Bachelor of Science with Honours in Cyber Security

COURSEWORK No. 2

Module Name : Secure Programming

Module Code : PSB600IT

Weighting : 60%

Due Date : 25 June 2026

Submitted By

Name : Don Loo Boon Kiat

CU Index No. : 16907748

Word Count : 5206

Table of Contents

1 Introduction 4

1.1 Project Overview 4

1.2 Objectives of the Secure System 4

1.3 Methodology Overview 4

2 Literature Review 4

2.1 Security in the Software Development Lifecycle 4

2.2 Secure Software Engineering and Programming Techniques 5

2.3 Standards-Based Approaches 5

2.4 Tools to Support Secure Development 5

2.5 Critical Comparison of Frameworks 6

3 System Design and Implementation 6

3.1 Platform Selection and Development Environment 6

3.2 Secure Architecture and Design Principles 7

3.3 Threat Modelling with STRIDE 8

3.4 Vulnerabilities and Secure Coding Countermeasures 9

3.4.1 SQL Injection 9

3.4.2 Plaintext Password Storage 10

3.4.3 Insecure Direct Object Reference 11

3.4.4 Privilege Escalation 13

3.4.5 Price Tampering 15

3.4.6 Stored Cross-Site Scripting 18

4 Security Testing and Analysis 20

4.1 Dynamic Testing (Manual Exploitation) 20

4.2 Static Analysis with Bandit 21

4.3 Towards More Comprehensive Security Testing 22

5 Discussion 23

5.1 Challenges and Limitations 23

5.2 Comparison with Industry Best Practices 23

5.3 Lessons Learned and Future Improvements 24

5.4 Additional Security Controls (Defence in Depth Roadmap) 24

6 Conclusion 25

7 References..... 25

1 Introduction

1.1 Project Overview

The more complex web applications become, the higher the likelihood that unintentional flaws leading to security breaches may occur. Modern software engineering mandates that security be considered part of the software development process (SDLC) from day one. In this document, the development and security tests conducted on MatchPoint, a tennis related web application created with Python programming language and Flask framework are described. Two functionally identical builds have been developed, a vulnerable version and a hardened secure version.

1.2 Objectives of the Secure System

The secure architecture aims at mitigating six specific weaknesses found within the OWASP Top 10 and the STRIDE security model, which includes SQL injection, storing passwords as plain text, broken access control, stored cross-site scripting and price manipulation. The intention is to counter these vulnerabilities by using parameterised queries, hashing passwords, performing server-side authorisation, escaping outputs and validating inputs, among others.

1.3 Methodology Overview

The opening part provides a literature review on secure software engineering and the use of standards-based frameworks such as OWASP, CERT, SANS and STRIDE. Following this is an overview of how the two versions of the application were developed, code hardening process that was utilised and static and dynamic testing that was carried out using Bandit.

2 Literature Review

2.1 Security in the Software Development Lifecycle

From an issue to address after deployment, security has become a built-in feature of the entire software development life cycle (SDLC). The Microsoft Security Development Lifecycle introduced by Howard & Lipner (2006) provides concrete evidence of how different phases in software development process have their own associated security activity. In this regard, McGraw (2006) suggests that it is cheaper to find defects when developing the software because it costs just a small percentage of the costs that arise once a defect is detected in production. Therefore, adopting security as a holistic activity and not just part of the testing phase is considered best practice for projects.

2.2 Secure Software Engineering and Programming Techniques

Secure software engineering involves the application of defensive software development principles, including least privilege, defence in depth, fail-safe defaults and complete mediation, during the designing and building processes (McGraw, 2006). With regards to implementing secure programming practices, there are measures against various types of vulnerabilities, including input / output checking for protection against injection and cross-site scripting attacks, use of parameterised queries to ensure separation between code and data to prevent SQL injection attacks, using strong hashing for password storage and server-side authorisation checks to address broken access control. According to Seacord (2013), numerous vulnerabilities are caused by trusting input that cannot be trusted, and that consistency and language awareness are key to prevention.

2.3 Standards-Based Approaches

Several community guidelines govern the secure software development process. OWASP Top 10 represents one of the widely used awareness documents which represent a consensus regarding the topmost vulnerabilities related to the development of web applications, where the latest OWASP Top 10 continues the placement of Broken Access Control at position A01, as well as places injection at position A05 (OWASP, 2025). The SEI CERT coding standards are represented by guidelines from Carnegie Mellon University (Software Engineering Institute n.d.) for certain programming languages, while CWE / SANS Top 25, developed by the MITRE Corporation, describes the topmost vulnerabilities in terms of severity (MITRE Corporation, 2025). One of the proactive approaches to the threat identification is threat modelling, which identifies threats according to the classification of the STRIDE model developed by Shostack (2014). The current project identifies threats through the lenses of both STRIDE and OWASP methodologies.

2.4 Tools to Support Secure Development

The automation of the process helps reveal vulnerabilities that might not have been recognised through manual testing. In Static Application Security Testing (SAST), source code is analysed without actually being run, for Python code, Bandit can be used to find things like secrets that are coded directly into code and risky settings. DAST focuses on finding vulnerabilities within the running application, whereas sandboxing and fuzz testing expand the test coverage. According to McGraw (2006), tools act as a supplement and not as an alternative to secure design and are most useful if used along with the SDLC process and human involvement. This project follows that philosophy by conducting both automated and manual vulnerability testing.

2.5 Critical Comparison of Frameworks

Despite similarities in these frameworks mentioned above, they have different goals and thus cannot be substituted for each other. It can be noted that the OWASP Top 10 operates as a document aimed at creating awareness through prevalence and exploitability data. It is easy to understand and always up to date but only descriptive and not prescriptive with respect to application verification completeness (OWASP 2025). In its turn, the OWASP Application Security Verification Standard (ASVS) fills the gap of the former with the help of providing testable requirements classified into different assurance levels. Thus, ASVS is more appropriate for use to evaluate application hardening comprehensiveness than OWASP Top 10 (OWASP, 2021). Meanwhile, SEI CERT standards operate on the level of coding constructs and are more effective in preventing implementation errors while CWE / SANS Top 25 ranks weakness by severity (MITRE Corporation, 2025). In addition, STRIDE framework differs from other frameworks since it is used for eliciting threats at the design stage, and not for providing information about known vulnerabilities (Shostack, 2014). With the help of this framework, one could elicit some risks that are not covered by vulnerability-based frameworks like repudiation.

One of the most consistent concepts described in literature on secure software engineering is that none of the mentioned frameworks is sufficient alone, and thus, defence in depth and security by design are implemented with the help of combining design time threat modelling with construction time coding standard and verification time test (Viega and McGraw, 2001; McGraw, 2006). The national guidance also represents a similar position in that it suggests using an integrated approach that includes these steps into the whole lifecycle and not just one part (NIST, 2022). In the context of the current project, all the mentioned frameworks were used simultaneously and not separately as it was necessary to reflect prevalence, severity, implementation, correctness and design level risk in evaluating the project.

3 System Design and Implementation

3.1 Platform Selection and Development Environment

MatchPoint is implemented using the Python programming language with the help of the Flask framework due to its easy routing capability and built-in support for Jinja2 template engine. The use of SQLite allowed avoiding a standalone database server thus making two versions completely self-contained. Two applications, functionally similar but different in terms of security were created, one vulnerable application operating on port 5000 and one hardened version running on port 5001 (Figure 1). The seed.py script was used to set up identical user accounts and test data, so the only difference

between the two versions was in the security mechanisms used. Three user accounts were set up, administrator account as well as member accounts of DonL and BobL.

