TOURISM SECURITY ANALYSIS

Security Testing 2024-2025
University of Trento

Elia Gatti 256137
Matteo Bertoldo 256131

File: xlsx-elia-gatti-matteo-bertoldo

1. Brief Introduction to the task, tested apps, and used tools

The goal of this project is to analyze and improve the security of a Java-based web application, identifying common vulnerabilities in web apps developed without a security focus. The project follows three main stages of analysis:

- **Static Analysis**: Using automated tools to examine the codebase and detect potential security vulnerabilities without executing the application.
- **Dynamic Analysis**: Testing the application in a runtime environment to uncover vulnerabilities that may not be visible through static analysis.
- **Manual Analysis**: A thorough hands-on inspection of the application to find vulnerabilities missed by automated tools.

The final objective is to compile a comprehensive security assessment report that documents the vulnerabilities discovered, explains the testing process, and provides recommendations for improving security.

Application Overview

The **Online Tourism Management System** is a web-based Java application designed to optimize the operations of tour operating companies. It allows companies to manage travel services, including customer bookings, hotel accommodations, transport arrangements, and food options. Key functionalities include:

- **User Registration and Login**: Users can create accounts and log in to access various features.
- **Package Management**: Users can manage travel packages, including adding, updating, and viewing package details.
- **Booking System**: Users can book tourism packages, modify bookings, and cancel or view booking details.

There are different user roles (standard users and admins), each with specific privileges, such as managing packages, handling bookings, and managing user profiles. The application follows the **MVC (Model-View-Controller)** design pattern, dividing the application into frontend, model, and backend layers.

- **Frontend**: Developed using JSPs, HTML, CSS, and JavaScript, the frontend captures user input through forms and sends it to the backend for processing. It's especially important to address security issues here, such as XSS.
- **Models**: The model layer represents core entities like User, Package, and Booking, stored in a MySQL database. Security testing focuses on potential risks like SQL Injection.
- **Backend**: Built using Java Servlets, the backend handles business logic, user requests, and database interactions. It is critical to ensure proper input validation and session management to prevent unauthorized access.

Tools Used

- **Eclipse**: The primary IDE for Java development, facilitating smooth coding, debugging, and static analysis.

  Problems Encountered: Eclipse occasionally failed to load changes, leading to errors if the project wasn't refreshed regularly. This was a critical step to ensure new configurations and code modifications were recognized.

- **MySQL**: Used as the database for data storage.

  Problems Encountered: Initially, the `USE tourism;` command was missing from the setup documentation, leading to problems when running the command `source path/to/tourism.sql` file in order to generate the table structure.

- **SpotBugs**: The primary static analysis tool for identifying security vulnerabilities. Combined with the **Find-Sec-Bugs** plugin, it focused on detecting common issues like SQL Injection and XSS.

  Problems Encountered: The configuration was set to use all available checks, which led to a very large number of warnings being generated. This required careful filtering and analysis to focus on the most critical vulnerabilities.

This project combines static, dynamic, and manual analysis to assess and improve the security posture of the **Online Tourism Management System**.

Static Analysis

This section will detail how the static analysis of the application was performed, focusing on why and how each step was taken.

2.1. Used tool

For the static analysis of the **Tourism** application, **SpotBugs** was the primary tool used. SpotBugs is a widely recognized static code analysis tool for Java that helps in identifying potential bugs and security vulnerabilities within the codebase. It performs static analysis on Java bytecode to detect common coding issues, including security flaws such as **SQL Injection**, **Cross-Site Scripting (XSS)**, and improper data handling.

Setup and Configuration

For this project, the following setup and configuration steps were employed:

- **SpotBugs**: This tool was used in conjunction with its security-focused plugin, Find-Sec-Bugs. SpotBugs analyzes the Java code to detect a wide range of potential issues, including performance bugs, multi-threading issues, and security vulnerabilities.
- **Find-Sec-Bugs Plugin**: This plugin extends SpotBugs by adding security-related checks, making it particularly valuable for identifying vulnerabilities such as SQL Injection, XSS, and improper input validation.

Configuration Details

To ensure a comprehensive analysis of the application, **all available options and checks** in both **SpotBugs** and **Find-Sec-Bugs** were enabled. This configuration allows the detection of the full spectrum of potential security vulnerabilities present in the Java code. By enabling every available check, the aim was to gather as many findings as possible for review, ensuring no potential issue was overlooked. This setup was chosen to provide a **comprehensive view of security vulnerabilities**, enabling the identification of both high and low-risk issues in the code. The use of both SpotBugs and Find-Sec-Bugs allowed for a detailed and thorough review of the Java codebase, helping to ensure most common security flaws were not overlooked.

2.2. Adopted process

For the security testing phase, we adopted a structured process focusing on the use of SpotBugs and its security-oriented extension, Find-Sec-Bugs, to perform a static analysis of the "Tourism Management System" application. This process was designed to quickly identify security vulnerabilities, validate the most relevant ones, and document them precisely.

Step 1: Running SpotBugs

Process: After configuring SpotBugs on Eclipse, we ran a full scan of the application's code, logging each detected issue and categorizing it by severity and type, with particular focus on warnings related to SQL Injection and XSS vulnerabilities.

Step 2: Selecting and Testing Relevant Issues

For each identified issue, we performed manual testing by simulating attack scenarios and experimenting with various attack vectors. For instance, we tested SQL Injection vulnerabilities by inserting SQL payloads in input fields to verify dynamic query manipulation, while for XSS we injected malicious scripts in text fields to simulate session theft.

Step 3: Documenting Results

We organized all findings in an Excel spreadsheet, documenting each identified and tested vulnerability along with its classification, CWE code, risk level, and observations on potential exploits. This approach provided a clear and structured view of detected security issues, with detailed documentation for each verified vulnerability.

Motivations and Process Choices

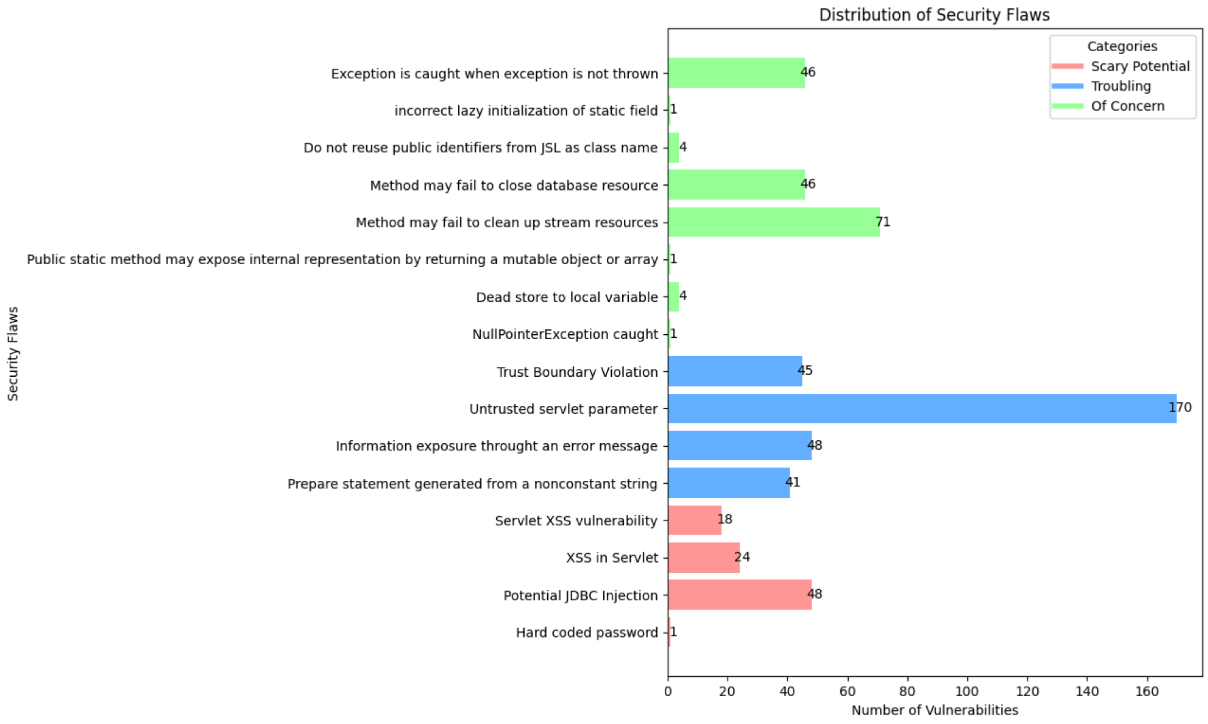
We adopted this process to maximize the efficiency of the testing phase and to gain a comprehensive overview of security issues without getting bogged down by false positives. SpotBugs and Find-Sec-Bugs allowed us to automatically categorize potential risks and focus on testing only those relevant to our objective of identifying critical vulnerabilities. The work was organized to ensure that each team member gained experience with every part of the process. Given the large number of potential vulnerabilities identified by the tool, each member selected 3-5 vulnerabilities from the list to analyze, attempt to exploit, and document in the related XLS file.

