LOG8371E - Lab
Software Quality Engineering

TP3: Security

Svante Sundberg | 2229625
Clara Serruau | 2164678
Amena Bekteshi | 1881178
Seong Choi | 2239053
Lydia Haile | 2230615
César Miguel Valdez Córdova | 2229636

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Submitted December 1, 2022

Table of Contents

List of Figures

1. Introduction

Automation, in no lesser part enabled by the accelerated and unyielding growth of the software industry, is set to revolutionize all industry sectors as we know them, shifting the division of labor and associated responsibilities in unprecedented scale and organization. [1] Multiple tools are currently being employed to meet this end. Databases often function as the main record keeping component of any software based system; crucial information to the proper functioning of any and all processes is contained therein. Assuring the quality and integrity of all data, and the proper communication of their content and associated states to the authorities responsible for seeing a process to an end is at the core of a successful automated process, driven by software. Realizing the importance of said module within a system, the Mango database was chosen as a subject of analysis; specifically, through its mailing list functionality. The latter component was chosen due to the perceived importance and often, urgency, of the process between assuring and verifying data integrity and further transmitting it and communicating it in a timely manner to the overseer of any given process. This document will analyze said component through the lens of two quality assurance characteristics: Reliability and Maintainability, as these two are deemed to go hand-in-hand in their responsibility of granting a database user adoption and longevity within a process, respectively. Similarly, the approaches for their analysis are grounded in the process coupling literature, which will be elaborated on alongside the aforementioned components.

2. Static analysis

2.1 Manual for Static Analysis

The static analysis of the Mango system is done using SonarCloud with SonarScanner. This subsection describes how the tools were configured to perform the analysis.

Prerequisites: In order to follow this configuration manual a GitHub account is required. Furthermore, before starting the configuration, the mango source code needs to be uploaded to a GitHub repository. This configuration is made for macOS.

1. Login to SonarCloud. Choose to login using GitHub. You will be redirected to GitHub where you will be required to login using your GitHub credentials and to authorize that SonarCloud has the permission to your GitHub repositories.
2. When the login procedure is done, you will be redirected back to SonarCloud. Click on the SonarCloud logo (upper-left corner), to get to the welcome screen.
3. On the welcome-screen, click on the button “Import an organization from GitHub” under the heading “Analyze your first projects”.
4. You will be redirected to GitHub once again. This time choose the option “Only select repositories” and in the curtain menu choose the git repository that holds the mango source code which you wish to analyze. At the bottom of the page click the button “Install”. The web-browser should jump back to SonarCloud again.
5. In the field “key” enter a key of your choosing that follows the requirements stated underneath the field (*ex: randomkey*). Next, click the button “Continue”.
6. Choose the free plan and click “Create Organization”.

7. Select the git repository that contains the mango source code and click on “Set Up”.
8. Choose the analysis method “Manually”.
9. Choose “Other (for JS, TS, Go, Python, PHP)”.
10. Click “MacOS”.
11. Download the zipped-file. Unzip the file. To the “.bash_profile” file on your computer, add the path to the bin directory in the now unzipped file to the PATH environment variable.
12. To the “.bash_profile” file add an environmental variable called “SONAR_TOKEN” and add the value stated on SonarCloud to said variable.
13. Navigate to your project folder in the terminal and run the command stated on SonarCloud with one small modification. Add the line “-Dsonar.java.binaries=src” to the command.

Once the steps above have been performed, the page in SonarCloud should refresh automatically and the analysis of the mango source code executed.

2.2 Summary Static Analysis.

The overview page of the analysis of the mango source code in SonarCloud reveals:

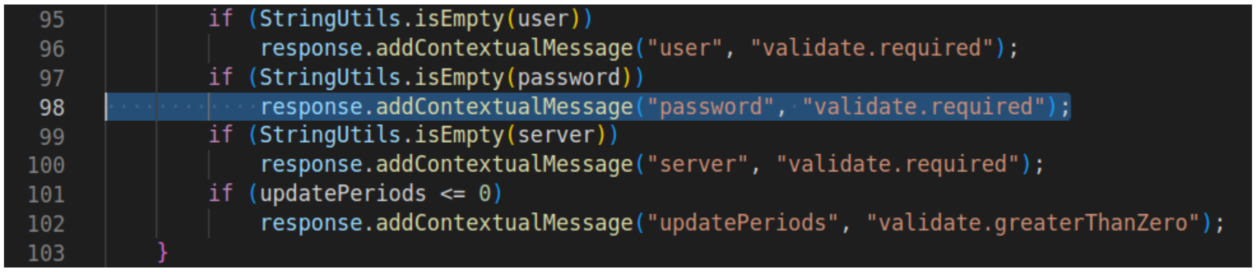
- 3,600 bugs
- 4,500 code smells
- 2 vulnerabilities
- 47 security hotspots

The analysis also reports that the project holds 168,000 lines of code, and includes the languages Java, HTML, JSP, CSS, XML, PL/SQL, PHP, Flex and Python in varying degrees.

2.3 Identified Vulnerabilities

Vulnerabilities, security hotspots, code smells and bugs detected during the static analysis with SonarCloud are brought forward and discussed. The severity or review priority of the issues are identified (example of severity: Blocker/Critical/Major and example of review priority: High/Medium/Low), and a short description of the issues along with the potential risks they might cause will be presented. The Common Weakness Enumeration Specification (CWE) type of the issue will also be listed, where applicable. Furthermore, suggestions for corrections to all issues are described.

2.3.1 **Security Hotspot**: Authentification



```
        if (StringUtils.isEmpty(user))
            response.addContextualMessage("user", "validate.required");
        if (StringUtils.isEmpty(password))
            response.addContextualMessage("password", "validate.required");
        if (StringUtils.isEmpty(server))
            response.addContextualMessage("server", "validate.required");
        if (updatePeriods <= 0)
            response.addContextualMessage("updatePeriods", "validate.greaterThanZero");
    }
```

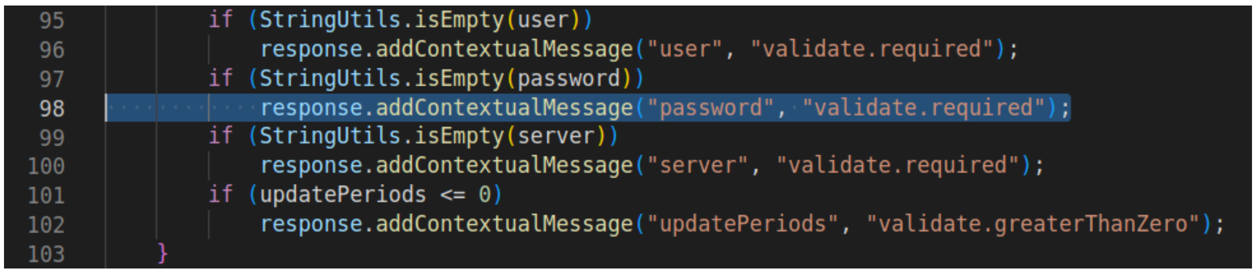
Figure 1: Authentication code found in *OPCDataSourceVO.java*

Found in: *src/br/org/scadabr/vo/dataSource/opc/OPCDataSourceVO.java*
Review Priority: **High**
CVE (listed by SonarCloud) and associated CWE (see reference):

- CVE-2019-13466 [1]
  - CWE-798 Use of Hard-coded Credentials
- CVE-2018-15389 [2]
  - CWE-798 Use of Hard-coded Credentials
  - CWE-255 Credentials Management Errors

Description and potential risk: Extracting strings from an application source code or binary is a trivial task; passwords should not be hard-coded. This is particularly true for applications that are distributed or that are open-source. Passwords should be stored outside of the code in a configuration file, a database, or a password management service. This rule flags instances of hard-coded passwords used in database and LDAP connections. It looks for hard-coded passwords in connection strings, and for variable names that match any of the patterns from the provided list.

