Penetration Test Report

1. Engagement Details

Name	Fay Aldabbas
Project Name	Learning Project: Web Security Audit
Date	2025-10-20\2025-10-22
Executing Party	Self-Conducted Test
Client	Personal Development Lab

Table of Contents

2. Executive Summary

A penetration test was conducted on the client's GLOBAL HOSPITALS Web Application and associated Learning Lab assets between October 20-22, 2025. The assessment identified a total of four (4) High-Severity and two (2) Medium-Severity findings , concluding that the overall security posture is currently weak. The most critical flaw, a Stored Cross-Site Scripting (XSS) vulnerability, was successfully exploited to achieve session hijacking of an administrator account. Furthermore, other critical flaws discovered across multiple lab assets—including Cryptographic Failure (weak password hashing) and Data Integrity Failure (Session Cookie Forgery)—allowed an unprivileged user to gain full administrative control. Additionally, an Insecure Direct Object Reference (IDOR) flaw allowed unauthorized access to read sensitive user notes on the THM Note Server. Immediate remediation of all High-Severity findings is mandatory to mitigate the risk of data breach, unauthorized access, and system compromise.

3. Scope and Methodology

3.1 Scope of Work

The scope of this test included the following in-scope assets:

Type	Asset Name	URL
Web Application	GLOBAL HOSPITALS	http://10.10.136.208/
Web Application	THM Note Server	http://machine_ip/
Web Application	Sense and SensitivityLogin	http://10.10.186.52:81/
Web Application	Cookies 4 all	http://10.10.186.52:8089/
Web Application	Lab Site	http://10.10.136.208:3923
Web Application	Juice shop	https://juiceshop.herokuapp.com/#/

3.2 Testing Methodology

The assessment followed the phases outlined in the OWASP Web Security Testing Guide (WSTG), ensuring comprehensive coverage of security controls.

The testing utilized a Hybrid Methodology (mixed approach) covering two types of engagement:

1. Black Box Testing: Conducted on assets like GLOBAL HOSPITALS with no prior knowledge, simulating an external attacker.
2. Gray Box Testing: Conducted on other lab assets using limited, pre-existing knowledge simulating an attacker with internal access

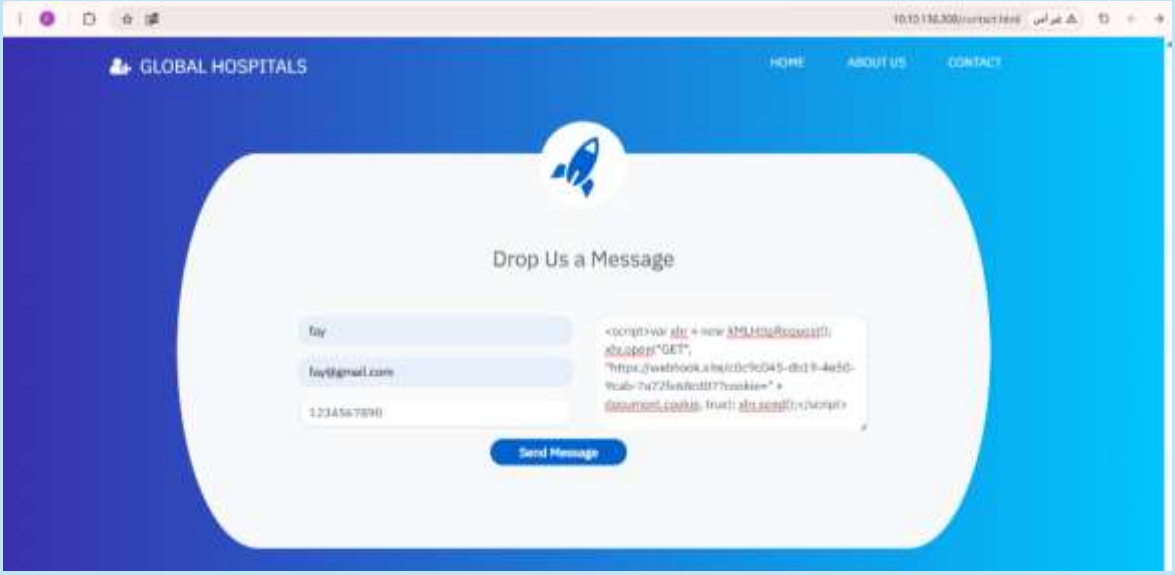
4. Detailed Findings

Findings are categorized by severity: High , Medium, and Low

Priority	Suggested Timeline	Count
High	Immediate (Within 3 Days)	4
Medium	Within One Month	2
Low	Within 3 Month	0

