Software Quality TP3

01/12/2022

Étienne Lescarbeault 1949656 etienne.lescarbeault@polymtl.ca	**Linda Nguyen** 2237354 linda.nguyen@polymtl.ca	**Yonnel Chen** 1992778 yonnel.chen-kuang-piao@polymtl.ca

François Barabé 1992767 francois.barabe@polymtl.ca	**Mohamed Yassir El Aoufir** 1885972 mohamed-yassir.el-aoufir@polymtl.ca	**Charles-Étienne Pronovost** 1958141 charles-etienne.pronovost@polymtl.ca

Abstract

After evaluating the quality and performance of the report generation functionality of the Mango software, we improved our analysis of the system by focusing on the security aspect of the whole software. First, we performed a static analysis using *SonarQube* to identify security vulnerabilities and we classified them using the community-developed list Common Weakness Enumeration. No *Vulnerabilities* requiring immediate remediation were detected. However, it identified 47 *Security Hotspots*, which are parts of the code that could pose a security issue and should be investigated by the developers. Among them, 44 were classified as Low, 1 as Medium and 2 as High. Eight of them were analyzed and discussed, and 6 of these were found to be vulnerabilities that need to be fixed. Subsequently, we performed penetration tests on the entire Mango system using OWASP ZAP. In this way, we have identified and explained vulnerabilities in Mango. We also provided solutions for each vulnerability. By using the two aforementioned approaches, we found that the security concerns detected by each of the respective tools are in essence different and that having both techniques complementing each other should be used as part of the quality assurance activities.

1. Introduction

The Mango system is a sensor management database. It offers a structuring mechanism to process the numerous data points received in real time and many functionalities to process the data, query it, generate informative reports as well as monitoring the points (the sensors) received to alert the users. This system acts like a pivot from the world of connected sensors to interpretability, it is connected to several other systems, so its security is an essential aspect and must be assessed. Indeed, security vulnerabilities can be used by an exploit, in the case of Mango this can mainly result in a loss of private data or/and or in incorrect behavior of the program, this obviously leads to other consequences. Thus, by identifying security vulnerabilities, we can improve and protect the system.

We can say that our work focuses on the Security characteristic of the ISO 25000 standard. This characteristic is officially defined as the “Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.”(ISO/IEC 25010”, (n.d.)).

Our goal is to assess the security of the whole software. To do this, we will first perform a static analysis with SonarQube. This will allow us to identify security vulnerabilities and to classify them using the list developed by the Common Weakness Enumeration community. Secondly, we will simulate malicious attacks by performing penetration tests on the entire Mango system using OWASP ZAP. We will be able to identify vulnerabilities in Mango and their degree of sensitivity. We will then provide solutions for them.

2. Static Analysis

In this section, we are discussing the static analysis that we performed on the entire software using *SonarQube*, an automated tool mainly used to identify security vulnerabilities. We chose SonarQube for its detailed security recommendations as well as for its feature to evaluate MangoDB’s source code locally. To define the type of vulnerabilities that were identified, we are using Common Weakness Enumeration, a community-developed list to classify and describe prevalent software and hardware weaknesses (CWE, 2022a).

2.1 Manual

First and foremost, as static analysis is a quality assurance activity that needs to be carried out on a continuous basis, we included the following manual to help with the installation and set-up of the required tools for the development team. Considering SonarQube is being introduced and tested out, all the analysis was done on the original Mango's source code.