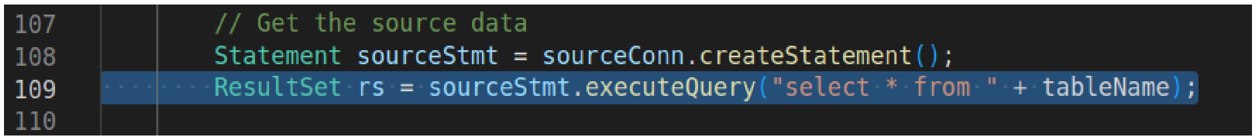
Solution: SonarCloud is detecting a contextual message as a security issue. Line 98 is in fact not an issue. The password is not hardcoded. The line is only stating that a non-empty password is required. This does not require a fix.



```
 95         if (StringUtils.isEmpty(user))
 96             response.addContextualMessage("user", "validate.required");
 97         if (StringUtils.isEmpty(password))
 98             response.addContextualMessage("password", "validate.required");
 99         if (StringUtils.isEmpty(server))
100             response.addContextualMessage("server", "validate.required");
101         if (updatePeriods <= 0)
102             response.addContextualMessage("updatePeriods", "validate.greaterThanZero");
103     }
```

Figure 2: Contextual message as a security issue code

2.3.2 **Security Hotspot**: SQL injection

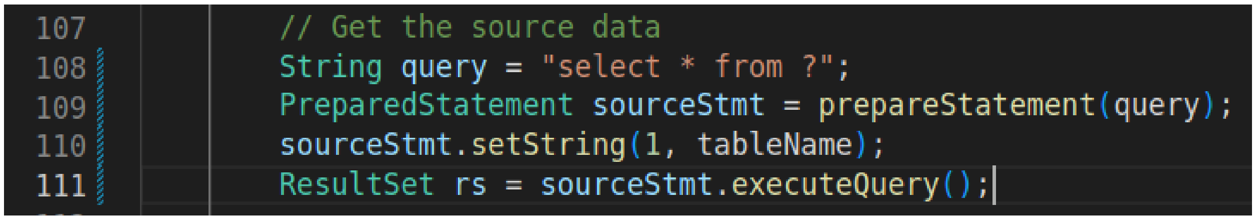


```
        // Get the source data
        Statement sourceStmt = sourceConn.createStatement();
        ResultSet rs = sourceStmt.executeQuery("select * from " + tableName);

```

Figure 3: SQL injection found in DBConvert.java

Found in: *src/com/serotonin/mango/db/DBConvert.java*
Review priority: **High**
CVE: Non listed in SonarCloud
Description and potential risk:
Formatted SQL queries can be difficult to maintain, debug and can increase the risk of SQL injection when concatenating untrusted values into the query. However, this rule doesn't detect SQL injections (unlike rule {rule:javasecurity:S3649}), the goal is only to highlight complex/formatted queries.

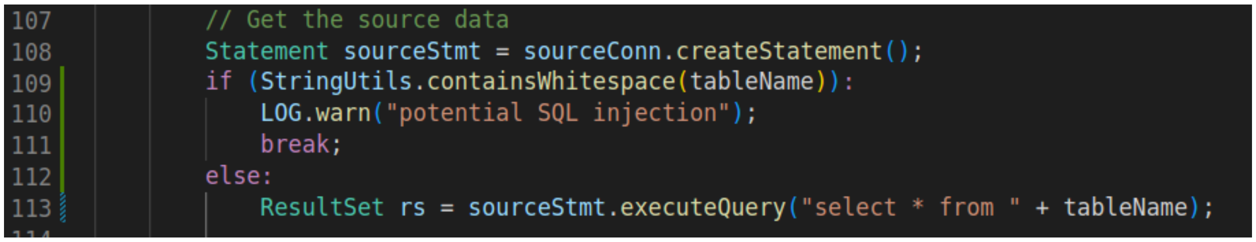
Solution: When looking at how to fix SQL injection, solutions that used parameterized queries in order to reduce the risk of sql injection were found [9].



```
// Get the source data
String query = "select * from ?";
PreparedStatement sourceStmt = prepareStatement(query);
sourceStmt.setString(1, tableName);
ResultSet rs = sourceStmt.executeQuery();
```

Figure 4: Formatted SQL queries

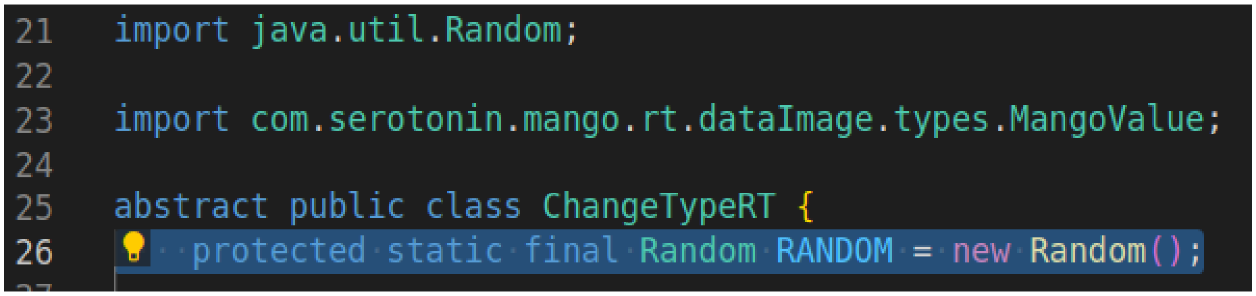
This would solve the issues in SonarCloud, but would break the code as it is not the correct syntax for using PreparedStatement, as it contains a variable table name [10].
A whitespace check in tableName could prevent an SQL injection:



```
// Get the source data
Statement sourceStmt = sourceConn.createStatement();
if (StringUtils.containsWhitespace(tableName)):
    LOG.warn("potential SQL injection");
    break;
else:
    ResultSet rs = sourceStmt.executeQuery("select * from " + tableName);
```

Figure 5: Formatted SQL queries

2.3.3 **Security Hotspot**: Weak Cryptography



```
import java.util.Random;

import com.serotonin.mango.rt.dataImage.types.MangoValue;

abstract public class ChangeTypeRT {
    protected static final Random RANDOM = new Random();
```

