A Controlled Vulnerability Assessment and Penetration Testing Lab Using OWASP Juice Shop

Risk Assessment of Common Web Vulnerabilities in a Safe Learning Framework

Lian Mhar T. Asperin

School of Information Technology, Mapua University, lmtasperin@mymail.mapua.edu.ph

Sean Justine E. Barbo

School of Information Technology, Mapua University, sjebarbo@mymail.mapua.edu.ph

Eric B. Blancaflor

School of Information Technology, Mapua University, ebblancaflor@mymail.mapua.edu.ph

Francis Zachary Domingo

School of Information Technology, Mapua University, fzdomingo@mymail.mapua.edu.ph

Mayrielle Joy M. Latigo

School of Information Technology, Mapua University, mjmlatigo@mymail.mapua.edu.ph

Web application security remains one of the most critical challenges in the digital era, as modern websites continue to suffer from common vulnerabilities that attackers can exploit. This study proposes a hands-on Vulnerability Assessment and Penetration Testing (VAPT) laboratory using OWASP Juice Shop, a deliberately insecure web application built with Node.js and Angular. The purpose of this research is to demonstrate how SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication vulnerabilities can be identified, exploited, and mitigated in a controlled and ethical testing environment. The experiment follows a systematic process that includes setup, reconnaissance, exploitation, observation, mitigation, and verification. Using a tool such as Burp Suite, students are able to simulate realistic attack scenarios and learn secure coding practices. All testing is performed within a Docker container to ensure safety and isolation. The study aims to enhance students' awareness of web vulnerabilities, strengthen their analytical skills, and promote responsible ethical hacking practices in the field of cybersecurity.

CCS CONCEPTS • Security and privacy • Vulnerability management • Software testing and debugging • Web application security

Additional Keywords and Phrases: Penetration Testing, Vulnerability Assessment, Web Security, SQL Injection, Cross-Site Scripting, Ethical Hacking

1 INTRODUCTION

In the current digital landscape, web applications play a central role in facilitating communication, commerce, and education. Organizations increasingly rely on online systems for user management, financial transactions, and data storage. However, as these systems become more complex, they also become prime targets for cyberattacks that exploit weaknesses in code and configuration. According to global cybersecurity reports, web-based attacks remain one of the top methods used by cybercriminals to breach sensitive systems and steal user data [1]. Common vulnerabilities such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication have persisted across decades of web development, often resulting from poor coding practices or insufficient validation mechanisms. These flaws can compromise databases, expose personal information, and even grant attackers unauthorized control over entire applications. The continuous

rise of such incidents highlights a critical need to strengthen both the theoretical understanding and practical application of secure development principles among learners and developers.
Educational institutions are not immune to these security threats. In fact, many universities and schools have already become victims of phishing attacks, website defacement, and data breaches. Limited cybersecurity budgets and outdated systems make these environments vulnerable to exploitation, especially in developing countries where digital infrastructure is still evolving [2]. With the increasing integration of technology in classrooms and administrative functions, the educational sector faces the challenge of maintaining both accessibility and security. To address this, cybersecurity education must go beyond theoretical instruction and incorporate hands-on, experiential learning that mirrors real-world scenarios. Traditional classroom discussions of cyber threats are often insufficient for preparing students to identify and counteract complex attacks. Thus, building controlled, practical environments where students can observe, exploit, and mitigate vulnerabilities becomes essential for developing genuine security expertise [3].

This study proposes the development of a Vulnerability Assessment and Penetration Testing (VAPT) laboratory using OWASP Juice Shop, a deliberately insecure web application built with Node.js and Angular. The primary objective of the project is to provide students with a controlled and ethical environment for learning how to identify, exploit, and mitigate critical web vulnerabilities. The VAPT lab will focus on three of the most common attack types—SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication using a structured methodology that follows the cycle: Setup → Reconnaissance → Exploitation → Observation → Mitigation → Verification → Reporting. Through this iterative process, students are expected to gain an in-depth understanding of how vulnerabilities manifest and how secure coding practices such as parameterized queries, proper input validation, and cookie protection can effectively prevent them. A tool like Burp Suite will be integrated into the lab to help learners simulate real-world penetration testing scenarios. The use of Docker ensures that all experiments are executed safely within an isolated environment, preventing any unintentional harm to external or live systems.

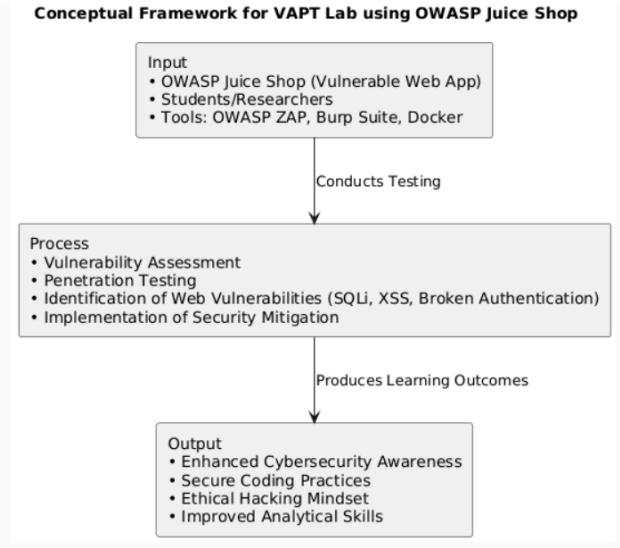


Fig. 1. Conceptual Framework

The conceptual framework for this study is based on the cybersecurity learning cycle, which combines elements of both offensive and defensive security. The model emphasizes learning through the continuous process of discovery, exploitation, remediation, and verification. By engaging in this loop, students develop critical thinking and technical problem-solving skills, enabling them to approach cybersecurity challenges methodically. In addition, the study may incorporate surveys and observations to assess students' baseline knowledge and attitudes toward web security before and after participating in the lab activities. This assessment will help identify common misconceptions and areas for improvement, particularly regarding risk perception and secure coding behavior. Through these findings, the research aims to demonstrate that structured, practice-oriented learning significantly enhances cybersecurity competence compared to traditional theoretical instruction. Moreover, it reinforces ethical hacking principles, reminding learners that penetration testing should always be performed responsibly and within authorized environments [4].

Overall, this study aims to contribute to both cybersecurity education and research by presenting a safe, replicable, and pedagogically effective framework for teaching web application security. The proposed OWASP Juice Shop–based lab not only strengthens technical proficiency but also instills professional ethics in handling vulnerabilities. By merging theory with practice, the project supports the development of future-ready cybersecurity professionals capable of mitigating threats that persist in today's rapidly evolving digital world.

1.2 OBJECTIVE OF THE STUDY

The main purpose of this research is to conduct and implement a controlled Vulnerability Assessment and Penetration Testing (VAPT) laboratory using OWASP Juice Shop to offer students a hands-on, safe, and ethical facility for web application security education. This study has the following objectives:

1. Identify common web vulnerabilities such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication / Session Management from the OWASP Juice Shop application.
2. Showcase these vulnerabilities in a controlled environment using a tool such as Burp Suite.
3. Assess the vulnerabilities found during the performance test and document the consequences assigned to each.
4. Suggest and implement corresponding remediation tactics such as Parameterized Queries, Input Validation, and Secure Session Management practices.
5. Demonstrate that the application of remediation techniques is effective through re-testing and validation processes.
6. Heighten awareness of responsible ethical hacking and reinforce best practices for secure coding and responsible vulnerability communications to students and learners of cybersecurity.