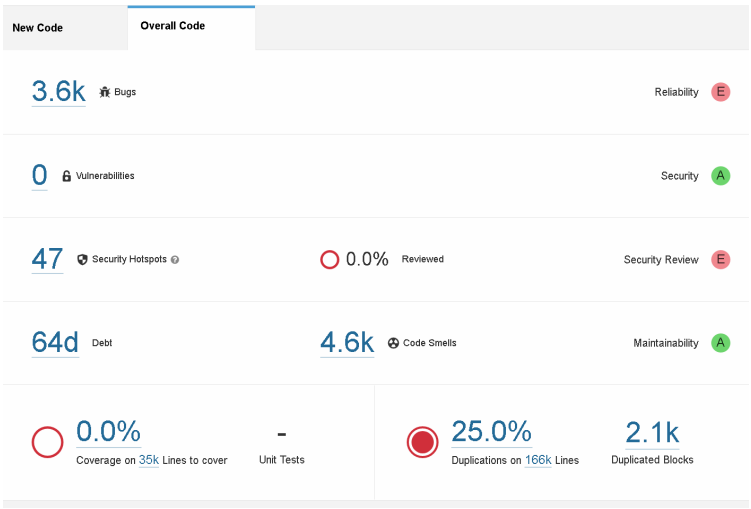
1. Download and install Java 11 if it hasn't already been done from https://adoptium.net/temurin/releases/?version=11. Make sure to set its folder path as the JAVA_HOME system variable (via Edit Environment variables). As well, edit the Path system variable in order to add Java 11's bin folder there too.
2. Download SonarQube from https://www.sonarqube.org/
3. Download SonarScanner from https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/
4. Add the SonarScanner's bin folder as Path via Edit Environment variables.
5. Run the file StartSonar.bat from SonarQube's bin folder.
6. Head to http://localhost:9000.
7. If using SonarQube as a first-time user, log in with the credentials "admin" as the username and "admin" as the password. The system will prompt the user to create a new password. If this isn't the first time, simply log in with the already established password.
8. Click "Create Project" at the top right and select "Manually".
9. Enter any Project Key name such as "mangoanalysis" for example.
10. Select the option for the analysis to be done locally.
11. Generate a project token by clicking on the "Generate" button.
12. Click on "Continue" and then on "Other" for the build.
13. Select the OS such as "Windows".
14. Open MangoSource folder. Open Git Bash in there and write the command "*sonar-scanner.bat -D"sonar.projectKey=mangoanalysis" -D"sonar.sources=." -D"sonar.host.url=http://localhost:9000" -D"sonar.login=admin" -D"sonar.password=PASSWORD" -D"sonar.java.binaries=MANGOPATH*" where "*PASSWORD*" is to be replaced by the newly created password from step 7 and where "*MANGOPATH*" is the complete folder path containing the Mango Source folder.
15. The scan may take several minutes to complete. Once done, refresh the dashboard page via http://localhost:9000/ to see SonarQube's analysis results.

One of the main challenges may come from the installation and setup process as SonarQube assumes users are familiar with editing environment variables (for both Java 11 and SonarScanner) which isn't explicitly outlined in the documentation. Shuffling between SonarQube, SonarScanner, Oracle and more websites had to be done to solve this. Additionally, in step 14, SonarQube does generate such a command line to be copy and pasted to be written in Git Bash. However, it could not be used as it initially was. The following modifications had to be done in order for SonarQube to successfully run the analysis: the part about *sonar.password* had to be added; *sonar.login*'s value had to be edited to "admin" rather than leaving it as the generated id; the part with *sonar.java.binaries* also had to be added, for otherwise the *org.sonar.java.AnalysisException* would be thrown. In essence, we highly recommend for users to follow our manual to ensure that the static analysis runs smoothly.

2.2 Results

Next, a summary of the main security concerns outlined by SonarQube is presented in this segment. In essence, the aforementioned tool discovered after an initial scan zero *Vulnerabilities*, which are problems that would require an immediate fix as they represent a solid threat to a software's security (SonarQube, 2022). On the other hand, it identified 47 *Security Hotspots* (see Figure 1), which are security-sensitive pieces of code that require a manual review from developers to assess whether refactoring must be done or not depending on if they are actual vulnerabilities (2022). The severity levels of those potential issues range from Low (e.g. leaving debugging feature on after production phase) to High (e.g. concerns around authenticating). Fortunately, 44 hotspots are classified as Low (i.e. most of them), leaving only 1 as Medium and 2 as High. A security review rating of E is given to MangoDB as currently none of the hotspots have yet gone under review, which is a work in progress for the development team, and some of them may indeed be false positives. It is important to note that while this report focuses on the security aspect, SonarQube may also be used to provide more insight on other quality metrics as depicted in Figure 1 : it found in total about 3.6k bugs (17 with a severity level of Blocker, 1 Critical, 2k Major and 1.6k as Minor), 25% of code duplication, 0% of testability coverage and around 4.6k of *Code Smells*, which are issues that may play an impact on the maintainability of the software (for example, using deprecated HTML attributes).

Figure 1 : Overview of the results from SonarQube's analysis



While all of the security hotspots need to be looked into, we are discussing eight of them as follows:

1. **Authentication**: The security hotspot stemming from an authentication issue is found in the file *src/br/org/scadabr/vo/dataSource/opc/OPCDataSourceVO.java*, line 98 (see Figure 2). In the block of code in question, the method *addContextualMessage* is called with the word "password" as one of the two parameters which is why SonarQube assessed it as a potential hard-coded password. According to CWE-259, the use of a hard-coded password can occur when the software validates what the user inputted during the authentication process against a hard-coded password (CWE, 2022b). The risk comes from the fact that one can easily access the source code to find such a password. If the latter is discovered / leaked, that will enable everybody, even unauthorized ones, to know and use the same password until a patch is done. Such a situation can happen especially when initially creating an administrator account within

the code. This security concern is classified as High considering the safety of sensitive data (e.g. sensors, user details) is at stake. To remedy this, a simple solution would be to store passwords outside of the code such as in a database. In Mango's case, it is not in fact a vulnerability as that statement merely validates whether the string from a password field is empty or not and a message mentioning that the password is required is provided as such. However, while the statement from the hotspot is of no concern, we did find a hard-coded password being set as "admin" for a default, administrative account elsewhere in the code (file *DatabaseAccess.java*, line 135, see Figure 3) and that requires an immediate fix. As such, not all security concerns can be detected through static analysis.