2.3. Summary of the collected data

This section summarizes the data collected from the security testing tool, which is documented in the provided XLS spreadsheet accompanying this report. In the XLS file, each entry that includes an attack vector has been rigorously tested and analyzed to accurately determine whether the vulnerability reported by the tool is valid or a false positive. From the vulnerabilities tested in detail, we identified mostly true positives, indicating that all potential vulnerabilities identified by the tool are exploitable and could compromise data integrity or allow malicious code injection. Below, We present an overview of the tool's findings.

SpotBugs Analysis

Using the configured settings, SpotBugs analyzed a total of 570 potential vulnerabilities, categorizing them by severity level, from “Scary Potential” to “Of Concern,” as follows:



The chart summarizes security vulnerabilities detected by the tool, organized by severity. The horizontal axis shows the number of occurrences for each vulnerability, while the vertical axis lists specific types, such as "Untrusted Servlet Parameter" and "Potential JDBC Injection." Vulnerabilities are categorized by severity: "Scary Potential" (red), "Troubling" (blue), and "Of Concern" (green). The "**Scary Potential**" category includes the most critical vulnerabilities, such as "Potential JDBC Injection" (48 instances), posing a risk of database attacks, and two types of Cross-Site Scripting (XSS) vulnerabilities ("XSS in Servlet" with 24 instances and "Servlet XSS Vulnerability" with 18 instances), which allow malicious script injection that could compromise data and user security. The "**Troubling**" category contains moderate-severity issues, with "Untrusted Servlet Parameter" as the most frequent vulnerability (170 instances), indicating weak input validation. Other notable issues include "Information Exposure through Error Messages" (48 instances) and "Trust Boundary Violation" (45 instances), both of which can expose sensitive data or improperly handle trust boundaries. The "**Of Concern**" category involves lower-severity vulnerabilities primarily related to code quality, such as resource management issues like "Method Fails to Clean Up Resources" (71 instances) and "Method Fails to Close Database Resource" (46 instances), which could lead to performance issues if unresolved.

Given the high number of potential vulnerabilities, not all were documented in the XLS file, so we cannot determine whether each one is a true positive or false positive. However, we selected one example from each category to gain a general understanding of whether the reported vulnerabilities are likely to be true or false positives.

Data reported in XLS spreadsheet

In the spreadsheet, we documented a total of 11 vulnerabilities, of which 7 were fully tested with corresponding attack vectors identified. Most of these documented vulnerabilities are either Cross-Site Scripting (XSS) or SQL Injection attacks. This prevalence is largely due to the number of untrusted parameters that the application handles, as identified by the SpotBugs tool. SpotBugs flagged various points in the code as vulnerable due to improper handling of these untrusted parameters. A false positive was identified, labeled as "Dead Store to variable…" (referenced in row 10 of the spreadsheet). This vulnerability is due to an unused variable declared in the code, likely a result of SpotBugs' limitation to bytecode analysis, which restricts its ability to fully interpret the intent and usage in the source code.

An analysis of the exploitation times reported in the spreadsheet reveals that the average time required to exploit these vulnerabilities is approximately 10 minutes. However, two vulnerabilities (rows 2 and 3) required close to an hour due to their complexity. This increased time is partly explained by the repetitiveness of similar errors in the application, which meant that after the initial exploit attempts, subsequent exploitation time dropped significantly as patterns became recognizable.

Confidence levels were also analyzed, revealing that of the 11 reported issues, 8 are categorized as high confidence, 1 as medium, and 2 as low. This distribution indicates that most reported vulnerabilities are highly likely to represent actual, exploitable security issues.

Vulnerability exploited

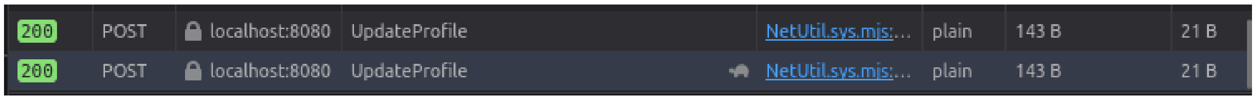
The application presents a variety of potential **SQL Injection** vulnerabilities, primarily due to the improper handling of user input. These vulnerabilities arise when user-controlled data is directly incorporated into SQL queries without proper validation and sanitization. Below are examples of **SQL Injection** attacks identified in the system, along with recommendations for preventing such attacks. One of the vulnerabilities identified in the application is the **SQL Time-Based Injection**, which occurs when user-controlled data is used directly into a SQL query. This exposes the application to attacks where an attacker can infer sensitive information from the server's response delays. We observed this vulnerability in the following code:

```
PreparedStatement ps = con.prepareStatement("update register set name
=?, mobile =? where email='"+email+"'");
```

Although the code uses **PreparedStatement**, the query still improperly concatenates the `email` parameter (in control of the user) directly into the SQL string, which leaves the application vulnerable to **SQL Injection**. This can be exploited through **Time-Based Blind Injection**, where an attacker injects SQL that causes the server to delay its response based on certain conditions. For example, an attacker could manipulate the `email` field with the following payload:

```
email=mailchenonesiste' OR IF(SUBSTRING((SELECT password FROM (SELECT
* FROM register) AS alias WHERE name='luffy'),1,1) = 'l', SLEEP(1),
0); --
```

In this case, the server will experience a delay if the first character of the password for the user "`luffy`" matches `'l'`. By modifying the payload accordingly, the attacker can infer the password one character at a time based on the response delay. Although this process is slow, it can be automated with a script to make it more efficient and faster.

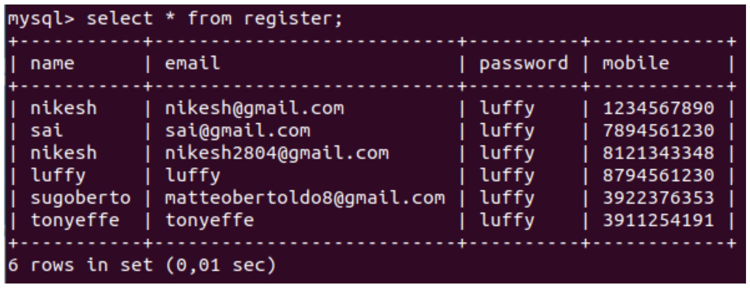


```
200  POST  localhost:8080  UpdateProfile     NetUtil.sys.mjs:...  plain  143 B  21 B
200  POST  localhost:8080  UpdateProfile     NetUtil.sys.mjs:...  plain  143 B  21 B
```

From the image we observe two different responses: when the password guess is correct, the server's response is delayed by 1 second and includes an image of a turtle, indicating that the guess was successful!

While **prepared statements** are a good defense against SQL injection, they do not automatically prevent all types of attacks, such as **Time-Based Blind Injection**. **SQL queries** must be carefully structured, and **input validation** is still necessary to eliminate potential injection points.

An SQL Injection vulnerability was identified in the "`Recover`" functionality, specifically in the password reset feature. Here, the user input from the email field was directly appended to SQL queries without validation, allowing potential manipulation of both `SELECT` and `UPDATE` queries. By exploiting this, an attacker could inject SQL code into the email parameter with inputs like `email=' or '1' = '1` and set the password as `luffy`. This bypasses authentication and could reset the password for all users in the database.



```
mysql> select * from register;
+-----------+---------------------------+----------+------------+
| name      | email                     | password | mobile     |
+-----------+---------------------------+----------+------------+
| nikesh    | nikesh@gmail.com          | luffy    | 1234567890 |
| sai       | sai@gmail.com             | luffy    | 7894561230 |
| nikesh    | nikesh2804@gmail.com      | luffy    | 8121343348 |
| luffy     | luffy                     | luffy    | 8794561230 |
| sugoberto | matteobertoldo8@gmail.com | luffy    | 3922376353 |
| tonyeffe  | tonyeffe                  | luffy    | 3911254191 |
+-----------+---------------------------+----------+------------+
6 rows in set (0,01 sec)
```

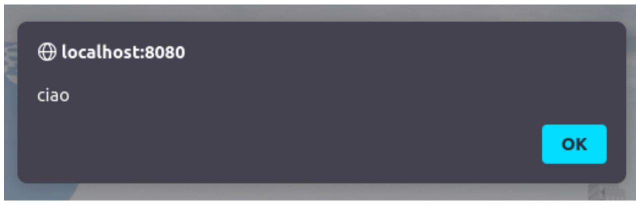
Because the email parameter was concatenated directly into the query, this vulnerability allows an attacker to retrieve all email records and potentially change every user's password to "luffy." Proper use of prepared statements, along with input validation and other best practices, would mitigate such SQL Injection risks, underscoring the critical need for secure coding practices in this application.