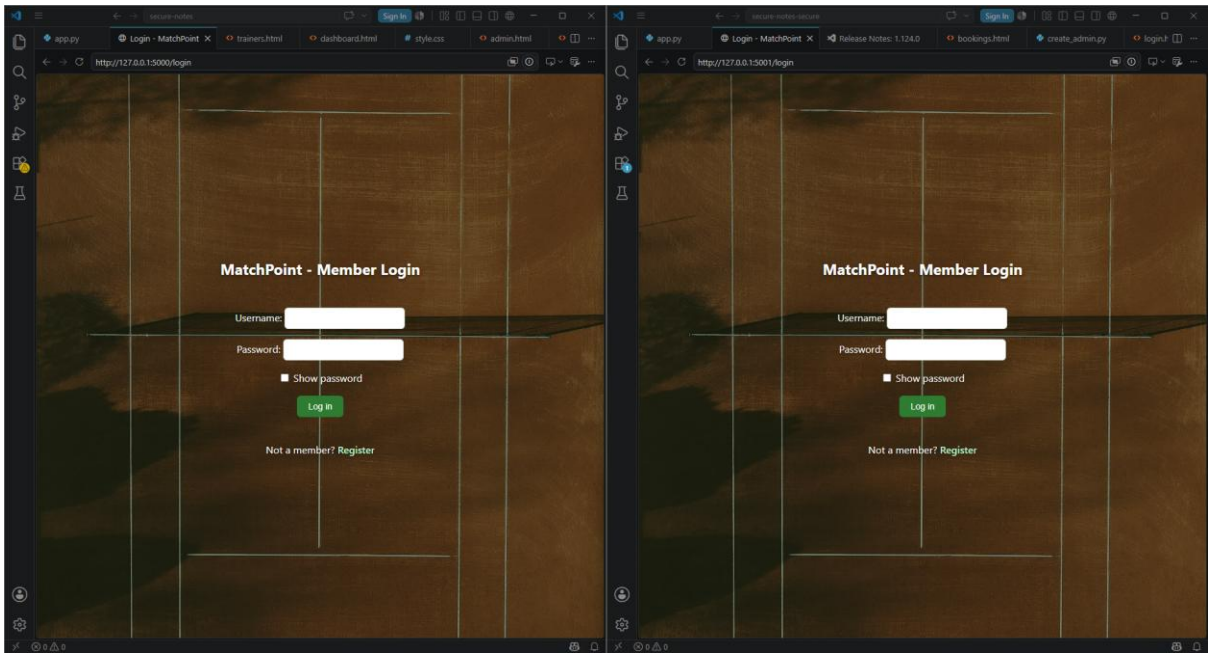


Figure 1 - Vulnerable Application (Left), Secure Application (Right)

3.2 Secure Architecture and Design Principles

In addition to mitigating individual vulnerabilities, the secure build itself is designed following a minimal list of security design principles to make sure that security is never dependent on any specific control. The application is structured with clear separation of trust boundaries between the browser / client supplied data (untrusted), the Flask application layer handling authentication, authorisation and validation and the SQLite database accessed via the application only. The complete mediation principle is realised via performing all authorisations on the server in each protected request, as opposed to hiding privileged operations via UI elements. The least privilege principle is represented in the session-based approach where the user has access only to their own records according to the server.

Defence in depth is provided via making sure that there should be multiple points of failure for an attacker to exploit successfully. In particular, the price tampering mitigation not only validates the number submitted by the user but ignores client input and considers the value from the database to be the only valid one, whereas IDOR countermeasure applies record ownership at the level of SQL queries instead of accepting this information as provided in the request parameter. The application is designed using the principle of secure default, Jinja2 auto-escaping is kept enabled, the debug mode is turned off, and the secret key is read from environment variables instead of being hardcoded. Such an approach represents the security by design paradigm

described in literature as the one in which security is the default and developers should take deliberate steps to increase risks (McGraw, 2006; Shostack, 2014).

3.3 Threat Modelling with STRIDE

In order to verify that the vulnerabilities that were considered originated from analysis as opposed to being randomly chosen, the MatchPoint application underwent the STRIDE assessment before the secure build was implemented. It involved the decomposition of the application into its main components including the login module, the booking form, the trainer-hire form, the per user information stored on SQLite, the administration section and the information flow through the boundary of the untrusted browser and trusted server. All the above-mentioned components have been analysed against the six STRIDE categories. The table below presents the threat matrix based on the STRIDE assessment.

Table 1 (STRIDE threat matrix for MatchPoint)

STRIDE Category	Threat in MatchPoint	Entry Point / Asset	OWASP Mapping	Mitigation In Secure Build
Spoofing	Authentication bypass through SQL injection on the login form	Login form and stored credentials	A05 Injection	Parameterised queries; credentials verified server-side against scrypt hashes
Tampering	Altering the trainer price and injecting script into booking notes	Trainer-hire form; special-requests field	A05 Injection and A06 Insecure Design	Authoritative price read from the database; Jinja2 output escaping enabled
Repudiation	Actions cannot be reliably attributed to a user owing to the absence of an audit trail	All state-changing actions	Security logging and monitoring	Not yet implemented (see Section 5.4): structured event logging
Information Disclosure	Exposure of stored passwords and of bookings belonging to other users	User table; booking records	A04 Cryptographic Failures and A01 Broken Access Control	Scrypt password hashing; per-query ownership checks
Denial of Service	Unthrottled repeated requests to the login and booking endpoints	Login and booking endpoints	Resource exhaustion (outside the core Top 10)	Not yet implemented (see Section 5.4): request rate limiting
Elevation of Privilege	Direct access to the admin panel by a standard user	The /admin route	A01 Broken Access Control	Server-side role check returning HTTP 403

It is clear from the matrix that the six risks have in total covered five out of the six STRIDE types, which provides a fairly good coverage of the STRIDE model. In addition, it becomes evident what the remaining risks are, namely repudiation / denial of service, because there is no auditing / audit logging performed and there is no request rate limiting in the application. The recognition of these weaknesses during the design phase is in itself a positive aspect of threat modelling, and both of them will be covered in more detail later in Section 5.4. Therefore, the matrix serves not only as a justification of the scope of the secure build, but also identifies further weaknesses which need to be addressed.

3.4 Vulnerabilities and Secure Coding Countermeasures

3.4.1 SQL Injection

This build of the vulnerable system purposefully exhibits six common vulnerabilities of web applications, classified using the OWASP Top 10 and STRIDE framework. On the other hand, the secure build incorporates a known and trusted solution to each of the above-mentioned security flaws. More specifically, the issue of SQL injection is handled. In this case, the vulnerable login function appends user input into an SQL command to allow for a payload of "'OR'1'='1" to be passed without authentication. The secure build counters such attacks by making use of parameterised queries using placeholders (Figure 2).

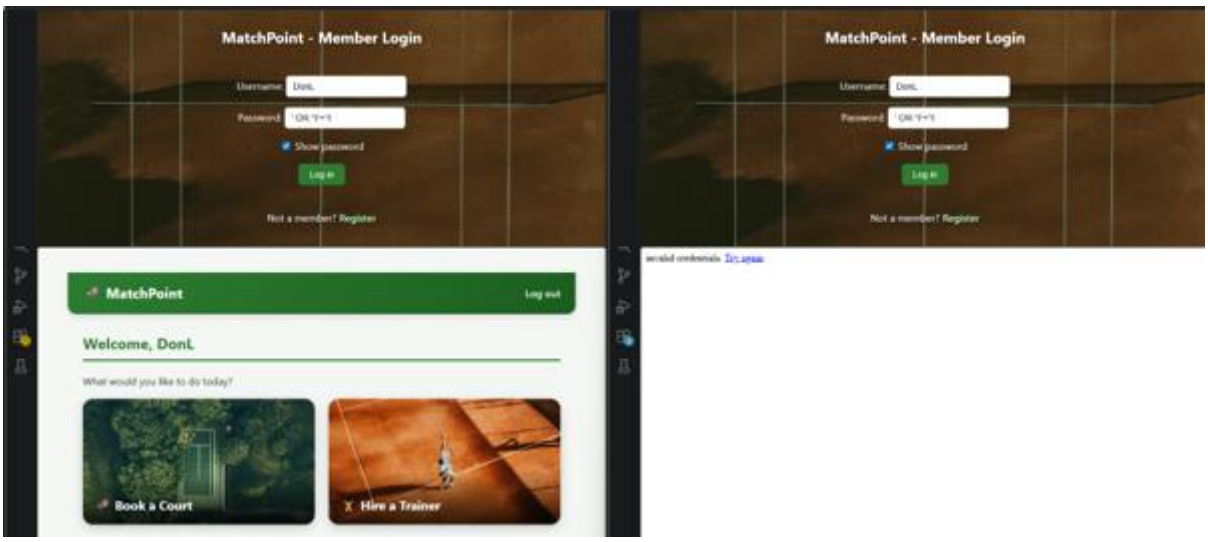


Figure 2 - SQL Injection on both Vulnerable (Left) & Secure (Right) Applications

3.4.1 SQL Injection Solution

The solution entails replacing the use of string concatenation to create the SQL statement with a query that utilizes the "?" placeholder and passes the values as distinct parameters. In turn, the SQL driver recognises this value only as data and does not execute it as code within the SQL statement. Thus, the payload "'OR'1'='1" is recognised as just a username and not executable code (Figure 3).

