



## **Fortify on Demand Security Review**

Tenant:	Other_740088900_FMA_686636323
Application:	web
Release:	test
Latest Analysis:	2025/02/13 06:02:51 AM
Latest Assessment Type:	Static Assessment

## Executive Summary

Tenant: Other\_740088900\_FMA\_686636323  
Application: web  
Release: test  
Business Criticality: High  
SDLC Status: QA/Test  
Static Analysis Date: 02/13/2025  
Dynamic Analysis ---  
Date:

### Fortify on Demand Security Rating



46 issues

Status: Failed

Static:



Dynamic:



Open



Source:

Monitoring:



### Application Details

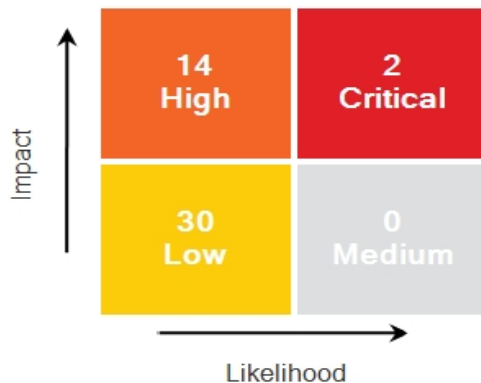
Application type: Other Development

Interface type: Web Service (SOA)

Project type: Other

Sample Application: True

#### Risk Totals by Severity



#### Issue Status

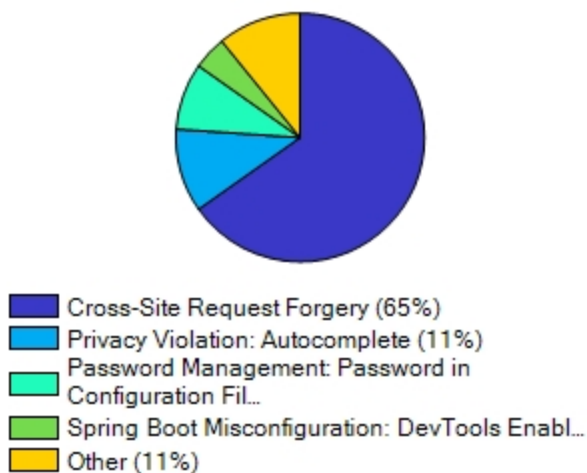
New	Existing	Reopened
27	19	0

#### Assignment Status



Assigned (2%)  
Unassigned (98%)

#### Most Prevalent Issues by Category



#### Developer Status



Open (100%)

#### Auditor Status



Pending Review (100%)

## Issue Breakdown

Issues are divided based on their impact (potential damage) and likelihood (probability of identification and exploit).

High impact / high likelihood issues represent the highest priority and present the greatest threat.

Low impact / low likelihood issues are the lowest priority and present the smallest threat.

See Appendix for more information.

Rating	Category	Test Type	
Critical	SQL Injection: MyBatis Mapper	Static As...	2
High	Password Management: Hardcoded Password	Static As...	1
High	Password Management: Password in Configuration File	Static As...	4
High	Privacy Violation: Autocomplete	Static As...	5
High	Spring Boot Misconfiguration: DevTools Enabled	Static As...	2
High	Unreleased Resource: Streams	Static As...	2
Low	Cross-Site Request Forgery	Static As...	30

## Issue Breakdown by OWASP Top 10 2017 PCI Sections 6.3, 6.5 & 6.6

The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.

The PCI compliance standards, particularly sections 6.3, 6.5, and 6.6, reference the OWASP Top Ten vulnerability categories as the core categories that must be tested for and remediated.

OWASP Category	Severity			
	Critical	High	Medium	Low
A1 - Injection	2			
A2 - Broken Authentication				
A3 - Sensitive Data Exposure		10		
A4 - XML External Entities (XXE)				
A5 - Broken Access Control				
A6 - Security Misconfiguration		2		
A7 - Cross-Site Scripting (XSS)				
A8 - Insecure Deserialization				
A9 - Using Components with Known ...				
A10 - Insufficient Logging and Monito...				
Total	2	12		

## Issue Breakdown by OWASP Top 10 2021 PCI Sections 6.3, 6.5 & 6.6

The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.

The PCI compliance standards, particularly sections 6.3, 6.5, and 6.6, reference the OWASP Top Ten vulnerability categories as the core categories that must be tested for and remediated.

OWASP Category	Severity			
	Critical	High	Medium	Low
A01 - Broken Access Control				30
A02 - Cryptographic Failures				
A03 - Injection	2			
A04 - Insecure Design		5		
A05 - Security Misconfiguration		6		
A06 - Vulnerable and Outdated Comp...				
A07 - Identification and Authenticatio...		1		
A08 - Software and Data Integrity Fail...				
A09 - Security Logging and Monitorin...				
A10 - Server-Side Request Forgery				
Total	2	12		30

## Issue Breakdown by Analysis Type

Issues are divided based on their impact (potential damage) and likelihood (probability of identification and exploit).