Figure 2 : Preview of a security hotspot in regards to authentication

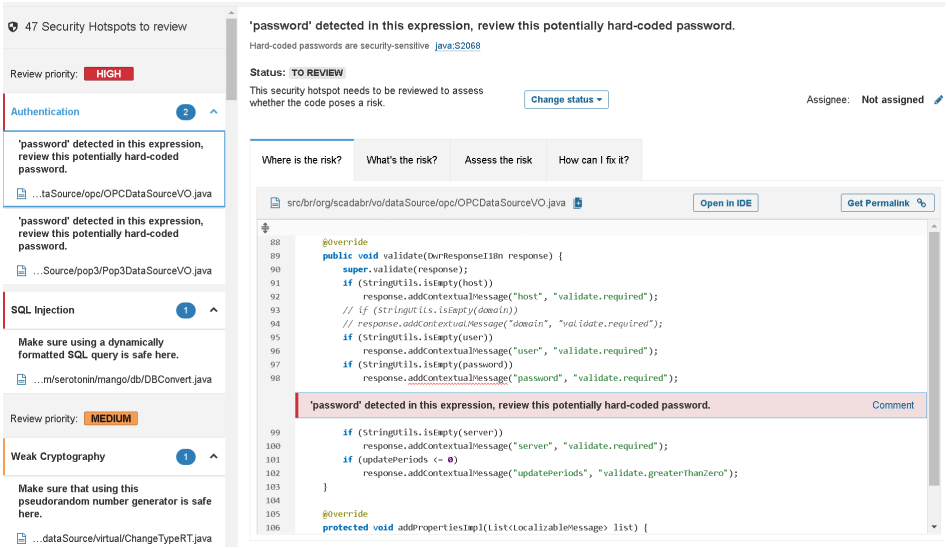


Figure 3 : Hard-coded password found outside of the hotspot

```
130                 else {
131                     // New database. Create a default user.
132                     User user = new User();
133                     user.setId(Common.NEW_ID);
134                     user.setUsername("admin");
135                     user.setPassword(Common.encrypt("admin"));
136                     user.setEmail("admin@yourMangoDomain.com");
137                     user.setPhone("");
138                     user.setAdmin(true);
```

2. **SQL injection:** The next hotspot resides in the file *src/com/serotonin/mango/db/DBConvert.java* line 109. The method *executeQuery* is called with "*("select * from " + tableName)*" as parameters. Since a string including the inputted table name is passed as parameter, this is considered as a dynamically formatted SQL query. The underlying risk is explained in CWE-89, Improper Neutralization of Special Elements used in an SQL Command, which mentions that such method does not neutralize special elements / characters which could ultimately lead to an attacker modifying the intended meaning of the SQL command (CWE, 2022c). For example, a normal user would have written "*SELECT * FROM Users*" to have access to some

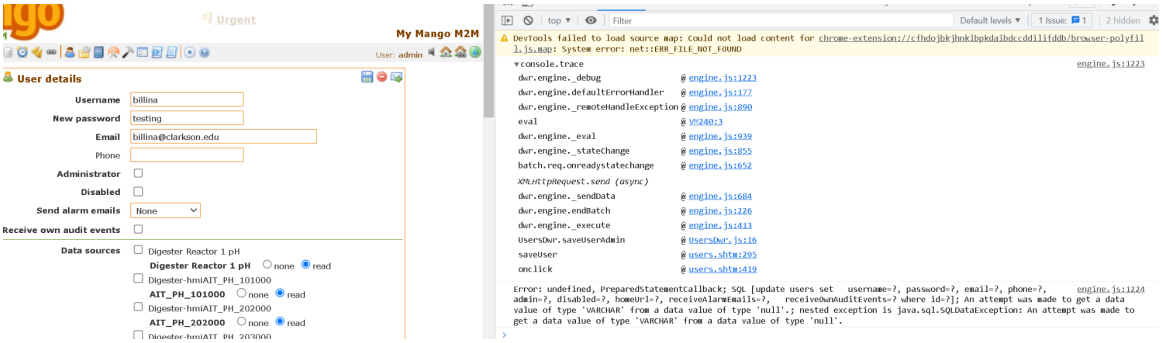
permissible data. On the other hand, a malicious user could add "*; DROP TABLE Users;*" next to the aforementioned string to get rid of users' data right after the query which would result in a dire situation for the company and which is why the severity of this security concern is High. A solution to this would be to first validate and store the user's input in an initial string. Secondly, another string is created to store the whole query (e.g. "*SELECT account_balance FROM user_data WHERE user_name = ?;*" (OWASP Cheat Sheet Series, 2022). Then, we pass the query in Java's PreparedStatement(). And finally, we use *setString* to bind the first string to the prepared statement, right before executing the query. This allows for the query as a whole and validated from special characters to be passed as the parameter and to help prevent SQL injection. In Mango's case, it is not a vulnerability as that file *DBConvert.java*'s use is to retrieve the database so it can be used in the browser and throughout all the features, including getting the tables' names. This security hotspot is not about the SQL functionality (/test/sql.shtm) where users can type their own queries and therefore *tableName* is not what any user actually types in (if it were, then it would have been a vulnerability).

