Department of computer and software engineering

LOG8371E: Software Quality Engineering

TP3: Security

Submitted by **Group 2**

Jérôme Cléris - 2017719

Philippe De Blois - 2022298

Louis-Alexandre Hébert - 2015537

Anthony Ileret - 2114076

Christophe St-Georges - 2024043

Submitted to P^{r} Heng Li

Fall 2022

December 1st, 2022

Tables

Table of Contents

Abstract **3**

1. Introduction **4**

2. Static Analysis **4**

Process 4

Results 5

Discussion on challenges and solutions 9

3. Penetration Testing **9**

Process 9

Results 10

Discussion on challenges and solutions 14

4. Comparison of Static Analysis and Penetration Testing **14**

5. Conclusion **15**

References **16**

Table of Figures

2. Static Analysis **4**

Results 5

Figure 2.1 Summary of static analysis results 5

Figure 2.2 Code of the first vulnerability 6

Figure 2.3 Code of the second vulnerability 6

Figure 2.4 Code of the third vulnerability 7

Figure 2.5 Code of the fourth vulnerability 7

Figure 2.6 Code of the fifth vulnerability 7

Figure 2.7 Code of the sixth vulnerability 8

Figure 2.8 Code of the seventh vulnerability 8

Figure 2.9 Code of the eighth vulnerability 8

3. Penetration Testing **9**

Results 10

Figure 3.1 Summary of the penetration test results 10

Figure 3.2 Overview of the first vulnerability 10

Figure 3.3 Overview of the second vulnerability 11

Figure 3.4 Overview of the third vulnerability 11

Figure 3.5 Overview of the fourth vulnerability 12

Figure 3.6 Overview of the fifth vulnerability 12

Figure 3.7 Overview of the sixth vulnerability 13

Figure 3.8 Overview of the seventh vulnerability 13

Figure 3.9 Overview of the eighth vulnerability 14

Abstract

Introduction: The objective of this paper is to describe Mango's security aspect with static code analysis and penetration testing.

Method: Firstly, we want to identify different security issues in the Mango project. To do so, we start with a static analysis of the source code of the project. This means we will use a tool such as SonarCloud[1] to analyze the code and detect vulnerabilities at different levels. Then, a different approach will be used: penetration testing. A tool such as OWASP ZAP[2] will be used to attack the system to find different security failures. Finally, a comparison of those two approaches will be done to clarify the differences of the given results.

Results:
While performing the static analysis of the code, the SonarCloud tool detected 4 vulnerabilities and 569 security hotspots. Those were mostly related to injection (SQL injection, XSS injection, etc.), but also some were related to denial of service, weak cryptography and even some broken access control. In the penetration testing phase of our analysis, the OWASP ZAP tool discovered 111 security vulnerabilities. The security issues have a variety of categories such as cross-site scripting, SQL injection, information leak, content security policy and buffer overflow.

Conclusion: To conclude, we have found that the Mango system had quite a lot of security hotspots and vulnerabilities of many types, ranging from cross-site scripting issues to cryptography issues, as well as unauthorized access to sensitive information. Cross-site scripting issues were found by both testing approaches, while some others were only found either through static analysis or penetration testing. It is thus preferable to combine different approaches to find as many security issues as we can.

1. Introduction

Mango is a software that offers a machine-to-machine solution to manage sensors, collect data, and visualize collected data. Since sensors are used for critical and high-precision work, for instance, to monitor the temperature of a greenhouse or to collect data for laboratory experiments, the Mango software must be of high quality. High-quality software contains a low number of bugs, corresponds to the requirements, is reliable, is maintainable, is performant and is secure. This work will aim at evaluating the entire Mango system security. The evaluation will be made in two parts. First, a static analysis of the system will be done. Sonar Cloud and Sonar Scanner are two free tools to perform static analysis. Those two tools will be used to conduct the analysis of the system [1][2]. The second part is to conduct penetration testing of the entire system. The penetration testing will be made with the OWASP ZAP tool. OWASP ZAP is a proxy and security scanner used mostly for penetration tests [3]. For each part of the evaluation, a manual guide will be issued, a summary of the results will be presented, eight vulnerabilities will be explained and challenges will be discussed. Finally, a comparison of the two methods will be made.

