UNIVERSITÀ
DI TRENTO

Security Testing

BeFloral Security Report

Academic year 2023/2024

Christian Sassi
Student ID: 248724

Submission date: 09/07/2024

Name of filled XLS: *Security Testing Project Report - Christian Sassi.xlsx*

Contents

1 Introduction

The goal of this project was to perform an analysis on a real web application to identify and evaluate potential security vulnerabilities that can be exploited to compromise system's integrity, confidentiality and availability. Given the web application context of the analysis, the primary focus was on identifying vulnerabilities like:

- Client-Side Validation
- Cross-Site Scripting (XSS)
- Cross-Site Request Forgery (CSRF)
- Indirect Direct Object Reference (IDOR)
- SQL Injection

However, as will be explained in the following chapters, additional vulnerabilities were discovered during the analysis.

1.1 Tested application

BeFloral is an e-commerce web application designed for selling flowers. Developed by students from the University of Salerno, it uses Java, JSP, and Servlets for backend functionality and MySQL for database management. The project is structured to run on an Apache Tomcat server and includes features like user authentication, product catalog management and order processing.

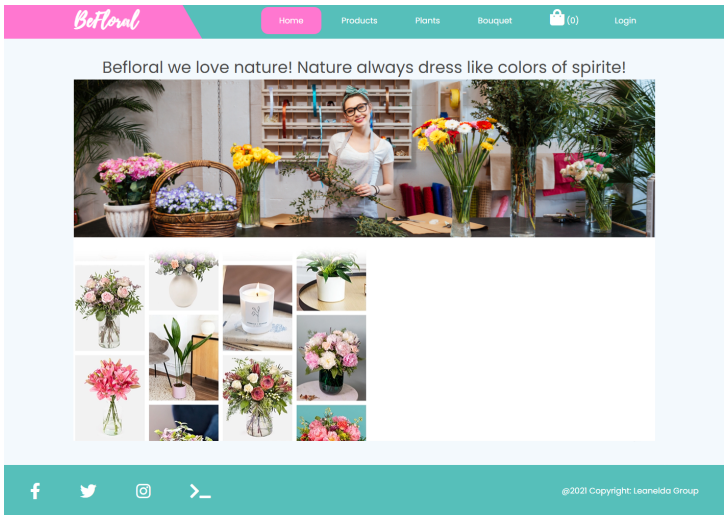


Figure 1: Befloral home page

1.2 Methodology

The first step was to manually test the app to better understand its functionality from a user's perspective and identify potential critical aspects that might hide vulnerabilities. After gaining an understanding of the application, the analysis begun. Several approaches can be used to conduct a security analysis, generally divided into two groups: static analysis and dynamic analysis. For this task, both methodologies were adopted along with manual checking to perform a more comprehensive and trustworthy analysis.

1.2.1 Static analysis

Static analysis examines an application's code without actually running it (hence the name "*static*"). There are two main approaches: pattern-based and flow-based. Pattern-based analysis searches for common vulnerability patterns in the code, while flow-based analysis aims to understand the program's execution flow and identify all possible paths. As a result, the pattern-based method is often seen as more general but less in-depth compared to the flow-based approach, which is more program-specific. However, it becomes harder to cover every execution path, especially in large programs. Additionally, static analysis can produce false positives (reporting vulnerabilities that are not real) or false negatives (missing actual vulnerabilities). For instance, a false positive might occur when static analysis identifies a vulnerable code path that the program never executes.
One tool that can be used for this type of analysis and that was used for this task is **SpotBugs**. SpotBugs is a popular open-source static analysis tool that helps developers find bugs and security vulnerabilities in Java applications. It uses a combination of pattern-based and flow-based analysis techniques to identify potential issues. SpotBugs assigns a severity level ("*Scary*", "*Troubling*" and "*Of Concern*") and a confidence level ("*High confidence*", "*Normal confidence*" and "*Low confidence*") to each identified vulnerability. It also provides detailed information about the vulnerability, pinpointing its location in the code and explaining its theoretical nature. There is also the possibility to use external plugins like **FindSecBugs**, a to extend SpotBugs' capabilities.