3. **Weak cryptography:** After that, we have the hotspot that pertains to cryptography via the file *src/com/serotonin/mango/rt/dataSource/virtual/ChangeTypeRT.java,* line 26. The application recognizes it as a security concern because the method *Random()* is called to generate a number between 0 and 1, and the value is then stored in a static final variable. However, the numbers we get from such generators are in fact pseudorandom which means they may be predictable and that there is a possibility for malicious users to derive them. Furthermore, the attribute has its accessibility as *Protected* within the package, meaning the generated numbers may be used multiple times. CWE - 338 explains how the Use of Cryptographically Weak Pseudo-Random Number Generator can lead to exposure to some attacks as PRNG's algorithm is not considered as cryptographically strong. An example of a potential risk would be if PRNGs were used during the authentication process to generate session IDs, which attackers may guess and gain access to the platform as a result. Due to the likelihood of one resorting to this method of attack, the severity level is at Medium (CWE, 2022d). A better and more secure alternative in Java would be to use random number generators (RNG) such as *SecureRandom()*, which is cryptographically strong. In Mango's case, with the fact that the method Random() is called instead of *SecureRandom()* and given that the generated pseudo random value is used in at least 21 classes (e.g. *BrownianChangeRT.java*, *RandomAnalog ChangeRT.java*), it does constitute a vulnerability and requires an immediate fix. The changing game between the two methods is that when *Random()* is used for cryptographic purposes, it only has 48 bits, whereas *SecureRandom()* provides up to 128 bits, making repeats less likely (GeeksForGeeks, 2022).

4. **Insecure configuration:** Then, our next security hotspot is found in the file *src/br/org/scadabr/rt/dataSource/dnp3/Dnp3DataSource.java*, line 149. This line has to go under review as *e.printStackTrace() was called in a* catch block handling exceptions. According to CWE-215, the Insertion of Sensitive Information Into Debugging Code can be problematic if the debugging mode is still activated while going into production which would lead to exposure of such sensitive information (CWE, 2022e). An example would be having a code block that writes users' details (e.g. address, names) in the browser while the debugging mode is on. While the severity of this is considered as Low, it is important not to leave active debug features in production servers. A better alternative would be to use the *Loggers* framework

instead of *printStackTrace()* as redirection of the error logs to a more secure destination (e.g. files, database) and the formatting of how the log entries should look like are permissible through its modules (Crowdstrike, 2022). In the case of Mango as a finalized product, the hotspot is a vulnerability as from the statement, exceptions' stack traces are printed in the browser's console which is too much information that should not be given to malicious users, returning the nature of the exception as well as the called methods' line numbers. For example, we verified as such by clicking on Mango's Users page, then on any user, and finally we set the New Password to something else. We can see that the stack trace is printed as depicted in Figure 4.

Figure 4 : Exception's stack trace being printed in the console



5. **Safety behind publicly writable directories:** Next, we have a hotspot that can be found in the file *src/com/serotonin/mango/vo/report/ReportChartCreator.java*, line 226, with a severity level of Low. This line is under review because it is writing into a public storage, in this case in the temporary files, which could mean that a malicious user could access, modify or create a file of the same name. This is especially a problem if the name of the file is predictable. In this case, while the name of the file, "tempZip.zip", is predictable, the Java function *File.createTempFile()* adds a random string of numbers at the end of the file making it unique and unpredictable. This vulnerability is classified under CWE as CWE-379: Creation of Temporary File in Directory with Insecure Permissions, which notes that if an attacker to list the processes on the system, the user could correlate this information with the created file and gain information regarding a user's actions. It also clarifies that an attacker could possibly read and modify the contents of the file when using the Java *createTempFile()*. In this case, for Mango, it means an attacker could read and modify the contents of the report that is sent as an email attachment, which could be private information, which means that it is a security vulnerability. To prevent this, the temporary file should be written to a directory that is not world readable, meaning that an attacker would need special privileges to access the content of that directory. A method that guarantees a unique file name should still be used to prevent a race condition.