High impact / high likelihood issues represent the highest priority and present the greatest threat.

Low impact / low likelihood issues are the lowest priority and present the smallest threat.

See Appendix for more information.

Category	Static	Dynamic	Open S...	Monitor...
Cross-Site Request Forgery	30	0	0	0
Password Management: Hardcoded Password	1	0	0	0
Password Management: Password in Configuratio...	4	0	0	0
Privacy Violation: Autocomplete	5	0	0	0
Spring Boot Misconfiguration: DevTools Enabled	2	0	0	0
SQL Injection: MyBatis Mapper	2	0	0	0
Unreleased Resource: Streams	2	0	0	0
Total	46	0	0	0

## Issue Detail

Below is an enumeration of all issues found in the project. The issues are organized by priority and category and then broken down by the package, namespace, or location in which they occur.

The priority of an issue can be Critical, High, Medium, or Low.

Issues from static analysis reported on at same line number with the same category originate from different taint sources.

### 7.1.1 SQL Injection: MyBatis Mapper

**Critical**

CWE-89

OWASP Top 10 2017: A1 - Injection

OWASP Top 10 2021: A03 - Injection

PCI 3.2: 6.5.1 Injection Flaws

#### Summary

On line **20** of **MemberMapper.xml**, a SQL query is built using input potentially coming from an untrusted source. This call could allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands. Constructing a dynamic SQL statement with input from an untrusted source might allow an attacker to modify the statement's meaning or 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.

In this case, data is concatenated to the SQL statement in **MemberMapper.xml** on line **MemberMapper.xml**.

MyBatis Mapper XML files allow you to specify dynamic parameters in SQL statements and are typically defined by using the **#** characters, as follows:

```
<select id="getItems" parameterType="domain.company.MyParamClass" resultType="MyResultMap">
SELECT *
FROM items
WHERE owner = #{userName}
</select>
```

The **#** character with braces around the variable name indicate that MyBatis will create a parameterized query with the **userName** variable. However, MyBatis also allows you to concatenate variables directly to SQL statements using the **\$** character, opening the door for SQL injection.

**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.

```
<select id="getItems" parameterType="domain.company.MyParamClass" resultType="MyResultMap">
SELECT *
FROM items
WHERE owner = #{userName}
AND itemname = ${itemName}
</select>
```

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 should 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';
```

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 guarantee that an application is secure against 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.

## Recommendation

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 3:** [Example 1](#) can be rewritten to use parameterized SQL statements (instead of concatenating user supplied strings) as follows:

```
<select id="getItems" parameterType="domain.company.MyParamClass" resultType="MyResultM
ap">
SELECT *
FROM items
WHERE owner = #{userName}
AND itemname = #{itemName}
</select>
```

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. MyBatis includes functionality to account for building dynamic queries within its XML schema, whilst still giving the ability to use parameterized queries [2].

**Example 4:** Example 3 can be rewritten as a dynamic query within the XML in order to first verify if itemName exists, whilst still using parameterized queries as such:

```
<select id="getItems" parameterType="domain.company.MyParamClass" resultType="MyResultM
ap">
SELECT *
FROM items
WHERE owner = #{userName}
<if test="itemName != null">
AND itemname = #{itemName}
</if>
</select>
```

This way of creating dynamic queries prevents SQL injection due to keeping the security of parameterized queries, leaving it far safer and easier to maintain than string concatenation. If this is unavailable for any reason or insufficient, 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.

## References

1. MyBatis 3 | Mapper XML Files, MyBatis, <http://mybatis.github.io/mybatis-3/sqlmap-xml.html>
2. MyBatis 3 | Dynamic SQL, MyBatis, <http://mybatis.github.io/mybatis-3/dynamic-sql.html>
3. SQL Injection Attacks by Example, S. J. Friedl, <http://www.unixwiz.net/techtips/sql-injection.html>
4. Stop SQL Injection Attacks Before They Stop You, P. Litwin
5. SQL Injection and Oracle, Part One, P. Finnigan, <https://haind.wordpress.com/2008/06/13/sql-injection-and-oracle-part-onepete-finnigan/>
6. Writing Secure Code, Second Edition, M. Howard, D. LeBlanc
7. IDS00-J. Prevent SQL Injection, <https://www.securecoding.cert.org/confluence/display/java/IDS00-J.+Prevent+SQL+Injection>

8. INJECT-2: Avoid dynamic SQL, <http://www.oracle.com/technetwork/java/seccodeguide-139067.html#3>
9. Fortify Taxonomy | SQL Injection: MyBatis Mapper, <https://vulnecat.fortify.com/en/detail?category=SQL+Injection&subcategory=MyBatis+Mapper>

## Instances

SQL Injection: MyBatis Mapper

Critical

Package: N/A

Instance	Analysis Info	Analyzer
ID 46447462 - src/main/resources/mapper/MemberMapper.xml:20	<b>Sink:</b> in src/main/resources/mapper/MemberMapper.xml:20 <b>EnclosingMethod:</b>	configur...
ID 46447463 - target/classes/mapper/MemberMapper.xml:20	<b>Sink:</b> in target/classes/mapper/MemberMapper.xml:20 <b>EnclosingMethod:</b>	configur...

## 7.2.1 Password Management: Hardcoded Password

High

CWE-798, CWE-259

OWASP Top 10 2017: A3 - Sensitive Data Exposure

OWASP Top 10 2021: A07 - Identification and Authentication Failures

PCI 3.2: 6.3.1 Hardcoded Sensitive Information, 6.5.3 Insecure Cryptographic Storage, 8.2.1 Render authentication credentials unreadable

### Summary

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

### Explanation

It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. After the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system must choose between security and availability. In this case, a hardcoded password was found in the call to in **Member.java** on line **16**.

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

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

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

