University of Trento
Department of Computer Science
Security Testing

Report

Author: Marcel Pascal Stolin
Matricola: 229702
E-Mail: `marcelpascal.stolin@studenti.unitn.it`
XLS Report: `SecurityTesting_Project_2023-2024(.ods,.xlsx)`
Dependency Check Report: `dependency-report.html`

Submission date: 04/08/2024

1 Introduction

This project was carried out as part of the final exam for the course Security Testing at the University of Trento for the year 2023/24. The goal of the project was, to present techniques which have been introduced and learned during the course, by performing an analysis to test an application's security degree.

The given application is called *Befloral*, a web application developed using Java EE. *Befloral* is an online shop where customers can buy and review flowers, plants, and bouquets. Additionally, *Befloral* provides an admin interface, where the admin can manage orders (view, edit, and delete), products (view, edit, delete, and create), and inspect the applications' log file. The applications' log includes events made by the application, such as, debugs, warnings, and errors.

The minimum goal was to inspect the application using static and dynamic analysis. For static analysis, *SpotBugs*[1] and *FindSecBugs*[2] have been used. *Zed Attack Proxy*[3] (ZAP) has been used for dynamic analysis. In addition, ZAP was extended with *FuzzDBFiles* and *FuzzDBOffensive*.
SpotBugs and *FindSecBugs* are static analysis tools, specifically implemented to scan Java applications for security vulnerabilities. *FindSecBugs* is an extension to *SpotBugs*, to detect vulnerabilities within Java frameworks and libraries.
Besides static analysis, the application security was further inspected using dynamic analysis with ZAP. ZAP is an open-source vulnerability scanner for web applications. It provides penetration testing through automated exploring, attacking, and reporting of web-related vulnerabilities. Additionally, ZAP provides fuzzing to further stress discovered vulnerabilities. Fuzzing is a dynamic security testing technique that automates the generation of possible inputs to discover as many execution paths as possible. Two types of fuzzing exist, dump and smart fuzzing. Dump fuzzing does not consider any context or state of the software and smart fuzzing has some intelligence it can use to generate inputs. ZAP generates inputs based on dictionaries, templates, or random, hence it is a (advanced) dump fuzzer. To inspect as many execution paths as possible during dynamic analysis, ZAP was extended with *FuzzDBFiles* and *FuzzDBOffensive.* Both are dictionaries, which provide common attack patterns, used to simplify the discovery of vulnerabilities. Using ZAP is considered black-boxed testing because it has no internal knowledge about the internals of the application.
Besides the minimum requirements, *Befloral's* security degree was further investigated by scanning its dependencies for publicly disclosed vulnerabilities. The tool of choice, used to check the application's dependencies is *OWASP Dependency-Check*[4].

[1]`https://spotbugs.github.io/`
[2]`https://find-sec-bugs.github.io/`
[3]`https://www.zaproxy.org/`
[4]`https://owasp.org/www-project-dependency-check/`

2 Adopted Process

This section describes the process adopted for discovering and inspecting application vulnerabilities during the static analysis, dynamic analysis, and dependency check phases.

It is important to mention, that static analysis was applied before dynamic analysis, to comply with the Software Development Life Cycle (SDLC) phases. Further, information gathered during static analysis was helpful to investigate vulnerabilities found during dynamic analysis.

2.1 Static Analysis

As mentioned previously in Section 1, *SpotBugs* and *FindSecBugs* have been used for static analysis. Both tools extend Eclipse by adding a new menu option, that lists all statically discovered vulnerabilities. Additionally, it provides a detailed description of the found vulnerability and ranks the vulnerability based on its threat level. A disadvantage of static analysis is, it tends to be over-restrictive and leads to a higher number of false positives. Therefore, it was required to inspect all reported vulnerabilities manually.

The following routine was applied for each discovered vulnerability during static analysis:

1. A vulnerability was discovered, either by *SpotBugs* or *FindSecBugs*.
2. The vulnerability was further inspected in the source code, to check if it is a true positive or false positive.
3. If the vulnerability was considered a true positive, it was manually tested within the web application.
4. Lastly, the vulnerability was reported to the XLS document.