2. Static Analysis

Process

To conduct the static analysis of the Mango system, two tools were used. The first tool, Sonar Cloud, is a SaaS tool to visualize results of static analysis. The second tool, Sonar Scanner, is an open-source command line utility that can perform static analysis of local folders.

The first step to perform the static analysis of the system is to configure Sonar Cloud. Indeed, an organization must be created. The organization will host the project and has teamwork capabilities. The organization can be created manually or imported from github. When creating an organization a key must be specified and a plan chosen. log8371group2 was the organization key and the free plan was chosen. When the organization configuration is done, a project must be created. When creating a project, an organization must be selected, a key chosen and a display name chosen. We put our project in the log8371group2 organization, we chose log8371tp3group2 as project key and as display name. When the project is created, it must be configured to fetch static analysis data. We chose the manual configuration. Then, we chose the “Other” language option and the Linux operating system

(take tour operating system). Also, the token or environment variable value must be copied and saved for later.

The second step is the static analysis environment configuration. Sonar Scanner command line utility must be downloaded and installed on your system. Refer to the right documentation to install Sonar Scanner on your system [2]. Also, you must add an environment variable for the Sonar Cloud token in your system. In linux, we did it with:

```
export SONAR_TOKEN=15451d9734bcc622de293918944a7d46772c0ec8
```

When all the configuration is done, Sonar Scanner can be runned. The parameters that need to be defined are the organization, the project key, the Sonar Cloud URL and the Java binaries path. The last parameter is very important since Sonar Scanner will not scan Java files and since Mango is manly programmed in Java. In linux, the command should look like:

```
./sonar-scanner
-Dsonar.organization=log8371group2
-Dsonar.projectKey=log8371tp3group2
-Dsonar.sources=~/TP3-LOG8371E/mangoSource
-Dsonar.host.url=https://sonarcloud.io
-Dsonar.java.binaries=~/TP3-LOG8371E/mangoSource/build/WEB-INF/classes/*
```

When the Sonar Scanner execution is done and if all the previous steps are done correctly, the results should appear in your Sonar Cloud project.

Results

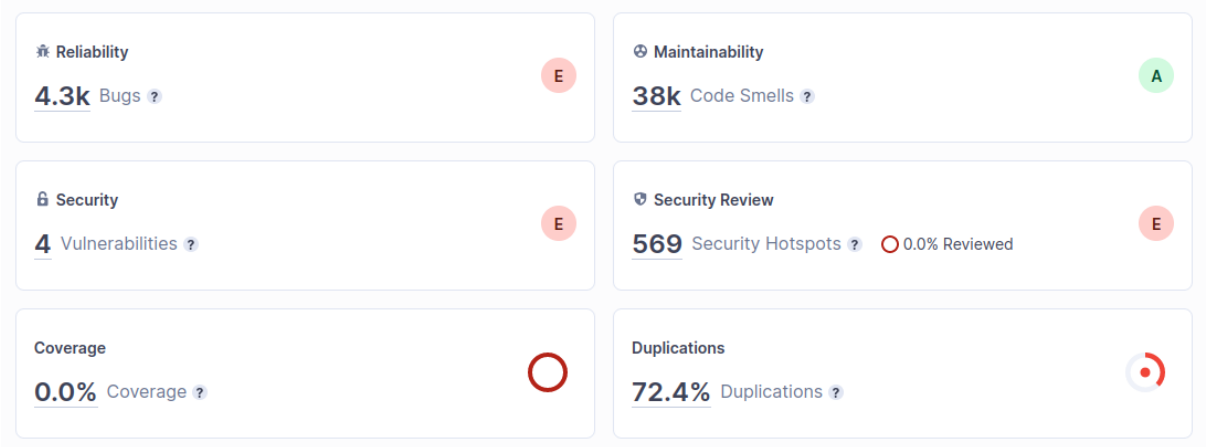


Figure 2.1 Summary of static analysis results

The figure above shows that there are a few confirmed vulnerabilities and, considering that the mango system has above 362k lines of code, a reasonable amount of security hotspots detected by the static security scan. Below we will discuss eight security hotspots or vulnerabilities that were found using the Sonar Cloud tool.