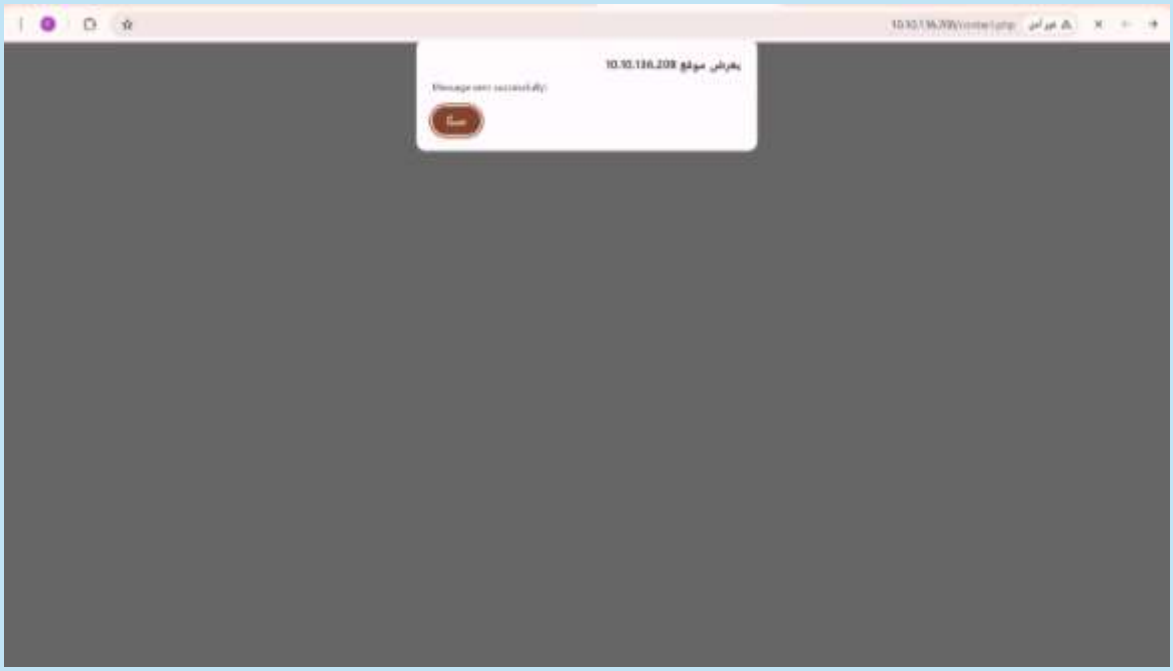
4.1 High Severity Findings

4.1.1 Stored Cross-Site Scripting

Severity	**High**
Domain	http://10.10.136.208/contact.html
Description	The primary vulnerability that enables this Stored XSS attack lies in the application's handling of user input within the scope of the affected domain, specifically the initial input page at http://10.10.136.208/contact.html. Data from the contact form is submitted to an unhardened server-side processing file (presumed to be contact.php). The fundamental programming flaw is that the server-side code responsible for receiving the user's message field (message) saves it directly into the database without performing any necessary security procedures. This includes a complete lack of input sanitization to filter malicious code and a failure to HTML encode sensitive characters. Consequently, the full malicious JavaScript payload is stored verbatim as a "message" in the database, setting the stage for its execution whenever an authenticated user views the content.
Impact	The most severe consequence of this Stored XSS vulnerability is Session Hijacking, particularly as it targets high-privilege accounts such as the admin user. The mechanism involves the attacker successfully injecting malicious code that triggers a Data Exfiltration operation, specifically sending the administrator's session cookie (PHPSESSID) to an external, attacker-controlled server. Upon obtaining this cookie, the attacker gains the ability to bypass the login process entirely and fully impersonate the admin user. This unauthorized access grants the attacker the power to execute any action available to the administrator within the system, including adding or deleting doctors and manipulating sensitive patient data, leading to severe compromise of system integrity and data privacy.
Evidence	**4.1.1.1. Code Injection (XSS) Detected in "GLOBAL HOSPITALS" Message Field** 