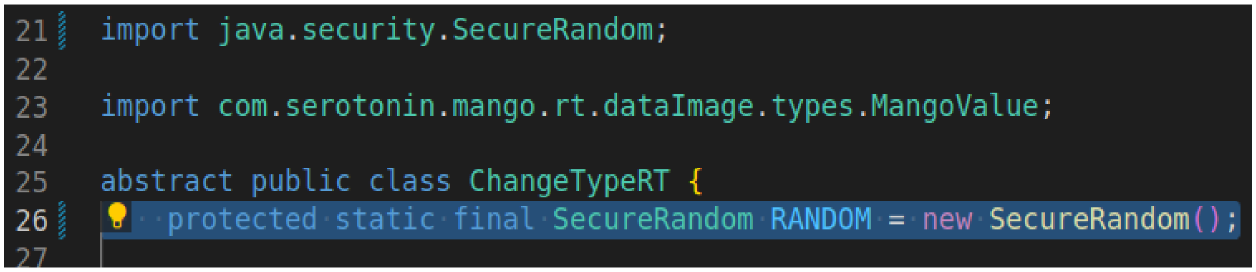
Figure 6: Weak cryptography found in ChangeTypeRT.java

Found in: *src/com/serotonin/mango/rt/dataSource/virtual/ChangeTypeRT.java*
Review priority: **Medium**
CVE (listed by SonarCloud) and associated CWE (see reference):

- CVE-2013-6386 [3]
  - CWE-310 Cryptographic Issues
- CVE-2006-3419 [4]
  - NVD-CWE-Other Other
- CVE-2008-4102 [5]
  - CWE-189 Numeric Errors

Description and potential risk: Using pseudorandom number generators (PRNGs) is security-sensitive. When software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information.

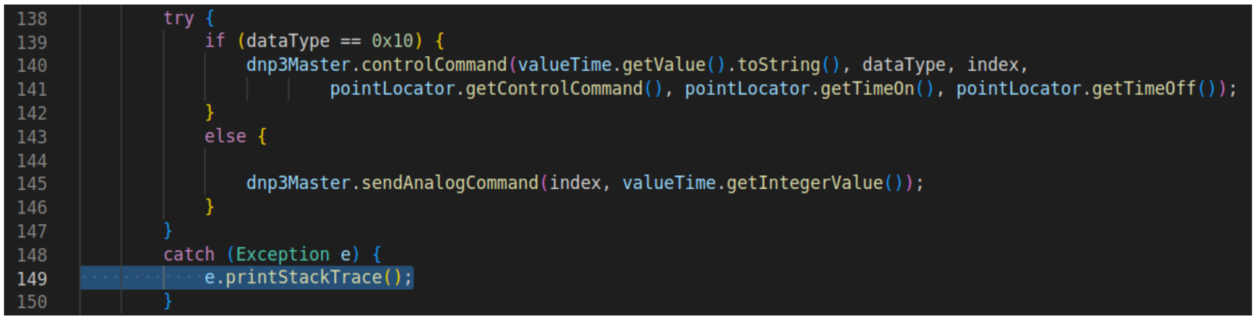
Solution: directly provided as a tip from SonarCloud and suggests to use java.security.SecureRandom instead of java.util.Random. This is because security-related applications should not rely on "pseudorandom number generator", but on "cryptographically strong random number generator".



```java
import java.security.SecureRandom;

import com.serotonin.mango.rt.dataImage.types.MangoValue;

abstract public class ChangeTypeRT {
    protected static final SecureRandom RANDOM = new SecureRandom();

```

Figure 7: Solution with java.security.SecureRandom

2.3.4 **Security Hotspot**: Insecure Configuration



```java
        try {
            if (dataType == 0x10) {
                dnp3Master.controlCommand(valueTime.getValue().toString(), dataType, index,
                        pointLocator.getControlCommand(), pointLocator.getTimeOn(), pointLocator.getTimeOff());
            }
            else {

                dnp3Master.sendAnalogCommand(index, valueTime.getIntegerValue());
            }
        }
        catch (Exception e) {
            e.printStackTrace();
        }
```

Figure 8: Insecure configuration found in Dnp3DataSource.java

Found in: *src/br/org/scadabr/rt/dataSource/dnp3/Dnp3DataSource.java*

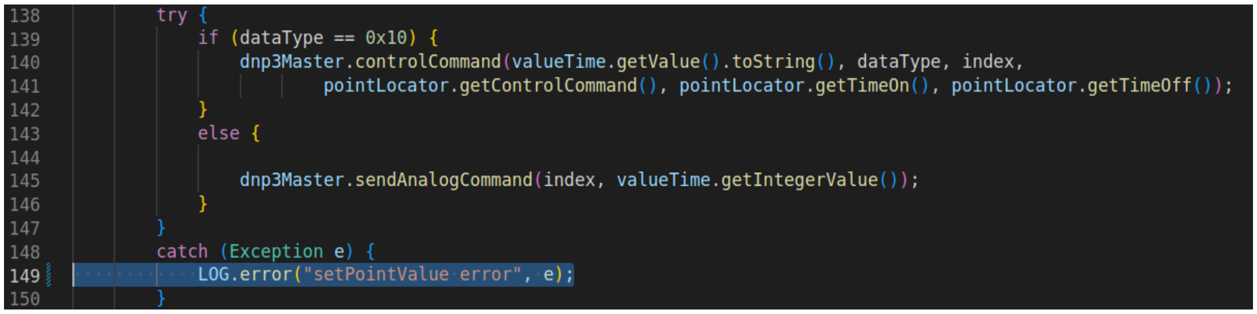
Review priority: **Low**

CVE (listed by SonarCloud) and associated CWE (see reference):

- CVE-2018-1999007 [6]
  - CWE-79 Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')
- CVE-2015-5306 [7]
  - CWE-254 7PK - Security Features
- CVE-2013-2006 [8]
  - CWE-200 Exposure of Sensitive Information to an Unauthorized Actor

Description and potential risk: Delivering code in production with debug features activated is security-sensitive. An application's debug features enable developers to find bugs more easily and thus also facilitate the work of attackers. It often gives access to detailed information on both the system running the application and users.

Solution: Instead of using a debug feature, we will log the error and its position in the execution flow. This keeps the information about the exception that happened in the log, which is not as readily available to the user.

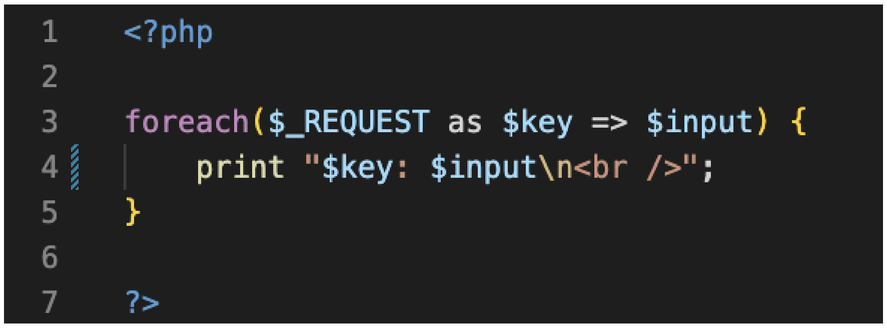


```
        try {
            if (dataType == 0x10) {
                dnp3Master.controlCommand(valueTime.getValue().toString(), dataType, index,
                        pointLocator.getControlCommand(), pointLocator.getTimeOn(), pointLocator.getTimeOff());
            }
            else {

                dnp3Master.sendAnalogCommand(index, valueTime.getIntegerValue());
            }
        }
        catch (Exception e) {
            LOG.error("setPointValue error", e);
        }
```

