# Penetration Test Report

Maise web application (*http://172.16.252.239)*

Security Testing

Safa Abiasis Bashir Yusuf

StudentID: YUS23603526

01.04.2025

# Table of Contents

## Overview

## Background

As part of my cybersecurity course at the University of Roehampton, I was given the task of performing a web application penetration test on the Maisie platform **(http://172.16.252.239).** This test was designed to feel like a real job for a client, with our lecturer acting as the client. The web application was built using PHP with a MySQL database and was hosted in a controlled lab environment in David Bell 117 (DB117).

The purpose of the test was to find and report any security problems in the application before it goes live. I followed a manual testing approach, meaning no automated tools were used, which made the process more realistic and closer to how ethical hackers work in real life.

The application had two types of user accounts: regular user accounts (which we could create and use) and one admin account (which we were not given access to). I tested the site using common techniques such as SQL injection, brute force, and cross-site scripting (XSS). I focused on the types of risks listed in the OWASP Top 10, which is a well-known guide in the cybersecurity field.

## Objective

The primary objective of this penetration test was to evaluate the overall security posture of the Maisie web application and identify any vulnerabilities or weaknesses that could be exploited by a malicious actor. This assessment aimed to simulate real-world cyber threats to determine how the application would respond under such conditions.

Testing involved manually inspecting the application for common security flaws, including but not limited to injection vulnerabilities (e.g., SQL injection), authentication bypass techniques, and client-side attacks such as cross-site scripting (XSS). These tests were conducted using ethical hacking methodologies and guided by the OWASP Top 10 framework, which outlines the most critical web application security risks.

All vulnerabilities discovered during the testing process were carefully documented. Each finding included a clear explanation of the issue, its potential impact—such as unauthorized data access, user account compromise, or privilege escalation—and practical recommendations for remediation. A risk rating was assigned to each vulnerability to indicate its severity and potential threat level.

The ultimate goal of this project was to support the developers and stakeholders in enhancing the application's security before deployment, thereby ensuring the protection of user data and maintaining the integrity of the system in a production environment.

## Scope

This penetration test's objective was to evaluate the Maisie web application's security, which was hosted on port 80 at http://172.16.252.71. To replicate realistic assault scenarios, testing was done by hand in the internal Cyber Lab (DB117).

The assessment focused on:

- Evaluating the application from an external, black-box perspective
- Testing functionality accessible through user-created accounts
- Identifying vulnerabilities based on the **OWASP Top 10**, including SQL injection, XSS, and authentication flaws
- Manually inspecting input fields, session handling, cookie behaviour, and access controls
- Highlighting real-world risks that could lead to data breaches, privilege abuse, or session hijacking

The aim was to uncover weaknesses that a potential attacker could exploit and to support developers in improving the application’s security posture before deployment.

## Constraints and Limitation

There were several constraints and limitations during the penetration testing of the Maisie web application (*http://172.16.252.239*):

1. **Automated tools** such as vulnerability scanners were prohibited. Because all evaluations had to be done  manually in order to imitate actual ethical hacking techniques, this limited the speed and comprehensiveness of some testing.

2. **Denial-of-Service (DoS) attacks**, including the use of tools such as Burp Suite Intruder for resource exhaustion, were strictly prohibited to avoid crashing or disrupting the lab environment.

3. **Testing for IDOR (Insecure Direct Object Reference) vulnerabilities** was not possible due to limited access to multiple user roles or object references. The application structure did not support testing scenarios involving varied user privileges.

4. **I was not given credentials** and was warned not to try to gain access or escalate privileges concerning the admin position, even though the existence of an admin account was known.

5. **There was no remote access available**, therefore testing was restricted to the lab setting in David Bell 117. This limitation increased time pressure, decreased flexibility, and prevented working from home.

6. Also, Due to **limited lab access and availability of machines**, testing had to be carefully planned and was often time-consuming, requiring coordination around classroom schedules and available equipment

# Methodology

## Methodology - Black box application penetration test

A penetration test of a Black Box web application was conducted. In this kind of test, the tester is completely ignorant of the web application's internal organization, source code, and server-side setup. The publicly available interface of the Maisie web application **(http://172.16.252.239)** was used for testing, which simulated how an outside attacker may try to find vulnerabilities from the outside.

There was no use of backend system information or administrator access. Testing was restricted to the application's exposed features and user-created accounts.