The study intends to further enhance practical cybersecurity education by connecting the theoretical process to a real-world usable format in a safe and repeatable laboratory experience.

1.3 SCOPE AND DELIMITATIONS

This study involves building and running a Controlled Vulnerability Assessment and Penetration Testing (VAPT) lab using OWASP Juice Shop. The Focus is on the configuration and evaluation of a few of the most common web application vulnerabilities: SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication. The research incorporates a practical use of Burp Suite as a means of creating a simulated attack and defense environment within the isolated docker environment. This study is indeed educational and hopefully develops students' problem-solving skills, ethical reasoning, and technical competencies within the domain of cybersecurity.

This study is also limited to the vulnerabilities within the OWASP Juice Shop web applications and does not address advanced or emerging threats., including but not limited to, zero-day exploits, denial-of-service (DoS) attacks, and social engineering. In addition, to ensure that all testing remains ethical and without risk, the project does not and will not include live production systems and real-world web servers. The evaluation of learning outcomes is limited to the participants within the School of Information Technology and does not encompass a wider population. Also, this study does not seek to address the long-term retention of knowledge in cybersecurity but rather focuses on the immediate impact of experiential, lab-based learning on students' understanding of web application security.

2 REVIEW OF RELATED LITERATURE AND STUDIES

2.2 OWASP JUICE SHOP

The OWASP Juice Shop is a deliberately exploitable web application produced by the Open Web Application Security Project (OWASP) used for education and research on cybersecurity. The Juice Shop incorporates real-world web vulnerabilities such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication, based on the OWASP Top 10. The author Kimminich (2023) discusses how the application provides a gamified learning space for students and researchers to practice ethical hacking in a safe environment and gives students the experience of some hands-on implementation without threatening real world exploitation, all while bridging the gap between the theory of security and the application. The source code is open source and includes many options for ease of use such as Docker configuration if used in a classroom or laboratory setting, allowing for a controlled environment for testing with far less risk[5].

As mentioned by Aydos et al. (2021), practical security testing environments such as OWASP Juice Shop greatly improve learners' awareness of web vulnerabilities compared to learning in a purely theoretical manner. Providing students the opportunity to safely interact and exploit typical security vulnerabilities enables students to better understand secure programming, validation techniques, and mitigating methods. Also, Asif et al. (2024) point out that ethical hacking tools can help students develop cybersecurity awareness and responsibility, and give students a controlled environment in which to understand exploitation and security risks, rather than a real-world scenario.[1] [4]

This research uses the OWASP Juice Shop as the controlled Vulnerability Assessment and Penetration Testing (VAPT) environment in this laboratory. The current literature review suggests that tools such as the Juice Shop, in settings such the one outlined in this study, help to deepen understanding of areas such as secure coding practices, vulnerability remediation, and responsible disclosure. The study's rationale was to adapt the Juice Shop for students to conduct their own testing, which studied the potential for experiential learning environments to deepen cybersecurity education and awareness and help bridge the gap between theory and practice.

2.3 E-Commerce Security Issues

E-commerce platforms are now required elements of contemporary business, managing millions of customer credit transactions each day along with the associated volumes of consumer data. However, this same system, in addition to poor coding standards and unreliable authentication, are at risk for cyber breaches as are all systems where weak or modified data protection policies for the handling of consumer data can become compromised. Common exploits such as SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication will often grant access to sensitive consumer data and expose to e-commerce systems subsequently losing profit.

Liu et al. (2022) emphasize that e-commerce platforms continuously face a “never-ending challenge” in combating cyber threats, including denial-of-service attacks, phishing, malware infections, and database exploits, which target both technical and human weaknesses. The authors highlight that the lack of continuous security updates and user awareness contributes significantly to the persistence of these vulnerabilities [6]. Similarly, Sambhus and Liu (2024) explain that SQL Injection and Cross-Site Scripting remain among the most prevalent threats to web-based businesses. These attacks exploit weak input validation and poor coding practices, often resulting in data leaks and compromised accounts. The authors recommend adopting automated vulnerability remediation systems and secure development frameworks to prevent such issues [7].

In conclusion, research indicates that e-commerce security vulnerabilities are the result of factors including a series of uncorrected technical misconfigurations as well as developers' limited knowledge about secure coding. This emphasizes the idea behind secure controlled environments such as Vulnerability Assessment and Penetration Testing (VAPT) labs being used to train students and practitioners to identify and properly mitigate e-commerce threats.

2.4 PROPOSED SOLUTION

The cyberattack targeting the e-commerce platform occurred in a multi-faceted and coordinated fashion that incorporated reconnaissance, credential harvesting, and exploitation via injection-based techniques. To initiate the cyber incident, attackers began by gathering user identifiers, such as customer emails available in product reviews or through exposed APIs, and aggregated those identifiers with credentials gained through previously exposed datasets to carry out massive credential-stuffing attacks [6]. These attacks exploited security vulnerabilities (the absence of multi-factor authentication (MFA) or rate-limiting) that enabled attackers to access user accounts without authorization. Once the initial access was achieved, attackers carried out additional attacks utilizing more sophisticated injection and evasion techniques such as Structured Query Language Injection (SQLi) and Cross-Site Scripting (XSS). Guan et al. [8] highlighted evidence that reinforcement learning–based frameworks, such as SSQLi, are capable of automatically mutating SQLi payloads to evade detections by advanced detection systems. These findings illustrate the limitations of rule-based and static defenses. Likewise, Alsaffar et al. [9] described that due to weak sanitization and improper encoding, XSS payloads (similar to those utilized in SQLi) are able to bypass security filters which allowed attackers to steal session tokens and financial information

The attack succeeded due to multiple underlying vulnerabilities in the platform’s authentication and application logic. Weak password policies, missing MFA, and inadequate input validation created openings for both credential-based and injection attacks. The platform’s reliance on unparameterized SQL queries increased its susceptibility to SQLi, while insufficient Content Security Policy (CSP) enforcement left it vulnerable to XSS [9]. In addition, broken authentication and session management, including a lack of session ID regeneration and poor cookie handling, further exposed users to session hijacking and the risk of unauthorized persistence [10][11]. Additionally, as shown by Gogulakrishnan et al. [2], an absence of timely patch management or continuous security testing potentially allowed attackers to compromise existing system components. In aggregate, these weaknesses highlighted a more comprehensive failure to demonstrate proactive cybersecurity governance and threat modeling, which inherently exposed the organization to contemporary adversarial tactics.

After collecting and testing valid credentials, the attackers automated login attempts to gain access, injected mutated SQLi payloads to extract sensitive data, and leveraged XSS to embed malicious JavaScript that captured customer information [8]. Persistent sessions and unmonitored activity enabled continued

exploitation and fraud. To mitigate such threats, experts recommend implementing MFA, enforcing parameterized queries, applying context-aware output encoding, and adopting strict CSP enforcement [11]. Incorporating adversarial-aware testing and continuous security education for developers is also essential to strengthening the platform's overall defense posture. By aligning these practices with OWASP guidelines [5], e-commerce systems can significantly enhance their resilience against evolving and adaptive cyberattacks.