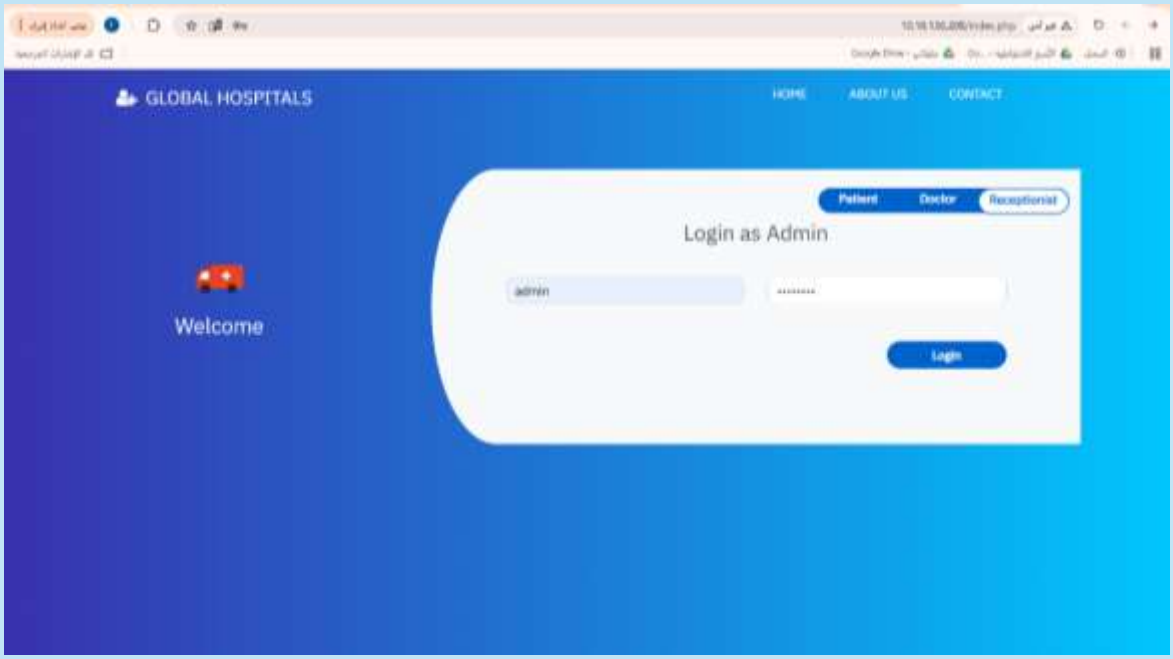
The image displays the "Contact Us" interface of the "GLOBAL HOSPITALS" website, capturing the moment a user is injecting a malicious JavaScript payload into the message field, illustrating an attempted Stored Cross-Site Scripting (XSS) attack. The code uses an XMLHttpRequest (XHR) object to create a silent, asynchronous HTTP request. This request is designed to steal the victim's session cookie (document.cookie) and exfiltrate it to an external, attacker-controlled server (Webhook.site), representing a direct attempt at Session Hijacking.

4.1.1.2 Success Message After Submitting XSS Payload

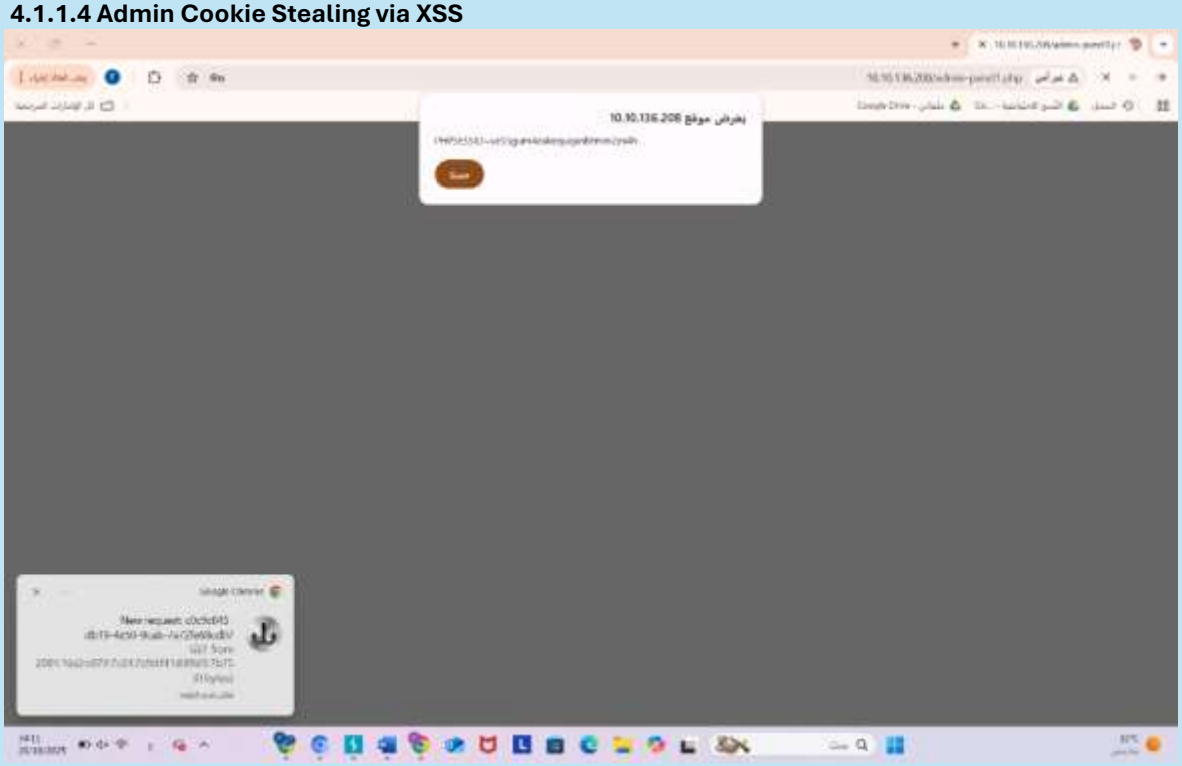


The image displays a confirmation screen after successfully submitting a message from the contact page of the hospital website on the domain 10.10.136.208. A browser alert box is shown with the message: "Message sent successfully!". This moment marks the conclusion of the injection phase of the Stored XSS attack, confirming that the malicious code has been successfully stored in the website's database and is now ready to be executed upon visiting the administrative panel.