```
javap -c ConnMngr.class

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

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

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

```

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

```

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

## Execution

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.

## Recommendation

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

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

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

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

```

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

```

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

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

## References

1. SQLCipher., <http://sqlcipher.net/>
2. MSC03-J. Never hard code sensitive information, <https://www.securecoding.cert.org/confluence/display/java/MS-C03-J.+Never+hard+code+sensitive+information>
3. Fortify Taxonomy | Password Management: Hardcoded Password, <https://vulncat.fortify.com/en/detail?category=Password+Management&subcategory=Hardcoded+Password>

## Instances

Password Management: Hardcoded Password

High

Package: com.check.thyme.vo

Instance	Analysis Info	Analyzer
ID 46447490 - src/main/java/com/check/thyme/vo/Member.java:16	<b>Sink:</b> FieldAccess: password in src/main/java/com/check/thyme/vo/Member.java:16 <b>EnclosingMethod:</b> setPassword	structural



## 7.2.2 Password Management: Password in Configuration File

High

CWE-13, CWE-260, CWE-555

OWASP Top 10 2017: A3 - Sensitive Data Exposure

OWASP Top 10 2021: A05 - Security Misconfiguration

PCI 3.2: 6.3.1 Hardcoded Sensitive Information, 6.5.3 Insecure Cryptographic Storage, 8.2.1 Render authentication credentials unreadable

### Summary

Storing a plain text password in a configuration file may result in a system compromise.

### Explanation

Storing a plain text password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plain text. In this case, a hardcoded password exists in **application.properties** on line **29**.

### Execution

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 and that the password entry contains plain text.
2. If the entry in the configuration file is a default password, require that it be changed in addition to requiring that it be obfuscated in the configuration file.

### Recommendation

A password should never be stored in plain text. An administrator should be required to enter the password when the system starts. If that approach is impractical, a less secure but often adequate solution is to obfuscate the password and scatter the de-obfuscation material around the system so that an attacker has to obtain and correctly combine multiple system resources to decipher the password.

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

### References

1. Fortify Taxonomy | Password Management: Password in Configuration File, <https://vulncat.fortify.com/en/detail?category=Password+Management&subcategory=Password+in+Configuration+File>

## Instances

### Password Management: Password in Configuration File

High

Package: N/A

Instance	Analysis Info	Analyzer
ID 46447446 - src/main/resources/application.properties:29	<b>Sink:</b> spring.datasource.password in src/main/resources/application.properties:29 <b>EnclosingMethod:</b>	configur...
ID 46447466 - src/main/resources/application.properties:9	<b>Sink:</b> recaptcha.secret in src/main/resources/application.properties:9 <b>EnclosingMethod:</b>	configur...
ID 46447445 - target/classes/application.properties:29	<b>Sink:</b> spring.datasource.password in target/classes/application.properties:29 <b>EnclosingMethod:</b>	configur...
ID 46447467 - target/classes/application.properties:9	<b>Sink:</b> recaptcha.secret in target/classes/application.properties:9 <b>EnclosingMethod:</b>	configur...

## 7.2.3 Privacy Violation: Autocomplete

High

CWE-525

OWASP Top 10 2017: A3 - Sensitive Data Exposure

OWASP Top 10 2021: A04 - Insecure Design

PCI 3.2: 3.2 Do not store sensitive authentication data after authorization

### Summary

The form in **login2.html** uses autocomplete on line **16**, which allows some browsers to retain sensitive information in their history. Autocompletion of forms allows some browsers to retain sensitive information in their history.

### Explanation

With autocomplete enabled, some browsers retain user input across sessions, which could allow someone using the computer after the initial user to see information previously submitted.

### Recommendation

Explicitly disable autocomplete on forms or sensitive inputs. By disabling autocomplete, information previously entered will not be presented back to the user as they type. It will also disable the "remember my password" functionality of most major browsers.