Vulnerability #1

```
opts[sp[0]] = 7 eval( 6 sp[1]);
```

Figure 2.2 Code of the first vulnerability

In build/resources/dojo/src/io/RepubsubIO.js there is a Blocker vulnerability. According to OWASP, this case is an injection issue (OWASP Top Ten A3) and more precisely a cross-site scripting vulnerability (CWE-79). The problem is that data that is entered by a user is stored in a variable. Then, this variable is reused in other variables, the final variable is used inside a Javascript eval call [7]. The eval function executes Javascript code as a string. So, it is dangerous since a user could manipulate what he enters in that first variable that will be later called by the eval and run it if it is code. Eval has the same permissions as the caller of the eval function. This means that malicious code can be injected in the program and run easily with the right permissions. Even if it can be tempting to use eval to dynamically execute code, it should be avoided. So the solution could be to remove the eval and change the code accordingly. Another way to fix this could be to validate what can be input by the user to prevent any special characters that could lead to scripting.

Vulnerability #2

```
mango.longPoll.pollSessionId = Math.round(Math.random() * 1000000000);
```

Figure 2.3 Code of the second vulnerability

In build/resources/common.js Weak Cryptography there is a Critical vulnerability. It is an instance of PRNG (CWE-338) or Cryptographic failure (OWASP Top Ten 2021 A2). The problem is that the Math.random() function is used to generate the poll session id. This means that an attacker can potentially predict a future or a list of poll session id and use a predicted token to steal a user session or potentially elevate its permission to gain admin access to Mango. The proposed solution is to change the Math.random() function by one that is not pseudo random. For example, the crypto.getRandomValues() would be a great choice to replace the original random generator.

Vulnerability #3

```
exportFile = File.createTempFile("tempCSV", ".csv");
reportCsvStreamer = new ReportCsvStreamer(new PrintWriter(new FileWriter(exportFile)), bundle);
```

Figure 2.4 Code of the third vulnerability

In src/com/serotonin/mango/vo/report/ReportChartCreator.java there is a Critical vulnerability. This is a Broken Access Control problem (OWASP Top Ten 2021 A1), which is more precisely an insecure temporary file (CWE-377). The problem is the creation of a public writable file. A malicious attacker could create the file before the app. This can also be a problem with elevated privileges, the attacker could access files that he shouldn't or even modify or delete files. A solution to this is to create a file with less permissions or use an API made to create secure temporary files.

Vulnerability #4

```
private void copyTable(Connection sourceConn, Connection targetConn, String tableName) throws SQLException {
    LOG.warn(" --> Converting table " + tableName + "...");

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

Figure 2.5 Code of the fourth vulnerability

In src/com/serotonin/mango/db/DBConvert.java there is a Major vulnerability. It is an instance of Injection (OWASP Top Ten 2021 A3). Specifically, it is an SQL Injection (CWE-89). It is problematic because the incoming tableName field is not validated and used directly in an SQL query. This means an attacker could inject SQL code and affect the database by dropping tables and inserting data. The attacker could also get access to data from other tables. To correct the vulnerability, input validation and input sanitization mechanisms should be added to make sure the tableName field only contains a table name and to make sure it does not contain SQL code.

Vulnerability #5