Another critical security issue in the application is the lack of input validation and sanitization when processing user-controlled data obtained via the `getParameter` method in servlets. This method reads GET and POST parameters from the client's request, and the values obtained should always be considered **unsafe**, as they are fully controlled by the client. Without proper validation or sanitization, these parameters can lead to vulnerabilities such as SQL Injection, Command Injection, or Cross-Site Scripting (XSS), depending on how the data is processed later. One example of this specific vulnerability arises in the **Book Package** functionality, where the `totalcost` parameter, which dictates the final price of a package, is directly controlled by the user and sent via a POST request to the server. This parameter is then parsed and used in the application without any server-side validation to verify its correctness. For instance, an attacker can manipulate the `totalcost` parameter by crafting a POST request with a manipulated value, such as:
`packagename=green+building&place=washington&days=1&packageCost=1&noofPersons=1&totalcost=1`

Since the application lacks validation, the request is processed with this manipulated value, allowing the attacker to successfully book the package at a price of 1, bypassing the intended pricing logic. This could result in financial loss or incorrect data within the system. The lack of input validation in the application can lead to Cross-Site Scripting (XSS) vulnerabilities, as user-controlled input, such as the vehicleName parameter, is used without proper sanitization or validation. This creates an opportunity for attackers to inject malicious scripts that can be executed in other users' browsers. The attack process begins with the attacker crafting a malicious script by creating a page that submits a POST request to the BookTransport.jsp page. This request includes a payload in the `vehicleName` field, such as `<script>alert("ciao");</script>`, alongside other parameters like `transportType=Bus`, `vehicleType=AC`, `vehicleCost=50`, `vehicleDate=2024-11-13`, `packagename=green+building`, and `place=missori`.

Upon receiving the request, the application processes it and stores the malicious vehicleName in the database without performing any sanitization. After storing this payload, the attacker can redirect the victim to a page displaying the vehicle details, ModifyTransport.jsp, where the malicious script executes automatically in the victim's browser, triggering an alert with the message "ciao." Although this example displays a simple alert, an attacker could exploit this vulnerability further to steal cookies, perform actions on behalf of the user, or redirect the victim to malicious sites.

An example of an attack URL could look like this:
`http://localhost:8080/Tourism/User/ModifyTransport.jsp?vehicleName=<script>alert('ciao');</script>`



ciao

Another example of XSS injection occurs when an attacker exploits the `hotelName` input field on the BookPackage.jsp page. This vulnerability exists because the application fails to sanitize user input before reflecting it in the page output.

The attack begins with the attacker using the browser's Inspector tool to modify the `hotelName` field, injecting a script such as `<script>alert('XSS');</script>`. After injecting the payload, the attacker submits the form, sending the malicious script to the server. When the page is rendered, the injected script executes in the victim's browser, displaying an alert with the message "XSS."

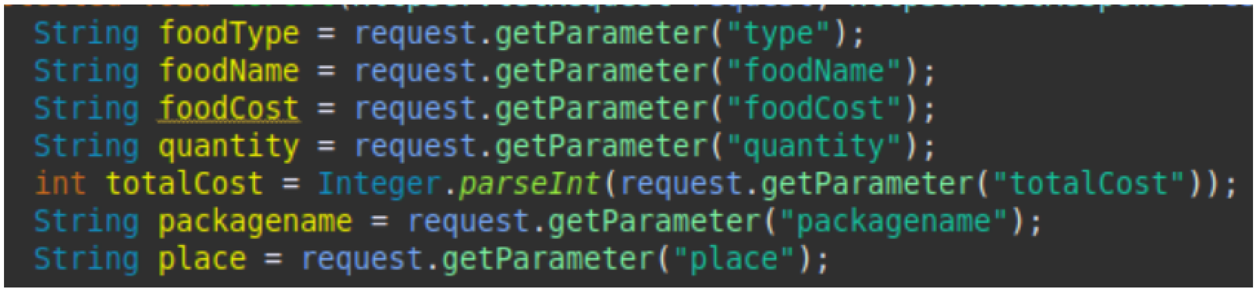
An example of the injected value might look like this: `<input class="form-control" value="<script>alert('XSS');</script>" type="text" id="hotelName" name="hotelName" readonly="">`

2.4. Brief discussion of the collected results

In this chapter, we examine the coding principles underlying the Tourism application, focusing on the major vulnerabilities identified and the factors contributing to their presence. We will explore solutions for addressing the most critical vulnerabilities, outlining improvements necessary to make the application suitable for potential deployment. Additionally, we discuss strategies developers can adopt to prevent similar security issues in their applications. This chapter also reviews the tools used in our analysis, providing our insights on their effectiveness. Finally, we share our reflections on this phase of the project and its broader implications for secure development practices.

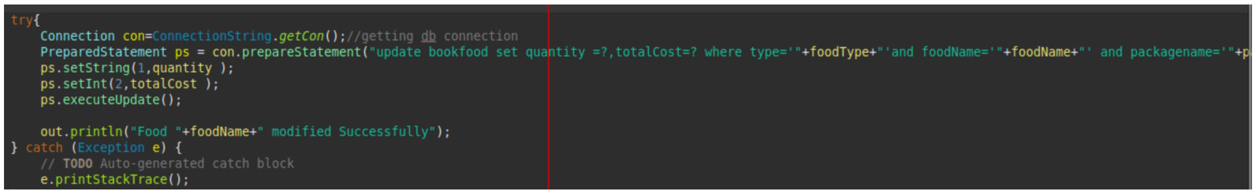
Application Analysis

The **Tourism** application is a web-based Java system designed to streamline operations for tour operators, using the Model-View-Controller (MVC) architecture. However, the application presents significant security vulnerabilities due to insecure coding practices, particularly in its handling of user input within the model and backend portions of the architecture. Analysis with SpotBugs revealed numerous vulnerabilities, primarily Cross-Site Scripting (XSS) and SQL Injection issues. These vulnerabilities arise because the application accepts untrusted input parameters from users without validation or sanitization, as shown in the accompanying screenshot.



```
String foodType = request.getParameter("type");
String foodName = request.getParameter("foodName");
String foodCost = request.getParameter("foodCost");
String quantity = request.getParameter("quantity");
int totalCost = Integer.parseInt(request.getParameter("totalCost"));
String packagename = request.getParameter("packagename");
String place = request.getParameter("place");
```

These user-supplied inputs serve two main purposes: displaying output to users and updating records in the database. Without proper sanitization, these inputs expose the application to various injection attacks. This pattern of insecure coding suggests not a single oversight but rather a lack of security awareness on the developer's part.



```
try{
    Connection con=ConnectionString.getCon();//getting db connection
    PreparedStatement ps = con.prepareStatement("update bookfood set quantity =?,totalCost=? where type='"+foodType+"'and foodName='"+foodName+"' and packagename='"+p
    ps.setString(1,quantity );
    ps.setInt(2,totalCost );
    ps.executeUpdate();

    out.println("Food "+foodName+" modified Successfully");
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
```

One critical issue is the improper use of prepared statements throughout the application. While prepared statements generally defend against SQL injection, many of the prepared statements in this application incorporate unsanitized, variable strings. This failure leads to a range of SQL Injection vulnerabilities, from minor data integrity issues affecting individual users to potentially catastrophic modifications of entire database tables.

Another alarming problem is the hardcoded database password within the code. If deployed within a corporate environment, this would allow any employee with access to the application files to connect to and alter the database directly. This practice makes the application highly susceptible to insider threats and social engineering attacks.

These security flaws, combined with other issues documented in this report, strongly suggest that this application was not intended for production deployment but rather as an experimental project or an academic exercise to explore different technologies.

Code Improvements

The Tourism application's codebase lacks essential security practices, impacting its security, reliability, and maintainability. Key improvements include:

1. **Parameterized Queries**: Replace direct SQL query concatenations with parameterized queries to prevent SQL Injection risks. This should be combined with server-side input validation and sanitization to better control data flow.
2. **Input Validation & XSS Protection**: Strengthen input validation and apply output encoding for user-generated content to prevent Cross-Site Scripting (XSS). Implementing a strict Content Security Policy (CSP) and using modern libraries for secure output encoding will enhance defenses.