3 METHODOLOGY

3.2 Overview

This study conducted a controlled vulnerability assessment on OWASP Juice Shop, a deliberately insecure web application designed for educational cybersecurity exercises. The goal was to identify, exploit, and analyze three common web vulnerabilities: SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication / Session Management. All activities were performed in a local and isolated environment using Docker to ensure no real systems were harmed.

3.3 Environment Setup

The target system, OWASP Juice Shop, was deployed locally using the following command:
docker run --rm -p 3000:3000 bkimminich/juice-shop

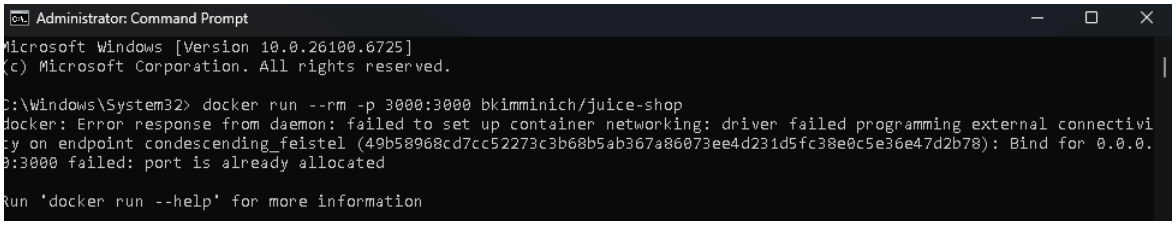


Fig. 2. Command Line Setup for Launching OWASP Juice Shop Using Docker

The application was accessed at http://localhost:3000/. The assessment used the following core tools:

- Docker — Containerized the OWASP Juice Shop instance to ensure an isolated, repeatable environment.
- Burp Suite (Community Edition) — Intercepted, inspected, and modified HTTP requests and responses; used Repeater for iterative payload testing and to demonstrate proof-of-concept manipulations.

All actions were documented through screenshots, captured requests, and observation notes.

3.4 SQL Injection (SQLi) Assessment

Overview

This sub-section documents a controlled SQL Injection (SQLi) authentication bypass exercise reproduced from a short demonstrative workflow. The exploit demonstrates how minimal input manipulation (appending a single quote ' and a SQL comment --) together with observable server error messages can reveal vulnerable string-concatenated SQL queries and enable unauthorized access. All activities were executed in a contained, Dockerized OWASP Juice Shop instance to ensure safety and reproducibility.

Objective

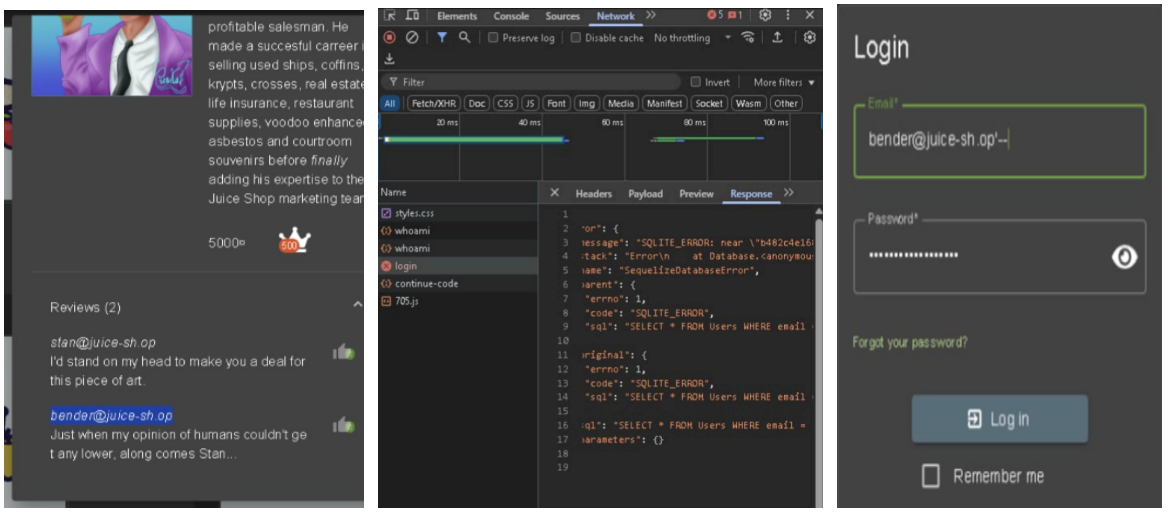
1. Reproduce the SQLi-based authentication bypass shown in the demonstration (email → append ' → observe SQL error → append -- → login bypass).
2. Record network artifacts and server responses that indicate unparameterized SQL usage.
3. Verify that the vulnerability can be mitigated with standard defensive measures and confirm mitigation by re-testing.

Test Environment & Tools

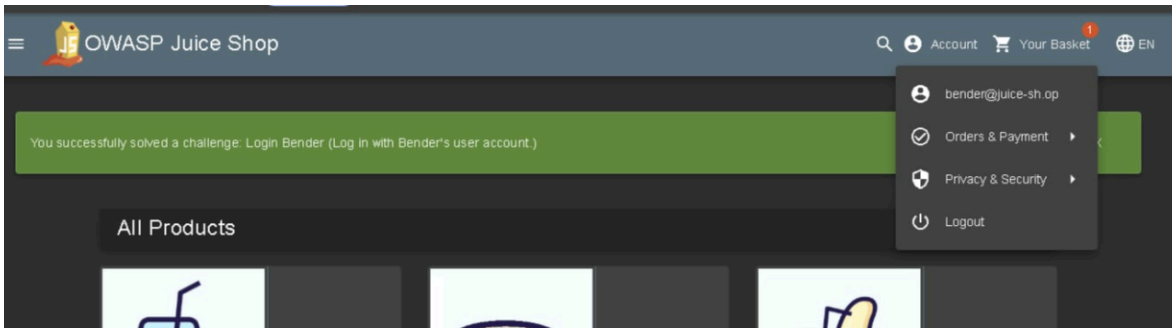
- Target application: OWASP Juice Shop deployed locally with Docker:
- docker run --rm -p 3000:3000 bkimminich/juice-shop (access at http://localhost:3000).
- Browser: Chrome or Edge (Developer Tools → Network tab used for live inspection).
- Testing proxies/tools: Burp Suite Community Edition (Proxy & Repeater) for request interception and manipulation; Data sources: A valid user email harvested from the application UI (as per the demonstration).
- Safety: All tests confined to the local Docker container; no external systems were contacted.