```
eval("var data = "+ dyn.value);
```

Figure 2.6 Code of the fifth vulnerability

In war/resources/view.js there is a Critical vulnerability. It is an instance of injection (OWASP Top Ten 2021 A3). Precisely, it is an eval injection or an improper neutralization of directives in dynamically evaluated code (CWE-95). The danger here is that it might allow code to be injected in the eval call and let an attacker run malicious code. To fix this problem, the dynamic code could be to use hard-coded code instead if possible here. If that is not

possible, the best solution is to run the eval part inside a controlled environment to prevent access.

Vulnerability #6

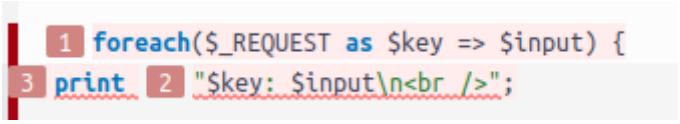


Figure 2.7 Code of the sixth vulnerability

In build/resources/dojo/tests/widget/showPost.php there is a Blocker vulnerability. It is an instance of cross-site scripting (OWASP Top Ten 2021 A7). Specifically, it is a reflected cross-site scripting (CWE-79). It is problematic because the field input is not validated and directly printed in an html page. This means an attacker could use a “script” tag in its request and execute javascript code on the page. To correct the problem, user data validation and sanitization mechanisms should be implemented to make sure the input field does not contain javascript code or “script” tag.

Vulnerability #7

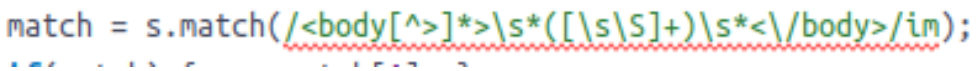


Figure 2.8 Code of the seventh vulnerability

The code above, located in build/resources/dojo/dojo.js.uncompressed.js, is a potential Critical hotspot. It is an instance of Injection (OWASP Top Ten 2017 A1) or more precisely a potential uncontrolled resource consumption (CWE-400). The regex is made to match with HTML files. The problem is that, since the regex contains “*” characters, it can match with very large HTML files. If the files are too large, the system could use all its resources to evaluate the regular expression, which could help an attacker to create a denial of service attack against the Mango system. To solve the problem, a reasonable limit could be used in the regular expression instead of the “*” characters.

Vulnerability #8

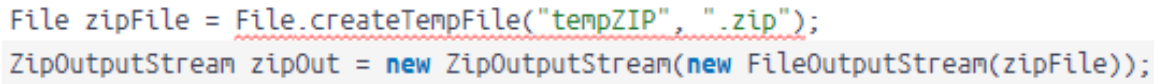


Figure 2.9 Code of the eighth vulnerability

In src/com/serotonin/mango/rt/maint/work/ReportWorkItem.java there is a Critical vulnerability. This is a Broken Access Control problem (OWASP Top Ten 2021 A1). Specifically, it is an insecure temporary file (CWE-377). The concern is that a malicious person could pre-create the file with the same name before the application. Because of race

conditions, this could allow the attacker to access, modify or even delete files that he shouldn't. The best solutions are using an API made to create secure temporary files or using a low permission folder to create temporary files in it.

Discussion on challenges and solutions