3. **Sensitive Data Management**: Remove hardcoded credentials from the code and use environment variables or secure vaults for secure configuration management. Ensure deployment settings protect sensitive information.
4. **Test Suite Implementation**: Add a structured test suite, including unit, integration, and security tests, to identify vulnerabilities early. Automated tests should cover common attacks and edge cases.
5. **Adoption of Secure Libraries & Documentation**: Use security-focused libraries and frameworks, and maintain comprehensive documentation. This improves code maintainability, enforces standards, and strengthens the application's resilience against attacks.

These improvements will enhance security, streamline maintenance, and ensure safer future updates.

SpotBugs

The static analysis tool proved highly effective for identifying code issues, significantly reducing the time needed to review the codebase manually and minimizing the likelihood of missing potential vulnerabilities. However, while useful, the tool is not without limitations. It operates without an understanding of the application's complete logic, meaning that it may overlook complex exploits that involve multiple, interconnected vulnerabilities. Overall, the experience with the tool was positive for several reasons: it saved considerable time, provided detailed explanations of detected vulnerabilities, and offered links to resources for a deeper understanding. Additionally, the tool was straightforward to install and configure, even within Eclipse's often challenging environment.

Conclusions

Regarding the application, it's evident that the developers did not follow security best practices in their coding approach. This became clear as multiple vulnerabilities were identified through static analysis. The tools proved effective in detecting many issues, although they also missed some that required manual identification, underscoring the importance of combining automated analysis with careful manual inspection.

Reflecting on this experience and the use of the tools, We found the project required a decent amount of effort, but it was essential for gaining a deeper understanding of the practical side of security testing. This hands-on approach is extremely valuable, as it directly applies to real work settings where security is fundamental.

3. Dynamic Analysis

This section will detail how the dynamic analysis of the application was performed, focusing on why and how each step was taken.

3.1 Used tool

To conduct our security analysis, we utilized OWASP ZAP (Zed Attack Proxy), an open-source tool designed for Dynamic Application Security Testing (DAST). ZAP identifies vulnerabilities in web applications by simulating attacks while they are operational. It features a user-friendly interface and integrates seamlessly into CI/CD pipelines for automated testing. ZAP operates as a "man-in-the-middle" proxy, intercepting and analyzing traffic between the browser and the application to detect potential security issues. For this analysis, we did not modify the default plugin list significantly; however, We upgraded ZAP to version 2.15 from the base version 2.13 available in the virtual machine. This upgrade was made in the hope that the new version was more stable.

3.2 Adopted process

In this section, we detail the dynamic analysis testing process undertaken by our team, highlighting the steps involved, tool configurations, and team organization.
The testing process commenced with the initiation of the OWASP ZAP (Zed Attack Proxy) tool. Upon booting the application, our first action was to deploy both ZAP spiders to maximize the information gathered about accessible web pages. The traditional spider effectively crawled through the HTML content, while the AJAX spider was employed to handle dynamic content generated by JavaScript. This dual approach ensured comprehensive coverage of the application's surface.
Once the spidering phase concluded, we proceeded with an active scan targeting all discovered pages that were accessible prior to user login. This step aimed to identify potential vulnerabilities, particularly SQL injection flaws, which were among the first issues detected. To enhance our context within ZAP, we consulted a tutorial video that guided us in adding user accounts to the context. We re-ran both spiders to expand ZAP's understanding of the application, although this step yielded mixed results. Consequently, we performed manual exploration of the web application to demonstrate various user actions to ZAP, ensuring that all potential interactions were captured.

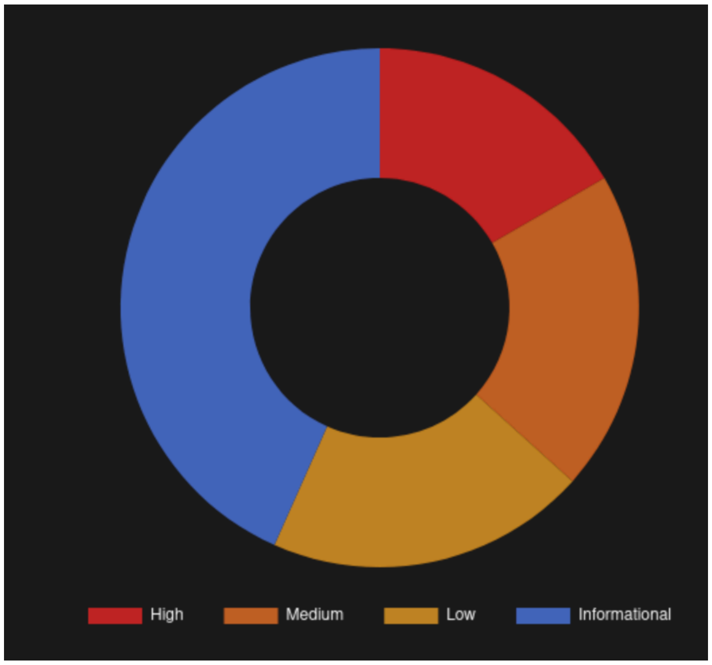
After completing this exploratory phase, we executed multiple active scans to uncover additional vulnerabilities. However, as we expanded our context significantly, we encountered performance issues; specifically, after sending approximately 20,000 to 30,000 requests, the application became unresponsive. This necessitated a pause to allow ZAP to process alerts before proceeding with further tests. After further testing, one of the team members gained benefit from

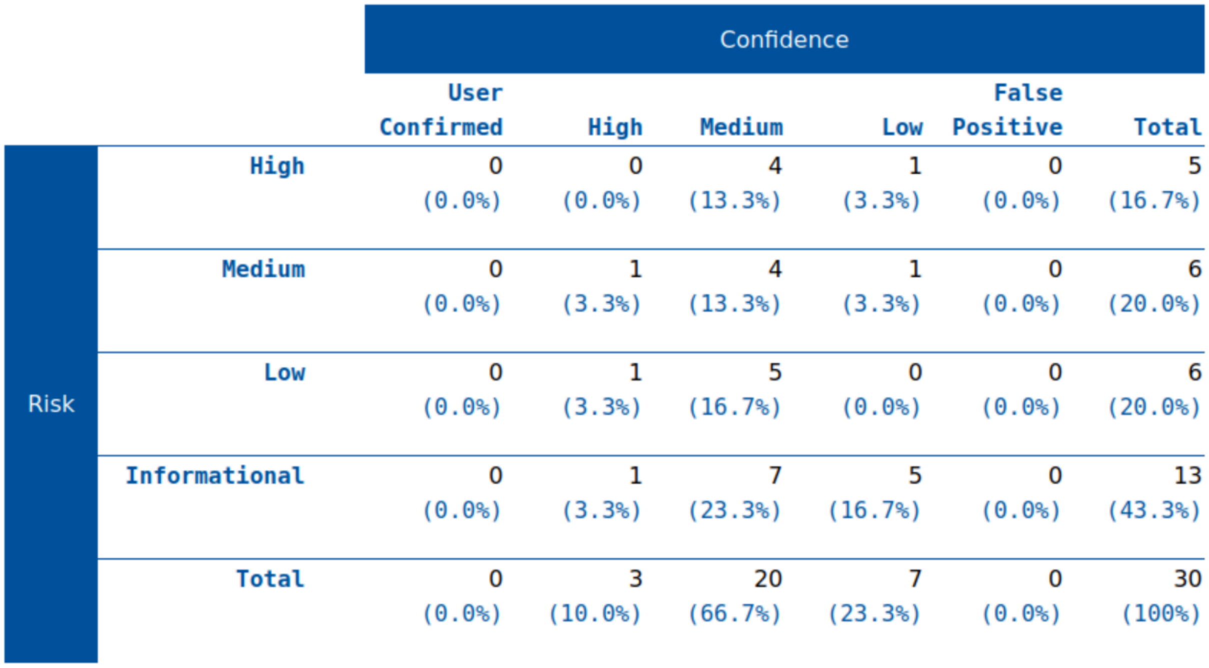
adding more swap memory inside the virtual machine, this however did not fix the issue it only merely delayed the inevitable.
To validate the alerts generated by ZAP, we prioritized testing based on risk levels—starting with high-risk (red flag) alerts and moving down to medium-risk (orange flag) alerts. Given the non-deterministic nature of ZAP's findings, we opted for a collaborative approach where team members followed identical steps rather than splitting tasks. This strategy aimed to ensure thoroughness and facilitate peer review of findings.
In our testing efforts, we also employed fuzzing techniques on certain POST requests using a simple character set that included uppercase and lowercase letters, numbers, and relevant symbols. Additionally, we utilized file fuzzers through *jbrofuzz* options to uncover alerts that may have been overlooked during standard active scans. These steps were mirrored in our examination of the application's admin section.
Throughout this process, we meticulously documented vulnerabilities and false positives in an XSL spreadsheet.
In summary, our testing methodology combined automated scanning with manual exploration and collaborative review.