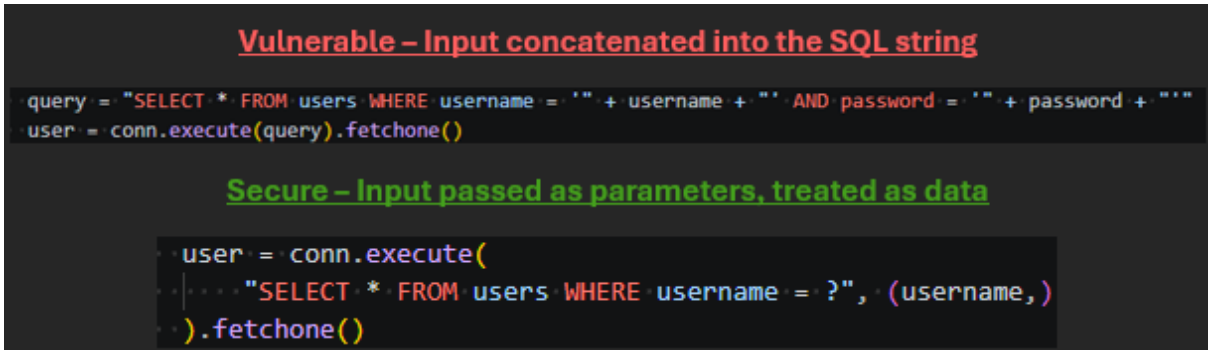


Figure 3 - SQL Injection Solution

3.4.2 Plaintext Password Storage

Storing passwords in plaintext is known as a cryptographic failure. The insecure system stores the passwords in a readable format. In the case of the secure system, however, passwords are hashed using the scrypt algorithm during the process of registering a user using the generate_password_hash() function from Werkzeug (Figure 4 and 5).

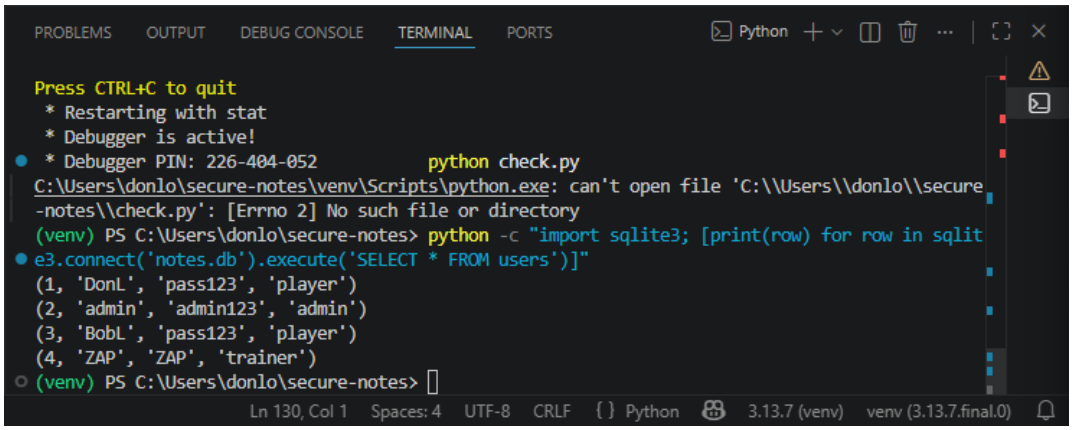


Figure 4 - Plaintext password in Vulnerable Application

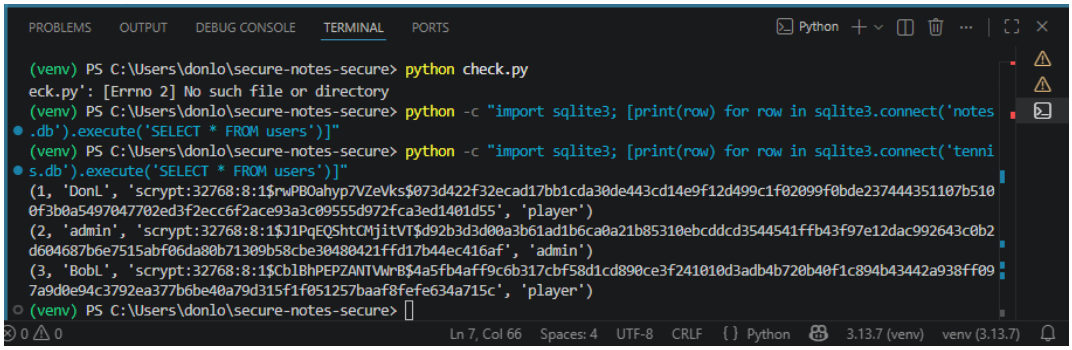


Figure 5 - Hashed password in Secure Application

3.4.2 Plaintext Password Storage Solution

The solution to this problem would be to hash the password before storing it through the use of “generate_password_hash()” from Werkzeug, which uses scrypt algorithm to hash it with a randomly generated salt. Since hashing is an irreversible process, the saved data will not be able to be reverted to its original state and the scrypt algorithm is

slow and requires lots of memory to try to prevent any brute-forcing attempts on it (Figure 6).

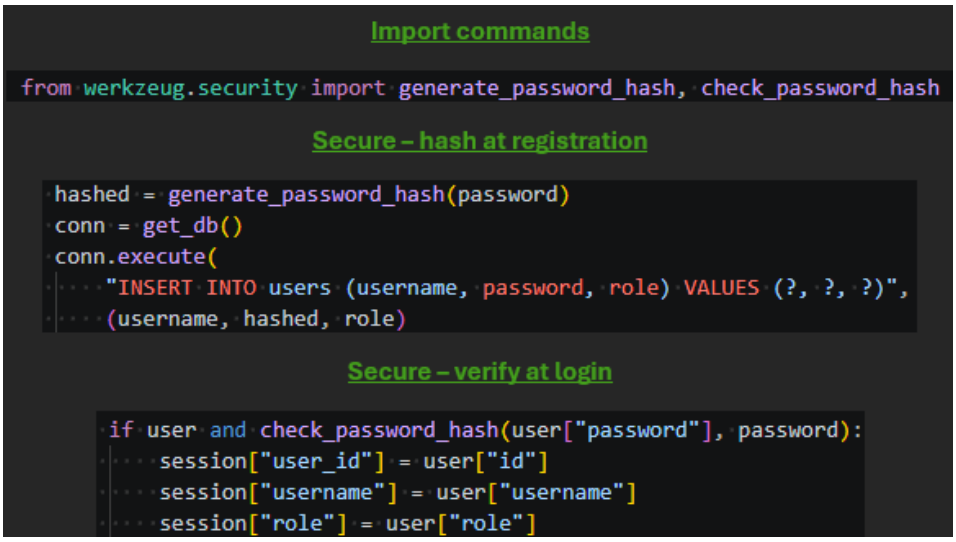


Figure 6 - Plaintext Password Storage Solution

3.4.3 Insecure Direct Object Reference

Insecure Version

First, we will cover the insecure version. When another user's reservation is requested based on their ID, their private information related to that ID is exposed. Referring to (Figure 7), we can observe that user 'DonL' has a court booking confirmed.

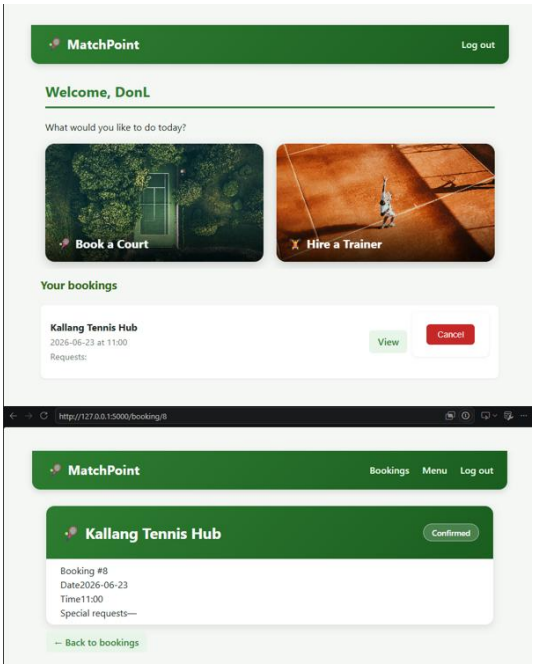


Figure 7 - Confirmed booking of court

However, due to the vulnerable application fetching its bookings through the ID of the URL, anyone will be able to view and access the booking as seen below in (Figure 8).