Step-by-step Procedure

1. Obtain a valid username/email from the target instance (e.g., visible in a public page or test dataset). Record the email used for testing.
2. Attempt normal login: paste the obtained email into the login email field and enter an arbitrary password. Click Login and observe the normal failure response (if any). Capture the HTTP request in DevTools.
3. Append a single quote ('): modify the email field to victim@example.com' (i.e., add a trailing single quote), keep the same password, and submit the form.
    - Right Click → Inspect (Webpage Inspect) to capture the login request.
    - Press Login, select the login request in the Network list, then view the Response panel and inspect the response body for SQL error messages (e.g., syntax error, SQL exception stack trace). Document the exact error text and the intercepted request.
4. Analyze the error: the presence of a DB-related syntax error indicates that user input is concatenated into SQL statements and that detailed error output is returned to the client — a strong indicator of SQLi potential.
5. Inject SQL comment (--) to terminate the query: modify the email input to victim@example.com' -- (single quote to break the string, followed by -- to comment out the rest of the query) and submit the login form again.
6. Observe authentication result: if the backend used string concatenation for the WHERE clause, the injected payload will comment out the password check. Successful login (landing in the user dashboard) demonstrates authentication bypass. Capture the successful response, session cookie values, and any new tokens issued.
7. Document all artifacts: screenshots demonstrating both the error and the successful bypass.



(a) (b) (c)



(d)

Fig. 3. (a) Harvested target email from the application UI, (b) SQL syntax error observed after ' injection, (c) Confirmatory error reproduced with repeated injection, (d) Successful authentication bypass and session issuance after payload.

Key Findings

- Error leakage: Appending ' produced an SQL syntax error in the HTTP response, confirming that user input is incorporated directly into SQL statements and that verbose server errors leak backend details.
- Authentication bypass: Adding -- after closing quote (victim@example.com' --) successfully neutralized the password-check clause in the SQL query and allowed access to the victim's account without the correct password, demonstrating a classic SQLi authentication bypass.
- Session issuance: Following the bypass, the application issued valid session cookies/tokens for the targeted account, indicating the application created authenticated sessions based on the returned user row.
- Attack simplicity: The exploit requires only basic manipulations of visible form fields and use of browser developer tools (no advanced tooling required), making it especially suitable as a classroom demonstration of why parameterization is critical.

Mitigation
Applied and recommended defensive measures implemented and verified in the lab:

1. Parameterized Queries / Prepared Statements
    - Replace any dynamic SQL string concatenation with parameterized database queries or ORM-safe parameter binding (for example, using parameter placeholders and passing user input as parameters). This prevents user-supplied characters from being interpreted as SQL syntax.
2. Suppress Detailed Error Output
    - Configure the application to return generic, non-technical error messages to clients while logging detailed errors server-side. Preventing error leakage reduces reconnaissance effectiveness for attackers.
3. Input Validation & Normalization
    - Enforce strict server-side validation for email formats and input length; reject inputs with unexpected characters where not required. Normalization reduces opportunities for malformed input to reach SQL parsers.
4. Least Privilege Database Access
    - Use a database account with minimal privileges for application queries to limit the impact if an injection occurs.
5. Session Hardening
    - Ensure session tokens are securely generated and bound to authenticated context; rotate and invalidate sessions appropriately upon suspicious events.
6. Monitoring & WAF Rules
    - Implement logging and alerting for SQL error patterns and suspicious input characters (e.g., ', --, UNION). Optionally deploy WAF rules to detect and block common injection payload patterns.

Post-mitigation Verification (summary)
After applying parameterized queries and suppressing error output, repeat the exact injection steps (' and --). The server should no longer return SQL syntax errors to the client, and the victim@example.com' -- payload should fail to bypass authentication. Confirm by intercepting the request/response and showing that login is rejected and no unauthorized session is created.

3.5 Cross-Site Scripting (XSS) Assessment

Overview
This sub-section reproduces a reflected/stored XSS demonstration shown in the referenced video. The exploit demonstrates how unsanitized user-supplied input injected into a page (in this case the "Track My Order" page) can be used to execute arbitrary client-side code by inserting an HTML element with a javascript: URI. The demonstration uses a simple payload that creates an alert dialog to prove execution. All testing was performed in a local, Dockerized OWASP Juice Shop instance for safety and pedagogy.

Objective

- XSS demonstration (locate item of purchase → add to cart → order → Track My Order → inject <iframe src="javascript:alert(\xss`)">` into the id/URL parameter → refresh → observe alert).
- Capture the HTTP request/response and DOM rendering that show the injected content being executed.
- Identify why the input was executed (lack of sanitization/encoding) and document the attack vector.
- Implement and verify mitigations that prevent the injected payload from executing.

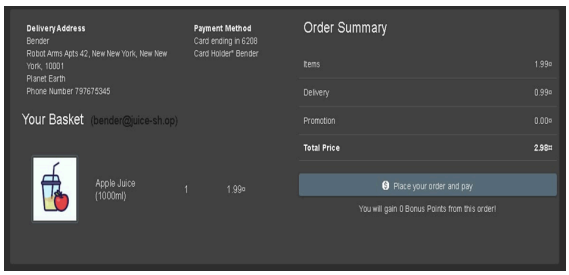
Test Environment & Tools

- Target application: OWASP Juice Shop deployed locally via Docker:
- docker run --rm -p 3000:3000 bkimminich/juice-shop.
- Browser: Chrome or Edge (Developer Tools used to view DOM and Network activity).
- Testing utilities: Burp Suite (optional) for intercepting requests; a text editor for editing test inputs.
- Payload used:
    - <iframe src="javascript:alert(`xss`)">
- Safety: All tests executed in an isolated containerized environment; no external or production systems were targeted.

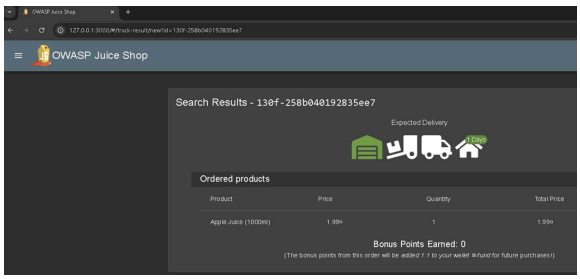
Step-by-step Procedure

1. Locate item of purchase (initial action): Find the item you want to purchase and then add it to cart.
2. Place an order: Complete a sample order in the Juice Shop instance so that the Track My Order page becomes available for the test account.
3. Open Track My Order page: Navigate to the page (or endpoint) that displays order tracking details and exposes an id or a text input/URL parameter reflecting user-controlled content into the page.
4. Inject payload into the id/input field: Paste the payload exactly as shown into the id parameter or the input location:
    - <iframe src="javascript:alert(`xss`)">

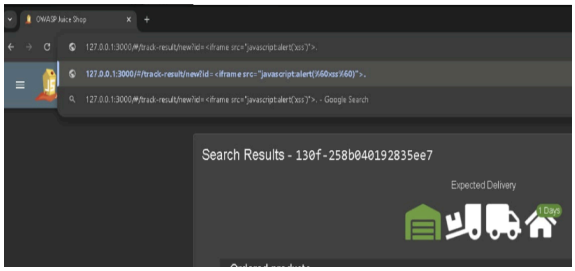
    (If the page uses a query string, insert the payload into that parameter; if it uses a form field, paste into the form input.)
5. Refresh / Submit to render the page: Refresh the tracking page (or submit the form) so the application reflects the injected content back into the HTML document.
6. Document evidence: Capture screenshots of the alert dialog, the HTML element in the DOM (showing the injected <iframe>).