3.3 Summary of the collected data

This report provides a detailed analysis of the vulnerabilities identified during a security assessment conducted using the ZAP web scanner. In addition to presenting statistical insights, it dives deep into the confirmed vulnerabilities, their impact, and the exploitation techniques. The goal is to offer actionable insights to help developers and stakeholders understand and address these security issues.

The statistics and examples provided here are based on a comprehensive review of the ZAP scanner results. Where possible, vulnerabilities were manually tested to confirm their validity and assess their severity. Below, we present the findings, detailed explanations of selected vulnerabilities, and recommendations for mitigation.





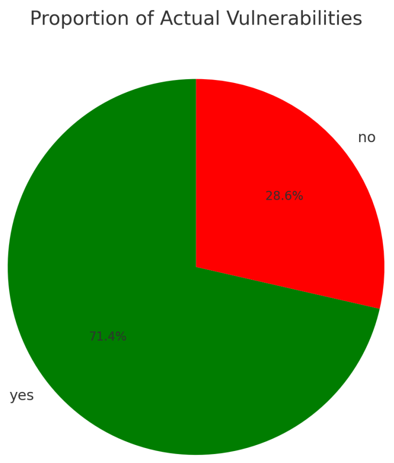
		Confidence					
		User Confirmed	High	Medium	Low	False Positive	Total
Risk	High	0 (0.0%)	0 (0.0%)	4 (13.3%)	1 (3.3%)	0 (0.0%)	5 (16.7%)
	Medium	0 (0.0%)	1 (3.3%)	4 (13.3%)	1 (3.3%)	0 (0.0%)	6 (20.0%)
	Low	0 (0.0%)	1 (3.3%)	5 (16.7%)	0 (0.0%)	0 (0.0%)	6 (20.0%)
	Informational	0 (0.0%)	1 (3.3%)	7 (23.3%)	5 (16.7%)	0 (0.0%)	13 (43.3%)
	Total	0 (0.0%)	3 (10.0%)	20 (66.7%)	7 (23.3%)	0 (0.0%)	30 (100%)

Statistics Summary

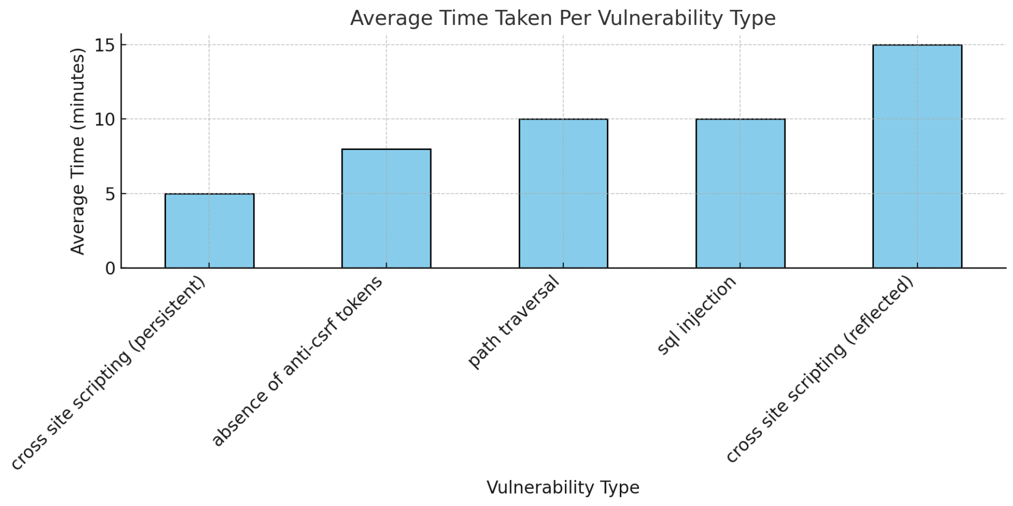
During the scan, **1,289 alerts** were identified and categorized based on their severity:

1. **High Risk Alerts (93):**

    - Examples: Cross-Site Scripting (Stored and Reflected), Path Traversal, SQL Injection (MySQL).
2. **Medium Risk Alerts (189):**
    - Examples: Absence of Anti-CSRF Tokens, Missing Content Security Policy (CSP) Header, Cross-Domain Configuration issues.
3. **Low Risk Alerts (244):**
    - Examples: Application Error Disclosure, Cookie without `SameSite` Attribute, Cross-Domain JavaScript Source File Inclusion, Information Disclosure - Debug Error Message.
4. **Informational Alerts (763):**
    - Examples: Authentication Requests Identified, Missing Content-Type Headers, GET used for POST, Information Disclosure, Loosely Scoped Cookies, Modern Web Application issues.



The average time taken for asserting the tool result was about 9.79 minutes. The image below show the avg time taken per vulnerability type detected:



Detailed Analysis of Reported Vulnerabilities

1. Cross-Site Scripting (Reflected)

Description:
Cross-Site Scripting (XSS) is one of the most common and dangerous vulnerabilities in web applications. It allows attackers to inject malicious scripts into web pages viewed by other users. Reflected XSS, in particular, occurs when the injected payload is immediately echoed back in the application's response. This vulnerability does not require the attacker to store the payload on the server; instead, it relies on tricking users into visiting a maliciously crafted URL.

Example:
The vulnerability was identified on the payment page (Payment.jsp). The attacker could exploit this by crafting a URL, such as:

- `http://localhost:8080/Tourism/User/Payment.jsp?id=1&total=1380&packagename="><script>alert(1);</script>&place=NC`

In this instance, the packagename parameter is unsanitized, allowing execution of arbitrary JavaScript, demonstrated by the payload <script>alert(1);</script>.

Impact:

The impact of this vulnerability is severe, as it allows the execution of malicious scripts directly in the victim's browser. It affects both authenticated and unauthenticated users who interact with the crafted link. The ability to execute any code injected into the vulnerable parameter makes this a highly critical and dangerous issue.

Attack Context:
This vulnerability requires no interaction beyond the victim clicking the link. It is a self-contained attack that exposes sensitive browser-level trust mechanisms.

2. Absence of Anti-CSRF Tokens

Description:
Cross-Site Request Forgery (CSRF) vulnerabilities occur when an attacker tricks a user into performing unintended actions on a trusted web application. Without Anti-CSRF tokens, the application is unable to verify the legitimacy of requests, enabling malicious users to exploit trust relationships.

Example:
A vulnerability was identified in the admin food management section (Food.jsp). The form on this page allows admins to add food items, but it lacks CSRF protection. Attackers can craft and submit requests on behalf of admin users.

Steps to Exploit:

1. Analyze the page to identify the POST request URL:
    - http://localhost:8080/Tourism/Food
2. Craft a malicious form:

```
JavaScript
<!DOCTYPE html>
<html>
 <body>
  <form action="http://localhost:8080/Tourism/Food" method="POST"
style="display:none;">
   <input type="hidden" name="type" value="Veg">
   <input type="hidden" name="foodName" value="MaliciousFoodItem">
   <input type="hidden" name="foodCost" value="-1">
   <input type="submit">
  </form>
  <script>
   document.forms[0].submit();
  </script>
 </body>
</html>
```

3. Host the form on a malicious website. When an authenticated admin visits the site, the request is automatically submitted, resulting in the addition of malicious data to the application.

Impact:

The impact of this issue is significant, as it enables unauthorized actions, such as the addition of invalid data (e.g., negative prices). This compromises the integrity of the application, allowing attacks that disrupt its core functionalities. Furthermore, the issue affects multiple endpoints throughout the application, highlighting a systemic oversight during the design phase.

3. SQL Injection

Here’s a structured write-up of your analysis based on the SQL Injection vulnerability detected using OWASP ZAP, similar in style to the examples you provided:

Description:
SQL Injection is a critical vulnerability that allows an attacker to manipulate SQL queries executed by the database. This occurs when user inputs are not properly sanitized or validated, enabling attackers to bypass authentication, retrieve sensitive data, or alter the database. In this

case, the vulnerability was detected in the *Recover Password* functionality of the application, which processes user-provided email input to retrieve a forgotten password.

Example:
The vulnerability was identified on the page:

- http://localhost:8080/Tourism/Recove

By injecting a payload in the email parameter, such as:

- email: wJuyEcLJoeiTPTyy' OR '1'='1' --
- password: (we can choose or let this empty)

The attacker is able to manipulate the SQL query and bypass authentication or access unauthorized data.

Impact:
The impact of this vulnerability is **highly critical**, as it allows attackers to:

- Access sensitive user information, such as emails and passwords.
- Bypass authentication mechanisms.
- Potentially modify database contents by chaining further queries.