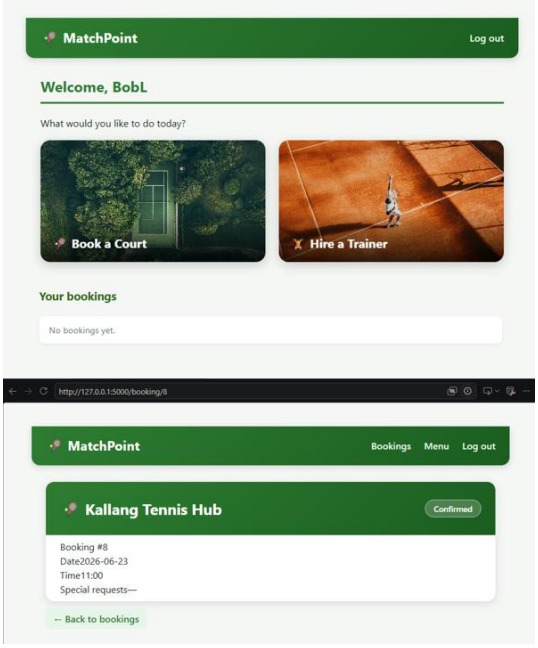


Figure 8 - BobL accessing DonL booking

Secure Version

Next, we will be covering the secure version. Where the ownership of the record is verified at the database level using the record ID as well as the session user, which in terms denies anyone access to view the bookings under other users (Figure 9 and 10).

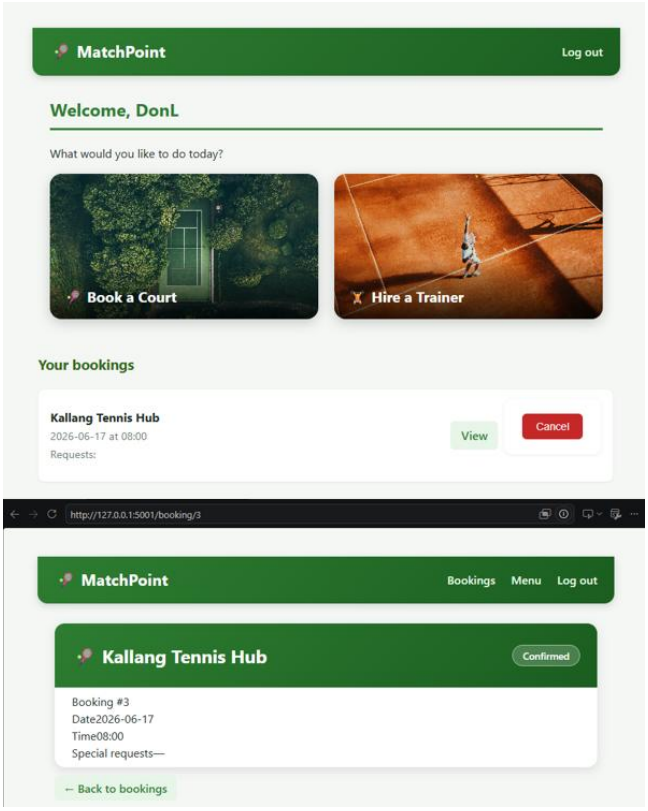


Figure 9 - DonL court booking

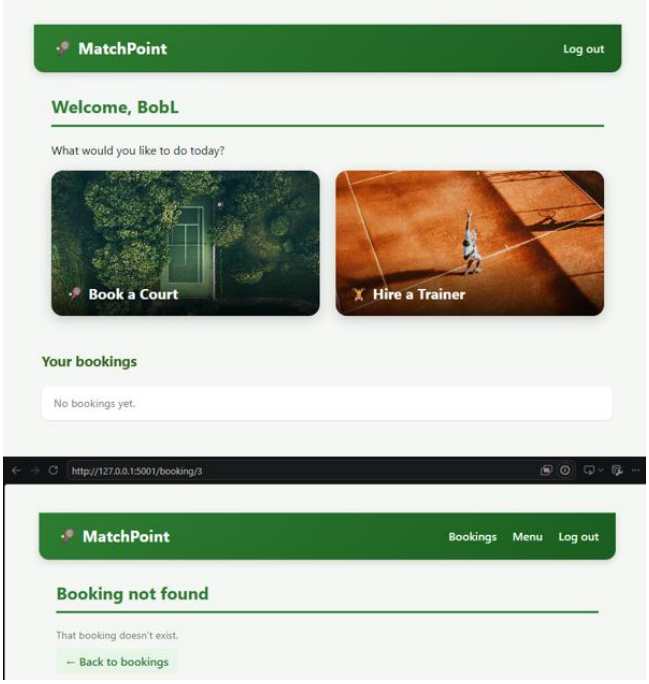


Figure 10 - BobL attempting to access DonL court booking

3.4.3 Insecure Direct Object Reference Solution

The mitigation process requires ownership validation on a per-query basis. In the susceptible code version, the booking record was fetched based on the ID embedded in the URL only. Thus, allowing any authenticated user to fetch any booking records. The fixed code incorporates an additional check clause, “AND user_id = ?”, where the parameter is obtained from the logged-in user’s ID which cannot be manipulated by the user. In case the booking does not belong to the requester, the query will fail to return any data (Figure 11).

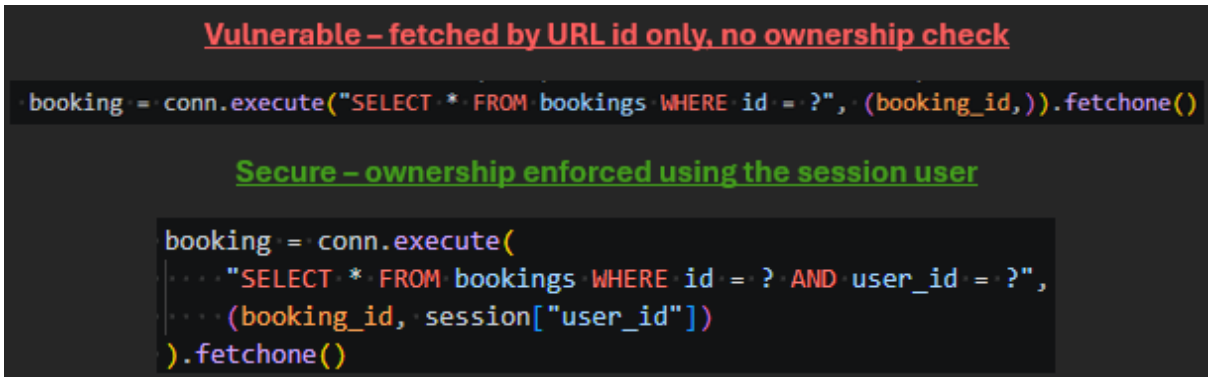


Figure 11 - Insecure Direct Object Reference Solution

3.4.4 Privilege Escalation

One major flaw in the vulnerable application is the ability to access the admin panel by any user who visits the link directly. Conversely, the secure build restricts users from accessing the panel unless they meet role requirements on the server, as seen below. Concealing links should not be considered access control (Figure 12).