**Example 1:** In an HTML form, disable autocomplete for all input fields by explicitly setting the value of the `autocomplete` attribute to `off` on the `form` tag.

```
<form method="post" autocomplete="off">
Address: <input name="address" />
Password: <input name="password" type="password" />
</form>
```

**Example 2:** Alternatively, disable autocomplete for specific input fields by explicitly setting the value of the `autocomplete` attribute to `off` on the corresponding tags.

```
<form method="post">
Address: <input name="address" />
Password: <input name="password" type="password" autocomplete="off"/>
</form>
```

Note that the default value of the `autocomplete` attribute is `on`. Therefore do not omit the attribute when dealing with sensitive inputs.

### References

1. Turn off autocomplete for credit card input, Pete Freitag, <http://www.petefreitag.com/item/481.cfm>
2. Fortify Taxonomy | Privacy Violation: Autocomplete, <https://vulncat.fortify.com/en/detail?category=Privacy+Violation&subcategory=Autocomplete>

## Instances

Privacy Violation: Autocomplete

High

Package: N/A

Instance	Analysis Info	Analyzer
ID 46447459 - src/main/resources/templates/login2.html:16	<b>Sink:</b> in src/main/resources/templates/login2.html:16 <b>EnclosingMethod:</b>	content
ID 46447455 - src/main/resources/webapp/WEB-INF/jsp/login.jsp:17	<b>Sink:</b> in src/main/resources/webapp/WEB-INF/jsp/login.jsp:17 <b>EnclosingMethod:</b>	content
ID 46447456 - src/main/webapp/WEB-INF/jsp/login.jsp:17	<b>Sink:</b> in src/main/webapp/WEB-INF/jsp/login.jsp:17 <b>EnclosingMethod:</b>	content
ID 46447457 - target/classes/templates/login2.html:16	<b>Sink:</b> in target/classes/templates/login2.html:16 <b>EnclosingMethod:</b>	content
ID 46447458 - target/classes/webapp/WEB-INF/jsp/login.jsp:17	<b>Sink:</b> in target/classes/webapp/WEB-INF/jsp/login.jsp:17 <b>EnclosingMethod:</b>	content

## 7.2.4 Spring Boot Misconfiguration: DevTools Enabled

High

OWASP Top 10 2017: A6 - Security Misconfiguration  
OWASP Top 10 2021: A05 - Security Misconfiguration  
PCI 3.2:

### Summary

The Spring Boot application is configured in developer mode.

### Explanation

The Spring Boot application has DevTools enabled. DevTools include an additional set of tools which can make the application development experience a little more pleasant, but DevTools are not recommended to use on applications in a production environment. As stated in the official Spring Boot documentation: "Enabling `spring-boot-devtools` on a remote application is a security risk. You should never enable support on a production deployment."

### Recommendation

Remove `spring-boot-devtools` dependency on production deployments.

### References

1. Spring Boot Reference Guide, <https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/>
2. Fortify Taxonomy | Spring Boot Misconfiguration: DevTools Enabled, <https://vulncat.fortify.com/en/detail?category=Spring+Boot+Misconfiguration&subcategory=DevTools+Enabled>

## Instances

Spring Boot Misconfiguration: DevTools Enabled

High

Package: N/A

Instance	Analysis Info	Analyzer
ID 46447460 - pom.xml:67	<b>Sink:</b> in pom.xml:67 <b>EnclosingMethod:</b>	configur...
ID 46447461 - target/classes/META-INF/maven/com.check/jstlthyme/pom.xml:67	<b>Sink:</b> in target/classes/META-INF/maven/com.check/jstlthyme/pom.xml:67 <b>EnclosingMethod:</b>	configur...

## 7.2.5 Unreleased Resource: Streams

High

CWE-772

OWASP Top 10 2017:

OWASP Top 10 2021:

PCI 3.2: 6.5.6 High Risk Vulnerabilities

### Summary

The function **ssrfTest()** in **BoardController.java** sometimes fails to release a system resource allocated by **getInputStream()** on line **203**. The program can potentially fail to release a system resource.

### Explanation

The program can potentially fail to release a system resource. In this case, there are program paths on which the resource allocated in **BoardController.java** on line **203** is not released.

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

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

### Recommendation

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.

## References

1. FIO04-J. Release resources when they are no longer needed, <https://www.securecoding.cert.org/confluence/display/java/FIO04-J.+Release+resources+when+they+are+no+longer+needed>
2. DOS-2: Release resources in all cases, <http://www.oracle.com/technetwork/java/seccodeguide-139067.html#1>
3. Fortify Taxonomy | Unreleased Resource: Streams, <https://vulncat.fortify.com/en/detail?category=Unreleased+Resource&subcategory=Streams>



## Instances

Unreleased Resource: Streams

High

Package: com.check.thyme.controller

Instance	Analysis Info	Analyzer
ID 46447483 - src/main/java/com/check/thyme/controller/BoardController.java:203	<b>Sink:</b> inputStream = getInputStream() in src/main/java/com/check/thyme/controller/BoardController.java:203 <b>EnclosingMethod:</b> ssrfTest	controlfl...
ID 46447482 - src/main/java/com/check/thyme/controller/BoardController2.java:204	<b>Sink:</b> inputStream = getInputStream() in src/main/java/com/check/thyme/controller/BoardController2.java:204 <b>EnclosingMethod:</b> ssrfTest	controlfl...