1.2.2 Dynamic analysis

Dynamic analysis involves executing the program to uncover bugs and vulnerabilities. While arguably less comprehensive than static analysis, it offers more reliable results. Dynamic analysis cannot realistically explore every possible program path, but discovered vulnerabilities are demonstrably exploitable since they occur during actual execution.
One tool that can be used for this type of analysis and that was used for this task is **ZAP**. ZAP is a popular security tool for dynamic analysis that operates in two modalities. The first one, "*Automated Scan*" allows the tool to automatically scan a web application, employing different techniques like pattern-matching and fuzzing to identify potential weaknesses that an attacker could exploit. Alternatively, ZAP can operates as a man-in-the-middle proxy via "*Manual Explore*", intercepting HTTP communication between the application and the tester's browser. This allows the tool to inspect and manipulate HTTP requests and responses.

1.2.3 Manual analysis

Finally, it's crucial to emphasize the importance of manual analysis alongside any chosen methodology. While both static and dynamic analyses tools offer valuable results and are generally reliable, they can occasionally produce incorrect results, as shown in this report. So, manually reviewing results before taking action is essential since ensures that identified vulnerabilities are genuine and warrants remediation efforts.

1.3 Environment

The following tools were used for this project and additional configurations are detailed below:

- Eclipse IDE for Enterprise Java and Web Developers v4.20.0
- MySQL v8.0.37

- Apache Tomcat v9.0.46
- SpotBugs v3.1.5
  - Added "*Security*" to the reported bug categories.
  - Set minimum confidence to report to "*Low*".
  - Added FindSecBugs plugin v1.12.0.
- ZAP v2.15.0. Unfortunately, due to an unresolved bug, the marketplace section of the tool was unreachable. By keeping the terminal open, it was possible to notice that exceptions were thrown when trying to access that section of the tool, making it impossible to install any additional plugins. For this reason, aside from the automatic scan, the fuzz testing provided by the tool was used as-is, meaning no additional changes were made, such as installing plugins to enhance its effectiveness.

2 Analysis Methodology

The conducted analysis followed this flow:

1. General testing
2. Static analysis
3. Dynamic analysis
4. Manual analysis

2.1 General testing

The initial approach involved testing the application to understand its structure and the user's capabilities. During this phase, the goal was to impersonate a normal user navigating the website. Special attention was given to areas requiring user input, such as the login page and search page. This allowed for a general idea of where to search for vulnerabilities and what results to expect from the tools.

2.2 Static analysis

For the static analysis, as mentioned in the previous chapter, SpotBugs and FindSecBugs were chosen. The reason behind this choice was that SpotBugs is straightforward to use and, despite its simplicity, it provides robust results. Therefore, the type of static analysis conducted mainly followed a pattern-based approach.