Some vulnerabilities required further investigation because the reported line (line in source code) during static analysis wasn't accurate. Then, it was required to further investigate the source by following the implementation structure. Afterward, it was possible to create an example payload to exploit the vulnerability.

2.2 Dynamic Analysis

As mentioned in the introduction in Section 1, ZAP provides fuzzing and penetration testing. It allows for automatically scanning the web application (penetration testing) using a traditional and an AJAX spider to discover vulnerabilities. Furthermore, reported vulnerabilities were further stressed using the fuzzing functionalities of ZAP. An advantage of dynamic analysis is, most found vulnerabilities are true positives. However, the disadvantage is, that it tends to be over-permissive and discovers fewer vulnerabilities compared to static analysis.

The following routine was used for dynamic analysis:

1. Penetration testing of *Befloral* was performed using the automated scanner and the AJAX scanner.
2. A reported vulnerability was inspected using the ZAP Firefox web browser.
3. Whenever possible, a vulnerability was further stressed using fuzzing.
4. A detailed description of the vulnerability was reported in the XLS file.

ZAP provides to run Firefox in a special execution mode with a specific ZAP-related configuration and a ZAP Firefox addon. This allows to investigate vulnerabilities through a proxy, for example, to perform a credential-sniffing attack. Furthermore, the fuzzing functionality of ZAP provided a way to stress specific endpoints of *Befloral*, for example, to perform a credential-stuffing, SQL injection, or an XSS attack.

2.3 Dependency Check

To check dependencies for known vulnerabilities, *OWASP Dependency Check* was used (introduced in Section 1). Its Command Line Interface (CLI) option was chosen, due to its simplicity. After the analysis of dependencies, the tool produces an HTML-formatted report. The report was attached to the archive of this submission. A check performed by *OWASP Dependency Check* is different from static analysis and dynamic analysis because all reported vulnerabilities are true positives. Hence, the reported vulnerabilities weren't checked manually.

First, *OWASP Dependency Check* searches for a Common Platform Enumeration (CPE)[5] identifier for each dependency. Then, if an identifier was found, the tool provides all related Common Vulnerability and Exposure (CVE)[6] entries for the dependency. Listing 1 shows the command to check all dependencies within *Befloral*. Afterward, all vulnerabilities have been reported to the XLS file.

```
$ dependency-check.sh --out . --scan ~/workspace/befloral/
```

Listing 1: Dependency check command

For example, the dependency `gson-2.8.7.jar` contains a vulnerability with the identifier *CVE-2022-25647*[7], which was reported by the tool. A more detailed summary is given in Section 3.4.

[5]https://nvd.nist.gov/products/cpe

[6]https://cve.mitre.org/

[7]https://nvd.nist.gov/vuln/detail/CVE-2022-25647

3 Summary

This section provides an overview of the collected vulnerabilities during the process mentioned in Section 2. Section 3.1 summarises the data collected via static analysis, Section 3.2 shows the results of the dynamic analysis, and Section 3.4 provides an overview of the dependency check. Additionally, Section 3.3 provides an example to perform a more advanced attack vector using the vulnerabilities discovered in the static and dynamic analysis phase.

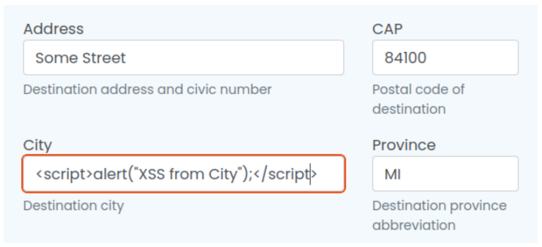
3.1 Static Analysis

The static analysis reported 225 vulnerabilities overall, including 18 different types. 150 out of 225 vulnerabilities were further inspected. Due to the high number of vulnerabilities and the time-consuming manual inspection during static analysis, the inspection of found vulnerabilities was focused on vulnerabilities introduced during the course. Vulnerabilities of the same threat are grouped into a single line in the XLS file (see *Bug Location* column). Otherwise, the XLS file would become too unreadable due to a massive number of vulnerabilities with the same properties, which only differ in the bug location.