## Methodology for manual testing

Neither automated scanners nor vulnerability tools were used during the manual test. This was done in order to adhere to the engagement's ethical guidelines and lab scope.

Manual testing involved:

- **Form input manipulation**: which involves inserting SQL payloads into search and login fields.
- **URL parameter tampering:** XSS was tested using carefully crafted inputs.
- **Brute Forcing:** creating a unique script to carry out a brute-force Basic Auth attack.
- **Session Analysis**: Involves reusing and capturing cookies from two test accounts.
- **Header Inspection:** employing Burp Suite (intercept only) and browser developer tools, this technique looks for server faults and the absence of secure flags.
- **Browser Interaction:** modifying, injecting, and inspecting behaviour across sites with developer tools.

The testing process closely reflected ethical hacking workflows and simulated realistic attack scenarios based on limited user-level access.

## CVSS

In this penetration test, the Common Vulnerability Scoring System (CVSS) version 3.1 was utilized to assess and explain the seriousness of vulnerabilities found. It offers an objective and standard way to evaluate security vulnerabilities according to their potential effect and ease of exploitation. With a numerical value and qualitative assessment (e.g., Low, Medium, High, Critical) assigned to each issue, the CVSS score helps prioritize remediation.

Informed decision-making about the urgency of patching or mitigation is supported by this approach, which makes it possible to clearly communicate risk levels to both technical and non-technical stakeholders.

Example CVSS Vector – SQL Injection

- **Score:** 9.8 (Critical)
- **Vector:** CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H

Explanation:

- **AV:N** – Can be exploited over the network
- **AC:L** – Easy to exploit, no special conditions
- **PR:N** – No login or permissions needed
- **UI:N** – No user interaction required
- **S:U** – Impact stays within the same system
- **C:H / I:H / A:H** – High impact on data confidentiality, integrity, and availability

## Executively Summary

In this report, the findings of a manual penetration test conducted on the Maisie web application, located at ***http://172.16.252.239,*** are presented. The purpose of the analysis was to model actual cyberattacks and find weaknesses that malicious users might exploit.

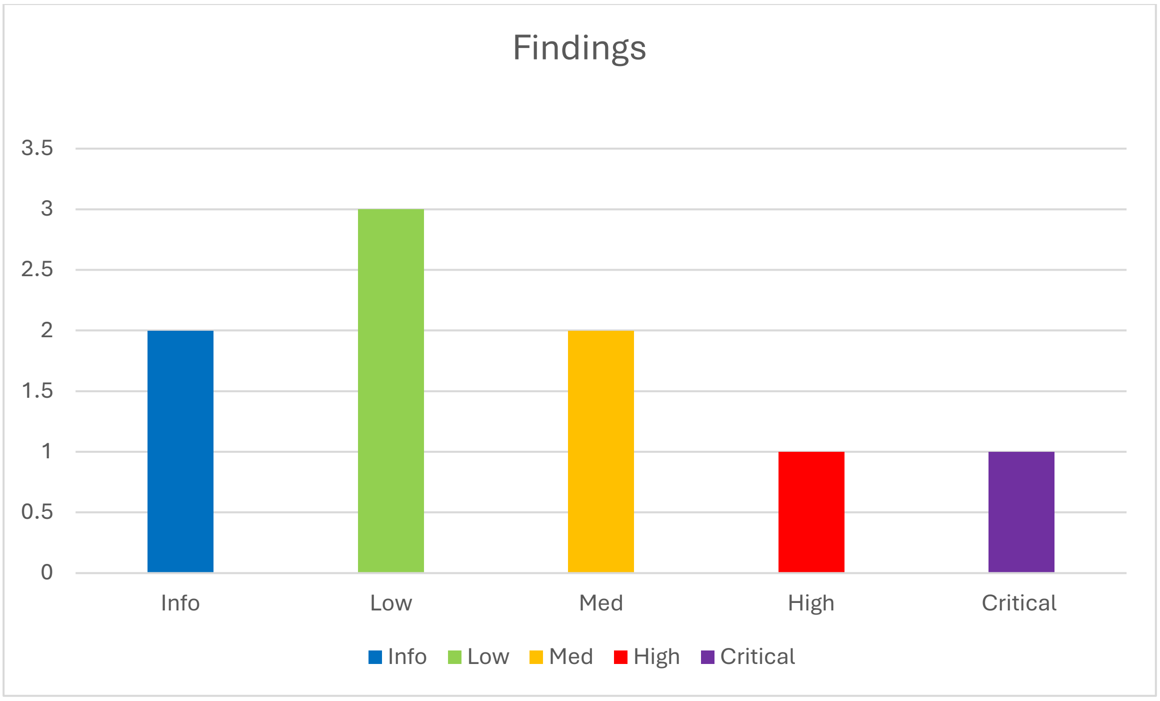
The testing identified several serious harmful security flaws that have significant effects on the application's security posture. Among them are:

- **SQL Injection**, which provided hashed credentials and access to the whole user database.
- **Cross-Site Scripting (XSS),** which makes it possible for malicious programs to enter and run in user browsers.
- **Brute-Force** Login because there are no lockout or rate-limiting safeguards.
- **HTTP (Unencrypted Communication),** which exposes data to interception.
- **Session hijacking and cookie poisoning,** which enable attackers to assume the identities of other users.

The website also lacks two-factor authentication (2FA), has weak password restrictions, has no terms and conditions, and has lengthy error messages that reveal technical information. These issues endanger the application as a whole as well as user data.

**Risk Breakdown by Severity**

The graph below summarizes the number of vulnerabilities discovered based on their severity:



**Recommendations**

Immediate actions should include:

- Resolving issues with input validation, including XSS and SQLi.
- Enforcing cookie settings and secure session processing.
- Getting a working TLS certificate and switching to HTTPS.
- Securing password policies and putting 2FA into practice.
- Restricting login attempts and keeping an eye out for brute-force attacks

These findings show that the system, in its current form, is not secure for public use. It is strongly recommended to address these issues before going live.

## Technical Summary

During my security assessment of the Maisie web application (172.16.252.239), I discovered and successfully exploited several critical vulnerabilities that, when chained together, resulted in full system compromise.

The first issue was a **SQL injection** vulnerability that allowed me to access the backend database without authentication. Using this, I extracted 155 user records, including credentials for the admin account. I then located a hidden admin page (/maisies_stash.php) exposed via robots.txt and used a **brute-force attack** to log in using the leaked credentials—no rate limiting or login protection was in place.

Once authenticated, I found that session cookies were not properly secured. I was able to perform **cookie poisoning**, reusing a valid session token to hijack another user’s session. Additionally, I identified a **reflected XSS vulnerability** in the **friends.php** page that allowed me to inject JavaScript using an SVG payload. This proved that the application would run untrusted scripts in users' browsers.

All communication with the site occurred over **unencrypted HTTP**, exposing all login credentials and session data to interception. These flaws were further supported by weak password policies, lack of two-factor authentication, and detailed error messages disclosing system information.

These vulnerabilities show that the application lacks the most basic security protections. I was able to move from no access to full control of the system using straightforward techniques that required no special tools. The combination of SQL injection, weak login security, session hijacking, and XSS leaves the application wide open to real-world attacks. Urgent action is needed to improve input validation, session management, password handling, and the use of HTTPS to protect users and system data.

# Technical Findings Details

## Critical Risk Findings

### Finding Title: SQL injection

| | | | |
|---|---|---|---|
| Host: | 172.16.252.239 | Port: | 80 |
| CVSSv3.1 | AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N | Severity: | CRITICAL |

### Background

**SQL Injection (SQLi)** is a critical security vulnerability that occurs when user-supplied input is improperly handled by a web application, allowing attackers to inject malicious SQL queries directly into the backend database. This can lead to unauthorized data access, modification, or even full compromise of the database server.

In this case, the Maisie web application failed to properly validate or sanitize input fields, allowing an unauthenticated attacker to inject custom SQL commands into the system. As a result, sensitive data—such as usernames and hashed passwords—was exposed directly from the database without the need for valid credentials.

This vulnerability represents a serious security risk, as it could allow a threat actor to compromise user accounts, escalate privileges, or map out the internal structure of the database and hosting environment.

### Technical Details

The vulnerability was exploited using the following SQL payload:

```
' OR 0=0 UNION SELECT NULL, username, password, NULL,
NULL FROM users --
```

For example, this payload above bypassed the application`s logic and successfully returned sensitive information to me from the users table which is very critical.

- A total of 155 user records were extracted
- The admin user “Maisie” was among the entries revealed.

Additional payloads were executed to also gather environment- specific information from the database server.

