract:** | The call to SecretKeySpec() at EncryptedPasswordUtils.java line 42 uses a weak encryption algorithm that cannot guarantee the confidentiality of sensitive data. | | |
| **Sink:** | EncryptedPasswordUtils.java:42 SecretKeySpec() | | |

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

## Category: Weak Encryption: Insecure Mode of Operation (3 Issues)




### Abstract:

The function encryptPassword() in EncryptedPasswordUtils.java uses a cryptographic encryption algorithm with an insecure mode of operation on line 47.

### 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);
...
```

In general, the mode of operation has minimal effect on RSA security. However, it is crucial to have secure padding, such as OAEP padding with SHA-2 hashing, for secure encryption, when you employ ECB or any other mode of operation.

Example 3: The following code performs encryption with an RSA public key using ECB with OAEP padding:

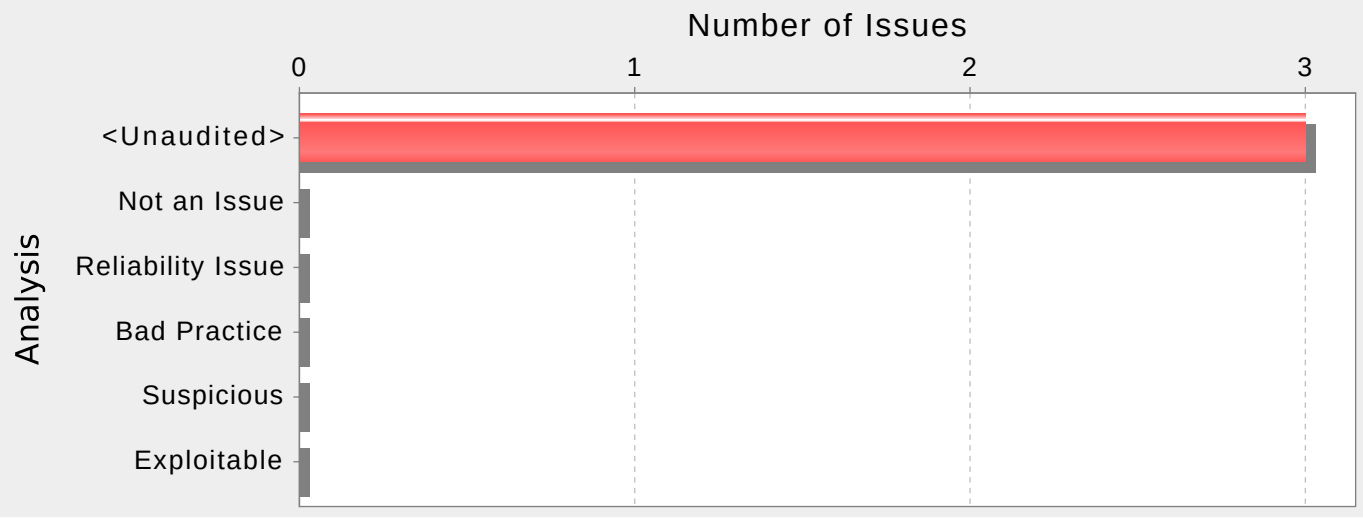
```
public Cipher getRSACipher() {
Cipher rsa = null;
try {
rsa = javax.crypto.Cipher.getInstance("RSA/ECB/OAEPWithSHA-256AndMGF1Padding");
}
catch (java.security.NoSuchAlgorithmException e) {
handleMissingAlgorithm();
}
catch (javax.crypto.NoSuchPaddingException e) {
```

```
handleMissingAlgorithm();
}
return rsa;
}
```

| EncryptedPasswordUtils.java, line 47 (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 47. | | |
| **Sink:** | EncryptedPasswordUtils.java:47 getInstance() | | |

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

## Category: XML External Entity Injection (3 Issues)




### Abstract:

XML parser configured in UserController.java:843 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:

Always configure an XML parser securely so that it does not allow external entities as part of an incoming XML document.

To avoid XXE injections the following properties should be set for an XML factory, parser or reader:

```
factory.setFeature("http://xml.org/sax/features/external-general-entities", false);
factory.setFeature("http://xml.org/sax/features/external-parameter-entities", false);
```

If inline DOCTYPE declaration is not needed, it can be completely disabled with the following property:

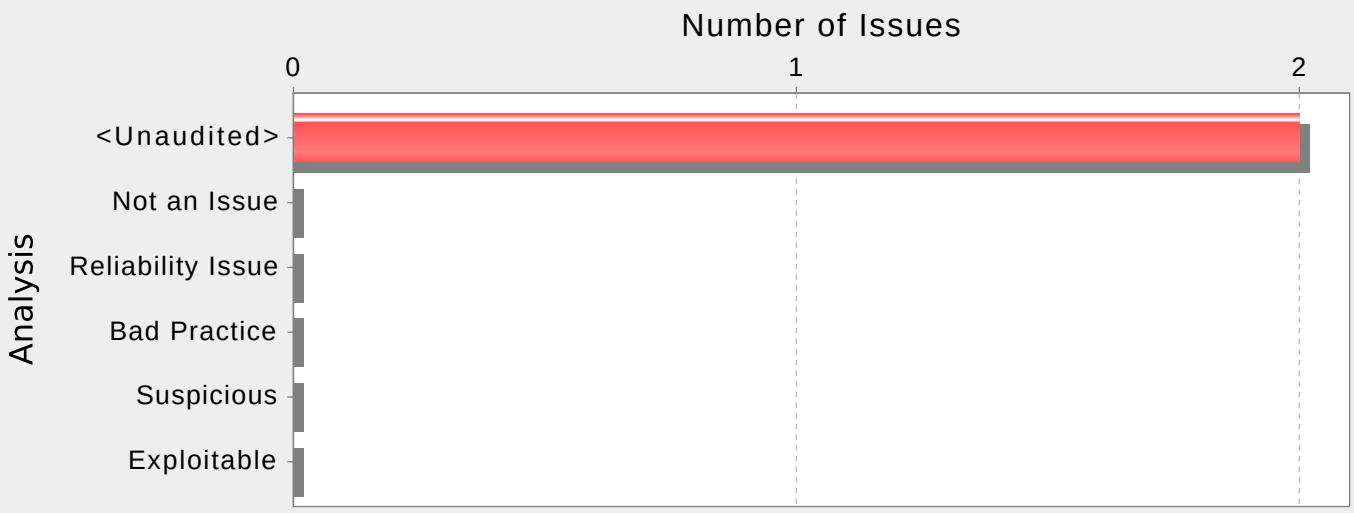
```
factory.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
```

## UserController.java, line 843 (XML External Entity Injection)

| | | | |
|---|---|---|---|
| **Fortify Priority:** | High | **Folder** | High |
| **Kingdom:** | Input Validation and Representation | | |
| **Abstract:** | XML parser configured in UserController.java:843 does not prevent nor limit external entities resolution. This can expose the parser to an XML External Entities attack. | | |
| **Sink:** | UserController.java:843 db.parse(...) : XML document parsed allowing external entity resolution() | | |