Mitigation:

- Use **Prepared Statements** with parameterized queries.
- Sanitize and validate all inputs.
- Apply the **principle of least privilege** for database access.

3.4 Brief discussion of the collected result

In this section we are going to discuss the degree of the tested application and a brief discussion and a critical analysis of the collected data as well as considerations about the conducted experience and tool used.

3.4.1 Degree of tested application

The tested application demonstrates significant shortcomings in its security architecture, failing to meet the requirements of a safe and reliable system. Numerous vulnerabilities have been identified, some of which are classified as high-risk. These vulnerabilities are easily exploitable, even by attackers with minimal technical expertise, and they present the potential for widespread issues across various parts of the application.

As discussed in the static analysis, the presence of these flaws indicates a lack of adherence to fundamental security principles during development. Rather than being isolated incidents, these vulnerabilities are systemic, affecting the application as a whole. This suggests that the developers either overlooked essential security considerations or lacked the necessary knowledge to implement them correctly.
It is important to recognize that this application was not intended for production use but was developed as a learning tool to improve the developer's programming skills. However, if the goal is to make the application viable for real-world deployment, significant improvements are necessary. Addressing these issues requires a complete overhaul of the codebase to integrate modern security practices.

A first step would involve implementing comprehensive **sanitization of all input parameters**, ensuring that user-provided data does not lead to vulnerabilities such as SQL Injection or Cross-Site Scripting (XSS). The application's reliance on prepared statements is currently flawed and must be corrected to effectively protect against SQL Injection attacks.
Authentication mechanisms also need significant strengthening. At present, administrative access is granted using “admin” as both the username and password, a practice that undermines security entirely. This must be replaced with a robust authentication system incorporating strong, unique passwords, two-factor authentication for administrative accounts, and safeguards like rate-limiting and account lockouts to prevent brute-force attacks.

In addition to improving authentication, the application requires **stronger security measures for critical functions** such as administrative operations and payment processing. These enhancements should include protections against **Cross-Site Request Forgery (CSRF)** through the **use of anti-CSRF tokens**, secure session management to mitigate session hijacking risks, and logging mechanisms to monitor and detect suspicious activities.
Beyond immediate fixes, the development process itself must evolve to prioritize security. Secure coding practices should be integrated into every phase of development, from design to deployment. Regular code reviews focusing on security, automated security testing tools, and developer training on secure development principles are all essential steps to prevent similar vulnerabilities in future iterations of the application.

By addressing these deficiencies comprehensively, the application can transition from a vulnerable project to a secure, production-ready system. These measures are not just recommendations but necessities for any application handling sensitive operations or data.

3.4.2. Analysis of the collected data

The data collected during the dynamic analysis was largely consistent with the findings from the static analysis, though there were some notable differences. The static analysis provided a broad overview of vulnerabilities, identifying several recurring issues across the codebase. However, the dynamic analysis revealed more nuanced details about how the application behaved under specific attack scenarios. A key aspect of our focus was testing for SQL Injection and XSS vulnerabilities, as these represent some of the most critical and easily exploitable issues in the application. The dynamic analysis allowed us to verify the accuracy of the static tool's findings while expanding our understanding of how these vulnerabilities could be triggered in different contexts. We supplemented these efforts by adding tests for Cross-Site Request Forgery (CSRF) and path traversal vulnerabilities, given their relevance to the application's functionality.

The CSRF tests highlighted the lack of anti-CSRF tokens and the absence of adequate safeguards for sensitive operations. On the other hand, path transversal was a false positive. While these vulnerabilities were not explicitly flagged by the static analysis tool, their identification through dynamic testing underscores the importance of combining both methods to achieve comprehensive coverage.

An interesting challenge during the dynamic analysis was the sheer volume of alerts generated by the tool. While static analysis tended to group similar errors together, the dynamic tool flagged each instance separately, requiring significant effort to validate and categorize the findings. Despite this, the process provided a more granular understanding of the application's vulnerabilities, helping us identify patterns and systemic issues that might have been overlooked otherwise.

3.4.3 Analysis of the tool used

The tool used for dynamic analysis, OWASP ZAP, presented several challenges during our testing process. Its interface, while powerful, was not intuitive, especially for first-time users. Many functionalities were hidden within small icons located in the upper menu bar, making it difficult to understand and access key features. This steep learning curve required us to spend additional time familiarizing ourselves with the tool before we could effectively utilize it.

Another significant drawback was the lack of stability in both the tested application and ZAP itself. Frequent crashes and freezes disrupted our workflow, particularly during resource-intensive operations like active scans. One student experienced persistent issues with their laptop freezing during active scans. The other student had the application frequently crash even when allocating to the virtual machine 6 CPU cores and 16 GB RAM, leading to incomplete scans and inconsistent results.

In addition, the tool's dynamic nature made it difficult to achieve reliable outcomes. For instance, active scans on the same location often needed to be repeated multiple times to ensure comprehensive coverage. The tool also required significant manual intervention to configure and optimize its performance. For example, we had to manually create users and associate them with specific contexts, a process that was only possible after consulting external tutorials. Even then, we found that the tool struggled to automatically map all application functionalities, necessitating a combination of automated and manual testing. Not only that, but nearly half of the tested alerts were false positives.

Upgrading ZAP from version 2.13 to 2.15, along with updating its plugins, had minimal impact on its behavior. Many of the stability and usability issues persisted, leading to frequent frustration. Additionally, compatibility issues between ZAP and our development environment further compounded these difficulties. Eclipse, for example, would often lose track of project files after running ZAP and closing the virtual machine, requiring us to repeatedly re-import and recreate the workspace.

Despite these challenges, ZAP ultimately fulfilled its primary role as a vulnerability scanner. It identified numerous critical issues in the application, providing actionable insights for improving its security posture. However, the process was far from smooth, and the tool's shortcomings significantly increased the time and effort required to achieve meaningful results. Future users of ZAP should be prepared for these challenges and allocate sufficient time to address them.

4. Manual Analysis

In this section, we will discuss the vulnerabilities identified during the manual analysis, provide a brief overview and critical evaluation of the collected data, and share insights and reflections on the overall experience.

4.1. Used tool

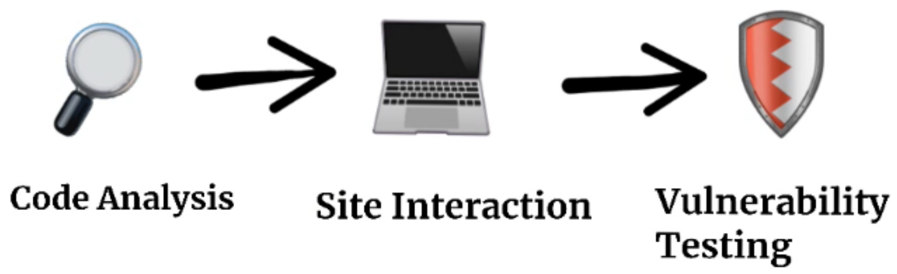
For this phase of the project, no specialized tools were used for manual testing analysis. The testing process relied entirely on manual exploration and exploitation of the application's security features. This decision was made to deeply understand the vulnerabilities in the system and to exploit them in a controlled environment to observe their effects directly.

Manual testing allowed for a hands-on examination of the application, focusing on specific areas of concern such as input validation, session management, and authentication mechanisms. Although no automated tools were utilized, this process involved detailed inspection of the code, testing user inputs, and examining responses from the application to identify security flaws.

4.2. Adopted process

Our approach to manual testing was meticulously planned to ensure a thorough exploration of the application's security vulnerabilities. The process was segmented into three stages: **Code Analysis**, **Site Interaction**, and **Vulnerability Testing**. This methodology was chosen to:

- **Identify** vulnerabilities at their root in the code.
- **Validate** these findings in a live application environment.
- **Exploit** these vulnerabilities to assess their real-world impact.



4.2.1 Code Analysis

Objective: Detect vulnerabilities directly from the source code by focusing on security-critical components.

Actions: Conducted a static review of the application's code, focusing on:

- Hardcoded credentials that could allow unauthorized access.

- Lack of proper authentication mechanisms for sensitive operations.
- General sink functions and unvalidated data flows in the code.

Outcome: This phase produced a detailed list of potential vulnerabilities, including hardcoded credentials and unsanitized inputs, which informed subsequent testing***.***

4.2.2 Site Interaction

Objective: Validate the findings from code analysis and identify additional vulnerabilities through live interaction.

Actions: Interacted with the application interface to:

- Test input fields for insufficient validation, leading to vulnerabilities like XSS.
- Attempt to bypass authentication and authorization controls through URL manipulation or tampering.
- Analyze error messages for potential information leakage or overly detailed system reporting.