The vulnerability with the highest frequency is the *Untrusted servlet parameter* with a total of 57 cases (reported in ID 1 to 20). In short, GET or POST parameters sent to the backend application via requests are not sanitized before being used at a potential security-critical point. 38 cases are found to be true positives, while 19 cases are false positives. However, for most false positives sanitization is still missing, but user defined-defined data is not further used at any security-critical point (see *Brief Vulnerability Description* column for further information). Furthermore, *Untrusted servlet parameter* vulnerabilities provide a way to perform more advanced attack vectors, e.g. a stored-XSS attack combined with a Cross-Site Request Forgery (CSRF).

3.1.1 Stored-XSS via Customer Address

A possible attack vector is to use a vulnerability reported as an *Untrusted servlet parameter* to carry out a stored-XSS attack. Whenever a customer creates or edits a (delivery) address, the input fields (except numeric values) are not sanitized when saved to the database (vulnerability with ID 1 in the XLS file). Given this vulnerability, an attacker can save a malicious script to the customer address field in the database. Eventually, this address is given to the admin, e.g. whenever an admin looks up orders. Then, the script is executed on behalf of the admin, which can become a Privilege Escalation threat. The attacker might save a script, which sends a request to a privileged API endpoint (e.g. reading sensitive customer data) and send the result back to the attacker.



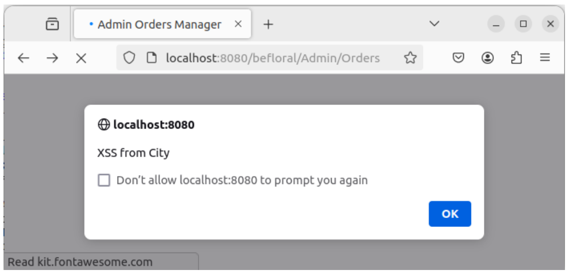
Address: Some Street
Destination address and civic number

CAP: 84100
Postal code of destination

City: <script>alert("XSS from City");</script>
Destination city

Province: MI
Destination province abbreviation

(a) Possible malicious script within the city input field



localhost:8080
XSS from City
Don't allow localhost:8080 to prompt you again

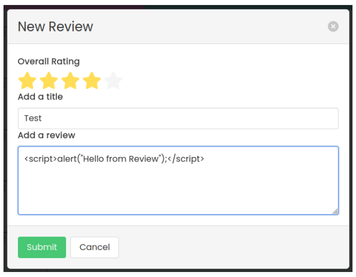
(b) Successfully executed script within the admin panel

Figure 1: XSS attack using the city data of a customer address

Figure 1a shows an example of a stored-XSS attack, where an attacker can insert a malicious script as part of the city property of an address (all non-numeric input fields can be exploited). Next, Figure 1b shows the execution of the script on behalf of the admin.

3.1.2 HTML/Script Injection via Customer Reviews

During the static analysis, an injection vulnerability concerning customer reviews was discovered (see ID 6 in the XLS file). It allows to insert any HTML content within a product review, which is rendered without any sanitization to the client. Figure 2a shows an example payload, while Figure 2b shows the execution of that script. Since reviews can be seen by any viewer of the website, this vulnerability is a potential sink for a CSRF attack.


New Review

Overall Rating

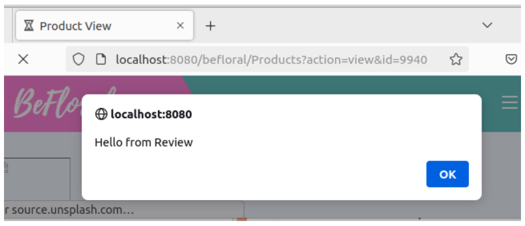
Add a title

Test

Add a review

<script>alert("Hello from Review");</script>

(a) Exploiting the product review input


Product View

localhost:8080/befloral/Products?action=view&id=9940

localhost:8080

Hello from Review

(b) Execution of a script, inserted as a review

Figure 2: Script injection via the review functionality of a product

It is important to mention, that the application performs a prevention by limiting the number of reviews on a product per customer to a single review. Listing 2 shows the error message given whenever a user tries to add more than one review to the same product. This prevention mechanism was discovered using the ZAP fuzzer, by fuzzing the vulnerability using advanced attack patterns.

```
{"statusCode":500,"message":"Duplicate entry \u00275-3\u0027 for key \u0027reviewes.pid\
    u0027"}
```