6. **Safety behind publicly writable directories** (second one): This hotspot is of the same kind as the previous one and can be found in the file *src/com/serotonin/mango/rt/maint/work/ReportWorkItem.java* at line 223. It also has a severity level of Low. The *File.createTempFile()* is also used here to create a temporary file in a public folder, so a malicious actor could potentially access private information inside the file or modify it before it is used. In this case, the file contains information related to events stored in the Mango database that will be used in

creating a report, so the information should indeed stay private and its integrity should be assured as to keep a correct report. This means that the line does contain a vulnerability and, like the previous one, is also of the type CWE-379: Creation of Temporary File in Directory with Insecure Permissions. The recommendation for solving this problem is the same and is to save the file to a directory that is not world readable, which means that no user should not be able to read or write anything inside of the directory without special privileges, which would protect it from an attacker without these special privileges.

7. **Lacks of *noopener* attribute**: This security hotspot is located in the *build/WEB-INF/snipped/eventList.jsp* file at line 120. It has a severity level of Low. It highlights that the link featured in a page opens up to a new window that has access back to the original window. This could pose a risk if, for example, an attacker puts a link on a site that opens up to their malicious website and changes the original page to a fake page that prompts the user to enter their credentials. As said in CWE-1022: Use of Web Link to Untrusted Target with window.opener Access, which describes this kind of vulnerability, adding the *rel* attribute with a value of *noopener* prevents the new window from having access to the old one via the *window.opener* object by setting it's value to *null*. In this specific case, however, the link opens to a website that is always the same and should be thrusted, so there should be no security risk. However, adding the attribute could still be useful if the new window having access to the original one is not necessary for it to function in case the redirected website becomes unsafe in the future, especially since the website is not responding at the time of this report being made and doesn't seem to be active. Consequently, we would qualify this line as a vulnerability, even if the risk of it being a problem is quite low.

8. **Safety behind the use of resource integrity feature**: The last security hotspot to be analyzed is the one located in the *build/WEB-INF/jsp/publicView.jsp* file at line 33. It has a severity level of Low. This potential security risk was detected because when importing the JavaScript library Dojo Toolkit from an external content delivery network (CDN), the script tag doesn't contain the integrity attribute. The potential risk comes from the fact that the software importing the resource won't know if the integrity of the resource is compromised, which poses a security risk if it has been modified by a malicious actor. Since in this specific case the toolkit comes from an external CDN, there is a risk in not verifying the integrity of the toolkit in the Mango code, which makes this a vulnerability. To fix this vulnerability, an *integrity* attribute containing the hash should be added to the tag. After downloading the file, the browser checks it to make sure the hash matches the one in the *integrity* attribute and will block it if it doesn't. (W3Schools, 2022) It is important to mention that, in this case, the toolkit is no longer available from the specified URL, which further highlights the importance of checking for a file's integrity when downloading it since it can't be known what will happen to it. This vulnerability, according to CWE, is of the type CWE-353: Missing Support for Integrity Check.

Table 1 : Recap of the identified security hotspots

CWE Type	File / Line	Potential risk example	Solution example	Severity Level	Upon review, is the line in fact a *Vulnerability* for MangoDB?
CWE - 259 - Use of Hard-coded Password	OPCDataSourceVO.java / line 98	Finding the password in the source code.	Store passwords outside of the code, in a database.	High	No
CWE - 89 - Improper Neutralization of Special Elements used in an SQL Command	DBConvert.java / line 109	Writing "; DROP TABLE tableName;" right after the initial SQL command.	Use Java's PreparedStatement().	High	No
CWE - 338 - Use of Cryptographically Weak Pseudo-Random Number Generator	ChangeTypeRT.java / line 26	Guessing a predictable session ID while authenticating.	Use a random number generator (RNG) such as SecureRandom().	Medium	Yes
CWE - 215 - Insertion of Sensitive Information Into Debugging Code	Dnp3DataSource.java / line 149	Printing the users' details in the browser while the debugging mode is activated.	Use the Loggers framework.	Low	Yes
CWE - 379 - Creation of Temporary File in Directory with Insecure Permissions	ReportWorkItem.java / line	A malicious user could access, modify or create a file, which shares the same name as the temporary file's.	The temporary file should be written to a directory that is not world readable.	Low	Yes
CWE - 379 - Creation of Temporary File in Directory with Insecure Permissions	ReportChartCreator.java / line	A malicious user could access, modify or create a file, which shares the same name as the	The temporary file should be written to a directory that is not world readable.	Low	Yes

		temporary file's.			
CWE - 1022 - Use of Web Link to Untrusted Target with window.opener Access	eventList.jsp / line	Putting a link on a site that opens up to their malicious website, prompting the user to enter their credentials.	Adding the attribute "rel=noopener" to the link tag.	Low	Yes
CWE - 353 - Missing Support for Integrity Check	publicView.jsp / line	Importing a resource (e.g. an external library) of which we are not sure if it has been compromised or not by a malicious user.	An integrity attribute containing the hash of the file should be added to the script tag.	Low	Yes

3. Penetration Testing

In the same manner, we perform penetration testing over the whole Mango system using OWASP ZAP. This software scans the application in two manners, passively and actively. While the first only reads request outputs to find vulnerabilities, the second launches attacks against the application such as SQL injections and XSS over the crawled URLs. In this section, we will first focus on the configuration of ZAP with Mango to act as a logged-in user while performing the attacks. Secondly, after running multiple scans across the application, we shall discuss the discovered security risks identified by CWE and the potential risk they cause in Mango as well as how they can be solved.