Using Sonar Cloud and Sonar Scanner was not very difficult. Indeed, the documentation of both those tools is very clear which helps using the tools more efficiently. The one challenge we faced was that Sonar Scanner is not natively able to scan Java files. Furthermore, Sonar Scanner can only scan Java class files. Since we did not check in the documentation when initially running the scan, we did not understand why the scan was failing. With the help of Google, StackOverflow and the documentation (and also a few trials and errors), we were able to identify the right Sonar Scanner parameter and format it correctly to have a successful scan.

3. Penetration Testing

Process

In order to perform penetration testing, we used the OWASP ZAP attack proxy. In order to setup OWASP ZAP to work with Mango, we first deployed Mango, then started Firefox from ZAP's dedicated button, which automatically configured the proxy of said Firefox instance [4]. We then logged into Mango, and used the generated POST URL in ZAP to create a new context for our penetration testing session [5]. In the context's options, we set the authentication method for ZAP to use to be form-based authentication. We then created a new user with "admin" as both its username and password, as those are the credentials we used to log into Mango initially. We then set up the regex pattern to be used for detecting a logged out state to be "Firefox 107 on Windows". After that, we right-clicked the previously mentioned POST URL in ZAP and flagged it as our new context [5]. Finally, we enabled Forced User Mode. We then started to crawl Mango's pages (passive scan) and ran an active scan to detect additional vulnerabilities.

Results

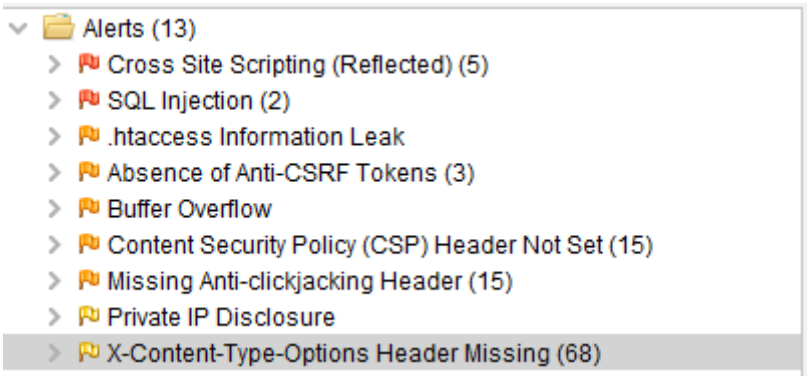


Figure 3.1 Summary of the penetration test results

As we can see, Mango is quite vulnerable to a myriad of different types of attacks. We'll focus on eight specific attacks noted by ZAP.

Vulnerability #1

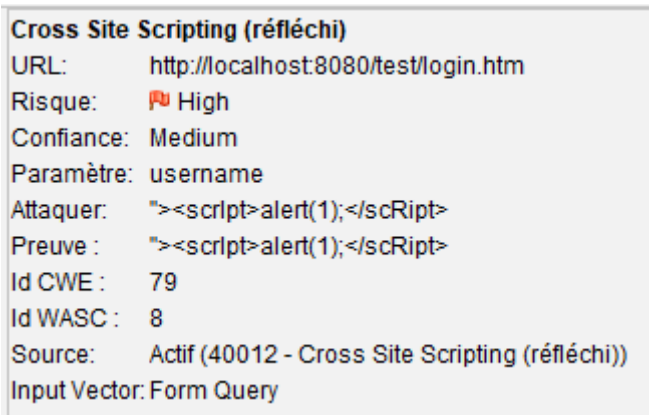


Figure 3.2 Overview of the first vulnerability

This vulnerability means that whatever string we input into the username field of the login form will be reused as-is when generating the page's HTML structure. Given this knowledge, it is possible to insert a <script> tag, allowing us to execute arbitrary code. One way to fix this would be sanitizing the input string of the username field. Another would be to just not generate HTML using the input string of the login form, as it should not be needed anyways (it should only be used for authenticating).

Vulnerability #2

Cross Site Scripting (réfléchi)	
URL:	http://localhost:8080/test/login.htm
Risque:	High
Confiance:	Medium
Paramètre:	password
Attaquer:	"><scrIpt>alert(1);</scRipt>
Preuve :	"><scrIpt>alert(1);</scRipt>
Id CWE :	79
Id WASC :	8
Source:	Actif (40012 - Cross Site Scripting (réfléchi))
Input Vector:	Form Query

Figure 3.3 Overview of the second vulnerability