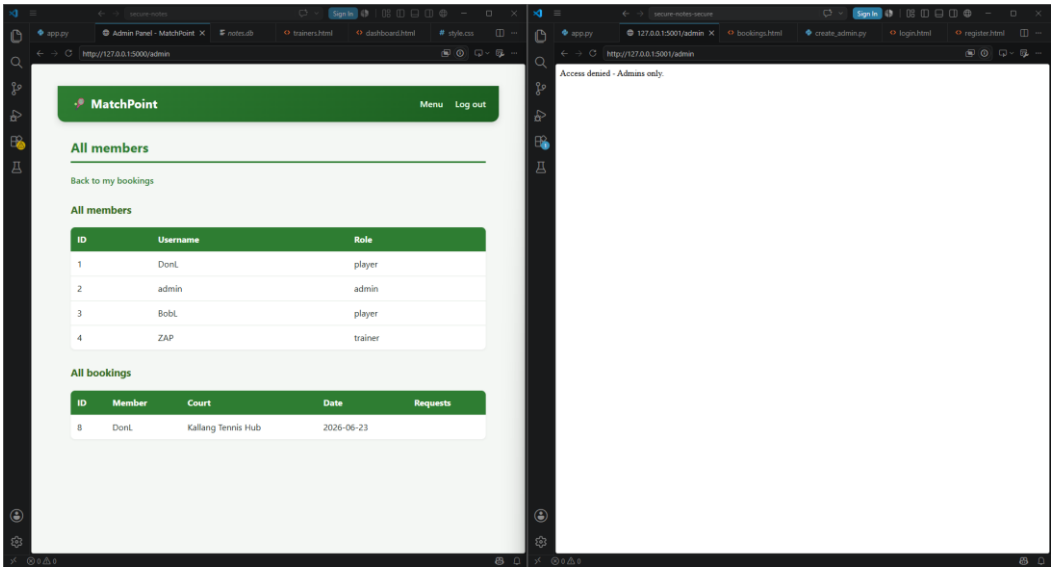


Figure 12 - DonL accessing Vulnerable Application's Admin Panel (Left), DonL denied access to Secure Application's Admin Panel (Right)

3.4.4 Privilege Escalation Solution

The solution for this is to implement a server-side authorisation that is added for the admin route. In the initial code snippet, any user that visited / admin would get access to the admin page. In this improved version, the program checks whether the user's role, saved in the session variable, is an administrator and returns "HTTP 403 Forbidden" for users who are not administrators. Here we can see that hiding the link from the user interface is not enough for implementing access control (Figure 13).

Vulnerable – no authorisation check

```
@app.route("/admin")
def admin():
    if "user_id" not in session:
        return redirect(url_for("login"))
    conn = get_db()
    all_bookings = conn.execute("""
        SELECT bookings.id, users.username, bookings.court, bookings.booking_date, bookings.requests
        FROM bookings
        JOIN users ON bookings.user_id = users.id
    """).fetchall()
    all_users = conn.execute("SELECT id, username, role FROM users").fetchall()
    conn.close()
    return render_template("admin.html", bookings=all_bookings, users=all_users)
```

Secure – server-side role check

```
@app.route("/admin")
def admin():
    if "user_id" not in session:
        return redirect(url_for("login"))
    if session.get("role") != "admin":
        return "Access denied - Admins only.", 403
    conn = get_db()
    all_bookings = conn.execute("""
        SELECT bookings.id, users.username, bookings.court, bookings.booking_date, bookings.requests
        FROM bookings
        JOIN users ON bookings.user_id = users.id
    """).fetchall()
    all_users = conn.execute("SELECT id, username, role FROM users").fetchall()
    conn.close()
    return render_template("admin.html", bookings=all_bookings, users=all_users)
```

Figure 13 - Privilege Escalation Solution

3.4.5 Price Tampering

Insecure Version

The insecure trainer-hire form uses a price value provided by the client that can be changed in the browser to register an amount of $0.

As we can observe from (Figure 14 and 15), accessing the developer tools for the trainer's page allows as to manipulate the pricing that is stated on the website when hiring a trainer.

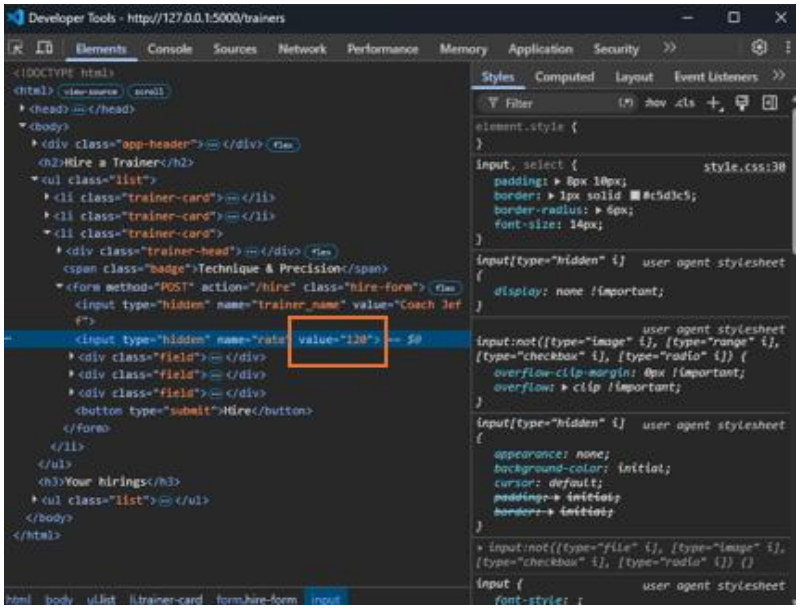


Figure 14 - Accessing Developer Tools (Insecure Version)

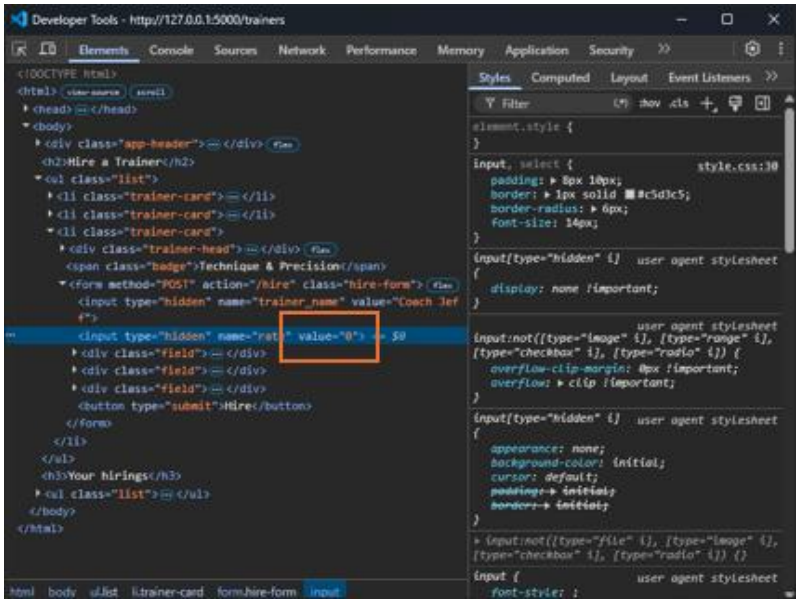


Figure 15 - Changing value of trainer hiring price (Insecure Version)

This ultimately allows us to hire a trainer for $0 as reflected in (Figure 16).

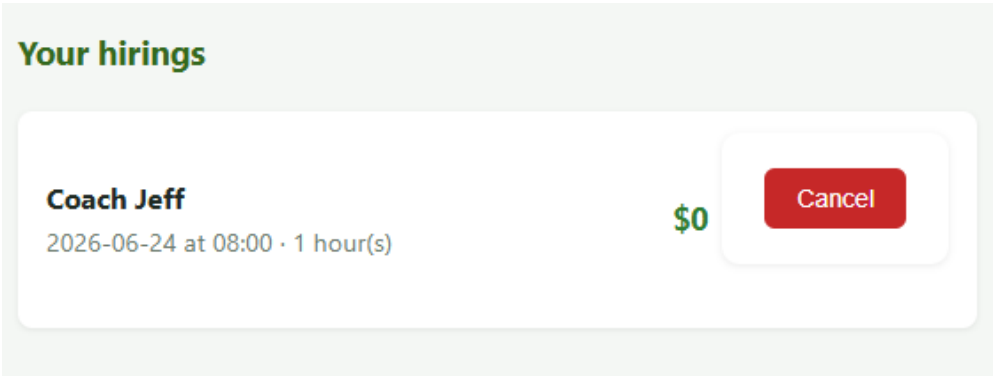


Figure 16 - Hired trainer for $0

Secure Version

Compared to the insecure version, the price in the secure version of the page is ignored, and the official price is obtained from the server making it impossible for the user to change the pricing of the trainer (Figure 17 and 18).