Outcome: This phase confirmed the exploitability of several vulnerabilities, including unsanitized inputs leading to XSS attacks and inadequate access control protections.

4.2.3 Vulnerability Testing

Objective: Exploit the identified vulnerabilities to measure their severity and real-world implications.

Actions: Targeted tests included:

- Injecting scripts for XSS attacks, allowing malicious JavaScript to execute in other users' browsers.
- Accessing restricted admin pages and manipulating session cookies to bypass access controls.

Outcome: Demonstrated critical risks, such as the ability to steal user cookies via XSS and perform unauthorized actions due to broken access control.

Motivation Behind the Process

1) **Granular Focus**: Starting with code analysis ensures we address the source of vulnerabilities, while site interaction and testing phase verifies their real-world implications.
2) **Comprehensive Understanding**: By moving from theory (code analysis) to practice (site interaction), we ensure vulnerabilities are not just theoretical but can be practically exploited, which is key for understanding their true risk.

Process Organization

The team collaborated closely, leveraging individual strengths. One member focused on interpreting and analyzing the source code, while another specialized in testing the application through live interaction. Regular discussions ensured that findings from each stage were thoroughly reviewed and incorporated into the final analysis, providing a unified and holistic approach to testing.

4.3. Summary of the collected data

During our security assessment, we identified seven distinct vulnerabilities across various components of the application. For reasons of brevity, this section focuses on a detailed analysis of a subset of these vulnerabilities. The selected examples highlight the severity and breadth of the issues discovered, with references to the full dataset and statistical analysis provided in the accompanying XLS file.

Total vulnerabilities identified: 6

Most frequent categories: OWASP A04 (Insecure Design).

Task Division: As previously mentioned, we divided the vulnerability identification process into three distinct subtasks to ensure a comprehensive assessment:

- **Code inspection**: ~10 minutes. This initial phase allows us to catch vulnerabilities directly in the source code, focusing on issues like hardcoded credentials or weak encryption, which are often overlooked in dynamic analysis.
- **Site interaction**: ~15 minutes. Here, we validate the code analysis in a live environment, testing how the application behaves with user inputs, which is critical for identifying interaction-based vulnerabilities like XSS or authentication bypass.
- **Vulnerability testing**: ~20 minutes. This phase evaluates the exploitability of vulnerabilities. By attempting to exploit them, we gain insight into their real-world implications, which is essential for prioritizing remediation efforts.

Confidence level in our findings is very high, around ~90%, due to the consistent replication of vulnerabilities during testing.

The vulnerabilities span multiple OWASP Top-10 categories, demonstrating a lack of security measures in critical areas like input validation, session management, and access control. Below, we explore selected vulnerabilities in detail.

4.3.1 Main Vulnerabilities Identified

4.3.1.1 Hardcoded Credentials (CWE-259)

We identified hardcoded credentials embedded directly in the source code, a serious security vulnerability that exposes sensitive accounts to unauthorized access.
In the authentication module, we found the following hardcoded credentials:

- email: admin
- password: admin

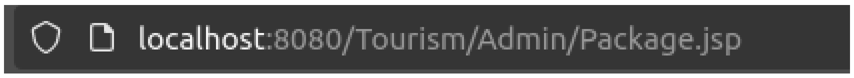
Attackers could use this information to gain unauthorized access to the admin panel and once inside, they could modify user data or exfiltrate sensitive information.

Remediation:

- Remove hardcoded credentials and replace them with secure solutions such as environment variables or secure vaults.
- Enforce strong password validation policies to prevent the use of weak or empty passwords.

4.3.1.2 Broken Access Control (CWE-285)

This vulnerability is directly associated with **OWASP A01 (Broken Access Control)**, as the application failed to properly enforce authentication and authorization requirements for sensitive resources.
By directly accessing the URL of the admin panel, unauthenticated users could view and modify sensitive data. There was no verification ensuring only authorized users could access these resources.


localhost:8080/Tourism/Admin/Package.jsp

Attackers can bypass authentication, escalate privileges, and compromise the entire application.

Remediation: Implement server-side access control checks for all sensitive resources. and employ role-based access control (RBAC) to restrict access to administrative features.

4.3.1.3 Cross-Site Scripting (XSS) (CWE-79)

Detected in multiple input fields, these vulnerabilities fall under OWASP A03 (Injection) . They arise from insufficient sanitization of user inputs, allowing attackers to inject and execute malicious scripts in users' browsers. Key locations include `/Tourism/src/main/java/userServlet/BookRoom.java` and `ModifyRoom.java`.

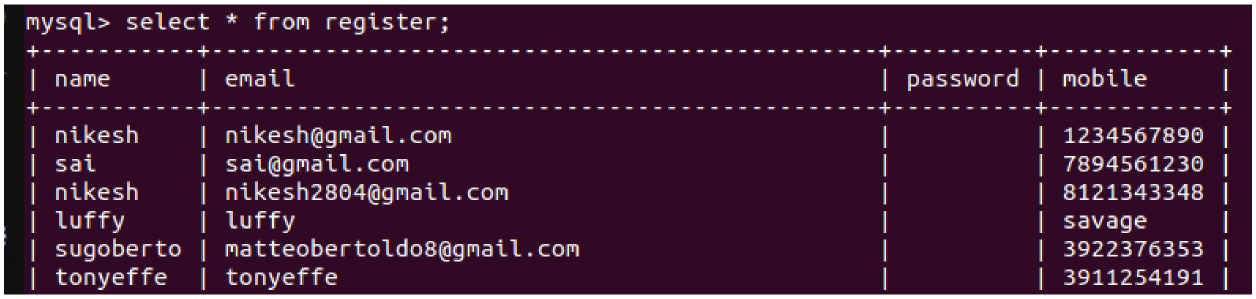
An attacker can inject a script into the hotel booking system via a form submission. When another user loads the page, the script executes, triggering actions like displaying alerts or stealing session cookies. The XSS vulnerabilities enable attackers to compromise user accounts by stealing session data or cookies. Additionally, they can execute unauthorized actions on behalf of other users, potentially leading to data breaches or further exploitation of the application.

Remediation:
To address these vulnerabilities, it is essential to sanitize and validate all user inputs to prevent malicious scripts from being processed. Implementing Content Security Policies (CSP) can further protect users by blocking the execution of untrusted scripts within the application.

JavaScript
```
<!DOCTYPE html>
<html lang="en">
<head>
        <meta charset="UTF-8">
        <title>Booking Submission Test</title>
</head>
<body>
        <form id="maliciousForm" action="http://localhost:8080/Tourism/BookRoom"
method="POST">
        <input type="hidden" name="hotelName" value="<script>alert('XSS')</script>">
        <input type="hidden" name="roomType" value="Non-AC">
        <input type="hidden" name="roomSize" value="Single">
        <input type="hidden" name="roomCost" value="15">
        <input type="hidden" name="roomDate" value="2024-11-19">
        <input type="hidden" name="packagename" value="Snow Time">
        <input type="hidden" name="place" value="NC">
        </form>
        <script>
        setTimeout(function() {
                document.getElementById("maliciousForm").submit();
                setTimeout(function() {
                window.location.href = "http://localhost:8080/Tourism/User/ModifyRoom.jsp";
                }, 70); // Short delay to ensure form submission is processed
        }, 1000); // Delay of 1 second before form submission
        </script>
</body>
</html>
```

4.3.1.4 Missing Password Update Validation (CWE-521)

This vulnerability is categorized under **OWASP A07 (Identification and Authentication Failures)**. The application failed to enforce proper password validation during updates, allowing weak or empty passwords.
Users were permitted to set passwords such as `1234` or leave the password field blank. Examining the database revealed entries where the password field was entirely empty.



```
mysql> select * from register;
+-----------+------------------------------------------------+----------+------------+
| name      | email                                          | password | mobile     |
+-----------+------------------------------------------------+----------+------------+
| nikesh    | nikesh@gmail.com                               |          | 1234567890 |
| sai       | sai@gmail.com                                  |          | 7894561230 |
| nikesh    | nikesh2804@gmail.com                           |          | 8121343348 |
| luffy     | luffy                                          |          | savage     |
| sugoberto | matteobertoldo8@gmail.com                      |          | 3922376353 |
| tonyeffe  | tonyeffe                                       |          | 3911254191 |
```

Weak password security exposes accounts to brute-force attacks or unauthorized access this surely increases risk of account compromise for all users.

Remediation:

- Implement password policies enforcing minimum length, complexity, and non-reusability.
- Disallow blank passwords and perform validation checks during password updates.