### 7.3.1 Cross-Site Request Forgery

Low

CWE-352

OWASP Top 10 2017:

OWASP Top 10 2021: A01 - Broken Access Control

PCI 3.2: 6.5.9 Cross-Site Request Forgery

#### Summary

The HTTP request at **board-list.html** line **255** must contain a user-specific secret to prevent an attacker from making unauthorized requests. HTTP requests must contain a user-specific secret to prevent an attacker from making unauthorized requests.

#### Explanation

A cross-site request forgery (CSRF) vulnerability occurs when: 1. A web application uses session cookies.

2. The application acts on an HTTP request without verifying that the request was made with the user's consent.

In this case, the application generates an HTTP request at **board-list.html** line **255**.

A nonce is a cryptographic random value that is sent with a message to prevent replay attacks. If the request does not contain a nonce that proves its provenance, the code that handles the request is vulnerable to a CSRF attack (unless it does not change the state of the application). This means a web application that uses session cookies has to take special precautions to ensure that an attacker can't trick users into submitting bogus requests. Imagine a web application that allows administrators to create new accounts as follows:

```
var req = new XMLHttpRequest();
req.open("POST", "/new_user", true);
body = addToPost(body, new_username);
body = addToPost(body, new_passwd);
req.send(body);
```

An attacker might set up a malicious web site that contains the following code.

```
var req = new XMLHttpRequest();
req.open("POST", "http://www.example.com/new_user", true);
body = addToPost(body, "attacker");
body = addToPost(body, "haha");
req.send(body);
```

If an administrator for **example.com** visits the malicious page while she has an active session on the site, she will unwittingly create an account for the attacker. This is a CSRF attack. It is possible because the application does not have a way to determine the provenance of the request. Any request could be a legitimate action chosen by the user or a faked action set up by an attacker. The attacker does not get to see the Web page that the bogus request generates, so the attack technique is only useful for requests that alter the state of the application.

Applications that pass the session identifier in the URL rather than as a cookie do not have CSRF problems because there is no way for the attacker to access the session identifier and include it as part of the bogus request.

## Execution

1. Fortify Static Code Analyzer flags all HTML forms and all XMLHttpRequest objects that might perform either a GET or POST operation. The auditor must determine if each form is valuable to an attacker as a CSRF target and whether or not an appropriate mitigation technique is in place.

## Recommendation

Applications that use session cookies must include some piece of information in every form post that the back-end code can use to validate the provenance of the request. One way to do that is to include a random request identifier or nonce, as follows:

```
RequestBuilder rb = new RequestBuilder(RequestBuilder.POST, "/new_user");
body = addToPost(body, new_username);
body = addToPost(body, new_passwd);
body = addToPost(body, request_id);
rb.sendRequest(body, new NewAccountCallback(callback));
```

Then the back-end logic can validate the request identifier before processing the rest of the form data. When possible, the request identifier should be unique to each server request rather than shared across every request for a particular session. As with session identifiers, the harder it is for an attacker to guess the request identifier, the harder it is to conduct a successful CSRF attack. The token should not be easily guessed and it should be protected in the same way that session tokens are protected, such as using SSLv3.

Additional mitigation techniques include:

**Framework protection:** Most modern web application frameworks embed CSRF protection and they will automatically include and verify CSRF tokens.

**Use a Challenge-Response control:** Forcing the customer to respond to a challenge sent by the server is a strong defense against CSRF. Some of the challenges that can be used for this purpose are: CAPTCHAs, password re-authentication and one-time tokens.

**Check HTTP Referer/Origin headers:** An attacker won't be able to spoof these headers while performing a CSRF attack. This makes these headers a useful method to prevent CSRF attacks.

**Double-submit Session Cookie:** Sending the session ID Cookie as a hidden form value in addition to the actual session ID Cookie is a good protection against CSRF attacks. The server will check both values and make sure they are identical before processing the rest of the form data. If an attacker submits a form in behalf of a user, he won't be able to modify the session ID cookie value as per the same-origin-policy.

**Limit Session Lifetime:** When accessing protected resources using a CSRF attack, the attack will only be valid as long as the session ID sent as part of the attack is still valid on the server. Limiting the Session lifetime will reduce the probability of a successful attack.

The techniques described here can be defeated with XSS attacks. Effective CSRF mitigation includes XSS mitigation techniques.