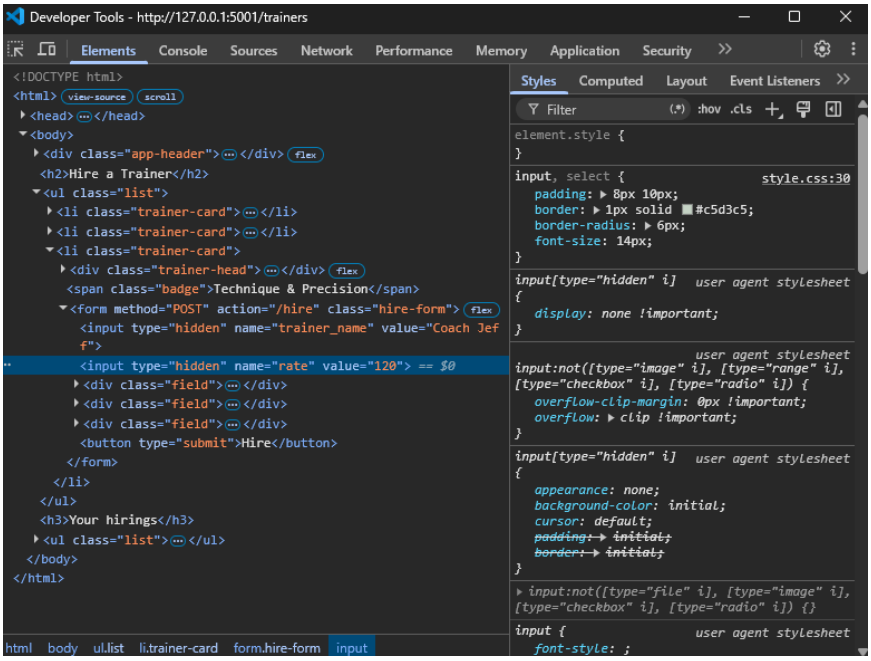


Figure 17 - Accessing Developer Tools (Secure Version)

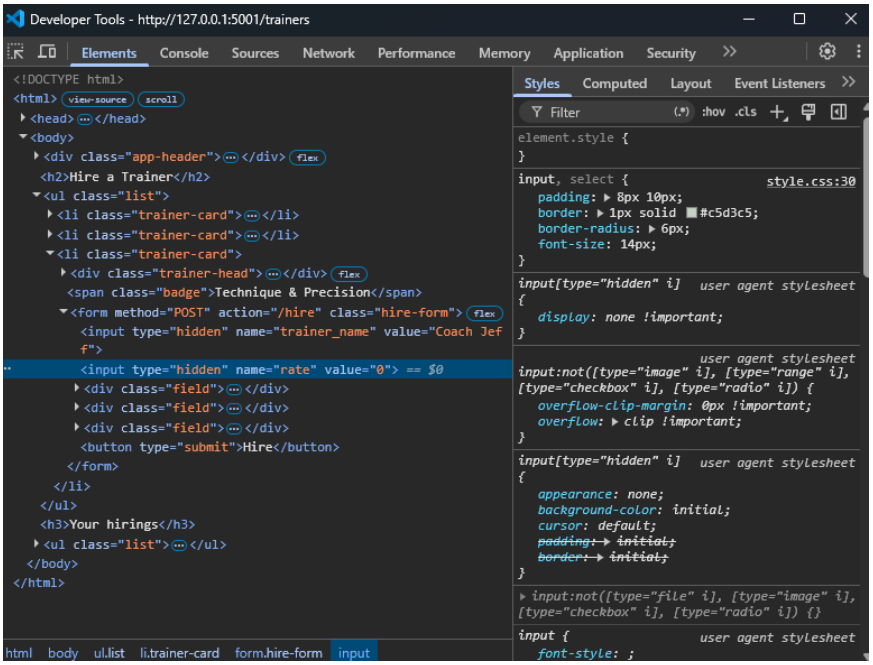


Figure 18 - Attempting to change price of trainer (Secure Version)

Even after changing the value to $0, the price shown on the hired page is the original price of $120. This tells us that the official price is obtained from the server instead of the page (Figure 19).

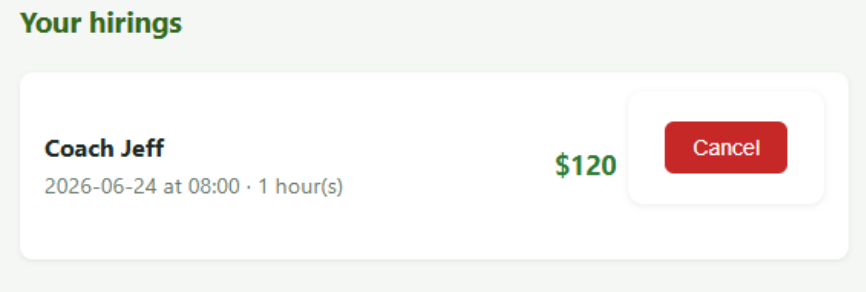


Figure 19 - Outcome of hiring trainer

3.4.5 Price Tampering Solution

The solution requires that there be an end to the use of the price as provided by the browser. The flaw here is that, in the compromised code, the price is determined from data provided in the form, which can be manipulated by the user to have the value of $0. However, in the fixed code, the submitted figure is ignored, and the actual figure is retrieved from the database (Figure 20).

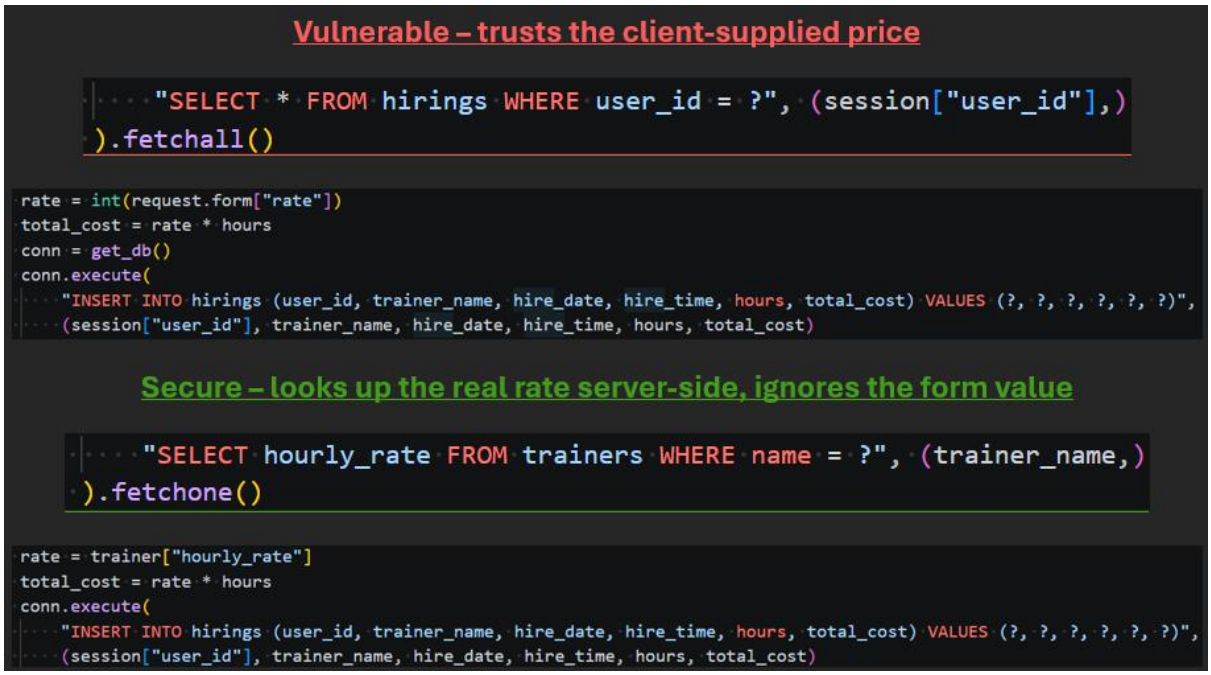


Figure 20 - Price Tampering Solution

3.4.6 Stored Cross-Site Scripting

Insecure Version

In the court booking form, there is a “special requests” box that can be found, whose content is also shown on the bookings page. In the vulnerable version, the “special requests” box is outputted unescaped since the autoescaping feature of Jinja2 has been disabled by the use of the “| safe” filter. This means that when there is some JavaScript code within the box, it is also saved and executed in the browser of anyone viewing that particular booking, as can be seen in (Figure 21) when entering “<script>alert(‘XSS’)</script>”.