**Table 1: SQL injections Payloads Used**

| Payload | Description |
|---|---|
| ' OR 0=0 ORDER BY 5 -- | Identified number of columns |
| ' OR 0=0 UNION SELECT 1, 2, 3, 4, 5 -- | Verified column structure |
| ' OR 0=0 UNION SELECT NULL, username, password, NULL, NULL FROM users -- | Extracted usernames and hashes |
| ' OR 0=0 UNION SELECT 1, @@version, 3, 4, 5 -- | Retrieved SQL Server version |
| ' OR 0=0 UNION SELECT 1, database(), 3, 4, 5 -- | Retrieved current database name |
| ' OR 0=0 UNION SELECT 1, user(), 3, 4, 5 -- | Retrieved current database user |
| ' OR 0=0 UNION SELECT 1, @@hostname, 3, 4, 5 -- | Disclosed database server hostname |

## Table 2: Data Extracted

| Data Revealed | Details |
|---|---|
| Users Table | 115 user entries retrieved, including known usernames |
| Password Hash Samples | Example SHA-256 hash: aeff5cd0a241613e75e9de13961ccae7 |
| SQL Server Version | Extracted using @@version |
| Current User | Retrieved using user () |
| Host System | Hostname revealed: c82996681235 |

- Validate **Input:** Check and clean all user input to block harmful data.
- **Use Prepared Statements:** Write database queries that keep user input separate from commands.
- **Limit Access:** Give the database only the access it needs—nothing more.
- **Use a WAF:** Add a Web Application Firewall to catch and block attacks.
- **Hide Error Messages:** Don’t show technical errors to users; they can help attackers.
- **Test Regularly:** Run security checks often to find and fix new problems.

## Finding Title: XSS cross-site scripting

| | | | |
|---|---|---|---|
| **Host:** | **172.16.252.239** | **Port:** | **80** |
| **CVSSv3.1** | **AV:N/AC:L/PR:N/UI:R/S:C/C:H/I:H/A:N** | **Severity:** | **CRITICAL** |

### Background

Cross-Site Scripting (XSS) allows attackers to run malicious JavaScript in the browser of anyone visiting a vulnerable web page. This can lead to session hijacking, credential theft, redirection to phishing sites, and content manipulation.

In this case, the Maisie application fails to properly filter user input passed through the img parameter in the URL. This allows attackers to inject JavaScript directly into the page. A specially crafted **SVG-based payload** was used to execute malicious code, and in this example, it was used to load an external image from another website (demonstrating that third-party media or scripts can be forced onto users)-

## Technical Details

The vulnerability was found in the **friends.php** page. The following obfuscated SVG payload was injected into the img parameter:

This payload uses JavaScript to open a new window that displays an external image:

- It decodes to **window.open('https://burnley.gov.uk/.../bully-american-bully.jpg', '_blank', 'width=800,height=600')**

### Proof of Execution:

- The image of an American Bully dog is shown in a new browser window (as seen in the screenshot you uploaded).
- This proves that external content can be injected and loaded via malicious user input.

**Remediation**

To fix this vulnerability:

- Sanitize user input**:** Reject or properly encode any input containing <, >, ", or JavaScript event attributes.
- Use output encoding**:** Escape any dynamic content that will be rendered into HTML pages.
- Implement CSP (Content Security Policy): Limit the sources of executable scripts and media.
- Validate URL parameters on server side: Ensure only safe values are accepted for inputs like img.
- Disable inline script execution: Avoid allowing direct script execution in HTML attribute

## High Risk Findings

### Finding Title - Brute forcing on Maisie stash

| | | | |
|---|---|---|---|
| Host: | **172.16.252.239** | Port: | 80 |
| **CVSSv3.1** | **AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:N/A:N** | Severity: | **HIGH** |