```
841    dbf.setFeature(XMLConstants.FEATURE_SECURE_PROCESSING, false);
842          DocumentBuilder db = dbf.newDocumentBuilder();
843          Document doc = db.parse(fpath.toFile());
844          try (ByteArrayOutputStream bytesOutStream = new ByteArrayOutputStream()) {
845           writeXml(doc, bytesOutStream);
```

## 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.

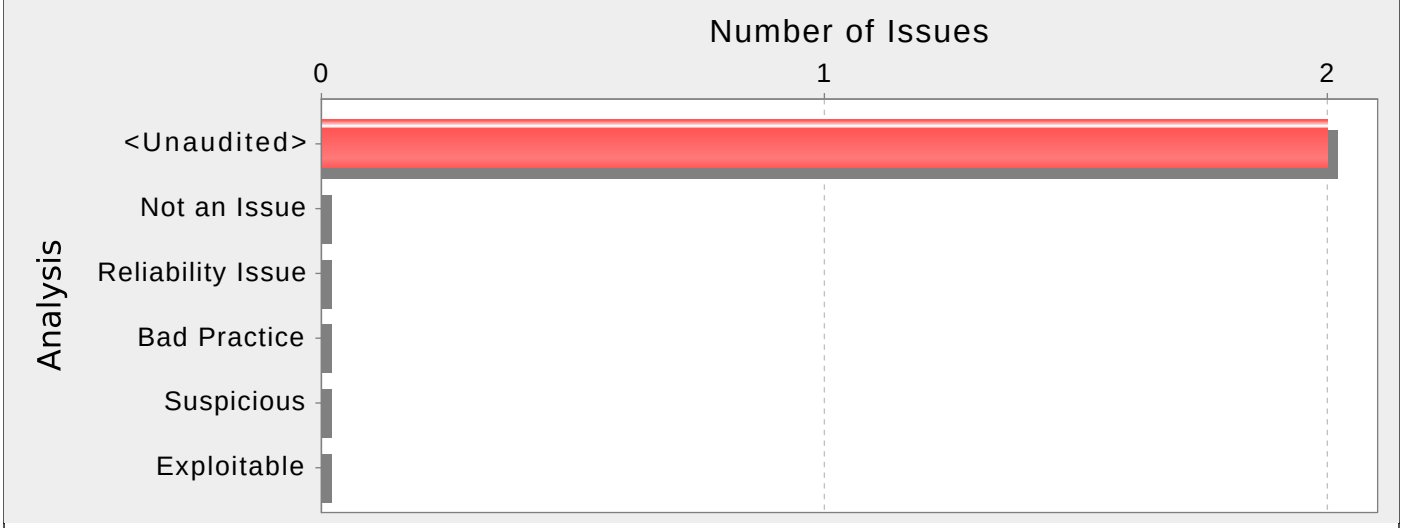
To modify a docker container to use a non-root user, the Dockerfile needs to specify a different user, such as:

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:11-jdk-slim` | | |
| 0 | `# A JDK 1.8 is needed if the WebInspect Runtime Agent is being used` | | |
| 1 | `#FROM openjdk:8u342-jdk-slim` | | |

## Category: Password Management: Empty Password in Configuration File (2 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:

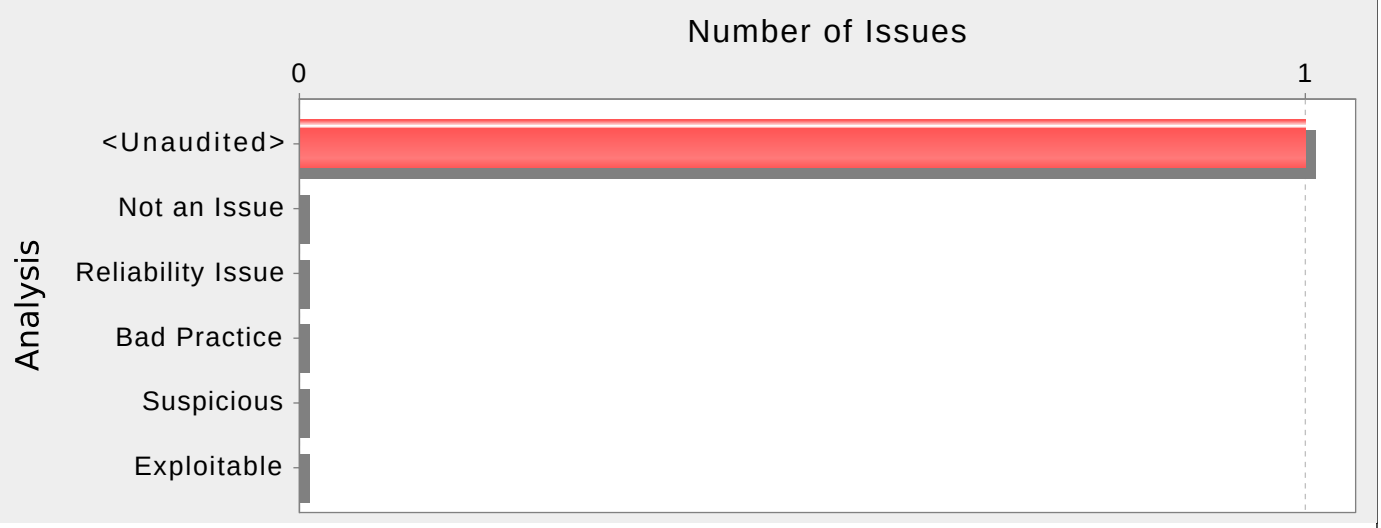
1. Fortify Static Code Analyzer searches configuration files for common names used for password properties. Audit these issues by verifying that the flagged entry is used as a password.

## application.yml, line 68 (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:68 spring.mail.password() | | |

```
66      host: smtp.sendgrid.net
67      username: apikey
68      password: # Your API Password
69      port: 587
70      test-connection: true
```

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




### Abstract:

Hardcoded encryption keys can compromise security in a way that is not easy to remedy.

### 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 42 (Key Management: Hardcoded Encryption Key)

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

**Sink:** EncryptedPasswordUtils.java:42 FunctionCall: SecretKeySpec()

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

| Issue Count by Category | |
|---|---|
| Issues by Category | |
| Insecure Transport: External Link | 60 |
| Password Management: Hardcoded Password | 35 |
| Password Management: Password in Comment | 17 |
| System Information Leak | 11 |
| Often Misused: File Upload | 4 |
| Privacy Violation | 4 |
| Weak Encryption | 4 |
| Weak Encryption: Insecure Mode of Operation | 3 |
| XML Entity Expansion Injection | 3 |
| XML External Entity Injection | 3 |
| Dockerfile Misconfiguration: Default User Privilege | 2 |
| Password Management: Empty Password in Configuration File | 2 |
| Key Management: Hardcoded Encryption Key | 1 |
| SQL Injection | 1 |

## Issue Breakdown by Analysis

### Issues by Analysis