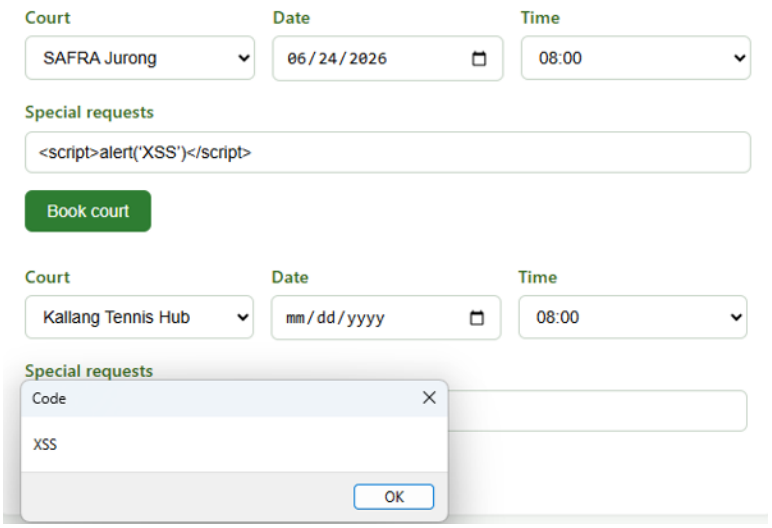


Figure 21 - Performing XSS attack (Vulnerable Application)

Secure Version

Under the secured setting, the automatic escaping feature offered by Jinja2 is again restored as the "|safe" filter is removed, allowing the input made to be treated as a simple test and not executable code. As seen in (Figure 22), the exact same payload becomes harmless since it is now seen as the actual text of the script and not executed.

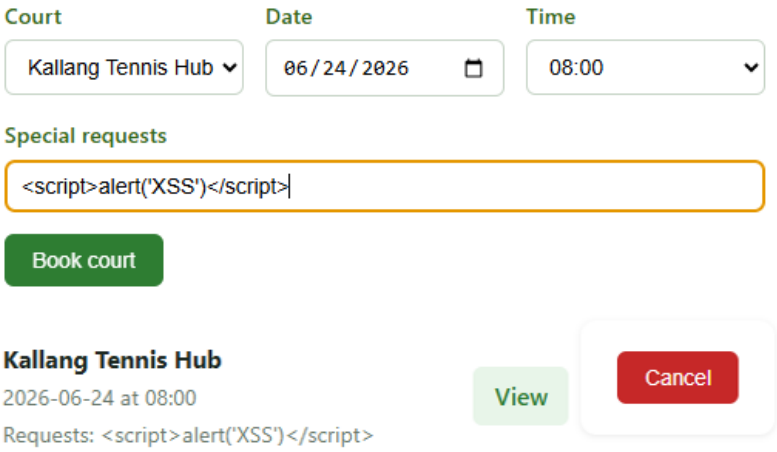


Figure 22 - Performing XSS attack (Secured Application)

3.4.6 Stored Cross-Site Scripting Solution

The solution has been applied through the template itself, and not through any modification to the Python code. The insecure code used the "| safe" filter from Jinja2 to render the "special requests" field. Thus, preventing the automatic HTML escaping of content in this part of the page. Consequently, the store script would run when the user viewed their own data. The removal of the "| safe" filter allows Jinja2's built in auto-escaping feature to take effect, rendering potentially dangerous characters as HTML entities (Figure 23).

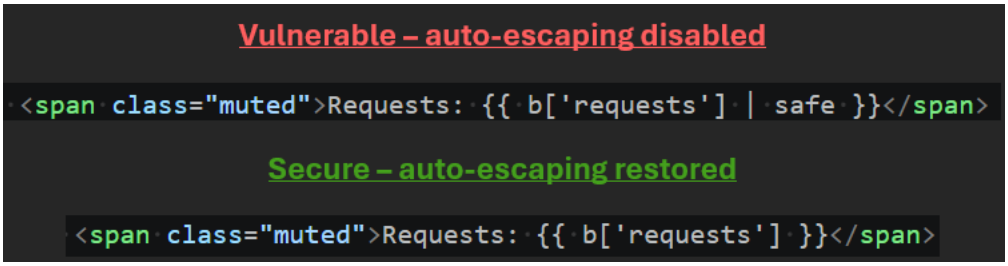


Figure 23 - Stored Cross-Site Scripting Solution

4 Security Testing and Analysis

For security testing and analysis, this was achieved by using the two methods together, testing dynamically through manual penetration of the functioning programs and analysis statically of the source code. This approach to analysis is consistent with academic opinion on the matter, whereby manual testing and automation achieve the best possible outcomes when used together (McGraw, 2006).

4.1 Dynamic Testing (Manual Exploitation)

For each of the vulnerabilities, the exploit was done manually for both versions to prove the expected behaviour. The attack works when the application version under test is vulnerable (port 5000), whereas it does not work when the application version under test is secured (port 5001). Multiuser vulnerabilities, including IDOR and privilege escalation tests, required two users being able to log in at once. Hence, two browsers, one insecure and one secured were used. Table 1 shows the result summarised into a table format.

Table 2 (Dynamic testing results across both builds)

Vulnerability	OWASP Top 10	STRIDE	Vulnerable (5000)	Secure (5001)
SQL Injection (login bypass)	A05 Injection	Spoofing	Login bypassed	Blocked
Plaintext password storage	A04 Cryptographic Failures	Information disclosure	Passwords readable	Hashed (scrypt)
Insecure direct object reference	A01 Broken Access Control	Information disclosure	Other user's booking shown	Blocked
Privilege escalation	A01 Broken Access Control	Elevation of privilege	Admin panel reached	Access denied (403)
Stored cross-site scripting	A05 Injection	Tampering	Script executed	Rendered as text
Price tampering	A06 Insecure Design	Tampering	Charge recorded as $0	Real rate ($120) enforced

As observed from all of these cases, the secure build was able to stop the attacks, indicating that the only difference between the two is the implementation of security mechanisms, not their functionality.

4.2 Static Analysis with Bandit

Bandit was used for the static analysis process. It is a Python program security tool. This process was carried out by running “bandit -r . –exclude ./venv ” Static scanning of the code without executing the code identified configuration vulnerabilities in the vulnerable system that could not be easily identified manually (Figure 24 to 27).

Insecure Version

```
Run started:2026-06-12 09:11:24.876336+00:00

Test results:
>> Issue: [B105:hardcoded_password_string] Possible hardcoded password: 'temporary-key-we-will-fix-this-later'
   Severity: Low   Confidence: Medium
   CWE: CWE-259 (https://cwe.mitre.org/data/definitions/259.html)
   More Info: https://bandit.readthedocs.io/en/1.9.4/plugins/b105_hardcoded_password_string.html
   Location: .\app.py:5:17
4       app = Flask(__name__)
5       app.secret_key = "temporary-key-we-will-fix-this-later"
6
```

Figure 24 - Vulnerability 1

```
>> Issue: [B608:hardcoded_sql_expressions] Possible SQL injection vector through string-based query construction.
   Severity: Medium   Confidence: Low
   CWE: CWE-89 (https://cwe.mitre.org/data/definitions/89.html)
   More Info: https://bandit.readthedocs.io/en/1.9.4/plugins/b608_hardcoded_sql_expressions.html
   Location: .\app.py:93:16
92              conn = get_db()
93              query = "SELECT * FROM users WHERE username = '" + username + "' AND password = '" + password + "'"
94              user = conn.execute(query).fetchone()
```

Figure 25 - Vulnerability 2

```
>> Issue: [B201:flask_debug_true] A Flask app appears to be run with debug=True, which exposes the Werkzeug debugger and allows
the execution of arbitrary code.
   Severity: High   Confidence: Medium
   CWE: CWE-94 (https://cwe.mitre.org/data/definitions/94.html)
   More Info: https://bandit.readthedocs.io/en/1.9.4/plugins/b201_flask_debug_true.html
   Location: .\app.py:214:4
213         init_db()
214         app.run(debug=True)
```