Background

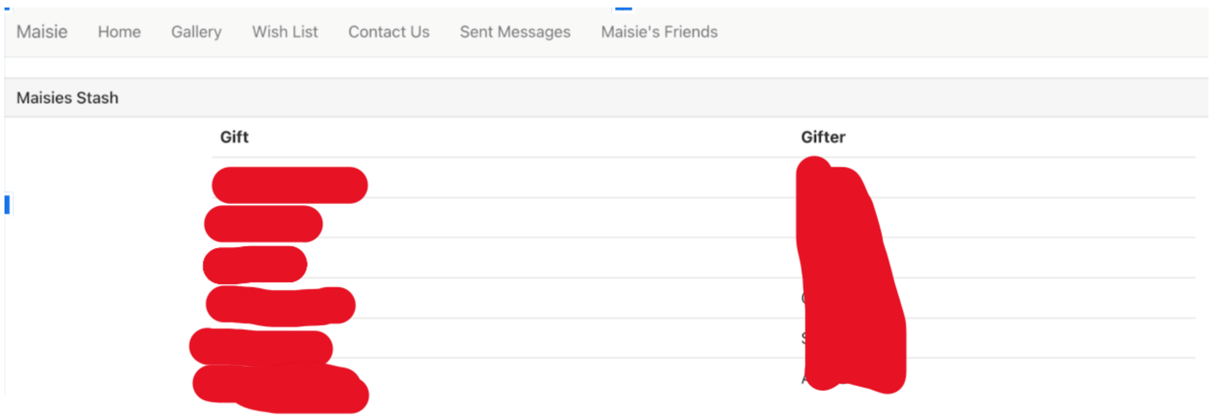
Brute-forcing is a technique where an attacker repeatedly attempts different username and password combinations in an effort to guess valid credentials. In this case, the target server permitted unlimited login attempts over a Basic Authentication form without any protective mechanisms in place. This allowed an attacker to successfully guess credentials and gain unauthorized access to a sensitive internal page.

#### Technical Details

- The hidden page **/maisies_stash.php** was discovered by examining the robots.txt file at http://172.16.252.239/robots.txt.
- A custom brute-force script was created in **Visual Studio Code** to automate login attempts.
- **Username:** Maisie (identified during earlier enumeration)
- **Passwords:** Common passwords sourced from passwords.txt
- **Authentication Method:** Basic Authentication (Base64 encoded)
- **Success Indicator:** HTTP/1.1 200 OK response upon valid login

Once valid credentials were found, the restricted page was accessed, confirming that both the exposed endpoint and lack of login protections left the application vulnerable to brute-force attacks.

Findings on MaisieStash



Maisies Stash

| Gift | Gifter |
|---|---|
| [redacted] | [redacted] |
| [redacted] | |
| [redacted] | |
| [redacted] | |
| [redacted] | |
| [redacted] | |

## Remediation Recommendations

- **Avoid Listing Sensitive Paths in robots.txt:** Do not expose hidden or admin-only pages through public files.
- **Implement Rate Limiting:** Limit the number of login attempts per IP to slow down brute-force attempts.
- **Enable Account Lockouts:** Temporarily disable accounts after a set number of failed logins.
- **Enforce Strong Password Policies:** Require complex, hard-to-guess passwords.
- **Enable Multi-Factor Authentication (MFA):** Add an extra layer of security to login processes.
- **Secure Internal Pages:** Restrict access to sensitive routes using authentication and authorization controls

## Finding Title  - HTTP Weak Protocol

| | | | |
|---|---|---|---|
| Host: | **172.16.252.239** | Port: | 80 |
| **CVSSv3.1** | **AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N** | Severity: | **HIGH** |

### Background

The Maisie web application uses HTTP instead of HTTPS, meaning the connection between the user and the site is **not encrypted**. This makes it easy for attackers to intercept or modify data during transmission, especially on shared networks. Browsers display a “connection not secure” warning, which increases the risk of users exposing sensitive data like usernames and passwords. The site is also vulnerable to spoofing and man-in-the-middle (MITM) attacks.
Technical Details

The application is served over **HTTP (http://172.16.252.239),** and there is no SSL/TLS certificate configured — HTTPS is not supported. As a result, the browser displays a warning:

***"Your connection to this site is not secure"***

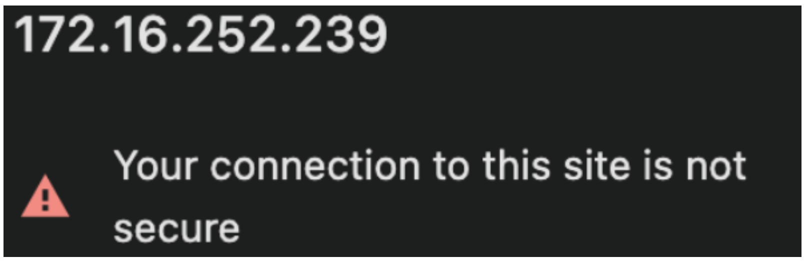
This means that all data sent between the user and the website is **unencrypted** and can be read in transit.