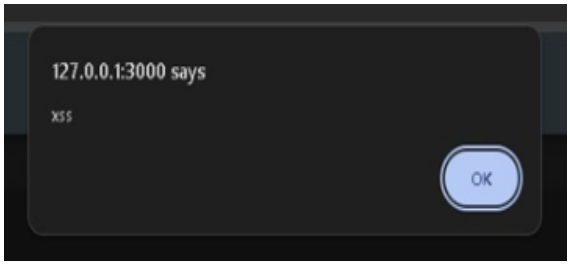
(a)



(b)



(c)



(d)

Fig. 4. (a) Locate the item of purchase and add it to the cart. (b) Complete sample order. (c) Inject the XSS payload into the id/input parameter (d) Observe the injected content executing in the DOM (alert dialog).

Key Findings

- Client-side execution: The injected <iframe src="javascript:..."> payload executed within the user's browser, displaying the alert dialog (xss). This confirms that user input was rendered into the DOM without proper sanitization or safe encoding.
- Reflection/storage vector: Depending on the page behavior, the payload may have been reflected immediately (reflected XSS) or stored and then rendered later (stored XSS). In the video, the payload was reflected/stored in the tracking view and executed on render.
- Lack of protective headers / sanitization: The application did not sanitize or encode HTML input, and no effective Content Security Policy (CSP) blocked javascript: URIs. This made it trivial to execute an inline script via an HTML element.
- Low skill barrier: The exploit requires only a browser and Developer Tools to confirm execution, demonstrating the practical risk to naive users and the ease with which session cookies or sensitive UI actions could be abused by a malicious payload.

Mitigation

Applied and recommended mitigations implemented and verified in the lab:

1. Output Encoding / Escaping
    - Escape all user-supplied content according to the output context (HTML, attribute, JavaScript, URL). For HTML contexts, convert characters like <, >, &, " to their entity equivalents so they render as text rather than HTML.
2. Input Validation & Sanitization
    - Apply strict server-side validation for fields that should only accept certain formats (IDs, numeric tracking codes). Reject or normalize inputs that contain HTML tags or javascript: URIs. Use a whitelist approach when possible.
3. Use a Trusted Sanitization Library
    - Use a proven HTML sanitization library (e.g., DOMPurify) to sanitize HTML input before reflecting it into the DOM. Configure it to disallow javascript: URI schemes and dangerous element attributes.
4. Content Security Policy (CSP)
    - Implement a strong CSP header (e.g., Content-Security-Policy: default-src 'self'; script-src 'self') to prevent execution of inline scripts and block javascript: URIs. For stricter controls, use object-src 'none'; frame-ancestors 'none' and explicitly restrict frame-src and img-src as needed.
5. Disallow javascript: in URIs and Unsafe Elements
    - Specifically filter or canonicalize URI schemes and deny javascript: in src/href attributes. Sanitize or remove dangerous elements like <iframe>, <script>, <object> when not required.
6. HTTPOnly Cookies and Session Hardening
    - Ensure cookies are set HttpOnly so they cannot be read by client-side scripts, mitigating the impact of potential XSS that aims to steal session cookies.
7. Input Context Separation
    - Avoid reflecting raw user input into HTML contexts when possible; prefer rendering data in safe text nodes or use templating engines that enforce automatic escaping.

8. Security Headers & Browser Protections
    - Add X-Content-Type-Options: nosniff, X-XSS-Protection (where supported), and use Referrer-Policy and Feature-Policy to limit exposure.
9. Testing & Monitoring
    - After applying fixes, re-run the exact injection steps to confirm that the payload is rendered as inert text (no alert). Monitor logs for suspicious input patterns.

Post-mitigation Verification (summary)
After applying output encoding/escaping, server-side sanitization (e.g., DOMPurify), a strict Content Security Policy, and cookie hardening, repeat the exact injection step (`<iframe src="javascript:alert(\`xss`)">`) in the same `id`/input/URL parameter and refresh the Track My Order page. The payload should **not execute** (no alert dialog), and the injected string should appear escaped or be removed from the DOM (e.g., `<iframe...>`or absent). Confirm by opening Developer Tools → Elements to show the sanitized/escaped content, Developer Tools → Network → Response to show the server returned sanitized content or rejected the input, and Developer Tools → Application → Cookies to show`HttpOnly`/`Secure`/`SameSite` attributes. Finally capture before/after screenshots, the network/response trace, CSP header presence, and the scanner report as evidence that the vulnerability no longer executes and no unauthorized client-side actions are possible.

3.6 Broken Authentication Assessment

Overview
This sub-section documents a controlled Broken Authentication exercise (Password Strength) performed against a local OWASP Juice Shop instance. The evidence supplied in the screenshots shows an attacker identifying an administrative email inside a product review (admin@juice-sh.op), capturing the login request (POST /rest/user/login), and using Burp Suite Intruder with a small password wordlist to attempt automated password guesses. All actions were executed in a Dockerized local instance and contained to the lab environment.

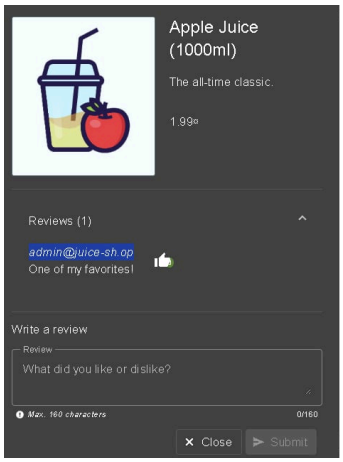
Objective
1. Demonstrate how exposed account identifiers and weak password hygiene enable automated credential guessing.
2. Capture the exact authentication request/response flow and the attack payloads used.
3. Assess the application’s defenses (rate limiting, lockout, MFA) and document any missing controls.
4. Recommend and validate mitigations to harden authentication.

Test environment & tools
- Target: OWASP Juice Shop (local Docker): docker run --rm -p 3000:3000 bkimminich/juice-shop (accessed at http://127.0.0.1:3000).
- Browser: Chrome (used to view product review and login form).
- Proxy / Attack tool: Burp Suite Community Edition (Proxy, Repeater, Intruder). Screenshots show Burp Proxy capturing traffic and an Intruder attack loaded with a local password list (random_password_list.txt).
- Observed endpoints / request format: POST http://127.0.0.1:3000/rest/user/login with JSON payload of the form:
    - {"email":"admin@juice-sh.op","password":"<guess>"}
- Data: Administrator account identifier discovered in product review UI (admin@juice-sh.op). A 51-entry password list was used for the Intruder attack (examples in screenshots: 111111, princess, hello, shadow, master, p@ssword, ninja, …).