## References

1. Divide and Conquer: HTTP Response Splitting, Web Cache Poisoning Attacks, and Related Topics, A. Klein,  
[http://www.packetstormsecurity.org/papers/general/whitepaper\\_httpresponse.pdf](http://www.packetstormsecurity.org/papers/general/whitepaper_httpresponse.pdf)
2. Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet, OWASP,  
[https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site\\_Request\\_Forgery\\_Prevention\\_Cheat\\_Sheet.html](https://cheatsheetseries.owasp.org/cheatsheets/Cross-Site_Request_Forgery_Prevention_Cheat_Sheet.html)
3. Fortify Taxonomy | Cross-Site Request Forgery, <https://vulncat.fortify.com/en/detail?category=Cross-Site+Request+Forgery>

## Instances

### Cross-Site Request Forgery

Low

Package: N/A

Instance	Analysis Info	Analyzer
ID 46447468 - src/main/resources/templates/board-list.html:255	<b>Sink:</b> AssignmentStatement in src/main/resources/templates/board-list.html:255 <b>EnclosingMethod:</b> lambda	structural
ID 46447469 - src/main/resources/templates/board-list.html:298	<b>Sink:</b> AssignmentStatement in src/main/resources/templates/board-list.html:298 <b>EnclosingMethod:</b> lambda	structural
ID 46447487 - src/main/resources/templates/board-list.html:151	<b>Sink:</b> in src/main/resources/templates/board-list.html:151 <b>EnclosingMethod:</b>	content
ID 46447488 - src/main/resources/templates/board-list.html:108	<b>Sink:</b> in src/main/resources/templates/board-list.html:108 <b>EnclosingMethod:</b>	content
ID 46447489 - src/main/resources/templates/board-list.html:20	<b>Sink:</b> in src/main/resources/templates/board-list.html:20 <b>EnclosingMethod:</b>	content
ID 46447472 - src/main/resources/templates/board-list2.html:298	<b>Sink:</b> AssignmentStatement in src/main/resources/templates/board-list2.html:298 <b>EnclosingMethod:</b> lambda	structural
ID 46447473 - src/main/resources/templates/board-list2.html:255	<b>Sink:</b> AssignmentStatement in src/main/resources/templates/board-list2.html:255 <b>EnclosingMethod:</b> lambda	structural
ID 46447476 - src/main/resources/templates/board-list2.html:20	<b>Sink:</b> in src/main/resources/templates/board-list2.html:20 <b>EnclosingMethod:</b>	content
ID 46447480 - src/main/resources/templates/board-list2.html:151	<b>Sink:</b> in src/main/resources/templates/board-list2.html:151 <b>EnclosingMethod:</b>	content
ID 46447481 - src/main/resources/templates/board-list2.html:108	<b>Sink:</b> in src/main/resources/templates/board-list2.html:108 <b>EnclosingMethod:</b>	content
ID 46447465 - src/main/resources/templates/index.html:36	<b>Sink:</b> FunctionPointerCall: get in src/main/resources/templates/index.html:36 <b>EnclosingMethod:</b> lambda	structural
ID 46447449 - src/main/resources/templates/login2.html:11	<b>Sink:</b> in src/main/resources/templates/login2.html:11 <b>EnclosingMethod:</b>	content
ID 46447452 - src/main/resources/webapp/WEB-INF/jsp/board.jsp:11	<b>Sink:</b> in src/main/resources/webapp/WEB-INF/jsp/board.jsp:11 <b>EnclosingMethod:</b>	content
ID 46447447 - src/main/resources/webapp/WEB-INF/jsp/login.jsp:12	<b>Sink:</b> in src/main/resources/webapp/WEB-INF/jsp/login.jsp:12 <b>EnclosingMethod:</b>	content
ID 46447450 - src/main/webapp/WEB-INF/jsp/board.jsp:11	<b>Sink:</b> in src/main/webapp/WEB-INF/jsp/board.jsp:11 <b>EnclosingMethod:</b>	content
ID 46447454 - src/main/webapp/WEB-INF/jsp/login.jsp:12	<b>Sink:</b> in src/main/webapp/WEB-INF/jsp/login.jsp:12 <b>EnclosingMethod:</b>	content
ID 46447470 - target/classes/templates/board-list.html:298	<b>Sink:</b> AssignmentStatement in target/classes/templates/board-list.html:298 <b>EnclosingMethod:</b> lambda	structural
ID 46447471 - target/classes/templates/board-list.html:255	<b>Sink:</b> AssignmentStatement in target/classes/templates/board-list.html:255 <b>EnclosingMethod:</b> lambda	structural
ID 46447484 - target/classes/templates/board-list.html:151	<b>Sink:</b> in target/classes/templates/board-list.html:151 <b>EnclosingMethod:</b>	content