**The site is vulnerable to:**

- Man-in-the-middle (MITM) attacks – attackers can intercept or modify traffic
- Spoofing – users may be tricked into visiting fake versions of the site

Because of this, login credentials and any submitted form data can be easily stolen**.**

**Below is a screenshot showing the browser warning:**


172.16.252.239

Your connection to this site is not secure

## Remediation

- Install an SSL/TLS certificate to support HTTPS
- Redirect all HTTP traffic to HTTPS
- Use HSTS to enforce secure connections for future visits
- Never send sensitive data over HTTP
- Verify SSL setup regularly using tools like SSL Labs

## Finding Title - Cookie poisoning

| | | | |
|---|---|---|---|
| Host: | **172.16.252.239** | Port: | 80 |
| **CVSSv3.1** | AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:N | Severity: | **HIGH** |

**Background**

During testing, it was discovered that the Maisie web application is vulnerable to cookie poisoning and session hijacking. The application uses plain-text session cookies that are not securely tied to a specific user or device. This allows an attacker to manually reuse another user's session cookie and gain unauthorized access to their account. The vulnerability demonstrates weak session handling and a lack of proper cookie protection.

**Technical Details**

- Burp Suite was used to intercept and analyse session cookies.
- Session cookies were observed in plain text, without encryption or secure flags.
- Two test accounts were created for demonstration.
- The session cookie from **User 1** was copied and manually inserted into **User 2**'s session via browser developer tools.
- Upon refreshing the page, User 2 was granted access to User 1's session, confirming session hijacking was successful.
- No additional security mechanisms (like IP binding, user-agent checks, or token validation) were in place to prevent this behaviour.

**Remediation**

- Use **secure, random session tokens** that cannot be guessed or reused.
- Set the **HTTP Only** and **Secure** flags on cookies to prevent access through client-side scripts.
- Implement **session binding** to user-specific attributes such as IP address or device.
- **Regenerate session IDs** after login and logout to prevent session fixation.

- Set appropriate **expiration times** and **invalidate sessions** after inactivity or logout.

## Finding Title: Unlimited Login Attempts – Missing Login Rate Limiting

| | | | |
|---|---|---|---|
| Host: | **172.16.252.239** | Port: | 80 |
| **CVSSv3.1** | AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N | Severity: | **HIGH** |

### Background

Login rate limiting is a one of the key defence against brute-force attacks. It limits how many times a user or IP can attempt to log in within a short period. During testing, it was discovered that the Maisie web application allows unlimited login attempts without any delay, warning, or blocking mechanism**.**

This makes the system highly vulnerable to automated brute-force attacks, where an attacker can repeatedly try different passwords until access is gained — especially then combined with weak password policies.

### Technical Details

- The application does not enforce a limit on the number of login attempts.
- I was able to submit incorrect credentials repeatedly without being blocked, delayed, or warned.
- There was no cooldown**,** CAPTCHA, or lockout after multiple failed login attempts.
- This behaviour allows an attacker to use automated brute-force scripts to guess user credentials.
- Combined with weak password policies, this significantly increases the risk of account compromise.

### Remediation

- Implement login rate limiting**:** Restrict the number of login attempts per IP or account in a short time frame.
- Enable account lockout or delay**:** Temporarily lock or slow down login attempts after several failures.

- Use CAPTCHA**:** Introduce CAPTCHA after repeated failed login attempts to block automated bots.
- Add alerting**:** Notify users or admins when multiple failed login attempts are detected.
- Combine with strong password policies and MFA for layered defence.

## Medium Risk Findings

### Finding Title: Server Error Message Reveals Sensitive Information

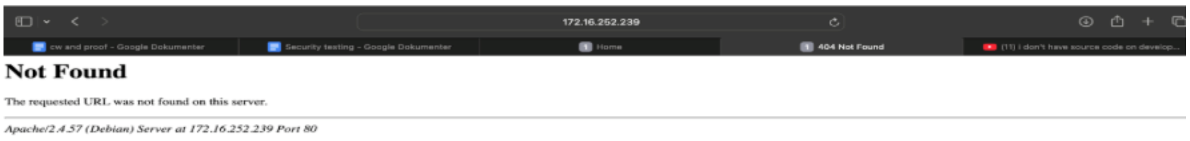
| | | | |
|---|---|---|---|
| Host: | 172.16.252.239 | Port: | 80 |
| CVSS v3.1 | AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N | Severity: | **MEDIUM** |

### Background

When a user accesses a non-existent page on the Maisie web application, the server returns a detailed 404 error message. This message reveals critical information such as the Apache version**,** operating system**,** and internal IP address**.**

Detailed error messages like this are dangerous because they give attackers insights into the server environment, which can help them plan more targeted and effective attacks.