Listing 2: HTTP 500 error for a second review

3.2 Dynamic Analysis

As mentioned in Section 1, dynamic analysis has been performed using ZAP. After performing the manual scanning (which took about 15min), it found 19 alerts containing 5748 vulnerabilities in total.

Not all vulnerabilities have been inspected manually and added to the XLS file, due to their redundancy and time constraints. For example, ZAP reported 4306 vulnerabilities of *Absence of Anti-CSRF Tokens*. The manual inspection of the first two reported vulnerabilities showed that these are true positives. However, ZAP reported over 100 vulnerabilities of this kind for the same URL[8]. Therefore, this vulnerability was not further investigated. Another example is the alert *User Agent Fuzzer*, with 408 reported vulnerabilities. Therefore, the investigation was focused on unique vulnerabilities, introduced during the course.

14 vulnerabilities have been documented in the XLS file, with only a single false positive. A deeper insight is given in Section 4.

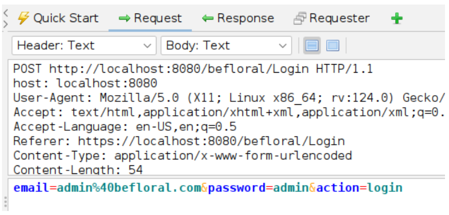
3.2.1 MITM/Proxy-Based Attack on Login Credentials

Figure 3a shows a potential session-side jacking attack (proxy-based), using ZAP proxy (ID 62 in XLS file). It is possible to see unencrypted credentials sent via HTTP. First, there is a possible vulnerability for a Man-In-The-Middle (MITM) attack. Secondly, it allows them to perform a credential stuffing attack, for example through fuzzing. This has proven, that the vulnerability is a true positive. Using the common username and password list, provided by *FuzzDB*, it was possible to get access as the admin user, shown in Figure 3b.

3.2.2 XSS Attack on URL Parameters

During the static analysis phase, see Section 3.1, a true positive vulnerability for a stored-XSS attack was discovered, by providing a malicious script for the `action` parameter (ID 43 in XLS file). The same vulnerability was detected by the *Active Scanner* of ZAP (ID 63, 64, 65, 66 in XLS file). This attack can be performed by visiting the URL `http://localhost:8080/befloral/User?action=<script>alert('HellofromXSS');</script>`. It will raise an exception, which is written without sanitization to

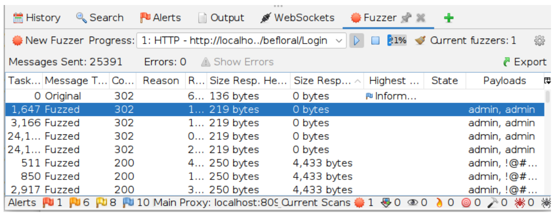
[8]`http://localhost:8080/befloral/Admin/Products`



```
POST http://localhost:8080/befloral/Login HTTP/1.1
host: localhost:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:124.0) Gecko/
Accept: text/html,application/xhtml+xml,application/xml;q=0.
Accept-Language: en-US,en;q=0.5
Referer: https://localhost:8080/befloral/Login
Content-Type: application/x-www-form-urlencoded
Content-Length: 54

email=admin%40befloral.com&password=admin&action=login
```