(a) Bug Explorer

(b) Bug Info of a potential *SQL Injection* vulnerability

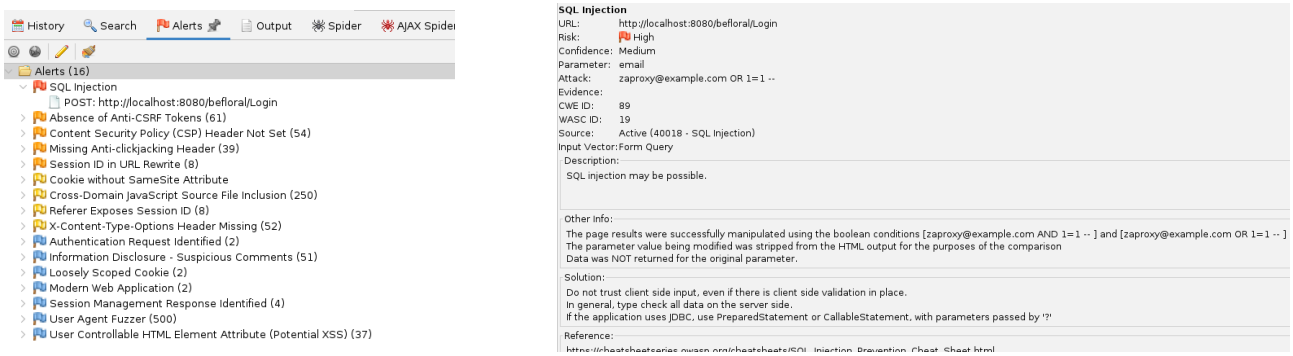
Figure 2: SpotBugs interface

Figure 2 shows the results collected by SpotBugs after a first run. The tool highlights vulnerabilities of varying severities and confidence levels. The approach used was to start with the most critical issues, beginning with "*Scary*", then "*Troubling*" and finally "*Of Concern*". The same approach was applied to the confidence levels: first "*High confidence*", then "*Normal confidence*" and lastly "*Low confidence*". The methodology for inspecting each vulnerability was pretty much always the same. First, the description of the vulnerability was read to understand the bug and its nature. Next, the vulnerable code was inspected to determine if the highlighted lines were indeed vulnerable.

If they were, the next step was to assess, similar to a flow-based analysis, whether potential code execution could reach that point. For this aspect, sometimes it was sufficient to statically inspect the nearby lines of code, while other times it was necessary to execute the code because the code's flow became difficult to follow through static inspection alone. It is important to note that while SpotBugs effectively highlighted issues in the code, sometimes the detected vulnerabilities were of low severity or their exploitation did not pose a critical risk to the web application.

2.3 Dynamic analysis

The dynamic analysis was divided into two phases. The first phase involved using the automated scan mode provided by the ZAP tool. During this analysis, starting from */befloral/Home*, the tool automatically executed requests to reach each endpoint of the web application. Specifically, in this phase, the tool used a so called "*Spider*" and an advanced version called "*AJAX Spider*". The Spider analyzed the content of the requests to identify exploitable vulnerabilities. This approach is efficient and applicable to most types of web applications. However, in today's web applications, JavaScript is extensively used to dynamically modify page content and, as a consequence, vulnerabilities introduced by JavaScript execution may not be evident by merely inspecting the HTML code of the page. AJAX Spider is designed to uncover these types of vulnerabilities by using a web browser to interact with each endpoint. Due to this approach, even if more effective, AJAX Spider operates slower than the traditional Spider.
Finally, upon completion of the scan, ZAP displays an page summarizing all collected results, as illustrated in Figure 3a.



(a) Collected results

(b) Inspection of a potential *SQL Injection* vulnerability

Figure 3: ZAP interface

During the second phase of the dynamic analysis, based on the results previously obtained by both static and dynamic analyses, the vulnerable endpoints where furtherly tested using fuzzing techniques that allows to verify them (in case of uncertainties) and find more complex attack vectors.