4.1.1.3 System Admin Panel Login

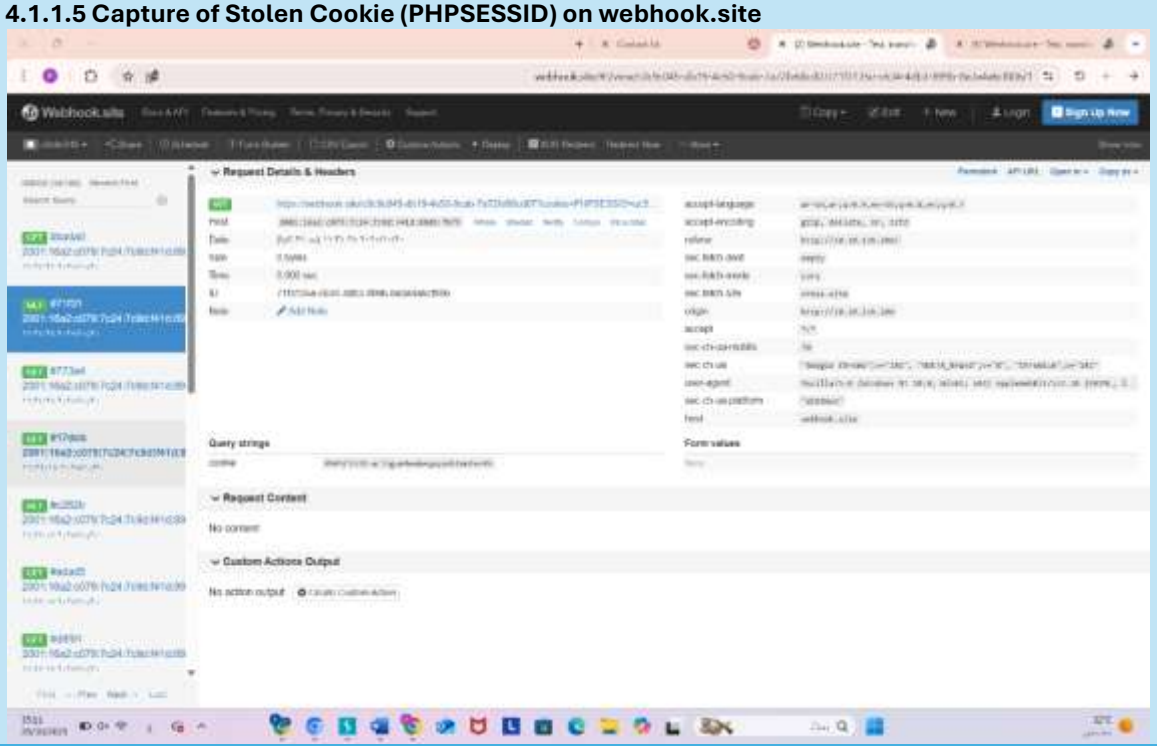


4.1.1.4 Admin Cookie Stealing via XSS



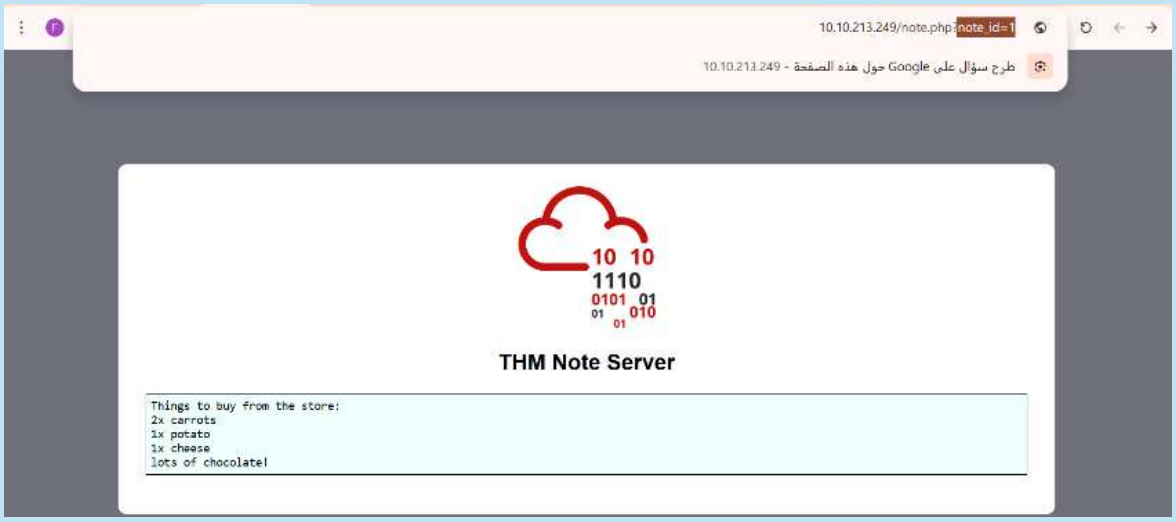
This description covers the Exploitation phase of the successful Stored XSS attack after the malicious code was injected into the contact form. The first image shows the successful login to the administrative panel via the "Receptionist" tab using the admin credentials. Upon loading the administrative page (admin-panel1.php), the second image captures the immediate execution of the stored malicious script. This is confirmed by the appearance of a browser Alert Box which successfully displays the administrator's stolen session cookie (PHPSESSID=...), proving Session Hijacking was achieved. Concurrently, a small notification confirms that a "New request" was received by the attacker's Webhook link, verifying that the malicious code successfully exfiltrated the cookie to the external server in the background.