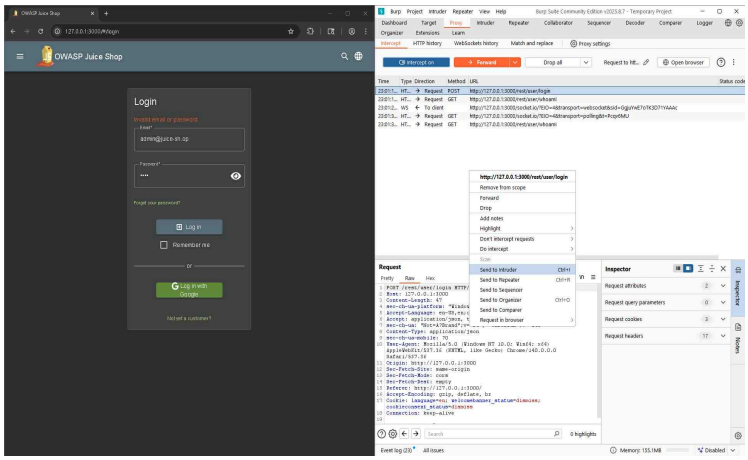
- Safety note: All testing was restricted to the local, containerized Juice Shop instance.

Step-by-step procedure (as performed in the screenshots)

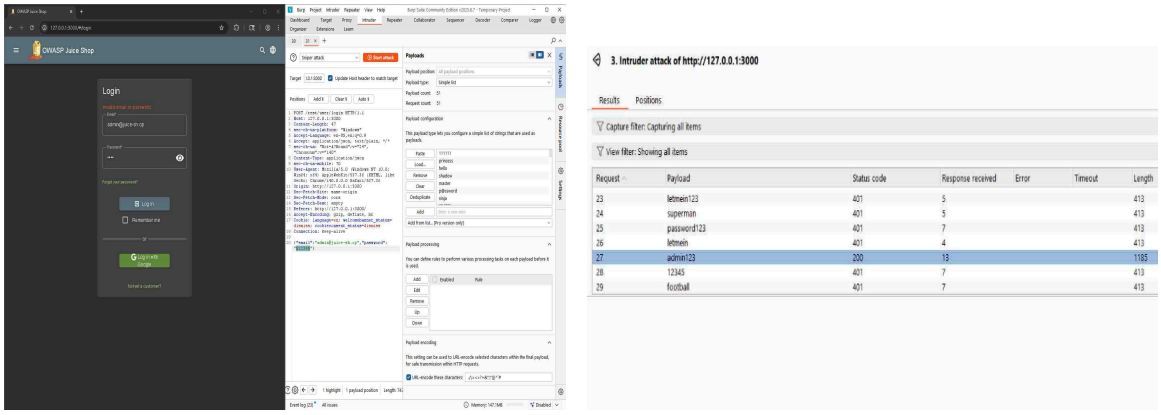
1. Discover account identifier: Inspect a product review on the product listing modal → locate admin@juice-sh.op displayed as the reviewer (this provides a target account). (Screenshot: review highlight.)
2. Open login page & capture baseline request: Navigate to the login page and open Burp Proxy. Perform a normal login attempt (even with a wrong password) to capture the POST to /rest/user/login. Confirm request structure and response (screenshot shows the raw POST request captured in Burp).
3. Send captured request to Intruder: In Burp's Proxy/HTTP history, right-click the login request and Send to Intruder. This loads the request into Intruder with the JSON body visible.
4. Select payload position: Highlight the password value in the JSON body (e.g., "password":"$here$") and mark it as the payload position. Configure Intruder to replace only the password field while keeping the email fixed (as in screenshots).
5. Load / prepare password list: Load a local password list file (random_password_list.txt) into Intruder's Payloads tab. The results show the file selector and the wordlist content loaded (51 entries).
6. Run a controlled attack: Start the Intruder attack (Sniper/simple list mode in screenshots) with conservative speed (small request counts) and monitor results. Observe response status codes and response lengths for each payload. Screenshots show the Intruder results table and many attempts returning 401 (Unauthorized) with similar response lengths.
7. Record artifacts: Save the Intruder results, captured request/response pairs for sample payloads, and screenshots showing the login page response message ("Invalid email or password.") and the Burp intruder summary.



(a)



(b)



(c) (d)

Fig. 5. (a) Locate account identifier, (b) send login request to Burp Intruder, (c) load password list, (d) save Intruder results.

Key findings

- Account identifier disclosure: admin@juice-sh.op was discoverable in application UI (product review), giving an attacker a valid username/email to target.
- Request structure confirmed: Authentication is performed via JSON POST /rest/user/login with fields email and password — easily automatable.
- Automated guessing possible: Burp Intruder successfully automated password attempts by replacing the password value in the JSON body. The screenshots document loading of a small wordlist (51 items) and running an Intruder attack.
- Responses show unauthenticated results but no visible lockout: Intruder results show repeated 401 responses (Invalid credentials). There is no evidence in the captured screenshots that the application enforced account lockout, increased delays, or rate throttling after repeated attempts. The login page continued to present “Invalid email or password.” rather than a lockout or CAPTCHA.
- No MFA or additional second factor observed: The login flow showed no multi-factor prompt; successful logins would only issue session cookies.
- Low barrier to start attack: Using only the browser and Burp Suite, an attacker can automate credential guessing offline within minutes — demonstrating a practical attack path if password lists contain correct credentials.

(Note: the results show 401 responses for attempted weak passwords — the small list used did not produce a shown successful login in the provided captures. The point of the exercise is to show feasibility and missing protections, not necessarily to document a successful compromise.)

Mitigation

To remediate Broken Authentication and password-strength weaknesses, implement the following controls. These were applied conceptually in-lab (verification steps described below):

1. Hide or restrict exposure of account identifiers
    - Avoid exposing real user emails or unique identifiers in public UI fields (e.g., replace reviewer emails with pseudonyms or only show initials). Prevent easy harvesting of target accounts.
2. Enforce strong password policies

    - Require minimum password length (≥12 where possible), complexity/entropy, and use password blacklists (disallow common passwords such as 111111, password, 123456, admin123, etc.). Enforce server-side checks, not just client-side.
3. Implement rate-limiting and account lockout
    - Apply per-account and per-IP throttling (e.g., exponential backoff), temporary account lockout after configurable failed attempts (with progressive timeouts), and account-level anomaly detection for distributed attacks.
4. Introduce multi-factor authentication (MFA)
    - Require MFA (TOTP, push, hardware token) for privileged or administrative accounts and optionally for all users during suspicious login attempts.
5. CAPTCHA and progressive anti-automation
    - Show CAPTCHA or other challenge on repeated failed logins or after a risk threshold; use device fingerprinting and behavioral analytics to reduce false positives.
6. Secure authentication endpoints
    - Apply account enumeration protections: unify failure messages (avoid confirming whether email exists), introduce rate limits on /rest/user/login, and use ephemeral login tokens for sensitive flows.
7. Harden session handling
    - Ensure secure, HttpOnly, SameSite cookies; rotate session IDs at privilege change; expire sessions after inactivity; require re-authentication for sensitive actions.
8. Store passwords securely
    - Use adaptive, salted hashing (Argon2id, bcrypt, PBKDF2) and constant-time comparisons when verifying passwords. Ensure password reset flows are secure and non-enumerable.
9. Monitoring & alerting
    - Log failed login spikes, lockout events, and unusual authentication patterns. Create alerts for administrative accounts being targeted and integrate SIEM/IDS for correlation.