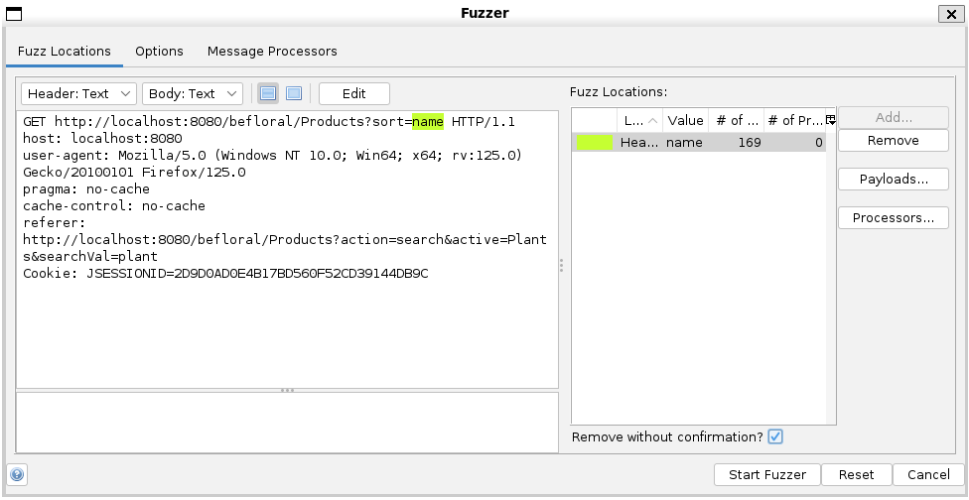


Figure 4: Fuzzing interface

Figure 4 demonstrates how to fuzz a specific field by simply selecting it (lime area) and specifying the fuzzing payload. In this case, a pre-existing dictionary specifically created for SQL Injection was chosen. Unfortunately, due to the issue with the marketplace explained in the previous chapter, fuzzing did not reveal much more interesting vulnerabilities beyond those already discovered. Nevertheless, it helps identify more complex attack vectors.

2.4 Manual analysis

In the end, it was decided to perform a final manual analysis to uncover any vulnerabilities that were not identified using the previously discussed methods. This decision was driven by the fact that, at this point, there was a deeper understanding of how the application works and its critical parts. The approach was akin to a black-box method (though not entirely, as a true black-box tester would not have this level of knowledge), meaning the goal was to simulate an attacker who only has access to the web application without knowing what is happening on the server. The focus was on parts of the web application that were not covered by the automated tools, particularly areas where users could input data, such as the review section, or where the web application imposes limitations or controls on user input.

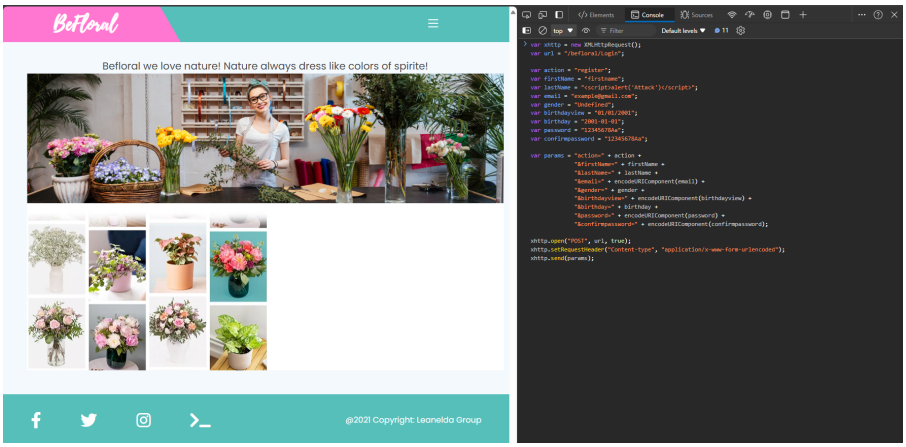


Figure 5: Manual attempting to bypass the client-side validation set on the registration fields

3 Collected data

The conducted analyses highlighted several critical aspects of this web application. Specifically, the majority of the vulnerabilities were discovered through static analysis, while dynamic analysis, although it also identified vulnerabilities, was particularly useful for deeper inspection of what was discovered through static analysis (and not only). Finally, manual analysis was performed to uncover vulnerabilities in areas not covered by these tools, successfully identifying some important critical issues.
Nineteen vulnerabilities were examined, of which 11 were true positives. Twelve vulnerabilities were identified through static analysis, five through dynamic analysis, and two through manual analysis. Despite these numbers, even though static analysis revealed a higher number of vulnerabilities, the ratio of true positives was only four out of twelve. In contrast, both dynamic and manual analyses had a ratio of 100%, meaning that all the discovered vulnerabilities using these techniques were true positives. This helps to better understand the initial thought exposed in the first chapter: while static analysis offers better coverage in terms of identifying vulnerabilities compared to dynamic and manual approaches, not all reported results can be considered reliable. On the other hand, dynamic and manual results, being tested in real-time, can be considered most of the time reliable.
Additionally, by inspecting the results, it can be observed how that static analysis identified four types of vulnerabilities out of a total of twelve discovered, dynamic analysis identified five out of five and manual analysis identified two out of two. This information underscores the strengths and weaknesses of these techniques. From one perspective, static analysis can identify a vulnerability and all its variants within an application. For example, an application may have different areas vulnerable to SQL Injection, each in a unique way. On the other hand, dynamic and manual analyses may not uncover all instances of a vulnerability, but they can detect a broader range of vulnerability types.
Note: In this context, "manual analysis" refers to tests performed dynamically, with no static testing being manually conducted.