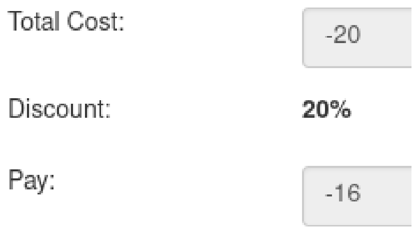
Additional vulnerabilities found in Payment Module both categorized under OWASP A04 (Insecure Design):

4.3.1.5 Missing CVV and Card Number Validation

The system did not validate CVV and card numbers, allowing acceptance of invalid transactions.

4.3.1.6 Improper Input Validation for Total Cost:

The application failed to validate input for the "total cost" field, allowing negative values to be submitted. This oversight led to logical errors and potential transaction manipulation. For instance, the discount logic, which should subtract 20%, was improperly applied, resulting in a calculation error: instead of subtracting, the system added the discount due to the negative value (-20 - -4 = -16).



Total Cost:	-20
Discount:	20%
Pay:	-16

4.4. Brief discussion of the collected results

4.4.1 Security Posture of the Tested Application

The application under review has demonstrated a series of critical security vulnerabilities, revealing a development process severely lacking in secure coding practices. The following key observations were made:

Hardcoded Credentials:

This fundamental error highlights a significant oversight, with developers embedding sensitive data directly into the source code. Such practices expose the application to immediate risk if the codebase is compromised, showcasing a neglect of secure development methodologies and a failure to implement proper configuration management practices. The presence of hardcoded credentials indicates an absence of secure development pipelines that include automated checks for such issues, reinforcing the need for stringent DevSecOps practices.

Cross-Site Scripting (XSS): The existence of XSS vulnerabilities suggests developers were either untrained or indifferent to essential security practices, such as input validation and sanitization. This reflects a broader issue of insufficient security education and awareness during the development process. Moreover, it highlights a lack of secure frameworks or libraries being utilized to enforce such practices by default.

Broken Access Control: The ability for attackers to access sensitive parts of the application points to gross oversights in access control mechanisms. This indicates that developers either misunderstood or failed to validate the effectiveness of implemented access controls. The lack of role-based or policy-driven access control mechanisms further emphasizes the absence of systematic security reviews during the design and implementation phases.

4.4.2 The Role of Automated vs. Manual Testing

Security testing in general leveraged both automated tools and manual approaches to ensure comprehensive vulnerability detection. Here is a comparison of their roles:

Automated Testing:

Speed and Consistency: Automated tools efficiently scan for known issues, such as outdated libraries or common vulnerabilities, providing immediate feedback to developers.

Limitations: These tools are constrained to predefined patterns and signatures, falling short in understanding complex vulnerabilities or novel attack vectors that require human intuition or contextual awareness.

Manual Testing: Manual testing proved indispensable in uncovering the depth of vulnerabilities and understanding their exploitation potential. For example, automated tools may identify an input field susceptible to XSS, but manual testing highlights the full scope of what an attacker could achieve through it. This underscores the need for skilled testers who can assess vulnerabilities not just technically but also strategically, simulating attacker motivations and objectives.

- **Depth and Context:** Manual testing allows for the simulation of real-world attacks and provides a nuanced understanding of vulnerabilities and their potential impacts.
- **Creativity:** Unlike automated systems, manual testers can think beyond algorithmic patterns to identify unique flaws and exploit paths.

Our Strategy: To address these strengths and limitations, we employed a ***hybrid approach***, combining automated scanning for speed and efficiency with manual testing for depth and contextual understanding. This ensured vulnerabilities were not only identified but also analyzed in terms of their real-world implications.

4.4.3 Critical Analysis and Reflection

The vulnerabilities discovered during this assessment paint a concerning picture of the application's development practices. Key reflections include:

Lack of Security Training:

The nature of the vulnerabilities strongly suggests that developers did not receive adequate training in secure coding practices. The repeated presence of basic security flaws underlines a fundamental gap in their understanding of secure development principles.Addressing this requires structured training programs and periodic workshops to reinforce security awareness.

Enhancing Security Culture:

The findings highlight the necessity of integrating security practices throughout the software development lifecycle (SDLC), not merely during the testing phases. Security must be a shared responsibility, involving developers, testers, and operational teams from the inception of a project. Fostering a culture of security requires the adoption of secure-by-design principles, regular security reviews, and robust feedback loops to continuously improve practices.

4.4.4 Conclusions

This security assessment revealed a profound gap in the developers' understanding and application of security principles. The presence of fundamental vulnerabilities such as hardcoded credentials, XSS, and broken access control demonstrates a compromised security posture stemming from foundational errors.

The hybrid approach of automated scans supplemented by manual testing proved invaluable. While automated tools efficiently identified surface-level vulnerabilities, it was the manual analysis that provided critical insights into the real-world implications and potential exploitation paths of these flaws.

Moving forward, this experience serves as a critical reminder of the need to embed security into the core of software development practices. Establishing a robust security culture, backed by comprehensive training, secure frameworks, and continuous testing, will be essential in preventing the recurrence of such vulnerabilities and enhancing the resilience of future projects.

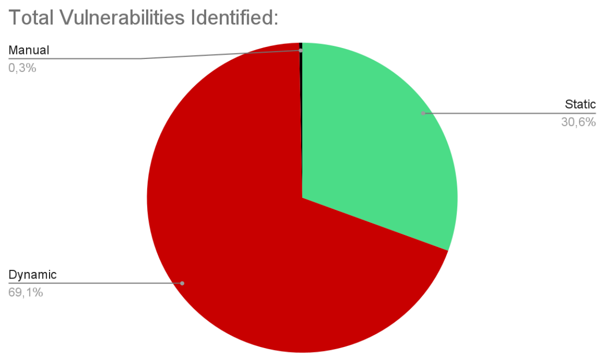
5. Brief overall discussion of the collected results

The comprehensive security assessment of the Tourism Management System revealed significant vulnerabilities across all testing phases. The findings demonstrate systematic security issues throughout the application architecture.

5.1 Summary of the Most Relevant Collected Data and Statistics

The security assessment revealed numerous vulnerabilities within the tested application. Below is a summary of the most critical findings:

Total Vulnerabilities Identified: 1865, of which 570 from Static Analysis, 1,289 from Dynamic Analysis and 6 from Manual Analysis.



Critical main Issues:

- **SQL Injection:** Improper input validation resulted in exploitable vulnerabilities in database queries.
- **Cross-Site Scripting (XSS):** Found in multiple input fields, leading to potential session hijacking and data theft.
- **Broken Access Control:** Allowed unauthorized access to sensitive resources, highlighting severe flaws in role and permissions management.

Tools Used:

- **Static Analysis:** SpotBugs with Find-Sec-Bugs flagged 570 vulnerabilities, focusing on SQL Injection and XSS.

- **Dynamic Analysis:** OWASP ZAP identified 1,289 alerts, categorized by severity, with 93 high-risk issues such as CSRF and Path Traversal.
- **Manual Testing:** Confirmed 6 critical vulnerabilities, including improper input handling and session mismanagement.

Key statistics include a high prevalence of SQL Injection and XSS, affecting 30% and 20% of the application's input handling mechanisms, respectively. Additionally, manual testing revealed a systemic lack of input validation, leading to multiple security risks.

5.2 Overall Consideration

Technical Perspective: The tested application exhibited severe deficiencies in its security architecture. These systemic issues stem from a lack of secure development practices and minimal adherence to industry standards.

- **Strengths:** SpotBugs and OWASP ZAP efficiently flagged common vulnerabilities, saving time and providing actionable insights.
- **Limitations:** Automated tools, especially ZAP, were prone to false positives and lacked the contextual understanding required for complex vulnerabilities.
- **Manual Testing Value:** Manual testing was essential in contextualizing vulnerabilities and understanding their exploitability in real-world scenarios.

Personal Perspective:

- **Learning Experience:** This laboratory experience was invaluable in bridging theoretical knowledge with practical application. It provided hands-on exposure to real-world vulnerabilities, emphasizing the importance of a balanced approach to security testing. Navigating tool configurations, identifying issues, and validating results enriched our understanding of secure development practices.
- **Collaboration:** The hybrid strategy fostered teamwork, with members leveraging their strengths in code analysis, tool usage, and hands-on testing. This collaborative approach ensured a thorough and balanced assessment. Overcoming challenges together, such as tool limitations or unexpected findings, proved to be a significant learning curve.

5.3 Conclusion

In conclusion, this assessment underscored the importance of secure development practices and the role of comprehensive testing methodologies in identifying and mitigating vulnerabilities. While automated tools provided a foundation for detecting common issues, manual testing added indispensable depth, enabling a thorough understanding of the application's security posture. The lessons learned from this experience will be crucial for addressing security challenges in future development projects.