### Technical Details


172.16.252.239
cw and proof - Google Dokumenter
Security testing - Google Dokumenter
Home
404 Not Found
(11) I don't have source code on develop...

Not Found

The requested URL was not found on this server.

*Apache/2.4.57 (Debian) Server at 172.16.252.239 Port 80*

As shown in the screenshot, accessing a random or invalid path such as:

**http://172.16.252.239/search1.phpfjhgv**

returns the following server message:

**Apache/2.4.57 (Debian) Server at 172.16.252.239 Port 80**

**This exposes:**

**Apache Version and OS**: Apache/2.4.57 (Debian) — Attackers can look up known vulnerabilities for this specific setup.

**Internal IP Address**: 172.16.252.239 — Helps attackers map internal infrastructure or develop local network exploits.

**Error Handling Weakness**: The application does not restrict error output or sanitize server headers.

Additionally, the application fails to properly handle incorrect URLs. Entering random strings after a valid route (e.g., /search1.phpfjhgv) results in the same verbose error instead of a generic 404 response. This indicates weak input validation and poor user experience.

## Remediation recommendations

- Disable Detailed Error Messages**:** Configure the server to return generic error pages (e.g., “404 Not Found”) without revealing server details.
- Sanitize Server Headers**:** Remove or mask version and OS info from server headers using Apache directives like Server Tokens and ServerSignature.
- Harden URL Handling**:** Implement stricter routing to catch malformed URLs and return safe, user-friendly 404 pages.
- Patch and Monitor**:** Ensure Apache and Debian systems are kept up to date with security patches and monitor for unusual traffic to invalid routes.

## Finding Title - Weak Password Policy

| | | | |
|---|---|---|---|
| Host: | 172.16.252.239 | Port: | 80 |
| CVSS v3.1 | AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N | Severity: | **MEDIUM** |

### Background:

The Maisie web application allows users to register with extremely weak passwords, such as short numeric values (1000, 1111) or even single characters (a). There are no password complexity requirements, and the system does not provide feedback or warnings when insecure passwords are used.

This lack of controls makes it significantly easier for attackers to guess or brute-force credentials and gain unauthorized access to user accounts. It also encourages poor password habits among users, further weakening the application's security.

### Technical Details:

Testing the application showed that the registration process accepts insecure passwords without any validation or enforcement.

**Example of Some Weak passwords accepted:**

- 1000
- 1111
- a
- abc

**It is no enforcement in the web application of:**

- Minimum length
- Uppercase and lowercase letter usage
- Numbers and special characters
- Checks against common or breached passwords

- Password strength indicators or warnings

As a result, attackers can use simple wordlists or scripts to easily compromise accounts with little effort or resistance

## Remediation

- **Enforce a strong password policy**: Require at least 8 characters with a mix of upper/lowercase letters, numbers, and symbols.
- **Block weak passwords**: Prevent the use of commonly used or compromised passwords.
- **Use a password strength meter**: Provide users with feedback during password creation.
- **Enable multi-factor authentication (MFA)**: Add an extra layer of protection.
- **Validate on the server-side**: Ensure rules cannot be bypassed via browser tools or manual requests.

## Finding Title - lack of using 2fa

| | | | |
|---|---|---|---|
| Host: | 172.16.252.239 | Port: | 80 |
| CVSS v3.1 | AV:N/AC:L/PR:N/UI:N/S:U/C:L/I:L/A:N | Severity: | **MEDIUM** |

### Background

Two-Factor Authentication (2FA) is a widely recommended security control that adds an extra layer of protection on top of traditional username and password logins. Without 2FA, if an attacker obtains or guesses a user's credentials, they can gain full access to the account with no additional barrier.

During testing, it was confirmed that the Maisie application **does not implement any form of 2FA**, leaving all accounts protected solely by passwords — which, in this case, can be weak and are not rate-limited

### Technical Details

The application does not offer any option for enabling two-factor authentication (2FA) during the login process or account setup. All user accounts are protected by a single layer of security: the username and password. No additional verification method, such as a one-time password (OTP), SMS code, or authentication app, is available.