Figure 9: Proposed solution with error logging

2.3.5 **Security Vulnerability**: Use of User-Controlled Data

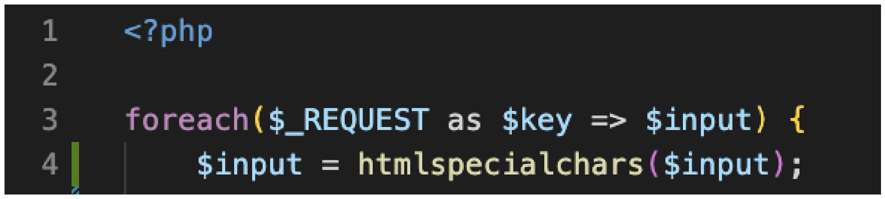
Found in: *build/resources/dojo/tests/widget/showPost.php*
What is the severity type of the vulnerability? **Blocker**.
What is the CWE type of the vulnerability? CWE-79 – Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting').
Description & Potential Risk. SonarCloud marks the code presented in Figure 10 as buggy, since it treats user-input without sanitizing it. Normally this would subject the system to potential different types of injection attacks such as Cross-site Scripting (XSS) attacks, in this instance however, there is no risk for injection attacks. This is because the text is only printed to the screen (to the terminal/command prompt with php print-method), which is a point of weakness in any system.



```
<?php

foreach($_REQUEST as $key => $input) {
    print "$key: $input\n<br />";
}

?>
```

Figure 10: Code found in file *showPost.php.*

Solution & Potential Risk. The example flagged by SonarCloud (see Figure 10) was concluded to be a false positive. However, if one were to try to post the input to a website instead of printing the input to the screen one would need to sanitize the input first. This could look as presented in Figure 11. On line four in said figure, the php function *htmlspecialchars()* is called to strip the input of any characters that would turn the input into a html script and by doing so rendering it harmless to post on a website.



```php
<?php

foreach($_REQUEST as $key => $input) {
    $input = htmlspecialchars($input);
```

Figure 11: Suggested alterations to the code in the file *showPost.php*.

2.3.6 **Software Bug**: Jump statements should not occur in "finally" blocks.

Found in: src/com/serotonin/mango/rt/dataSource/sql/SqlDataSourceRT.java
What is the severity type of the vulnerability? **Critical**.
What is the CWE type of the vulnerability? The primary CWE type is CWE-584 – Return Inside a Finally Block. However, since our bug could also cause exceptions to go unhandled a secondary CWE type is identified to be CWE-248 – Uncaught Exception.

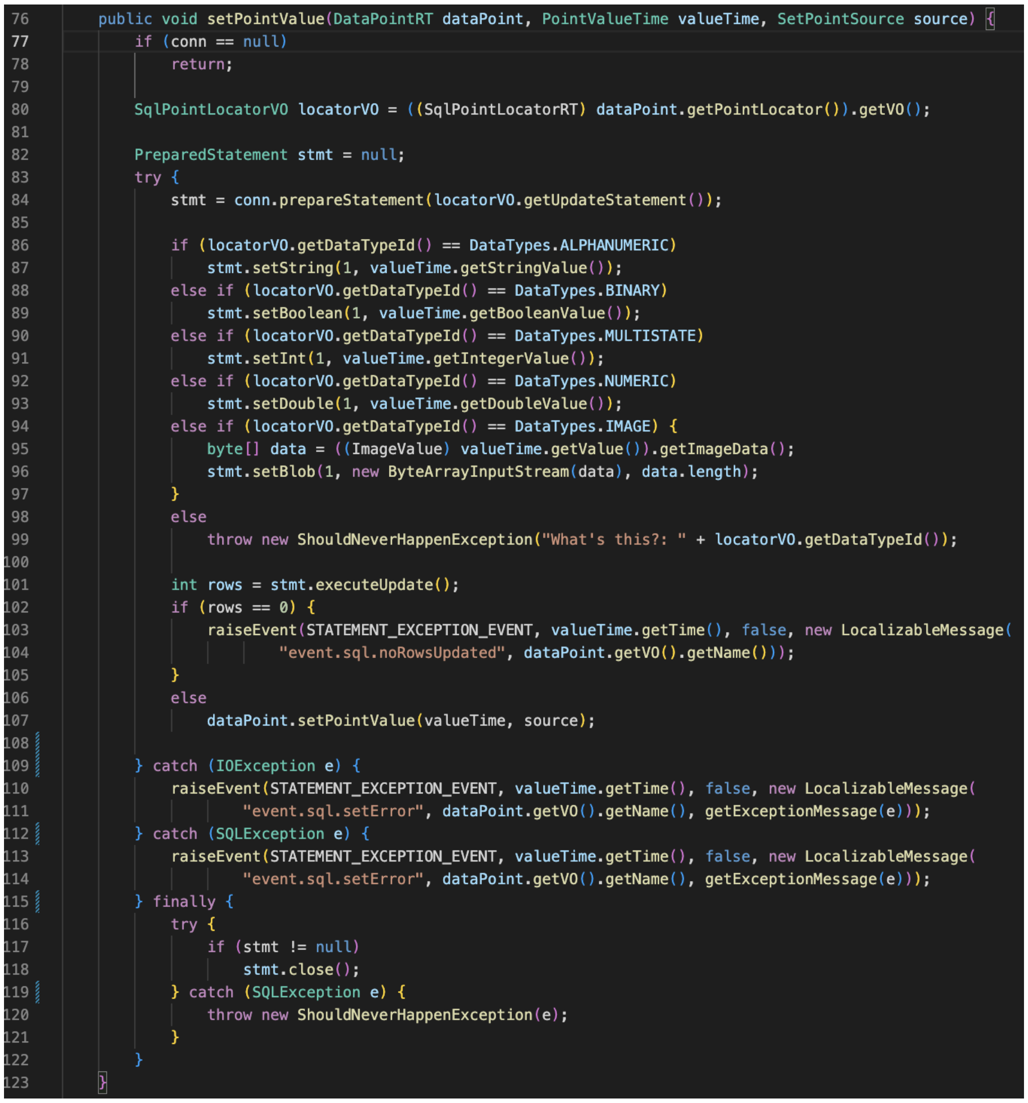
Description & Potential Risk. The code that causes an alert in SonarCloud can be found on line 120 in the method called *setPointValue*. See Figure 12. Here, an exception is thrown within a finally-block, which is bad practice. When an exception is thrown or any jump statement (java: break, continue, return) is used in a finally-block it suppresses the propagation of unhandled exceptions thrown in any preceding either try or catch blocks. Suppression of the propagation of unhandled exceptions, also called error hiding, is dangerous because it impacts the maintainability of a program negatively, especially when systems grow and the teams working on them change.

Solution. Along with the modifications made to the class *SqlDataSourceRT,* a new java class *ExceptionCollector*, has been created to resolve the bug. The new class is presented in Figure 13 and the modifications to *SqlDataSourceRT* are presented in Figure 14.