Package: N/A		
Instance	Analysis Info	Analyzer
ID 46447485 - target/classes/templates/board-list.html:108	<b>Sink:</b> in target/classes/templates/board-list.html:108 <b>EnclosingMethod:</b>	content
ID 46447486 - target/classes/templates/board-list.html:20	<b>Sink:</b> in target/classes/templates/board-list.html:20 <b>EnclosingMethod:</b>	content
ID 46447474 - target/classes/templates/board-list2.html:298	<b>Sink:</b> AssignmentStatement in target/classes/templates/board-list2.html:298 <b>EnclosingMethod:</b> lambda	structural
ID 46447475 - target/classes/templates/board-list2.html:255	<b>Sink:</b> AssignmentStatement in target/classes/templates/board-list2.html:255 <b>EnclosingMethod:</b> lambda	structural
ID 46447477 - target/classes/templates/board-list2.html:151	<b>Sink:</b> in target/classes/templates/board-list2.html:151 <b>EnclosingMethod:</b>	content
ID 46447478 - target/classes/templates/board-list2.html:108	<b>Sink:</b> in target/classes/templates/board-list2.html:108 <b>EnclosingMethod:</b>	content
ID 46447479 - target/classes/templates/board-list2.html:20	<b>Sink:</b> in target/classes/templates/board-list2.html:20 <b>EnclosingMethod:</b>	content
ID 46447464 - target/classes/templates/index.html:36	<b>Sink:</b> FunctionPointerCall: get in target/classes/templates/index.html:36 <b>EnclosingMethod:</b> lambda	structural
ID 46447453 - target/classes/templates/login2.html:11	<b>Sink:</b> in target/classes/templates/login2.html:11 <b>EnclosingMethod:</b>	content
ID 46447451 - target/classes/webapp/WEB-INF/jsp/board.jsp:11	<b>Sink:</b> in target/classes/webapp/WEB-INF/jsp/board.jsp:11 <b>EnclosingMethod:</b>	content
ID 46447448 - target/classes/webapp/WEB-INF/jsp/login.jsp:12	<b>Sink:</b> in target/classes/webapp/WEB-INF/jsp/login.jsp:12 <b>EnclosingMethod:</b>	content

## Static File Listing

The static file listing displays all files scanned by the SCA scanner.

Filename	Size (bytes)	Modified Date
.mvn/wrapper/maven-wrapper.properties	951	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	842	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	21409	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	21410	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	1140	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	1582	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	848	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	365	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	838	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	21405	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	21406	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	1136	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	1578	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	844	2025/02/13
D:/work/work/sca24.4/build/366430-229565-Other_7400...	361	2025/02/13
HELP.md	1942	2025/02/13
pom.xml	3673	2025/02/13
src/main/java/com/check/thyme/controller/BoardController...	16312	2025/02/13
src/main/java/com/check/thyme/controller/BoardController...	16334	2025/02/13
src/main/java/com/check/thyme/controller/MemberControll...	6405	2025/02/13
src/main/java/com/check/thyme/controller/WebConfig.java	803	2025/02/13
src/main/java/com/check/thyme/JspController.java	1259	2025/02/13
src/main/java/com/check/thyme/JstlthymeApplication.java	316	2025/02/13
src/main/java/com/check/thyme/repository/BoardReposito...	448	2025/02/13
src/main/java/com/check/thyme/repository/MemberReposi...	296	2025/02/13
src/main/java/com/check/thyme/service/BoardSvc.java	377	2025/02/13
src/main/java/com/check/thyme/service/BoardSvcImpl.java	1028	2025/02/13
src/main/java/com/check/thyme/service/CaptchaValidation...	964	2025/02/13
src/main/java/com/check/thyme/service/FileDownloadServ....	1351	2025/02/13
src/main/java/com/check/thyme/service/LoginAttemptServi...	1721	2025/02/13
src/main/java/com/check/thyme/service/MemberSvc.java	260	2025/02/13
src/main/java/com/check/thyme/service/MemberSvcImpl.ja...	888	2025/02/13