Below, vulnerabilities are discussed by type and the technique used to identify them. Additionally, for each vulnerability it is specified the number of true and false positives to better understand the effectiveness of the tools and methods used.

3.1 Static analysis results

- Indirect Direct Object Reference (IDOR): Path Traversal (0 / 4 true positives).

  One of the IDOR vulnerabilities discovered involved the potential direct access to a log file. However, further analysis revealed that this action could only be performed by an administrator. Even if someone was logged in as an administrator, direct access to the log file was not possible because it was retrieved and processed directly by the server. Other false positives of this vulnerability behave more or less the same.

- SQL Injection (1 / 4 true positives).

  The only vulnerability was a code instruction where an SQL injection was executed without using a prepared statement. This means that an attacker could potentially manipulate the parameter used in the query to perform an SQL Injection attack. Specifically, the vulnerability was identified in *befloral/Products?sort=name;QUERY*, where *QUERY* could be replaced with

a malicious query. False positives were reported in code areas where query were dynamically created but using parameters that were constant or not directly modifiable by the user.

- Unvalidated Redirects (0 / 1 true positives).

  SpotBugs highlighted various code areas where the redirection endpoints were dynamically created. However, upon closer inspection of the code, the parameters used to construct the redirection endpoints were always constants or not directly modifiable by the user.

- Cross-Site Scripting (XSS) (3 /4 true positives).

  SpotBugs highlighted a point where the cart content was returned to the user's web page. Because the cart content included product objects and each object had its own attribute fields, it was possible to exploit these fields to perform a persistent/type-1 XSS attack. Other true positives of this vulnerability behave more or less the same. The only false positive was found in a code section where the server returned fixed text that cannot be modified, thus posing no risk to the user.

3.2 Dynamic analysis results

- SQL Injection (1 / 1 true positives).

  ZAP, along with fuzzing, successfully identified and confirm the same SQL Injection vulnerability discovered during the static analysis (see previous section). It is interesting to note that ZAP did not detect the three false positives identified during the static analysis.

- Cross-Site Request Forgery (CSRF) (1 / 1 true positives).

  ZAP identified vulnerable endpoints like */befloral/Api/Cart*, */befloral/Api/Orders* and */befloral/Api/Products* that could be called from an external domain to force an authenticated user to perform an unwanted operation on the web application via a malicious redirect. This vulnerability existed because the web application lacked an anti-CSRF token to prevent such attacks.

- Cookie Without SameSite Attribute (1 / 1 true positives).

  ZAP identified the absence of the SameSite attribute for the JSESSIONID cookie. This could allow an attacker to steal the cookie while the victim visits a malicious website, thereby impersonating him on the web application where the victim is authenticated.

- Content Security Policy (CSP) Header Not Set (1 / 1 true positives)

  This vulnerability affects the entire web application, potentially allowing the loading of malicious scripts from untrusted sources outside the web application's domain.

- Vulnerable JS Library (1 / 1 true positives)

  ZAP reports several outdated and vulnerable JavaScript libraries used in the web application.

3.3 Manual analysis results

- Cross-Site Scripting (XSS)

  The focus was placed on the review system, which was not touched by the automated tools. The vulnerability allowed a potential attacker to write a review and insert malicious web code (HTML, CSS, JavaScript) into it. This could result in a persistent/type-1 XSS attack, causing the execution of malicious code by anyone who viewed the tainted review.

- Client-Side Validation

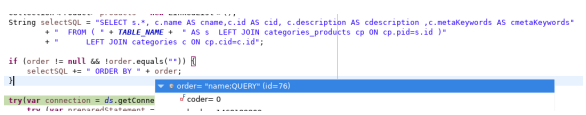
  The focus was placed on the validation performed during the registration of a new user. Specifically, there was a client-side validation mechanism that forced the user to only insert specific characters in specific fields of the registration. However, by injecting some JavaScript code, it was possible to bypass these controls.