This vulnerability means that whatever string we input into the password field of the login form will be reused as-is when generating the page's HTML structure. Given this knowledge, it is possible to insert a <script> tag, allowing us to execute arbitrary code. One way to fix this would be sanitizing the input string of the password field. Another would be to just not generate HTML using the input string of the login form, as it should not be needed anyways (it should only be used for authenticating).

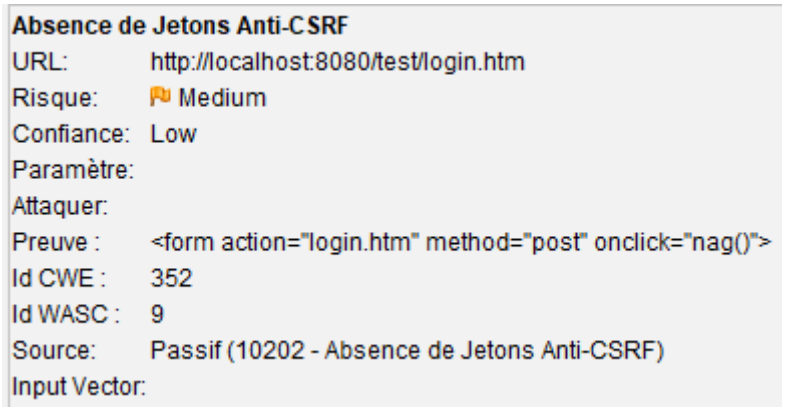
Vulnerability #3

SQL Injection	
URL:	http://localhost:8080/test/dwr/call/plaincall/DataPointDetailsDwr.getImageChartData.dwr
Risk:	High
Confidence:	Medium
Parameter:	c0-id
Attack:	0 AND 1=1 --
Evidence:	
CWE ID:	89
WASC ID:	19
Source:	Active (40018 - SQL Injection)
Input Vector:	Direct Web Remoting

Figure 3.4 Overview of the third vulnerability

This vulnerability means that we are able to run arbitrary SQL commands against the database, meaning we can drop tables and corrupt the SQL database. Fixing this can be done by sanitizing the input or using parameterized queries.

Vulnerability #4



Absence de Jetons Anti-CSRF	
URL:	http://localhost:8080/test/login.htm
Risque:	Medium
Confiance:	Low
Paramètre:	
Attaquer:	
Preuve :	<form action="login.htm" method="post" onclick="nag()">
Id CWE :	352
Id WASC :	9
Source:	Passif (10202 - Absence de Jetons Anti-CSRF)
Input Vector:	

Figure 3.5 Overview of the fourth vulnerability

Anti-CSRF (Cross-site request forgery) tokens are not found in the HTML submission form. A CSRF attack forces a user to execute unwanted actions on a web application in which they are currently authenticated. An anti-CSRF token pair would allow the user to validate their requests and prevent ones from attackers.

Vulnerability #5

Buffer Overflow	
URL:	http://localhost:8080/test/data_point_details.shtm?dpid=78
Risk:	Medium
Confidence:	Medium
Parameter:	dpid
Attack:	eJrXWdqEejlRfrUTCSLeShybJlPcPCOnxukHtWuHoHXHKiW wlIQBfCmRowfosTaVPhtloZteAHhPHBXGSwHFXkyagTjWLF uGlrNlujamKLPBlOCstvkOuaYSCPgFdKUTOXowxpbVibWXc eHangQPZiJpsLofsrbUUnlnbvFplTjpXkHwHACGLHONjQHF ZOxDntldgXiHmvOZrAPrdPpEusEcKiPTwWdSiuIsVrYVBUyG dncGnKWQNCHqXkBKluwsxxZhcLgWsNeKhJBfYalCwHwSs hQfEYBCcDBDAwZJWClilvYvPWjyWnMcqZDAJDtxLewXdNLF bpQblKoHvYyBPnTSlAcdYtXeUVksiQwhsIRuIZQDSxOhXvuC mvmvGAlbYBgrNKtJQVjLwXmkHYJseudSITeoMCIUEfycVwd gVhTiyDbvCMLkGknjcwgeXRJrxyEhNuQhgjGXcwZRyxbLTEx hqQxeFXLFgWlgYFQfdhWYIiZIAScikydUAUIWyPCjCKpWLBd
Evidence:	Connection: close
CWE ID:	120
WASC ID:	7
Source:	Active (30001 - Buffer Overflow)
Input Vector:	URL Query String

Figure 3.6 Overview of the fifth vulnerability

This buffer overflow vulnerability means that Mango's backend is filling up a buffer, but not checking the buffer size to ensure that it matches the size of the data destined to be inserted into it. This can lead to a multitude of issues such as a server crash, or worse, potentially leaking some sensitive information. This is similar in nature to what caused the infamous

SSL Heartbleed issue. Adding a check on a buffer's size before filling it should prevent such errors.