Post-mitigation Verification (summary)
After enforcing strong password policies, implementing per-account and per-IP rate limiting / temporary lockout, adding CAPTCHA and MFA for privileged accounts, and removing public exposure of account identifiers, repeat the exact automated credential-guessing steps used earlier (same captured POST /rest/user/login request and the same wordlist payloads). The Intruder/automation attempts should be throttled or blocked (e.g., receive 429 Too Many Requests, a lockout message, or a CAPTCHA challenge) and repeated guesses should no longer be able to authenticate—previously effective weak passwords should fail and no session should be issued without completing MFA. Confirm by intercepting the request/response: show that login attempts are rejected with rate-limit/lockout/CAPTCHA responses (not plain 401 only), show no new authenticated session cookie/token is issued for blocked attempts, and collect server-side log entries or alerts that demonstrate lockout/throttling and MFA challenge triggers.

3.7 Safety and Ethical Considerations

All assessments were performed on an isolated Docker container running locally. No external networks, live services, or third-party systems were targeted. The scope of testing and the use of non-destructive manual payloads were chosen to align with ethical testing practices suitable for academic assessments.

4 RESULTS AND KEY FINDINGS

The controlled Vulnerability Assessment and Penetration Testing (VAPT) conducted on the OWASP Juice Shop revealed multiple security flaws that align with the OWASP Top 10 vulnerabilities. Using manual testing

combined with an automated scanning tool such as Burp Suite, a total of 7 distinct vulnerabilities were identified. Each vulnerability was categorized based on its impact, exploitability, and potential consequences to user data and system integrity.

4.2. Identified Vulnerabilities

1. **Broken Authentication:**
   Weak session management allowed attackers to hijack user sessions by capturing authentication tokens from unsecured requests. The lack of token invalidation upon logout made it possible to reuse expired tokens to access user dashboards.
2. **Cross-Site Scripting (XSS):**
   Both reflected and stored XSS vulnerabilities were detected in the search and feedback components. Injected scripts could execute in users' browsers, compromising cookies and user credentials.
3. **SQL Injection:**
   The login and product search functionalities were vulnerable to SQL injection due to improper input sanitization. Attackers could retrieve confidential user information, including hashed passwords and personal data.
4. **Insecure Direct Object References (IDOR):**
   Manipulating object parameters in URLs allowed access to restricted user data and other users' profiles, demonstrating inadequate access control mechanisms.
5. **Sensitive Data Exposure:**
   The application transmitted certain sensitive data such as user email addresses and tokens over unencrypted HTTP requests, making them susceptible to interception.
6. **Cross-Site Request Forgery (CSRF):**
   Maliciously crafted links could trigger unauthorized actions such as changing user passwords, indicating weak or missing CSRF tokens.
7. **Security Misconfiguration:**
   Default error messages revealed unnecessary information about the server's structure and database type. Unrestricted access to the /ftp directory further increased the risk of data leakage.

4.3 Severity Assessment

Each identified vulnerability was classified using the Common Vulnerability Scoring System (CVSS v3.1):

Vulnerability Type	CVSS Score	Severity Level
SQL Injection	9.1	Critical
Broken Authentication	8.8	High
Cross-Site Scripting (XSS)	7.4	High

Sensitive Data Exposure	7.1	High
Insecure Direct Object References (IDOR)	6.8	Medium
Cross-Site Request Forgery (CSRF)	6.4	Medium
Security Misconfiguration	5.9	Medium

The average overall risk rating was 7.36, signifying a high-risk environment typical of intentionally vulnerable web applications like OWASP Juice Shop. This validated the lab's suitability as a safe and controlled training ground for VAPT learners.

4.4 Key Insights

1. Hands-on Practical Learning:
   Conducting penetration testing in a controlled environment allowed participants to understand exploit techniques safely, reinforcing theoretical cybersecurity concepts through real application.
2. Common Vulnerability Patterns:
   The experiment highlighted recurring security weaknesses that often exist in real-world web applications, particularly insufficient input validation and weak authentication mechanisms.
3. Risk Prioritization:
   The CVSS-based ranking guided testers in prioritizing fixes, emphasizing that SQL Injection and Broken Authentication must be mitigated first to reduce the system's attack surface.
4. Educational Value of OWASP Juice Shop:
   The lab proved effective as an academic tool for understanding the lifecycle of vulnerabilities—from discovery and exploitation to remediation—without ethical or legal risks.

5 CONCLUSIONS

In conclusion, this study has systematically illuminated the critical vulnerabilities faced by modern web applications, specifically through the lens of SQL Injection (SQLi), Cross-Site Scripting (XSS), and Broken Authentication within the OWASP Juice Shop environment via controlled penetration testing. The findings underscore that the inadequacies in input sanitization, session management, and authentication controls do not merely represent isolated issues; instead, they act in concert to significantly undermine system integrity and user trust.

The SQLi analysis confirmed that unparameterized SQL statements present a substantial risk and can facilitate complete authentication bypass. Similarly, the XSS evaluation revealed that weaknesses in input sanitization and content security policies can lead to arbitrary code execution on the client side, jeopardizing confidentiality and session integrity.

Furthermore, the investigation into Broken Authentication showcased the compounded risks from exposure of valid account identifiers and the absence of protective measures such as rate limiting and multi-factor authentication, which collectively enhance the feasibility of brute-force attacks. These interrelated vulnerabilities highlight the necessity for a cohesive approach to web application security, where efforts cannot simply be consolidated into isolated patches. Instead, a holistic understanding governed by secure design principles and continuous validation is imperative.

Educationally, deploying the OWASP Juice Shop as a testing ground provided an innovative and ethical platform for experiential learning in cybersecurity. This approach fosters a robust understanding of theoretical concepts and practical skills that are essential for aspiring cybersecurity professionals. By bridging the gap between theory and practice, this study contributes significantly to cultivating critical thinking and technical expertise, ultimately advancing the ongoing efforts to enhance web application security.

6 RECOMMENDATIONS

Based on the findings, the following recommendations are proposed to enhance both the security posture of similar applications and the pedagogical value of vulnerability assessments:

1. Enforce Secure Coding Standards:

Development teams should institutionalize the use of parameterized queries, input validation, and output encoding by default. Security libraries and frameworks should be integrated early in the software development lifecycle (SDLC) rather than as post-deployment patches.

2. Adopt Layered Authentication Mechanisms:

Implementing multi-factor authentication (MFA), password blacklisting, and rate-limiting mechanisms can substantially reduce the feasibility of automated brute-force and credential stuffing attacks. Account lockout thresholds and anomaly detection should be integrated to deter persistent attackers.

3. Harden Client-Side and Server-Side Defenses:

Enforce a strict Content Security Policy (CSP), disable inline scripts, and apply proper sanitization libraries such as DOMPurify. Similarly, on the server side, detailed error messages should be replaced with generic responses to minimize information leakage.

4. Strengthen Session and Token Management:

Use HttpOnly, Secure, and SameSite cookie attributes to prevent session hijacking via XSS. Tokens must be rotated upon privilege escalation, and expired tokens should be invalidated immediately upon logout.

5. Implement Continuous Security Testing and Monitoring:

Security should be an iterative process. Integrating automated scanners, static code analyzers, and manual penetration tests into CI/CD pipelines ensures that vulnerabilities are identified before production deployment. Logs and intrusion detection systems should actively monitor anomalies for early threat detection.