3.1. Manual

The following steps explain how to perform the penetration testing with OWASP ZAP on Mango. This assumes you already installed Mango along with ZAP. Firefox and ZAP should also be configured following the instructions given in the lab.

1. Open Mango in Firefox and ZAP
2. Login into Mango using **admin** as both username and password
3. In the *Sites* tab of ZAP, click on *http://localhost:8080*, *test* then right click on *POST:login.htm()(password, username)*. This is the entry point of Mango, captured after logging in
4. Select *Include in context* then *New context*. This will open a pop-up window.
5. Click on *POST:login.htm()(password, username)* and change *Context Name* to *mango login*.
6. In *Authentication*, select *Form-based Authentication*, This will open new options in the menu.

7. To configure the authentication method, select *POST:login.htm()(password, username)* in *Login Form Target URL*. The parameters will be discovered automatically.
8. Logout from Mango and open the inspector on Firefox. Select any element from the page that does not appear while being logged in, such as browser compatibility check or the username field. Copy the outer html and paste it into the *Regex pattern to identify Logged Out messages* field of the *Authentication* page of the pop-up menu in ZAP. You can then log back in.
9. Click on *Users* and *Add,* enter *Admin* as *User Name* and admin in both *password* and *username* fields.
10. Click *OK* to close the menu and save changes.
11. Right-click on *POST:login.htm()(password, username)* on the *Sites* tab and select *Flag as Context*, *mango login: Form-based Auth Login Request* then click *OK*.
12. On the toolbar of ZAP, enable *Forced User Mode*.
13. To start the attack, right-click again on *POST:login.htm()(password, username)* and *Attack*, then *Active Scan*. A pop-up window should appear.
14. To perform attacks as a logged-in user, choose *mango login* for *Context* and *admin* for *User*.
15. Click on *Start Scan* to perform the attacks. In the *Output* tab, a message saying *Authentication successful* should appear.
16. It is recommended to do some manual walkthrough of each page found in the Mango web app and perform scans on them. However, it is not necessary to perform the login procedure with ZAP again as long as you stay logged in.

Some problems were encountered while setting up Mango and Zap. The following list identifies them and provide the associated fixes:

- Mango can disconnect itself from time to time while running locally. In order to resume the current session, reload the page, disconnect and reconnect before pursuing the attacks and scans.
- If http://localhost:8080 is blank or shows a connection error, restart ZAP and close your browser. Try connecting again by restarting ZAP first and Firefox along with Mango afterwards. It can sometimes be helpful to open Mango on a browser with default settings (i.e. Chrome) to see if connection problems come from Mango itself or ZAP Proxy.

3.2. Results

The table below summarizes the security alerts identified using OWASP ZAP for the entire Mango system.

Table 2: Summary of the identified security alerts using OWASP ZAP for the entire Mango system

CWE Type	Risk level	Sample URL	Description
79 - Cross Site Scripting (Reflected)	High	http://localhost:8080/test/login.htm	Code can be injected in the browser by an attacker. While visiting the site, the victim can execute the code, which will then run in the context of the current site they are visiting.

693 - Content Security Policy Header not set	Medium	http://localhost:8080/test/data_sources.shtm	Without a CSP header, acting as a security layer against XSS and injection attacks, it makes the job much easier for an attacker.
352 - Absence of Anti-CSRF tokens	Medium	http://localhost:8080/test/sql.shtm	No tokens against Cross-Site Request Forgery were found.
1021 - Missing anti-clickjacking header	Medium	http://localhost:8080/test/compound_events.shtm	The application could be infected by an XSS attack containing clickjacking elements.
91 - XSLT Injection	Medium	http://localhost:8080/test/	It is possible for an attacker to inject XSLT code in the system, allowing it to read and write XML files in the browser.
1275 - Cookie without SameSite attribute	Low	http://localhost:8080/test/watch_list.shtm	A cookie was found without the SameSite attribute, which means it could be sent as a response to an XSS attack or cross-site request forgery.
200 - Private IP disclosure	Low	http://localhost:8080/test/data_sources.shtm	A private IP from another service has been found in the response body of a request.
693 - X-Content-Type-Options Header Missing	Low	http://localhost:8080/test/compound_events.shtm	The X-Content-Type-Options Header was not set to “nosniff” mode, allowing older browsers to do MIME-sniffing (KeyCDN, 2018).