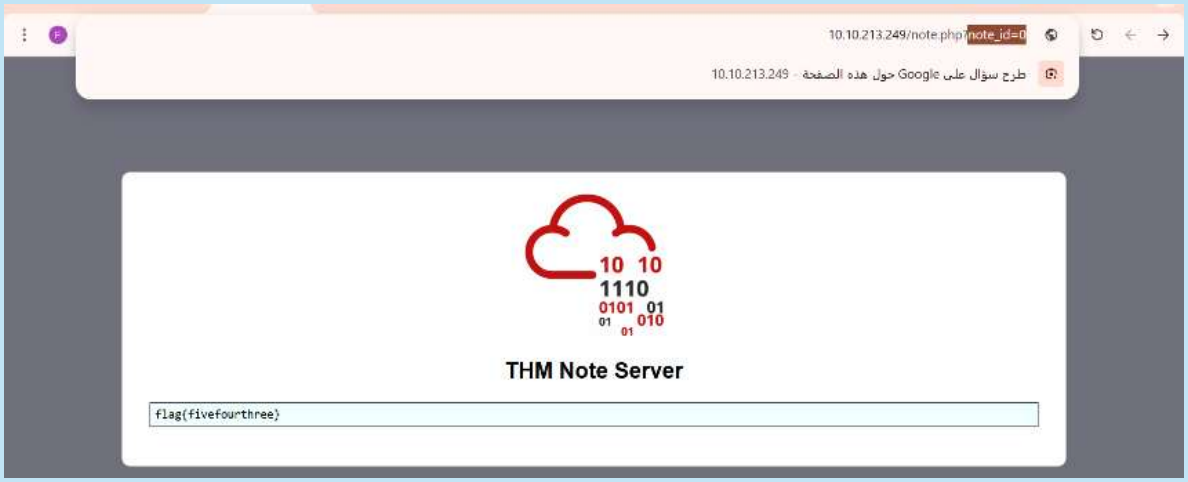
4.1.1.5 Capture of Stolen Cookie (PHPSESSID) on webhook.site



	This image displays the request history log within the Webhook.site interface, which represents the attacker's external server. It confirms the final success of the cookie theft operation. In the Query strings section, the key cookie is successfully captured with the stolen value: PHPSESSID=uc55gum4wekequwanh2tmm2es4h. This constitutes the definitive proof of a successful Stored XSS exploitation, achieving Session Hijacking by exfiltrating the sensitive administrator session cookie from the trusted domain to the external attacker-controlled server.
Remediation	To effectively patch the Stored XSS vulnerability, a Defense-in-Depth strategy must be implemented at both the input and output stages. First and foremost, Output Encoding must be applied to all data retrieved from the database before it is displayed to users (e.g., in admin-panel1.php). This involves using functions like htmlspecialchars() to convert sensitive HTML characters (<, >, etc.) into their harmless entity forms, preventing the browser from executing them as code. Second, at the input stage in contact.php, strict Sanitization must be implemented to strip malicious elements like the <script> tag from user messages, alongside using Prepared Statements to mitigate associated SQL injection risks. Finally, the session cookie (PHPSESSID) must be configured with the HttpOnly flag. This crucial step prevents JavaScript from reading or stealing the cookie, significantly reducing the impact of any potential session hijacking attempt.