(a) Unencrypted login credentials



Progress: 1: HTTP - http://localho../befloral/Login 21% Current fuzzers: 1

Messages Sent: 25391 Errors: 0

Task...	Message T...	Co...	Reason	R...	Size Resp. He...	Size Resp...	Highest ...	State	Payloads
0	Original	302		6...	136 bytes	0 bytes	Inform...		
1,647	Fuzzed	302		1...	219 bytes	0 bytes			admin, admin
3,166	Fuzzed	302		1...	219 bytes	0 bytes			admin, admin
24,1...	Fuzzed	302		0...	219 bytes	0 bytes			admin, admin
24,1...	Fuzzed	302		2...	219 bytes	0 bytes			admin, admin
511	Fuzzed	200		4...	250 bytes	4,433 bytes			admin, !@#...
850	Fuzzed	200		1...	250 bytes	4,433 bytes			admin, !@#...
2,917	Fuzzed	200		3...	250 bytes	4,433 bytes			admin, !@#...

Alerts 1 6 8 10 Main Proxy: localhost:809 Current Scans 1 0 0 0 0 0 0

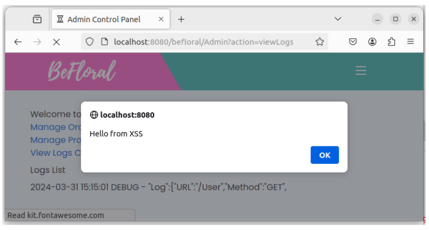
(b) Successfully fuzzed admin credentials

Figure 3: Possible credential stuffing attack

the application log, as shown in Listing 3. Whenever the admin visits the *View Logs* section[9] via the admin panel, the malicious script is rendered without sanitization and executed on behalf of the admin, as shown in Figure 4. Instead of executing an alert, like in the example attack, a more advanced attack can be performed. A possible attack on this vulnerability is presented next in Section 3.3.

```
2024-03-31 15:15:01 DEBUG - "Log":{"URL":"/User","Method":"GET",<script>alert('Hello from
    XSS');</script>"},
```

Listing 3: Log file content after performing the attack



Hello from XSS

Logs List

2024-03-31 15:15:01 DEBUG - "Log":{"URL":"/User","Method":"GET",

Figure 4: Execution of the stored script via the *Admin Logs*

3.3 Advanced Attack Vector

Given the previously introduced vulnerabilities, it is possible to combine these with a CSRF attack to perform a more advanced attack vector. For example, given the vulnerability introduced in Section 3.2.2, it is possible to insert the url-encoded[10] form of the script presented in Listing 4. This script will delete a specific product with the ID 13. The overall idea is, to perform a privileged operation that can only be performed as the admin. Therefore, this attack vector combines a stored XSS with a CSRF attack. Besides deleting a product, this attack vector can result in more harm, e.g. receiving sensitive customer data for social engineering.

```
<script>await fetch("http://localhost:8080/befloral/Admin/Products", { method: "POST",
    headers: { "Content-Type": "application/x-www-form-urlencoded", }, body: "action=
    delete&id=13" });</script>
```

Listing 4: Malicious script to perform a CSRF on behalf of the admin

3.4 Dependency Check

The dependency check reported 39 vulnerabilities in total. Table 1 summarises all vulnerabilities found by the *OWASP Dependency Check*. Unlike vulnerabilities found during static analysis and dynamic analysis, these haven't been checked further, as described in Section 2.3. However, some discovered vulnerabilities don't affect the tested application. These cases are described within the XLS file in the column *Brief Vulnerability Description*. One example is the vulnerability *CVE-2022-21824*[11] (ID 85 in the XLS file), which only affects applications developed using the *Node.js* runtime.

[9] `http://localhost:8080/befloral/Admin?action=viewLogs`
[10] For example using `https://www.urlencoder.io/`
[11] `https://nvd.nist.gov/vuln/detail/CVE-2022-21824`

Dependency	CVE Count	Evidence Count
`gson-2.8.7.jar`	1	26
`jstl-1.2.jar`	1	38
`log4j-core-2.14.1.jar`	4	45
`mysql-connector-java-5.1.0-bin.jar`	30	19
`mysql-connector-java-8.0.22.jar`	3	46

Table 1: Found vulnerabilities through dependency check

4 Discussion

This section ends this report by providing a summary of the security degree of the inspected application. Additionally, the author provides insights about the effectiveness of the used tools and proposes a possible solution to increase the security of *Overleaf*.