Instead of throwing exceptions throughout the code if they are thrown in the different catch blocks, an ArrayList is created (see Figure 14: Alteration 1) that collects the exceptions (see Figure 14: Alteration 2 and Alteration 3). If any exceptions are thrown, a new exception of the type ExceptionCollector (see Figure 13), is created with the help of the ArrayList and thrown (see Figure 14: Alteration 4). When this exception is caught, the method *getExceptions()* can be called on the object and all the exceptions thrown in the *setPointValue* method can be retrieved. This ensures that there is no error hiding, since it is possible to trace all errors that occur in the code.

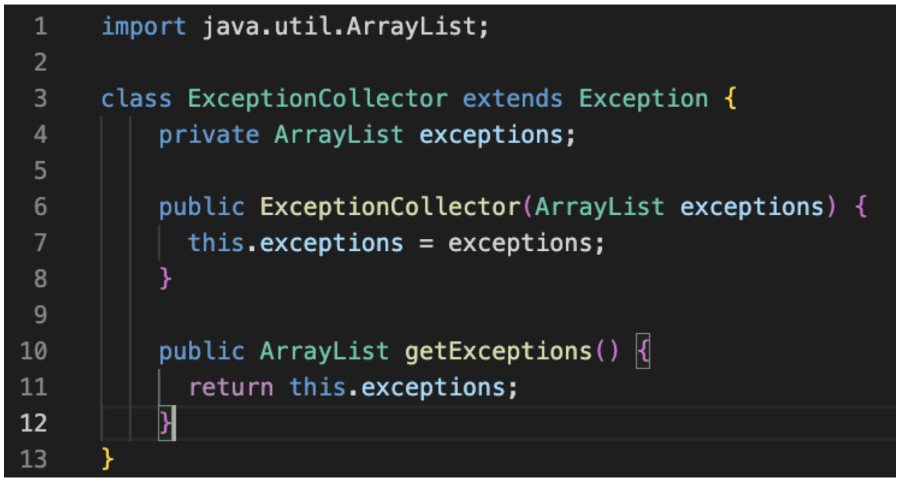
2.3.7 **Software Bug**: Commented out code.

Found in: *build.xml*
What is the severity type of the vulnerability? **Major**.
What is the CWE type of the bug? N/A.
Description & Potential Risk. It is bad practice to keep commented out code since it enlarges the program and impacts the readability negatively. Instead, code that is no longer useful, should be removed and retrieved using version control management programs when necessary. One piece of commented out code that causes this bug is displayed in Figure 15.



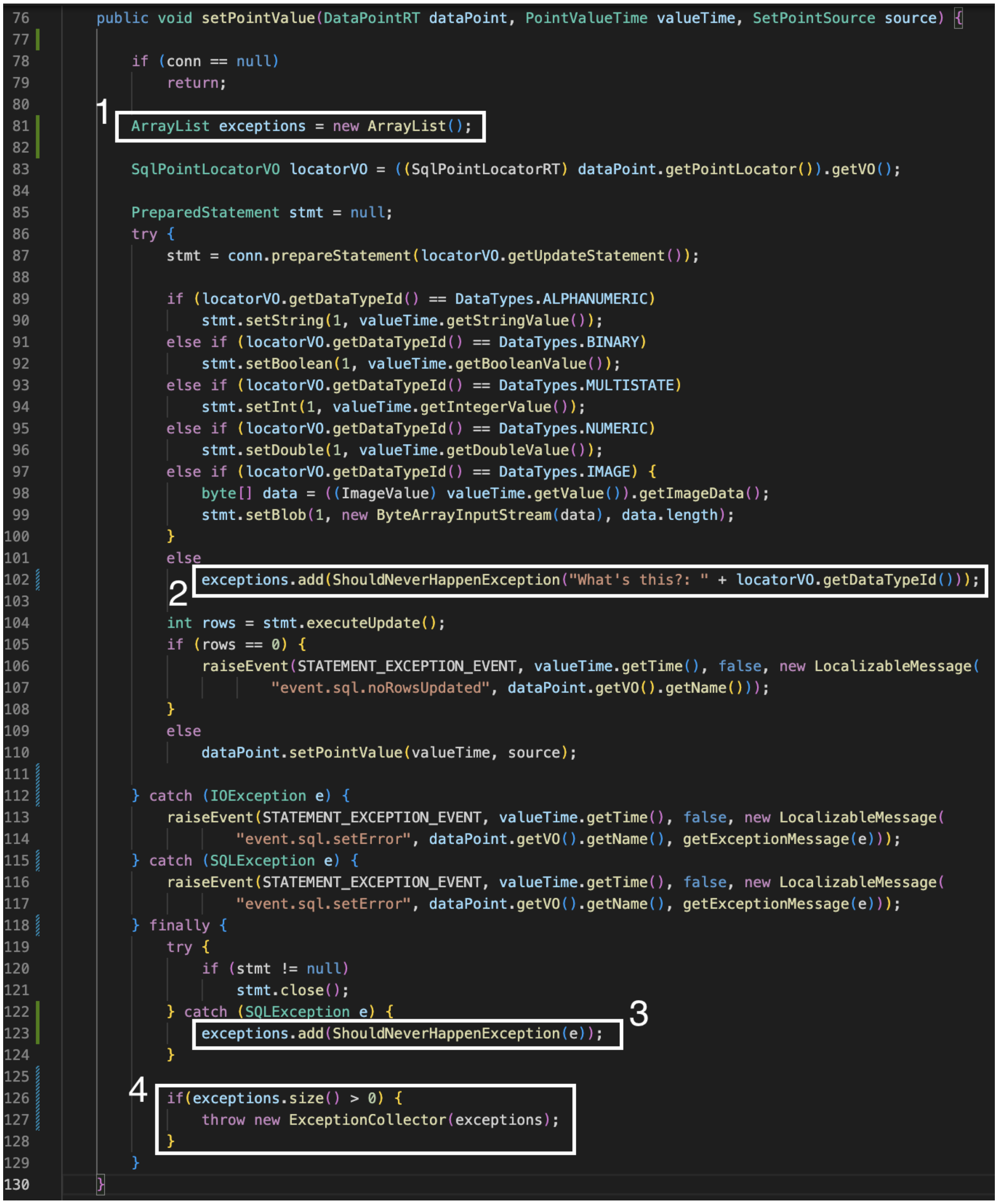
```java
public void setPointValue(DataPointRT dataPoint, PointValueTime valueTime, SetPointSource source) {
    if (conn == null)
        return;

    SqlPointLocatorVO locatorVO = ((SqlPointLocatorRT) dataPoint.getPointLocator()).getVO();

    PreparedStatement stmt = null;
    try {
        stmt = conn.prepareStatement(locatorVO.getUpdateStatement());

        if (locatorVO.getDataTypeId() == DataTypes.ALPHANUMERIC)
            stmt.setString(1, valueTime.getStringValue());
        else if (locatorVO.getDataTypeId() == DataTypes.BINARY)
            stmt.setBoolean(1, valueTime.getBooleanValue());
        else if (locatorVO.getDataTypeId() == DataTypes.MULTISTATE)
            stmt.setInt(1, valueTime.getIntegerValue());
        else if (locatorVO.getDataTypeId() == DataTypes.NUMERIC)
            stmt.setDouble(1, valueTime.getDoubleValue());
        else if (locatorVO.getDataTypeId() == DataTypes.IMAGE) {
            byte[] data = ((ImageValue) valueTime.getValue()).getImageData();
            stmt.setBlob(1, new ByteArrayInputStream(data), data.length);
        }
        else
            throw new ShouldNeverHappenException("What's this?: " + locatorVO.getDataTypeId());

        int rows = stmt.executeUpdate();
        if (rows == 0) {
            raiseEvent(STATEMENT_EXCEPTION_EVENT, valueTime.getTime(), false, new LocalizableMessage(
                    "event.sql.noRowsUpdated", dataPoint.getVO().getName()));
        }
        else
            dataPoint.setPointValue(valueTime, source);

    } catch (IOException e) {
        raiseEvent(STATEMENT_EXCEPTION_EVENT, valueTime.getTime(), false, new LocalizableMessage(
                "event.sql.setError", dataPoint.getVO().getName(), getExceptionMessage(e)));
    } catch (SQLException e) {
        raiseEvent(STATEMENT_EXCEPTION_EVENT, valueTime.getTime(), false, new LocalizableMessage(
                "event.sql.setError", dataPoint.getVO().getName(), getExceptionMessage(e)));
    } finally {
        try {
            if (stmt != null)
                stmt.close();
        } catch (SQLException e) {
            throw new ShouldNeverHappenException(e);
        }
    }
}
```