Figure 26 - Vulnerability 3

```
Code scanned:
        Total lines of code: 203
        Total lines skipped (#nosec): 0
        Total potential issues skipped due to specifically being disabled (e.g., #nosec BXXX): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 1
                Medium: 1
                High: 1
        Total issues (by confidence):
                Undefined: 0
                Low: 1
                Medium: 2
                High: 0
Files skipped (0):
```

Figure 27 - Bandit scan details (Insecure Version)

Secure Version

Referring to Bandit's scan of the insecure version, the first finding pointed out that the debug mode was active (debug =True) and this corresponds to CWE-94. Debug mode activates the interactive Werkzeug debugger, thereby posing a risk of remote code execution should it happen in production environment. The second finding pointed out a secret key which is being hard coded (CWE-259) and allows any attacker who happens to know the value of this key to spoof session cookies. The last finding pointed out a possible SQL injection. In this secure build however, the debug mode is turned off, the secret key is read from the environment variable and made randomised and lastly, the login query made parameterized thus eliminating the problematic use of concatenation that caused the warning of SQL injection vulnerability. This results in the Bandit scan of the secure version to return with no issues (Figure 28).

```
Run started:2026-06-11 08:09:52.009398+00:00

Test results:
        No issues identified.

Code scanned:
        Total lines of code: 218
        Total lines skipped (#nosec): 0
        Total potential issues skipped due to specifically being disabled (e.g., #nosec BXXX): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 0
        Total issues (by confidence):
                Undefined: 0
                Low: 0
                Medium: 0
                High: 0
Files skipped (0):
```

Figure 28 - Bandit scan details (Secure Version)

The union of dynamic injection exploit and its associated Bandit warning corroborate the position in literature that dynamic and static approaches work complementarily but not mutually exclusively (McGraw, 2006). While there are other ways, such as fuzzing, sandboxing and runtime analysis for conducting thorough tests, the combination of dynamic testing by hand with Bandit static analysis proved adequate in verifying the efficacy of all countermeasures applied in this project.

4.3 Towards More Comprehensive Security Testing

The manual exploitation and Bandit analysis described above indicate that all deployed countermeasures work correctly yet, there are additional aspects to distinguish through an assurance process. Dynamic Application Security Testing (DAST) provided by means of a proxy scanning solution like OWASP ZAP would provide automatic crawling of

builds and identifying security problems such as missing security headers, reflected parameters and vulnerable cookies which take much effort to detect manually. Although Bandit is used for static analysis of the source code of the project, Software Composition Analysis solutions like pip-audit or Safety would analyse third party dependencies looking for vulnerable versions in order to cover supply chain security risks highlighted in recent guidelines by OWASP and not addressed by the manual code review.

There are also two other techniques that could be used to cover more cases. Fuzz testing would provide sending significant amounts of malicious or unexpected input to login, booking and hire-trainer endpoints in order to identify any handling problems that targeted attacks may fail to expose, while automatic validation of security headers (e.g. according to Mozilla Observatory recommendations) would be used to ensure that required headers like Content Security Policy and Strict Transport Security have been added properly. In the most comprehensive approach, each of the methods mentioned above, along with Bandit, would be performed automatically on each build in CI/CD pipeline so that any decline in the security status of the application would result in failed build. This is consistent with the literature stating that tooling delivers its greatest value when it is part of the software development process (McGraw, 2006).

5 Discussion

5.1 Challenges and Limitations

The creation of two equally functional builds has proved to be difficult because they should only be different when it comes to security mechanisms. Therefore, the same seed data had to be used in both builds so that any difference between the builds could be linked to the security mechanism alone. It should be noted that this work is limited since only six vulnerabilities have been used as an example from the OWASP Top 10 list, and not the whole audit process as well as the fact that SQLite was used as opposed to a real database system. There has been no need for transport security because both builds are run on HTTP.

5.2 Comparison with Industry Best Practices

These adopted countermeasures, parameterised queries, use of scrypt password hashing and server-side authorisation fall in line with recommended practices as per the OWASP coding guide and the SEI CERT standards mentioned in the literature review. Though these countermeasures are adequate to be used in production, they can only constitute a part of the overall best-practice strategy. The best-practice security strategies include layers of defence such as using HTTPS / TLS, CSRF protection, security headers and logging and monitoring systems (OWASP A09) in addition to

dependencies and supply-chain scanning which have received increased attention in the OWASP Top 10. Importantly, such countermeasures need to be integrated into the continuous Software Development Life Cycle, wherein static and dynamic analysis become a part of the automated build process.

5.3 Lessons Learned and Future Improvements

The project re-established the importance of incorporating security right at the beginning rather than trying to implement it afterwards, whether the particular feature was safe or not depended entirely on the chosen code. Moreover, it showed the advantage of using several types of tests since the vulnerability was found by both static and dynamic analysis. Possible areas for improvement might be adding HTTPS support, adding protection against CSRF attacks, logging and rate-limiting, integrating Bandit into the CI / CD process and testing with other dynamic testing tools like OWASP ZAP. Conducting threat modelling for each component could prove useful during the design stage.

5.4 Additional Security Controls (Defence in Depth Roadmap)

While the hardened build fixes the identified issues of application logic in the specified scope, a truly hardened system deployment will involve additional controls to address the identified residual risk through threat modelling. Transport layer security will be achieved by providing the application over HTTPS / TLS with enforcement using the HTTP Strict Transport Security, which prevents interception of the session cookies and credentials via the HTTP protocol. Cross Site Request Forgery protection will be provided with inclusion of per session tokens on all forms, which modify the application state, conveniently done using the Flask WTF's CSRFProtect functionality. A set of security response headers, such as Content Security Policy, X Content Type Options and X Frame Options, would provide additional defence in depth protection against cross site scripting and clickjacking, this functionality is easily added to the Flask application via the Flask Talisman module.

The threat model also uncovered issues regarding repudiation and availability. Logging of authentication events, failed authorisations and admin actions would provide an audit trail needed to attribute any action to a specific user and to uncover any attack attempts in progress, thus directly addressing the identified repudiation issue. Rate limiting of requests, such as provided by Flask Limiter extension, will limit the number of attempts to perform brute force and denial of service attacks against both login and booking endpoints. Multi factor authentication, using TOTP, will also limit the effect of password theft. In summary, the identified controls will move the implementation from correct code to defendable deployment, and their phased implementation is the obvious next step in the project.

6 Conclusion

In this research report, the design, implementation and thorough evaluation of a secure web-based application is presented using the example of MatchPoint Tennis Club Application to understand the importance of secure programming. In order to showcase how secure applications depend on the programming approach used and not the feature provided, two functionally identical applications have been created, a vulnerable and a secured version. Parameterised queries, password hashing, server-side authorisation, input sanitisation and safe configuration settings each proved to be the countermeasure against any vulnerability present in the code. This has been validated by both exploiting the weaknesses manually and performing Bandit static analysis on the code. This project adheres to OWASP Top 10 and the STRIDE model, indicating that it is the most appropriate to integrate the concept of security throughout the software development lifecycle.

7 References

Howard, M., & Lipner, S. (2006). *The security development lifecycle*. Microsoft Press.

Howard, M., & LeBlanc, D. (2003). *Writing secure code* (2nd ed.). Microsoft Press.

McGraw, G. (2006). *Software security: Building security in*. Addison-Wesley.

MITRE Corporation. (2025). *2025 CWE top 25 most dangerous software weaknesses*. https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html

National Institute of Standards and Technology. (2022). *Secure software development framework (SSDF) version 1.1* (NIST Special Publication 800-218).

Open Worldwide Application Security Project. (2021). *Application Security Verification Standard 4.0.3*.

Open Worldwide Application Security Project. (2025). *OWASP Top 10:2025*. https://owasp.org/Top10/2025/

Seacord, R. C. (2013). *Secure coding in C and C++* (2nd ed.). Addison-Wesley.

Shostack, A. (2014). *Threat modeling: Designing for security*. Wiley.

Software Engineering Institute. (n.d.). *SEI CERT coding standards*. Carnegie Mellon University. Retrieved June 14, 2026, from https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

Viega, J., & McGraw, G. (2001). *Building secure software: How to avoid security problems the right way*. Addison-Wesley.