Overall, the application has many web-related vulnerabilities of different priority. Many found vulnerabilities will lead to threads with huge risks and can be chained together to conduct an attack vector (some have been introduced in Section 3). A possible reason for the high number of found vulnerabilities within *Overleaf* is, that standard security guidelines, for example, the *OWASP Top 10*[12] have not been followed. Most reported vulnerabilities have a corresponding *OWASP Top 10* entry (as seen in column *OWASP Top-10* in the XLS file).

	Static Analysis	Dynamic Analysis
Found Vulnerabilities	225	5748
Reported Vulnerabilities	150	14
True Positives	48 (32%)	13 (93%)
False Positives	102 (68%)	1 (7%)

Table 2: Overall statistics of found vulnerabilities during static and dynamic analysis

Table 2 presents the overall statistics of the static analysis and dynamic analysis. The overall number of found vulnerabilities in the dynamic analysis is higher than the vulnerabilities found during the static analysis. This contradicts the fact that static analysis can discover more execution paths and, therefore more vulnerabilities, compared to dynamic analysis. However, as mentioned in Section 2 and Section 3, many reported vulnerabilities during dynamic analysis are redundant and refer to the same vulnerability. Due to this high redundancy, the inspection of found vulnerabilities was focused on vulnerabilities that have been reported uniquely and which have been discussed during the lectures. Therefore, the overall number of reported vulnerabilities within the XLS file is fewer compared to the number of reported vulnerabilities by the security tools.

Furthermore, the statistics provide insights into the effectiveness of the static and dynamic analysis tools. ZAP can perform penetration testing in addition to fuzzing to discover and inspect vulnerabilities. However, because ZAP is a dump fuzzer, it is not able to distinguish between similar alerts due to the same vulnerability and it cannot perform more advanced attack patterns based on the internals of the application. *SpotBugs* and *FindSecBugs* have discovered a great number of vulnerabilities. Further manual inspection, for both static and dynamic analysis, of reported vulnerabilities has resulted in the discovery of several attack vectors (introduced in Section 3). However, during static analysis, most vulnerability alerts simply showed the source code line of the sink. To find the real vulnerability in the implementation, further code inspection and debugging were required most of the time.

The static analysis tool reported more false positives compared to true positives. In comparison, the dynamic

[12]`https://owasp.org/www-project-top-ten/`

analysis tool reported more true positives than false positives. This result aligns with the course learning outcome, that static analysis tends to be over restrictive, while dynamic analysis tends to be over permissive. Additionally, dynamic analysis tends to discover fewer vulnerabilities, because it is not able to discover all execution paths of an application. However, the likelihood that a found vulnerability during dynamic analysis is a true positive is higher compared to static analysis.

The dependency check provided a great overview of vulnerabilities within the applications' dependencies. Found vulnerabilities are true positives and have already been investigated by security researchers, however, not all vulnerabilities apply to the investigated application. As mentioned in Section 3.4, some vulnerabilities affect different technologies, not used in the investigated application.

An assumption by the author is, the reason for the high number of vulnerabilities is due to the lack of following security guidelines during development and deployment. Therefore, the developers of *Overleaf* can increase the applications' security by following the System Development Life Cycle (SDLC) including *DevSecOps*. SDLC divides the lifecycle of an application into several phases to provide a structured approach for identifying and eliminating vulnerabilities and risks. In addition, SDLC can be extended with *DevSecOps* to further improve security. With *DevSecOps*, the developers can take advantage of Continuous Integration (CI) to automate the process of testing the application and validating security requirements, while using Continous Delivery (CD) to make sure that a release-ready artifact of the application is prepared at any time.

5 References

The author assures, that the only resources used for this report, are those given by Prof. Marchetto during the course Security testing of the year 2023/24 and the links provided in this report.