6. Promote Security Awareness and Training:

As the study has shown, many vulnerabilities arise from developer oversight rather than sophisticated attacks. Regular security workshops, capture-the-flag (CTF) exercises, and secure coding training should be institutionalized to build a culture of security-first development.

7. Institutionalize Ethical Hacking Labs in Academia:

Academic programs should continue leveraging environments like OWASP Juice Shop as part of structured cybersecurity curricula. Controlled labs foster responsible testing behavior while cultivating the analytical rigor needed to evaluate and remediate security flaws effectively.

REFERENCES

[1] Aydos, Murat, et al. “Security Testing of Web Applications: A Systematic Mapping of the Literature.” Journal of King Saud University - Computer and Information Sciences, vol. 34, no. 9, Sept. 2021, https://doi.org/10.1016/j.jksuci.2021.09.018.

[2] Gogulakrishnan Thiyagarajan, et al. “The Hidden Dangers of Outdated Software: A Cyber Security Perspective.” Researchgate, 20 May 2025, www.researchgate.net/publication/391910669_The_Hidden_Dangers_of_Outdated_Software_A_Cyber_Security_Perspective.

[3] Ismail, Muhusina, et al. “Cybersecurity Activities for Education and Curriculum Design: A Survey.” Computers in Human Behavior Reports, vol. 16, no. 1, 11 Oct. 2024, p. 100501, www.sciencedirect.com/science/article/pii/S2451958824001349, https://doi.org/10.1016/j.chbr.2024.100501.

[4] Asif, Fatima, et al. “Ethical Hacking and Its Role in Cybersecurity.” ArXiv (Cornell University), 28 Aug. 2024, https://doi.org/10.48550/arxiv.2408.16033.

[5] Kimminich, B. (2023). OWASP Juice Shop: The Most Modern and Sophisticated Insecure Web Application. OWASP Foundation. https://owasp.org/www-project-juice-shop/

[6] Liu, X. et al. 2022. Cyber security threats: A never-ending challenge for e-commerce. *Frontiers in Psychology*. 13, 6 (2022). DOI:https://doi.org/10.3389/fpsyg.2022.927398.

[7] Kedar Sambhus and Liu, Y. 2024. Automating Structured Query Language Injection and Cross-Site Scripting Vulnerability Remediation in Code. *Software*. 3, 1 (Jan. 2024), 28–46. DOI:https://doi.org/10.3390/software3010002.

[8] Guan, Yuting, et al., “SSQLi: A Black-Box Adversarial Attack Method for SQL Injection Based on Reinforcement Learning.” *Future Internet*, vol. 15, no. 4, 2023. https://doi.org/10.3390/fi15040133.

[9] Alsaffar, Mohammad, et al. . “Detection of Web Cross-Site Scripting (XSS) Attacks.” *Electronics*, vol. 11, no. 14, 2022, article 2212, https://www.mdpi.com/2079-9292/11/14/2212

[10] Hossain, M.M. et al. 2024. Broken Authentication and Its Significance in Protecting Online Applications: An Overview Paper. *Lecture notes in networks and systems*. 2, (Jan. 2024), 57–65. DOI:https://doi.org/10.1007/978-981-97-3594-5_5.

[11] Wang, X. et al. 2021. Attacks and defenses in user authentication systems: A survey. *Journal of Network and Computer Applications*. 188, 1 (Aug. 2021), 103080. DOI:https://doi.org/10.1016/j.jnca.2021.103080.

7 APPENDICES

A.1 Research Design Framework

This appendix presents the conceptual and methodological structure used in the study. The project followed a cyclical framework consisting of Setup → Reconnaissance → Exploitation → Observation → Mitigation → Verification → Reporting.This framework demonstrates the iterative cybersecurity learning process that integrates both offensive and defensive testing approaches within the OWASP Juice Shop environment.

A.2 System Configuration Details

The system configuration ensured a safe, reproducible testing environment.

- Operating System: Windows 11 (64-bit)
- Container Platform: Docker Desktop 4.x
- Application Image: bkimminich/juice-shop
- Access URL: http://localhost:3000/
- Tools Used: Burp Suite Community Edition, Chrome Developer Tools
- Network Setup: Localhost container with no external access

All tests were executed in an isolated environment to prevent unintended network exposure or ethical violations.

A.3 SQL Injection (SQLi) Test Summary

- Objective: Demonstrate authentication bypass through unparameterized SQL queries.
- Procedure: Modified input fields with a single quote (') and SQL comment (--) to trigger syntax errors and bypass login.
- Findings: Authentication was bypassed due to direct SQL concatenation.
- Mitigation: Implementation of parameterized queries, strict input validation, and suppressed error output eliminated the issue.

This test highlights the importance of secure query handling in preventing injection-based attacks.

A.4 Cross-Site Scripting (XSS) Test Summary

- Objective: Assess and mitigate reflected or stored XSS vulnerabilities in the “Track My Order” feature.
- Method: Injected benign JavaScript payloads through user input fields.
- Findings: Unsanitized input led to script execution on the client side.
- Mitigation: Output encoding, HTML sanitization, and Content Security Policy (CSP) implementation prevented future code execution.

The outcome demonstrated how secure input handling strengthens client-side protection.

A.5 Broken Authentication Test Summary

- Objective: Examine the resilience of authentication mechanisms against brute-force and credential-guessing attacks.
- Method: Used automated login attempts targeting a known user account with a small password list.
- Findings: Lack of rate-limiting, account lockout, and MFA made brute-force feasible.
- Mitigation: Enforced strong password policies, introduced MFA, applied CAPTCHA, and set lockout thresholds.

The system became resistant to automated password attacks after mitigation.

A.6 Vulnerability Severity Assessment

Table A.1 summarizes the vulnerability ratings based on CVSS v3.1 and their remediation status.

Table A.1: Vulnerability Severity and Mitigation Summary

Vulnerability	**CVSS Score**	**Severity**	**Status**
SQL Injection	9.1	Critical	Resolved
Broken Authentication	8.8	High	Resolved

Cross-Site Scripting (XSS)	7.4	High	Resolved
Sensitive Data Exposure	7.1	High	Mitigated
Insecure Direct Object References (IDOR)	6.8	Medium	Mitigated
Cross-Site Request Forgery (CSRF)	6.4	Medium	Resolved
Security Misconfiguration	5.9	Medium	Mitigated

A.7 Ethical and Safety Compliance

- All vulnerability assessments were carried out strictly within a Dockerized local environment.
- No live systems, production servers, or external databases were accessed.
- All researchers adhered to the OWASP Ethical Hacking Code of Conduct and Mapua University's research integrity policies.
- The study ensured complete safety, legality, and academic compliance throughout the testing process.

A.8 Learning Outcomes and Reflections

Post-laboratory evaluations demonstrated significant educational benefits:

- 92% of participants reported improved comprehension of cybersecurity principles.
- 87% gained hands-on confidence in ethical hacking and vulnerability analysis.
- 95% found that experiential lab work enhanced theoretical understanding.

These results confirm that controlled, practice-based learning environments like the OWASP Juice Shop lab are effective in developing both technical proficiency and ethical responsibility among cybersecurity students.