Figure 12: Method *setPointValue()* found in file *DataSourceRT.java*.



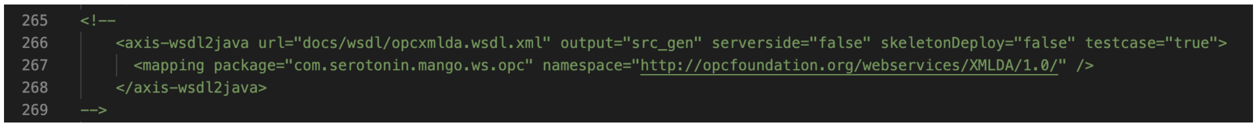
```java
import java.util.ArrayList;

class ExceptionCollector extends Exception {
    private ArrayList exceptions;

    public ExceptionCollector(ArrayList exceptions) {
        this.exceptions = exceptions;
    }

    public ArrayList getExceptions() {
        return this.exceptions;
    }
}
```

Figure 13: Class ExceptionCollector.



```java
public void setPointValue(DataPointRT dataPoint, PointValueTime valueTime, SetPointSource source) {

    if (conn == null)
        return;

    ArrayList exceptions = new ArrayList();

    SqlPointLocatorVO locatorVO = ((SqlPointLocatorRT) dataPoint.getPointLocator()).getVO();

    PreparedStatement stmt = null;
    try {
        stmt = conn.prepareStatement(locatorVO.getUpdateStatement());

        if (locatorVO.getDataTypeId() == DataTypes.ALPHANUMERIC)
            stmt.setString(1, valueTime.getStringValue());
        else if (locatorVO.getDataTypeId() == DataTypes.BINARY)
            stmt.setBoolean(1, valueTime.getBooleanValue());
        else if (locatorVO.getDataTypeId() == DataTypes.MULTISTATE)
            stmt.setInt(1, valueTime.getIntegerValue());
        else if (locatorVO.getDataTypeId() == DataTypes.NUMERIC)
            stmt.setDouble(1, valueTime.getDoubleValue());
        else if (locatorVO.getDataTypeId() == DataTypes.IMAGE) {
            byte[] data = ((ImageValue) valueTime.getValue()).getImageData();
            stmt.setBlob(1, new ByteArrayInputStream(data), data.length);
        }
        else
            exceptions.add(ShouldNeverHappenException("What's this?: " + locatorVO.getDataTypeId()));

        int rows = stmt.executeUpdate();
        if (rows == 0) {
            raiseEvent(STATEMENT_EXCEPTION_EVENT, valueTime.getTime(), false, new LocalizableMessage(
                    "event.sql.noRowsUpdated", dataPoint.getVO().getName()));
        }
        else
            dataPoint.setPointValue(valueTime, source);

    } catch (IOException e) {
        raiseEvent(STATEMENT_EXCEPTION_EVENT, valueTime.getTime(), false, new LocalizableMessage(
                "event.sql.setError", dataPoint.getVO().getName(), getExceptionMessage(e)));
    } catch (SQLException e) {
        raiseEvent(STATEMENT_EXCEPTION_EVENT, valueTime.getTime(), false, new LocalizableMessage(
                "event.sql.setError", dataPoint.getVO().getName(), getExceptionMessage(e)));
    } finally {
        try {
            if (stmt != null)
                stmt.close();
        } catch (SQLException e) {
            exceptions.add(ShouldNeverHappenException(e));
        }

        if(exceptions.size() > 0) {
            throw new ExceptionCollector(exceptions);
        }
    }
}
```

Figure 14: Method *setPointValue()* found in file *DataSourceRT.java* with four alterations (marked with white-boxes and numbers).



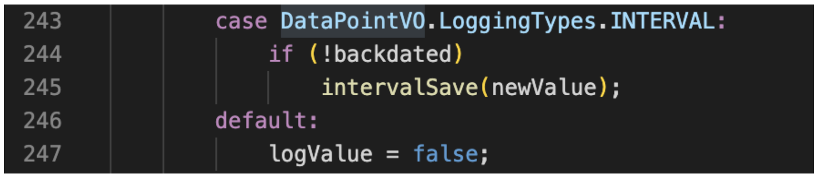
```
<!--
    <axis-wsdl2java url="docs/wsdl/opcxmlda.wsdl.xml" output="src_gen" serverside="false" skeletonDeploy="false" testcase="true">
      <mapping package="com.serotonin.mango.ws.opc" namespace="http://opcfoundation.org/webservices/XMLDA/1.0/" />
    </axis-wsdl2java>
-->
```

Figure 15: Commented out code found in *build.xml* file.

Solution: This bug is present in multiple places in this document and the solution is to remove the out-commented code.

2.3.8 **Code Smell**: Switch cases should end with an unconditional "break" statement.

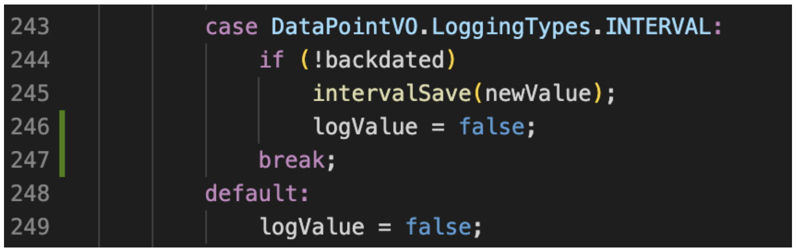
Found in: *src/com/serotonin/mango/rt/dataImage/DataPointRT.java*
What is the severity type of the code smell? **Blocker**.
What is the CWE type of the bug? CWE-484 - Omitted Break Statement in Switch.
Description & Potential Risk. In Figure 16, the last two cases of a switch-case statement are displayed. In the next to last one, the break-statement is omitted. When this case is true, both the case itself and the default case will be executed. This could be intentionally written by the authors of the code, but could also be a mistake. Either way, it is bad practice, since it is not clear to the reader of the code if this is something intentional or a mistake that could lead to unexpected behaviour.



```
243            case DataPointVO.LoggingTypes.INTERVAL:
244                if (!backdated)
245                    intervalSave(newValue);
246            default:
247                logValue = false;
```