OWASP highlights the top ten vulnerabilities (OWASP, 2021) found in web applications to help identify the most common security issues and be aware of them during conception and testing. As we will see, some of them have been found in the Mango system. The following list enumerates these vulnerabilities, in decreasing order of frequency:

1. Broken Access Control
2. Cryptographic Failures
3. Injection
4. Insecure Design
5. Security Misconfiguration
6. Vulnerable and Outdated Components
7. Identification and Authentication Failures
8. Software and Data Integrity Failures
9. Security Logging and Monitoring Failures

10. Server-Side Request Forgery

Knowing the different major vulnerabilities, we can then review the identified ones in Mango.

1. 79 - Cross-Site scripting (reflected): Cross-site scripting is an injection attack in which the attacker's code gets sent to a victim by the bias of a vulnerable web application. The malicious code sent can then be executed inside the vulnerable application, where it cannot assess if the script can be trusted or not. Therefore, it becomes possible to steal personal information from the client. In the case of Mango, it has been found possible to execute code inside both username and password fields in the login page. Sending *"><script>SOME CODE</script>* did execute the code in the browser. This is a case of an **injection** attack as identified by OWASP Top-10. A simple solution to this problem is to use a vetted library or framework that helps avoid this vulnerability.

2. 693 - Content Security Policy Header not set: Content Security Policy (CSP) helps the detection and mitigation of certain types of attacks, like Cross Site Scripting and data injection. It provides a set of standard HTTP headers that allow website owners to identify which sources of content should be allowed by the web browser. A potential risk of this vulnerability is that without CSP, it can be hard to identify legitimate sources of content, which could lead to data theft, for example. It is a case of a **Security Misconfiguration** attack as identified by OWASP Top-10. This can be fixed by making sure that the web server/application server is configured to set the Content-Security-Policy header.

3. 352 - Absence of Anti-CSRF tokens: A cross-site request forgery (CSRF) is an attack that consists of sending an HTTP request to a target destination as an unknowing or unwilling victim. It exploits the situations where a web site trusts the user. A potential risk of this vulnerability is that a victim could unknowingly send sensitive data to an external source. It is a case of a **Broken Access Control** attack as identified by OWASP Top-10. A possible solution could be to use a vetted library or framework that helps avoid this vulnerability.

4. 1021 - Missing anti-clickjacking header: This vulnerability is exposed when a response does not include either Content-Security-Policy with 'frame-ancestors' directive or X-Frame-Options to protect against 'ClickJacking' attacks. Clickjacking is an attack that consists of tricking a user into clicking a webpage element that is either invisible or masked. For instance, it could retrieve credit card information or password data. It is a case of a **Security Misconfiguration** attack as identified by OWASP Top-10. Since modern browsers support CSP and X-Frame-Options headers, a solution to this is to make sure that all web pages returned by the website/app have one of the headers set for them.

5. 91 - XSLT Injection: This attack consists of an **injection** using XSL transformations. XSLT is a language designed from transforming an XML document into another XML document. Hence, it is possible for an attacker to inject XSLT code in the system, allowing to read and write XML files in the browser and even read system information and local files, thus leading to the theft of personal information or infecting the client. It became possible to get the vendor name using the following attack: *http://localhost:8080/test/ %3Cxsl:value-of%20select=%22 system- prop - erty ('xsl:vendor') %22%2F%3E*, which returned *Apache* as vendor name. A fix would be to use input sanitization on client side and validation to avoid such requests.

6. 1275 - Cookie without SameSite attribute: A cookie was found without the SameSite attribute, which means it could be sent as a response to an XSS attack or cross-site request forgery.

Typically, this attribute is effective to protect against such attacks. Knowing that we found both vulnerabilities in the system, the absence of the cookie attribute makes the job easier for an attacker. It is a case of **Security Misconfiguration** as identified by OWASP Top-10. To fix this, set the 'SameSite' attribute to either 'lax' or ideally 'strict' for all cookies.

7. 200 - Private IP disclosure: Mango server disclose private IP addresses. These can give information about the internal network structure of the server. This information can be used by a hacker to break into the system. This can be fixed by removing the IP address from the response body and by using JSP/ASP/PHP comments instead of HTML/Javascript.

8. 693 - X-Content-Type-Options Header Missing: "X-Content-Type-Options" is a header used in the MIME (Multipurpose Internet Mail Extensions), when not enabled the browser figures out the type of the content by itself. It can lead to XSS attacks. For example, in the case where the header is missing, if the browser accepts a video, but receives a file containing a script, it will run the file in order to determine the type of the content and then run the script. It is a case of **Security Misconfiguration** as identified by OWASP Top-10. This can be fixed by enabling the header : 'X-Content-Type-Options=nosniff'.