4 Discussion of the collected results

After conducting several analyses, the web application cannot be considered secure since there are too much security issues that could seriously compromise the provided service. It suffices to say that the analysis revealed nine different types of vulnerabilities, totaling nineteen exploitable vulnerabilities.
While this project was developed to accomplish specific tasks (different from ours) and for an educational purpose, in a real-world scenario, this web application cannot and must not be used due to insufficient security measures. It is evident that during development, the developers did not follow any secure guidelines, which otherwise would have significantly reduced the number of vulnerabilities found. To be objective, it has to be said that achieving total security is nearly impossible, but developers nowadays have access to valuable resources such as OWASP, CWE, CVSS, etc... to enhance software security. Also because in this case, these are extremely well-known and widespread vulnerabilities (like SQL Injection, XSS and CSRF), so there are numerous methods to prevent them.

Regarding the collected data, as previously mentioned, there are several vulnerabilities affecting different areas of this web application. Among these, vulnerabilities like SQL Injection and XSS are particularly noteworthy. This is not only due to their notoriety, but also because attackers can exploit them using basic and common attack vectors. This is not to imply that other web applications are immune to these attacks (although it is very rare to find a web application vulnerable to this nowadays) but rather that they typically implement at least basic countermeasures. For instance, SQL Injection can be mitigated through various methods, such as sanitization procedures. While some procedures may be more effective than others, they still provide a level of protection. However, in this web application, no measures have been taken to prevent such scenarios.

(a) XSS attack where the name of a new product was filled with JS code



(b) SQL Injection attack where a malicious *QUERY* was passed to the server

Figure 6: Examples of two of the most serious vulnerabilities detected during the analysis demonstrate how easy it was to perform these attacks.

My personal experience was generally positive. The knowledge gained during the course was adequate to address most aspects of this project. Although some parts required additional information but it was easily acquired by consulting the web.
Regarding the tools being used, it must be noted that the initial setup was not straightforward, despite being well-documented by the provided materials. I encountered several challenges with some tools such as Apache Tomcat and, sometimes, Eclipse seemed overly complex for the tasks at hand. However, these issues were generally resolved with additional guidance found online.
The installation of SpotBugs and FindSecBugs was straightforward and its interfaces were very intuitive. Given the simplicity of the web application we were analyzing, SpotBugs easily identified a large number of vulnerabilities, although some were false positives, which is a known risk with static analysis tools. SpotBugs' effectiveness was further enhanced by the high severity of the vulnerabilities in this web application, requiring minimal effort for tools such this to identify them.
Installing ZAP was also straightforward. However, I encountered a problem with the marketplace,

preventing the use of additional plugins that could have improved some techniques like fuzzing. To mitigate this issue, I attempted the installation of the tool on different machines (fresh Ubuntu installation on a desktop PC and WSL on a laptop) but none of these methods worked. Research revealed that this is a known bug also affecting other users and despite consulting classmates and online resources, I did not find an official solution. Fortunately, this issue did not significantly impact the results collected by ZAP.
Despite these challenges, ZAP performed well, with its results compared favorably with those obtained through static analysis. One of its strengths lies in providing real potential attack vectors for identified vulnerabilities and offering techniques such as fuzzing to stress-test the application, potentially uncovering attack vectors that other tools may overlook.
After all, the tools used were very powerful, though sometimes overly complex and time-consuming to learn the basics.
In conclusion, I believe that combining static and dynamic analysis tools is the optimal approach for conducting thorough and comprehensive software analysis, without forgetting manual checking which, as already mentioned, is essential. Naturally, this strategy should be assessed on a case-by-case basis, but it generally yields the best results.

In conclusion, my experience as a security tester was positive. Before starting this project, I only had theoretical knowledge, so a practical approach was very useful, not only to test what I had learned, but also to better understand how the security testing world works, starting by approaching some of the most known and used tools of this field. Despite having a strong IT background, I had never encountered these topics before, so I am pleasantly satisfied with the overall experience.