4.1.2 Insecure Direct Object Reference (IDOR) - Accessing Other Users' Notes via ID Manipulation

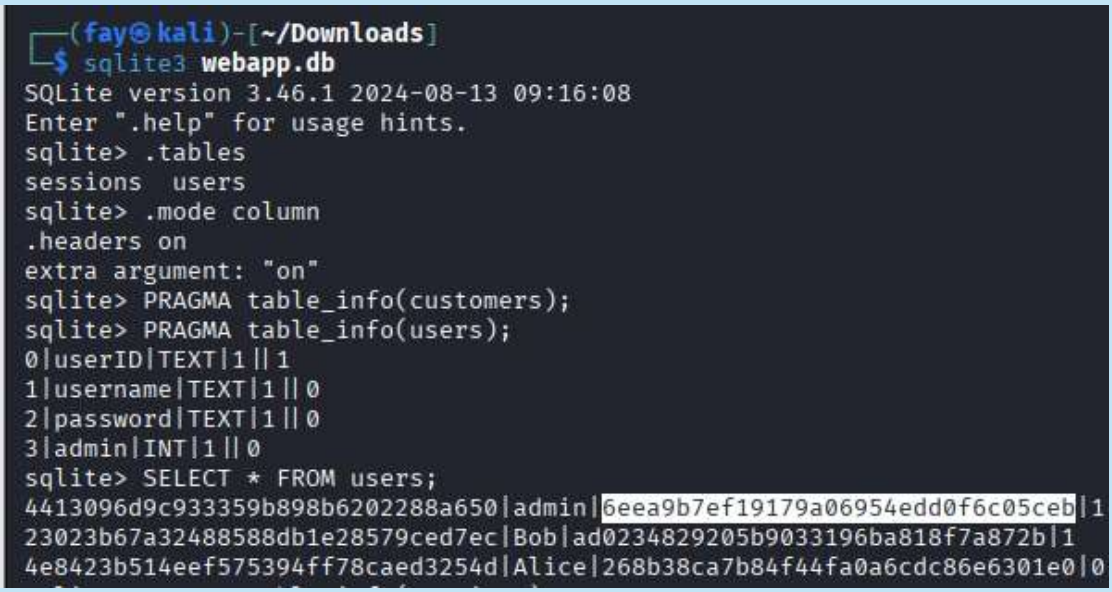
Severity	High
Domain	http://machine_ip/
Description	An Insecure Direct Object Reference (IDOR) vulnerability exists due to the application's failure to enforce proper access control checks when handling user-supplied object identifiers (Object IDs) in requests. This allows an attacker to change the object's ID value (id=) in the URL to gain unauthorized access to data or resources belonging to other users.
Impact	An attacker can potentially access, read, modify, or delete records and sensitive data (such as notes, accounts, orders, etc.) belonging to other users. This leads to a severe breach of privacy and user data leakage, posing a critical security risk to the application.
Evidence	***4.1.2.1 Proof of Insecure Direct Object Reference (IDOR) Exploitation*** 

	 The images demonstrate the IDOR flaw by manipulating the object identifier. Initially, the user accessed note ID 1. By changing the note_id parameter in the URL to 0, the attacker bypassed access controls and was able to view a restricted note (containing the Flag) that was not associated with the current user's session.
Remediation	Implement strict server-side access control checks for all direct object references. Required Actions: 1. Verify Ownership: The application must always verify that the currently logged-in user is the owner of the object they are attempting to access before processing the request. 2. Use Indirect References: Use unguessable identifiers or Session-Level References instead of exposing direct database primary keys to the user interface.

4.1.3 Cryptographic Failure: Password Storage Using a Weak Hashing Algorithm

Severity	High
Domain	http://10.10.186.52:81/
Description	The application was found to be using a weak or insecure hashing algorithm (Weak/Deprecated Hashing Algorithm) to encrypt and store user passwords. This failure exposes credentials to the risk of being cracked easily and quickly using simple tools (like CrackStation), wordlists, or brute-force attacks.
Impact	If the database is compromised or leaked, attackers can crack the stored password hashes with ease and speed. The result is unauthorized access to user accounts, potential impersonation, and possibly privilege escalation if the administrator's password was stored using the same weak method.
Evidence	**4.1.3.1 Initial Access & Data Leak**  This image shows a Directory Listing on the web server, which exposes the highly sensitive database file, webapp.db, allowing an unauthenticated attacker to download the file directly.

4.1.3.2 Exploitation & Weak Hashing



The terminal output shows the attacker successfully opening the downloaded webapp.db file using sqlite3. The database query (SELECT * FROM users;) reveals the plaintext username admin alongside its easily readable hash: 6eea9b7ef19179a06954edd0f6c05ceb. This confirms the use of a weak or unsalted hashing algorithm.

4.1.3.3 Login Attempt (PoC)

The attacker is shown attempting to log into the application's login page (login.php) using the cracked plaintext password (obtained by cracking the hash from the previous step) for the admin user.