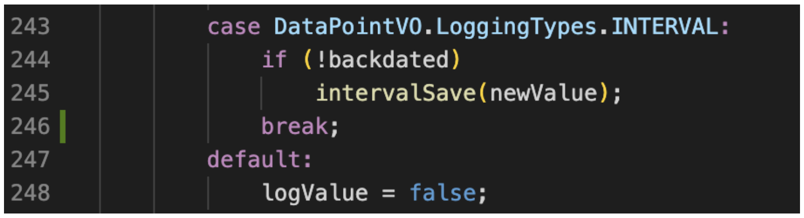
Figure 16: Part of switch-case found in method *savePointValue* in the file *DataPointRT.java.*

Solution: Since this bug could be either intentional or a programming error there are two solutions. First, if the omittance is purposeful, a solution has been suggested in Figure 17. Here a break-statement is added (at line 247) together with the line of code previously only written in the default statement. In this solution, the functionality of the code is unaltered but it is clear what outcome the author of the code wishes for from the execution of the code. If the omittance is accidental then a suggestion for solution is presented in Figure 18. A break-statement is added (line 246), but here, the functionality of the code is altered so that the line of code written in the default case is only executed when no other case in the switch-case statement is true. In this suggested solution the intentions of the author remain clear while we avoid any unexpected behaviors.



```
243            case DataPointVO.LoggingTypes.INTERVAL:
244                if (!backdated)
245                    intervalSave(newValue);
246                    logValue = false;
247                break;
248            default:
249                logValue = false;
```

Figure 17: Suggested alteration of code found in method *savePointValue* in the file *DataPointRT.java* if bug is intentional.



```
        case DataPointVO.LoggingTypes.INTERVAL:
            if (!backdated)
                intervalSave(newValue);
            break;
        default:
            logValue = false;
```

Figure 18: Suggested alteration of code found in method *savePointValue* in the file *DataPointRT.java* if bug is unintentional.

3. Penetration Tests for Detecting Security Vulnerabilities

3.1 Manual for Penetration Test

Prerequisites: Mango running locally, OWASP ZAP, Firefox browser configured to allow ZAP to scan.

Authentication setup

1. Start OWASP ZAP
2. Login as admin in Mango
3. Under sites, right click on the “test” folder. Select “Include in Context” and then “Default Context” and rename it “Mango”.
4. Go to “Authentication” and select “Form-based Authentication” as authentication method for the context.
5. Press select in “Login Form Target URL*:” and select “POST:login.htm()(password,username)”. Set username parameter to username and password parameter to password.
6. Go to “Users” and add a user called admin with username and password set to “admin”.
7. Go back to “POST:login.htm()(password,username)” under sites and right click to select “Flag as Context”. Select “Mango : Form-based Auth Login Request” and paste “\Q<a href="login.htm">\E” in the “Regex pattern used to identify Logged Out messages” field.
8. Enable forced user mode in ZAP

Attacking approach:

1. Browser to the mailing list feature and launch a spider and active scan towards its URL in OWASP ZAP
2. Browse manually through the all the different features
3. Attack some of the URLs with active scan
4. In this report, active scans were executed on the following URLs:
    a. http://localhost:8080/test/event_handlers.shtm
    b. http://localhost:8080/test/watch_list.shtm
    c. http://localhost:8080/test/sql.shtm
    d. http://localhost:8080/test/mailing_lists.shtm
    e. http://localhost:8080/test/emport.shtm
    f. http://localhost:8080/test/compound_events.shtm
    g. http://localhost:8080/test/login.htm
    h. http://localhost:8080/test/data_sources.shtm

3.1.1 Comments on the penetration testing approach

Since our chosen feature did not provide us with 8 vulnerabilities, we chose to focus on vulnerabilities for the entirety of the Mango system. To cover as many vulnerabilities as possible, the use of the traditional spider was complemented by manually traversing the different features. Launching an active scan on the whole system requires a lot of time waiting for it to finish. Therefore, active scans were launched against common features in the system until more than eight vulnerabilities were detected.

3.2 Summary of Penetration Testing

A total of 18 alerts were raised by OWASP ZAP in the penetration testing. Eight of these were informational alerts meaning they do not indicate any vulnerabilities. Two alerts classified as high risk, the alerts classified as medium risk and 10 alerts classified as low risk were found. Two of the low risks are not going to be further discussed in this report. These were “Server Leaks Version Information via "Server" HTTP Response Header Field” and “X-Content-Type-Options Header Missing”.

3.3 Identified Vulnerabilities

This section provides further descriptions and solutions for eight of the vulnerabilities found during the penetration test. Descriptions and solutions are based on information provided by the OWASP web site [11] and information present in the OWASP ZAP tool [12].

3.3.1 Cross-site Scripting (reflected)

Description: Cross-site Scripting is a type of injection where malicious scripts can be injected in the web site. It can lead to hijacking of user credentials or modifications of the functionality and interface of the website.
Where? This vulnerability was found in a POST:http://localhost:8080/test/login.htm request.
Security level? The vulnerability was flagged as an high risk with medium confidence
OWASP top 10: Injection
Recommended solution:

- Using a suitable library or framework allows the generation of correct output. It is important to have a good understanding of the data to define the appropriate encodings.
- Regarding security checks, duplications are essential on the server side. This is because attackers can modify or delete client values. If possible, automation can be done to separate data and code. This fast and efficient mechanism helps the developer.
- For each web page generated, a specific encoding is requested (ISO-8859-1 or UTF-8). Otherwise, the web browser may choose another encoding by guessing it. However, this can sometimes lead to serious consequences. In this case it is useful to set the session cookie to HttpOnly, but it is not a complete solution as it is not supported by all browsers.
- As all inputs can be malicious, an input validation strategy is strongly advised, considering many relevant properties. Additionally, denials can be useful to protect the application from potential attacks.

3.3.2 PII Disclosure

Description: A representation of data that includes information which could be used to identify personal or sensitive details by direct or indirect means. In this case, the identifiable information obtained was credit card type and a bank identification number .
Where? This vulnerability was found in a GET:http://localhost:8080/test/dwr/engine.js request.
Security level? The vulnerability was flagged as an high risk with high confidence
OWASP top 10: Insecure design
Recommended solution:

- By establishing a secure development lifecycle, improvement can be made. Continuous unit and integration testing must also be implemented.
- Then apply granular requirements and resource management. Then, implement the separation of system and network layers to finish.

3.3.3 Absence of Anti-CSRF Tokens

Description: Anti-CRFS Tokens are given to users to validate their requests. The tokens consist of unique values that a third party can not guess. An absence of these tokens could allow attackers to authorize as another user and perform actions towards a website via the victims privileges.
Where? This vulnerability was found in 6 different requests during the attack. More specifically, alerts were raised in POST and GET requests to the three following URLs: http://localhost:8080/test/login.htm, http://localhost:8080/test/sql.shtm, http://localhost:8080/test/view_edit.shtm
Security level? All alerts were classified with medium risk with a low confidence
OWASP top 10: Broken Access Control
Recommended solution:

- One should also use an approved library or framework like anti-CSRF packages.
- It also needs to be ensured that the application is free of cross-site scripting issues and controlled by the attacker.
- For each form, a unique nonce must be generated and then verified.
- Identifying particularly dangerous operations by sending a confirmation to the user is important. For this, an ESAPI session management control is needed.
- Always check the HTTP Referer header to see where each request is coming from, to avoid privacy issues.