Vulnerability #6

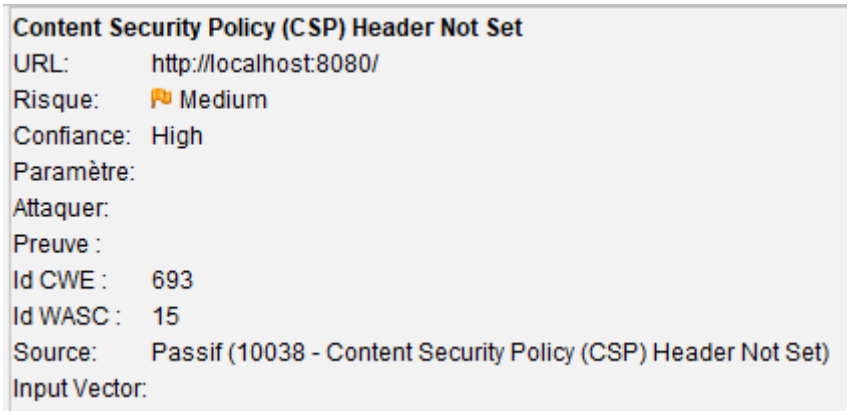


Figure 3.7 Overview of the sixth vulnerability

Content Security Policy (CSP) adds a layer of security to help detection of mitigation of specific types of attacks. It provides standards for HTTP headers that allow servers to declare approved sources of content that can be loaded. Fixing this alert simply consists of making use of such headers, in order to prevent unapproved sources of content from being used.

Vulnerability #7

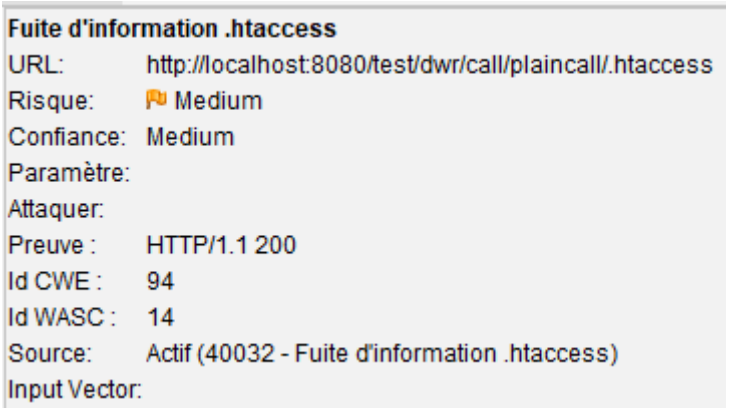


Figure 3.8 Overview of the seventh vulnerability

This vulnerability means that we can get access to the .htaccess configuration files of the Apache web server. We could then edit the server's configuration to alter its functionalities, characteristics and behaviors for malicious intent. To fix this, access to the .htaccess config should be restricted to backend administrators only, and only be modifiable from the backend host machine.

Vulnerability #8

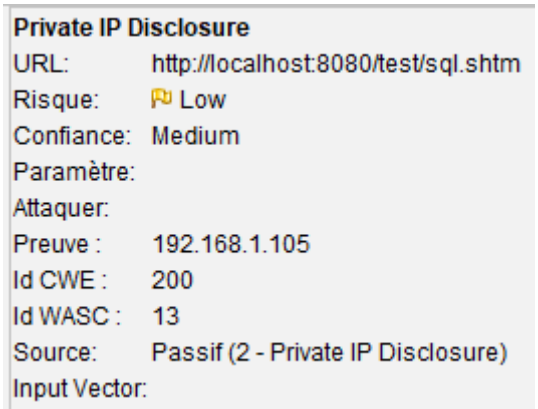


Figure 3.9 Overview of the eighth vulnerability

This vulnerability gives information about the server's internal system to the user. A malicious user could use this information to attack other parts of the system. To prevent this from happening, the server should remove the private IP address from the response body it gives to the requests.

Discussion on challenges and solutions

Getting OWASP ZAP to work wasn't exactly intuitive, as there were multiple steps involved in setting it up correctly, which we did not successfully do on our first go. It did not help that many links referenced in the lab's requirements as well as the Moodle tutorial were initially broken, meaning we had to piece together information from multiple sources, which all had their own quirks. Fortunately, after asking the lab assistant and receiving an email from the professor, we were able to get the links and information we needed.

4. Comparison of Static Analysis and Penetration Testing

Comparing the results obtained using both approaches, we can see that both approaches were able to identify some cross-scripting and SQL injection opportunities, which are the most dangerous vulnerabilities.

However, not all vulnerabilities were found by both approaches. For example, the static analysis approach was able to identify cryptography issues, which ZAP did not find. This is likely due to the fact that SonarCloud had access to the underlying code of the application, and could thus more easily detect flaws in the cryptography methods used. Likewise, SonarCloud was able to detect the creation of a publicly writable file by "reading" the code, which ZAP could not detect since we did not execute the related code during our penetration

testing. Overall, SonarCloud was able to identify a plethora of cross-site scripting issues, mainly due to the fact that the source code contained many “eval” JavaScript statements.