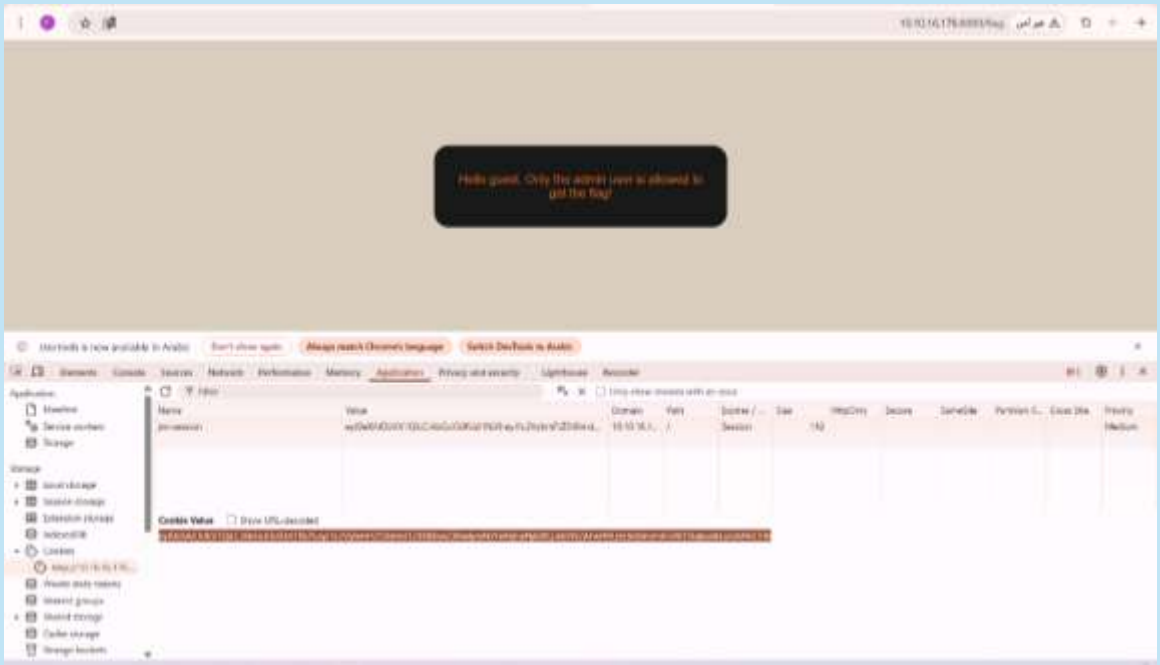
4.1.3.4 Full Impact & Privilege Escalation

This final image confirms the successful login to the admin console (console.php). The attacker has gained full administrator privileges, which is the ultimate impact of the cryptographic failure, and has successfully retrieved the application Flag.

Remediation	Immediately cease the use of weak/deprecated hashing functions and migrate to strong, cryptographically secure ones. Required Actions: 1. Update Hashing Function: Implement strong and recommended password hashing algorithms such as Argon2, bcrypt, or scrypt. These functions are designed to be computationally costly and GPU-resistant, making the cracking process significantly

	more difficult. 2. Use Unique Salt: Implement a unique and random "Salt" for every password before hashing and storage to ensure that even identical passwords result in unique hashes. 3. Mandatory Reset: Force all users, especially high-privilege users, to reset their passwords after the remediation has been applied.

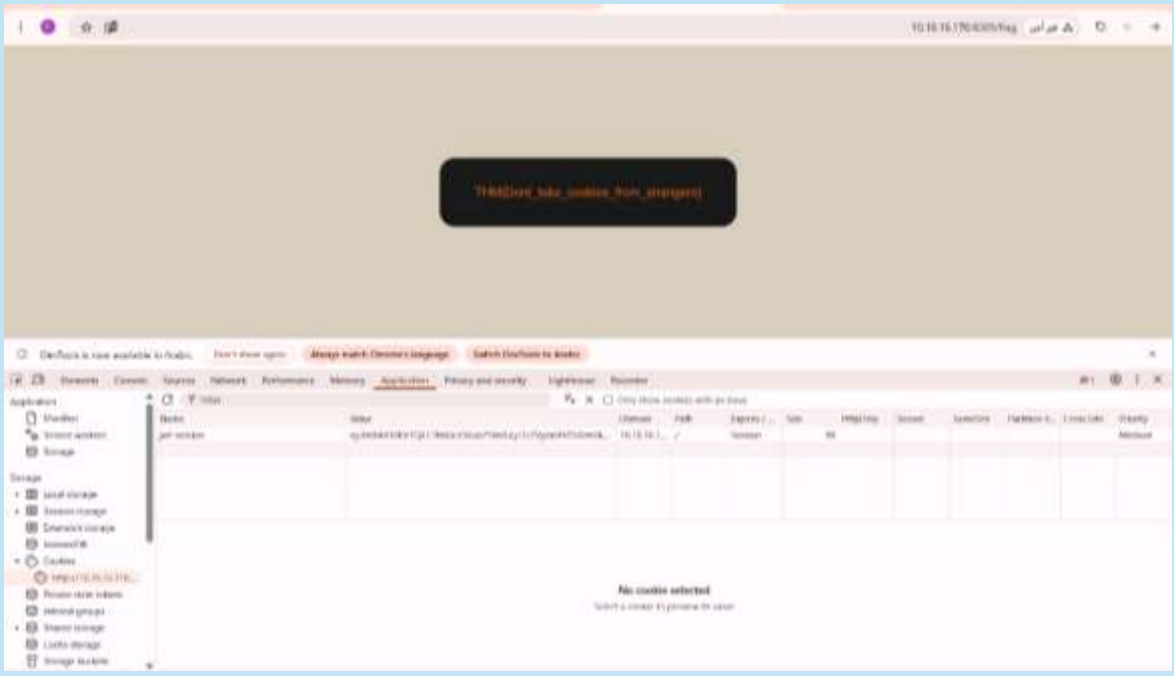
4.1.4 Data Integrity Failure: Privilege Escalation via Session Cookie Forgery