Filename	Size (bytes)	Modified Date
src/main/java/com/check/thyme/thymeController.java	2053	2025/02/13
src/main/java/com/check/thyme/ViewResolverConfig.java	1797	2025/02/13
src/main/java/com/check/thyme/vo/BoardVo.java	1239	2025/02/13
src/main/java/com/check/thyme/vo/CaptchaResponse.java	1076	2025/02/13
src/main/java/com/check/thyme/vo/Member.java	569	2025/02/13
src/main/resources/application.properties	1636	2025/02/13
src/main/resources/keystore.jks	2746	2025/02/13
src/main/resources/mapper/BoardMapper.xml	1611	2025/02/13
src/main/resources/mapper/MemberMapper.xml	1272	2025/02/13
src/main/resources/templates/account-locked.html	832	2025/02/13
src/main/resources/templates/board-list.html	14451	2025/02/13
src/main/resources/templates/board-list2.html	14441	2025/02/13
src/main/resources/templates/index.html	1315	2025/02/13
src/main/resources/templates/login2.html	934	2025/02/13
src/main/resources/templates/thyme.html	557	2025/02/13
src/main/resources/templates/thyme2.html	229	2025/02/13
src/main/resources/webapp/WEB-INF/jsp/board.jsp	641	2025/02/13
src/main/resources/webapp/WEB-INF/jsp/fail.jsp	243	2025/02/13
src/main/resources/webapp/WEB-INF/jsp/login.jsp	744	2025/02/13
src/main/webapp/WEB-INF/jsp/board.jsp	640	2025/02/13
src/main/webapp/WEB-INF/jsp/fail.jsp	243	2025/02/13
src/main/webapp/WEB-INF/jsp/login.jsp	746	2025/02/13
src/test/java/com/check/thyme/JstlthymeApplicationTests.j...	210	2025/02/13
target/classes/application.properties	1636	2025/02/13
target/classes/keystore.jks	2746	2025/02/13
target/classes/mapper/BoardMapper.xml	1611	2025/02/13
target/classes/mapper/MemberMapper.xml	1272	2025/02/13
target/classes/META-INF/maven/com.check/jstlthyme/pom...	204	2025/02/13
target/classes/META-INF/maven/com.check/jstlthyme/pom...	3673	2025/02/13
target/classes/templates/account-locked.html	832	2025/02/13
target/classes/templates/board-list.html	14451	2025/02/13
target/classes/templates/board-list2.html	14441	2025/02/13
target/classes/templates/index.html	1315	2025/02/13
target/classes/templates/login2.html	934	2025/02/13
target/classes/templates/thyme.html	557	2025/02/13




























Filename	Size (bytes)	Modified Date
target/classes/templates/thyme2.html	229	2025/02/13
target/classes/webapp/WEB-INF/jsp/board.jsp	641	2025/02/13
target/classes/webapp/WEB-INF/jsp/fail.jsp	243	2025/02/13
target/classes/webapp/WEB-INF/jsp/login.jsp	744	2025/02/13

## Appendix - Descriptions of Key Terminology

### Security Rating

The Fortify on Demand 5-star assessment rating provides information on the likelihood and impact of defects present within an application. A perfect rating within this system would be 5 complete stars indicating that no high impact vulnerabilities were uncovered.

Rating	
    	Fortify on Demand awards one star to applications that have undergone a security review that identifies critical (high likelihood and high impact) issues.
    	Fortify on Demand awards two stars to applications that have undergone a security review that identifies no critical (high likelihood and high impact) issues. Vulnerabilities that are trivial to exploit and have a high business or technical impact should never exist in business-critical software.
    	Fortify on Demand awards three stars to applications that have undergone a security review that identifies no high (low likelihood and high impact) issues and meets the requirements needed to receive two stars. Vulnerabilities that have a high impact, even if they are non-trivial to exploit, should never exist in business critical software.
    	Fortify on Demand awards four stars to applications that have undergone a security review that identifies no medium (high likelihood and low impact) issues and meets the requirements for three stars. Vulnerabilities that have a low impact, but are easy to exploit, should be considered carefully as they may pose a greater threat if an attacker exploits many of them as part of a concerted effort or leverages a low impact vulnerability as a stepping stone to mount a high-impact attack.
    	Fortify on Demand awards five stars, the highest rating, to applications that have undergone a security review that identifies no issues.

### Likelihood and Impact

#### Likelihood

Likelihood is the probability that a vulnerability will be accurately identified and successfully exploited.

#### Impact

Impact is the potential damage an attacker could do to assets by successfully exploiting a vulnerability. This damage can be in the form of, but not limited to, financial loss, compliance violation, loss of brand reputation, and negative publicity.

### Fortify on Demand Priority Order

#### Critical

Critical-priority issues have high impact and high likelihood. Critical-priority issues are easy to detect and exploit and result in large asset damage. These issues represent the highest security risk to the application. As such, they should be remediated immediately.

SQL Injection is an example of a critical issue.

## High

High-priority issues have high impact and low likelihood. High-priority issues are often difficult to detect and exploit, but can result in large asset damage. These issues represent a high security risk to the application. High priority issues should be remediated in the next scheduled patch release.

Password Management: Hardcoded Password is an example of a high issue.

## Medium

Medium-priority issues have low impact and high likelihood. Medium-priority issues are easy to detect and exploit, but typically result in small asset damage. These issues represent a moderate security risk to the application. Medium-priority issues should be remediated in the next scheduled product.

Path Manipulation is an example of a medium issue.

## Low

Low-priority issues have low impact and low likelihood. Low-priority issues can be difficult to detect and exploit and typically result in small asset damage. These issues represent a minor security risk to the application. Low priority issues should be remediated as time allows.

Dead Code is an example of a low issue.

## Issue Status

### New

New issues are ones that have been identified for the first time in the most recent analysis of the application.

### Existing

Existing issues are issues that have been found in a previous analysis of the application and are still present in the latest analysis.

### Reopened

Reopened issues have been discovered in a previous analysis of the application but were not present in subsequent analyses. These issues are now present again in the most recent analysis of the application.