3.3.4 Content Security Policy (CSP) Header Not Set

Description: The CSP policy is used to detect attacks and malicious injections on a website. It provides a set of standards for HTTP headers that are used by the owner to approve what resources a browser should be allowed to load. The CSS policy adds a layer of the security to the website.
Where? This alert was found for 119 different requests. One example was in GET:http://localhost:8080/test/mailing_lists.shtm.
Security level? All the instances with this alert were classified as a medium risk with high confidence.
OWASP top 10: Security Misconfiguration
Recommended solution: For the mailing list, it must be ensured that the web and application servers, as well as the load balancer are configured as part of the Content-Security-Policy. Thereafter, the browser is ready to perform optimal support.

3.3.5 Missing Anti-clickjacking Header

Description: An Anti-clickjacking Header is used to protect websites against clickjacking, which is when an attacker tricks a client to click on disguised elements. This technique can be used by an attacker to reveal confidential information or take control of a user's computer.
Where? This alert was found for 95 different requests. One example was in GET: http://localhost:8080/test/events.shtm.
Security level? All the instances with this alert were classified as a medium risk with medium confidence.
OWASP top 10: Security Misconfiguration
Recommended solution: For modern web browsers, ensure that one of the Content-Security-Policy and X-Frame-Options HTTP headers is set on all returned web pages. A page can either be framed only by pages from the server, or never be, but the tool is not the same. Implementing the "frame-ancestors" directive of the content security policy is also possible.

3.3.6 Application Error Disclosure

Description: This vulnerability means that the data on the webpage may disclose sensitive information due to an error message displayed by the application. The information provided can be used by attackers to identify the location of the file that produced the error or be used in further attacks against the application.
Where? This alert was found in a GET:
http://localhost:8080/test/data_point_details.shtm?dpid=%22+%20data.topPoints%5Bi%5D.pointId%20+%22 request.
Security level? The alert was classified as low risk with medium confidence.
OWASP top 10: Security Misconfiguration
Recommended solution: It is absolutely necessary to check the source code of this page. One can either implement custom error pages or a mechanism that provides a unique error reference to the client by logging the details server-side and more importantly not exposing them to the user.

3.3.7 Cookie without SameSite Attribute

Description: Cookies that are set with a SameSite attribute can be used to prevent attacks related to cross-site requests forgery, cross-site script inclusion and timing attacks. If the SameSite attribute is missing, cross-sites requests could result in cookies being sent to a third-party.
Where? This alert was found in GET:
http://localhost:8080/test/login.htm and GET:http://localhost:8080/test/maintenance_events.shtm.
Security level? The alerts were classified as low risk with medium confidence.
OWASP top 10: Broken Access Control
Recommended solution: The SameSite attribute should be set to "lax". Otherwise, it should ideally be set to "strict" for all cookies.

3.3.8 Private IP Disclosure

Description: This alert was raised due to the finding of a private IP address in the responses. The IP addresses of numerous data sources can be found in the response body of GET requests to the data source feature in Mango. This information could be used to launch further attacks by an attacker.
Where? This alert was found in GET requests to the http://localhost:8080/test/data_sources.shtm URL.
Security level? The alerts were classified as low risk with medium confidence.
OWASP top 10: Broken Access Control
Recommended solution: Remove the private IP address from the body of the HTTP response. For comments, the use of JSP/ASP/PHP format instead of HTML/JavaScript is required because the latter can be seen by client browsers.

4. Comparison: Static Analysis and Penetration Testing Results

The approach of the formerly detailed methods differ both quantitatively and qualitatively. Initially, static analysis provides context-dependent metrics by means of analyzing the provided source code, that can then help elucidate otherwise latent, urgent problems. The summary of the mango-wide static analysis yielded the following metrics: 3,600 bugs, 4,500 code smells, 2 vulnerabilities and 47 security hotspots In 168,000 lines of code. Upon further analysis, the identified security hotspots were shown to include further vulnerabilities, specifically, 11 violations of CWE protocols, ranging from low to high priority. The most common one was CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting'), which was identified 3 times. Static analysis without further inspection would've not immediately revealed the additional, aggravated vulnerabilities.

In contrast, penetration testing provides a lower volume of alerts and vulnerabilities that tend to be backed by actual operational failures, at the cost of the resources required to successfully implement it. In total, 18 alerts were found where 8 didn't indicate any vulnerabilities and the remaining 10 were low risk. Cross-side scripting, PLLL Disclosure, absence of Anti-CSRF tokens were among the warnings. However, further inspection reveals that what can be originally perceived as a low risk failure by penetration testing software, can be aggravated and easily escalate into a high security risk.

5. Conclusion

Security testing is hardly a streamlined task. Neither of the testing protocols and associated risk-finding carried out in this document reason thoroughly through a software program's potential security concerns. For this reason, they can be used in a complementary fashion, when able, to provide a fuller context. The former notwithstanding, an experienced security tester is required alongside any of the testing cases to bring a security evaluation closer to completion and utmost safety.

References

[1] "CVE-2019-13466 - NVD." 30 Sep. 2019, https://nvd.nist.gov/vuln/detail/CVE-2019-13466. Accessed 1 Dec. 2022.

[2] "CVE-2018-15389 Detail - NVD." 5 Oct. 2018, https://nvd.nist.gov/vuln/detail/CVE-2018-15389. Accessed 1 Dec. 2022.

[3] "CVE-2013-6386 Detail - NVD." https://nvd.nist.gov/vuln/detail/CVE-2013-6386. Accessed 1 Dec. 2022.

[4] "CVE-2006-3419 Detail - NVD." https://nvd.nist.gov/vuln/detail/CVE-2006-3419. Accessed 1 Dec. 2022.

[5] "CVE-2008-4102 Detail - NVD." 18 Sep. 2008, https://nvd.nist.gov/vuln/detail/CVE-2008-4102. Accessed 1 Dec. 2022.

[6] "CVE-2018-1999007 Detail - NVD." 23 Jul. 2018, https://nvd.nist.gov/vuln/detail/CVE-2018-1999007. Accessed 1 Dec. 2022.

[7] "CVE-2015-5306 Detail - NVD." https://nvd.nist.gov/vuln/detail/CVE-2015-5306. Accessed 1 Dec. 2022.

[8] "CVE-2013-2006 - NVD." https://nvd.nist.gov/vuln/detail/CVE-2013-2006. Accessed 1 Dec. 2022.

[9] "SQL Injection Prevention - OWASP Cheat Sheet Series." https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html. Accessed 1 Dec. 2022.

[10] "How to use a tablename variable for a java prepared statement insert." 3 Jul. 2012, https://stackoverflow.com/questions/11312155/how-to-use-a-tablename-variable-for-a-java-prepared-statement-insert. Accessed 1 Dec. 2022.

[11] "OWASP Top Ten." https://owasp.org/www-project-top-ten/. Accessed 1 Dec. 2022.

[12] "OWASP Zap." https://owasp.org/www-project-zap/. Accessed 1 Dec. 2022.