Despite that, ZAP was still able to find other types of vulnerabilities. For example, it was able to detect and prove the existence of a buffer overflow. It also found that it had access to potentially sensitive files or data, such as private IPs or the .htaccess config file of the Apache web server. It could also analyze packet data and detect that some were poorly constructed or lacked some useful security measures, as they were missing both Content Security Policy headers and Anti-CSRF tokens.

Since both approaches were able to find different types of vulnerabilities, we can conclude that to conduct the best security analysis of a system, both approaches must be used in combination. This combination will allow a team to find and correct most of the vulnerabilities of their system.

5. Conclusion

This paper focused on analyzing and testing the Mango system (a machine-to-machine software solution to manage sensors, collect data, and visualize said data), from a security viewpoint. First, we ran a static analysis tool on Mango’s code using SonarCloud, which was able to identify some security vulnerabilities as well as general security hotspots. After that, we used the OWASP ZAP attack proxy to try and detect additional vulnerabilities. Finally, we compared the results obtained using both approaches. We found that while both SonarCloud and ZAP were able to find cross-site scripting issues, some issues were only found by one approach. Namely, cryptography and file write permission issues were found by SonarCloud, while ZAP found a buffer overflow, accessed sensitive information and found that many packets were lacking security measures. It is thus preferable to conduct security analysis by using a variety of different approaches if we want to detect as many security issues as we can.

References

[1] SonarSource. (2007) SonarCloud (Version 9.5) [Online]. Available: https://www.sonarsource.com/products/sonarcloud/

[2] SonarSource. (2007) SonarScanner (Version 4.7). [Online]. Available: https://docs.sonarqube.org/latest/analysis/scan/sonarscanner/

[3] OWASP. (2010) OWASP ZAP (Version 2.12.0). [Online]. Available: https://www.zaproxy.org/

[4] Polytechnique Montréal. (2022) Tutorial for DVWA Penetration Testing with OWASP ZAP. [Online]. Available : https://moodle.polymtl.ca/pluginfile.php/1031368/mod_resource/content/5/Tutorial%20for%20Penetration%20tests%20of%20DVWA%20with%20OWASP%20ZAP.pdf

[5] SecureIca. (2019) Authenticated Scan using OWASP-ZAP. [Online]. Available : https://medium.com/@secureica/authenticated-scan-using-owasp-zap-f0a71dafe41

[6] Cosmin Stefan. (2013) ZAP Tutorial - Authentication, Session and Users Management. [Online]. Available : https://www.youtube.com/watch?v=cR4gw-cPZOA

[7] Mozilla. (2022) eval(). [Online]. Available : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

[8] Mitre. (2022) CWE. [Online]. Available : https://cwe.mitre.org/

[9] OWASP. (2022) OWASP Top Ten. [Online] Available : https://owasp.org/www-project-top-ten/