This leaves user accounts vulnerable to unauthorized access, especially when combined with other identified issues, such as weak password enforcement and unlimited login attempts. Without a secondary verification step, an attacker who obtains or guesses valid credentials can gain full access without any further barrier.

### Remediation

To reduce the risk of account compromise, the application should implement two-factor authentication. This can include methods such as time-based OTPs via authenticator apps, SMS-based codes, or email verification codes. Two-factor authentication should be mandatory for administrative or privileged accounts and optionally available for regular users.

The system should also provide clear guidance for users to set up 2FA and notify them of its availability. Implementing and encouraging 2FA significantly increases account security by requiring an attacker to have both the password and access to the second authentication factor.

## Finding Title - No Ability to Change Username or Password

| | | | |
|---|---|---|---|
| Host: | 172.16.252.239 | Port: | 80 |
| CVSS v3.1 | AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:L/A:N | Severity: | **MEDIUM** |

### Background

Allowing users to update their username or password is a basic feature in secure web applications. It enables users to correct mistakes, change their identity information, or update credentials after a suspected compromise. During testing, it was discovered that the Maisie web application does not offer any way for users to change their username or password after account creation.

### Technical Findings

- After registering an account, there are no options in the interface to change the username or password.
- No “Edit Profile,” “Change Password,” or similar functionality was available anywhere in the application.

This limits user control over their account and increases the risk of long-term exposure in case credentials are leaked.

#### Remediation

- Add a feature that allows users to change their password at any time through a secure form.
- Provide an option to change usernames (if appropriate for the application).
- Ensure password change requests require current password verification and follow strong input validation.

## Info Risk Findings

### Finding Title - Terms and Conditions

| | | | |
|---|---|---|---|
| Host: | 172.16.252.239 | Port: | 80 |
| CVSSv3.1 | Not scored using CVS | Severity: | **Info** |

#### Background:

During testing, it was identified that the web application does not include a Terms and Conditions page. This is important because users are not informed about how the site should be used, what rules apply, or how their data may be handled. The absence of this legal information can lead to confusion, reduce trust, and may raise compliance issues depending on the type of data being collected or stored.

#### Technical Findings

- No Terms and Conditions, Privacy Policy, or User Agreement was found on the site.
- I looked for this information in common areas but found nothing.
- The application allows users to register and interact without being shown any legal or data-related terms.
- This lack of transparency presents legal and trust concerns, especially for applications handling personal data.

#### Remediation

- Create a terms and conditions page that explains user responsibilities, acceptable use, and legal disclaimers.
- Add a privacy policy that outlines how user data is collected, stored, and used.
- Display links to these pages on key parts of the site, such as the registration and login pages.
- Include a checkbox during account creation to confirm user agreement to the terms.

## References

- OWASP Foundation, n.d. *SQL Injection Prevention Cheat Sheet*. [online] Available at: https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html [Accessed 20 Mar. 2025].

- OWASP Foundation, n.d. *Cross Site Scripting (XSS) Prevention Cheat Sheet*. [online] Available at: https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html [Accessed 20 Mar. 2025].

- OWASP Foundation, 2021. *OWASP Top 10: The Ten Most Critical Web Application Security Risks*. [online] Available at: https://owasp.org/Top10/ [Accessed 20 Mar. 2025].

- FIRST.org, 2019. *Common Vulnerability Scoring System v3.1: Specification Document*. [online] Available at: https://www.first.org/cvss/specification-document [Accessed 20 Mar. 2025].

- PortSwigger, n.d. *SQL Injection*. [online] Available at: https://portswigger.net/web-security/sql-injection [Accessed 20 Mar. 2025].
- PortSwigger, n.d. *Cross-site Scripting (XSS)*. [online] Available at: https://portswigger.net/web-security/cross-site-scripting [Accessed 20 Mar. 2025].

- SSL Labs, n.d. *SSL Test: Best Practices for HTTPS Configuration*. [online] Available at: https://www.ssllabs.com/ssltest/ [Accessed 20 Mar. 2025].

- University of Roehampton, 2025. *Cybersecurity Labs – Portfolio Assignment Brief*. [online] Available at: https://moodle.roehampton.ac.uk/mod/page/view.php?id=2092267

- [Accessed 20 Mar. 2025].OpenAI, 2025. *ChatGPT 4 – AI Language Model Assistance Tool*. [online] Available at: https://chat.openai.com [Accessed 20 Mar. 2025].