4. Comparison of static analysis and penetration testing

In this section, the penetration testing results are compared with the static analysis results in order to display the different vulnerabilities detected by both approaches.

First of all, the penetration testing and the static analysis approach did not detect any common vulnerabilities from the same CWE categories but we did find vulnerabilities from the same OWASP category.

Indeed, both approaches detected similar vulnerabilities in the injection category. They found different types of injection, the penetration testing found multiple XSS injection vulnerabilities while the static analysis only found some SQL injection vulnerabilities as displayed in Table 1 and 2. The reason that the penetration test did not detect SQL injection even though it has been identified as a high risk, is because it requires manual operations (active scanning from ZAP) thus we might have missed the specific vulnerabilities detected by the static analysis. Furthermore, the static analysis is an automatic process that goes through code source to find vulnerabilities with predefined templates thus it might have been inaccurate for the SQL injections. Also, it could have been for the same reason that it did not detect any XSS injections.

On the opposite of penetration testing, the static analysis found multiple vulnerabilities outside of the OWASP injection category. Indeed, as shown in Table 1, the static analysis found vulnerabilities in those CWE categories: hard-coded password, weak pseudo-random, creation of temporary file in directory with insecure permissions, use of web kink to untrusted target with window.opener access and insertion of sensitive information into debugging code. The reason that the penetration approach was not able to detect those vulnerabilities is because the penetration tests that we performed were limited in resources as if it was a customer using the mango software since the tests were performed in the form of a black box. Thus, it is easier to detect those vulnerabilities with static analysis since it has direct access to the source code unlike the penetration tests. For example, a hard code password in the source code can not be found unless there is access to the source code. As for the insertion of

sensitive information into debugging code it can also only be found by having the source code unless the application is running in debugging mode.

5. Conclusion

In conclusion, we performed a static analysis using *SonarQube* to identify security vulnerabilities and we classified them using the community-developed list Common Weakness Enumeration. 47 *Security Hotspots* were identified, among them, 44 are classified as Low, 1 as Medium and 2 as High. No *Vulnerabilities* were detected, as per SonarQube definition (issues requiring immediate attention). While the static analysis managed to detect many security concerns that need to be fixed (6 out of 8 reviewed hotspots), it also missed out from detecting an actual vulnerability (the one pertaining to hard-coded passwords), which we found only upon manually inspecting the code (outside of the code pointed out in hotspots). Secondly, we performed penetration tests on the entire Mango system using OWASP ZAP to identify vulnerabilities with both passive and active scanners, which lead to further analysis and potential fixes. Furthermore, we compared the results from static analysis and penetration testing and highlighted their differences in discovered vulnerabilities and how they could be explained. We performed a high coverage of Mango's security by combining multiple approaches, giving the future developers of the system a clear overview of the vulnerabilities and their fixes to obtain a much more secure application.

6. References

Common Weakness Enumeration. (2022a). *About CWE*. MITRE. https://cwe.mitre.org/about/index.html

Common Weakness Enumeration. (2022b). *About CWE*. MITRE. https://cwe.mitre.org/data/definitions/259

Common Weakness Enumeration. (2022c). *About CWE*. MITRE. https://cwe.mitre.org/data/definitions/89

Common Weakness Enumeration. (2022d). *About CWE*. MITRE. https://cwe.mitre.org/data/definitions/338

Common Weakness Enumeration. (2022e). *About CWE*. MITRE. https://cwe.mitre.org/data/definitions/215

Crowdstrike. (2022). *JAVA LOGGING GUIDE: THE BASICS*. https://www.crowdstrike.com/guides/java-logging/

GeeksForGeeks. (2022). *Random vs Secure Random numbers in Java*. https://www.geeksforgeeks.org/random-vs-secure-random-numbers-java/#:~:text=Size%3A%20A%20Random%20class%20has,which%20the%20seed%20was%20generated

ISO/IEC 25010. (n.d.). ISO 25000. https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

KeyCDN. (2018). *What is MIME Sniffing?* https://www.keycdn.com/support/what-is-mime-sniffing

OWASP. (2021). *OWASP Top-Ten*. https://owasp.org/www-project-top-ten/

OWASP Cheat Sheet Series. (2022). *SQL Injection Prevention Cheat Sheet*. Cheat Sheet Series Team. https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

SonarQube. (2022). *Security Hotspots*. https://docs.sonarqube.org/latest/user-guide/security-hotspots/

W3Schools. (2022). *HTML <script> integrity Attribute*. https://www.w3schools.com/tags/att_script_integrity.asp

ZAP Documentation. (2022). *Content Security Policy (CSP) Header Not Set*. https://www.zaproxy.org/docs/alerts/10038/