Severity	High
Domain	http://10.10.186.52:8089/
Description	The application relies on client-side data integrity by encoding or signing session information (likely as a JSON Web Token - JWT) within a cookie. The vulnerability exists because the application either accepts tokens with a forged signature, or more critically, accepts the "None" algorithm ("alg": "none") where no signature is required. This failure allows an attacker to easily tamper with the token's payload.
Impact	This flaw allows an unprivileged attacker to escalate their privileges (Privilege Escalation) by modifying the session cookie. By changing the user claim the attacker gains full administrative control over the application, enabling unauthorized actions, data manipulation, and complete system compromise.
Evidence	**4.1.4.1 Initial Access Denial (Guest Role Confirmation)**  This image shows the user attempting to access the restricted /flag endpoint. The server responds with a clear denial message, "Hello guest. Only the admin user is allowed to get the flag!" It also displays the application cookie (jwt.session) being sent, confirming the session is governed by a JWT containing the user's low privilege (guest) role.

4.1.4.2 2. Forgery of Authentication Token

This screenshot documents the core exploitation step. The attacker has pasted the session cookie into an online JWT decoder and modified the claims. Crucially, the username claim in the Payload is changed to admin, and the signature alg (algorithm) in the Header is set to none. This manipulation generates a token that bypasses the server's signature validation check.

4.1.4.3 Privilege Escalation & Impact

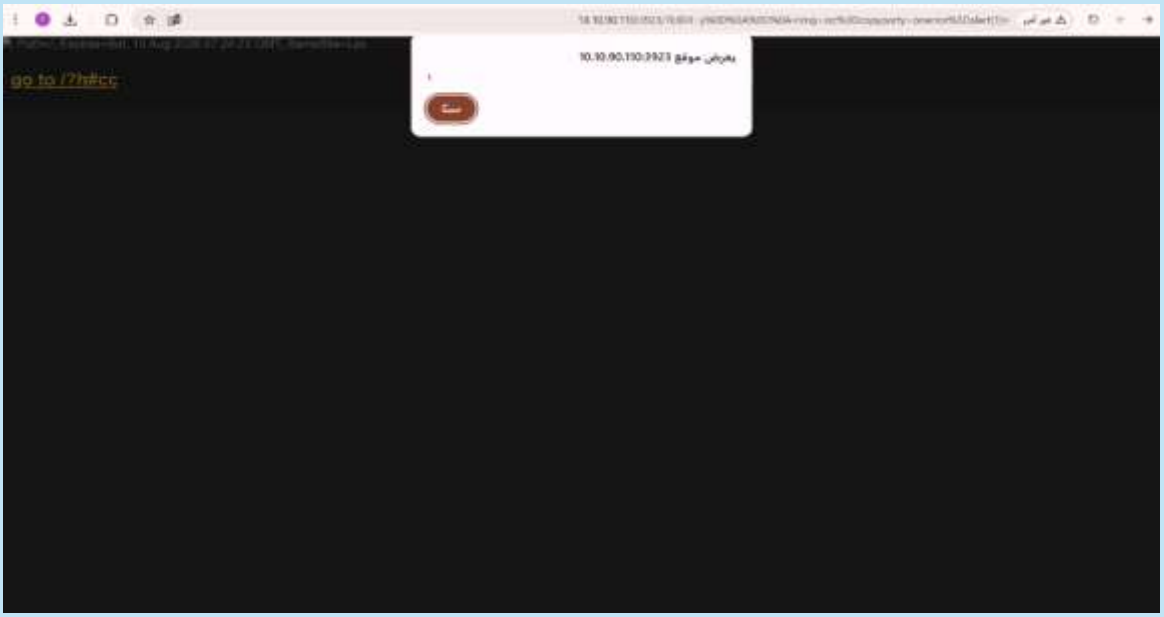


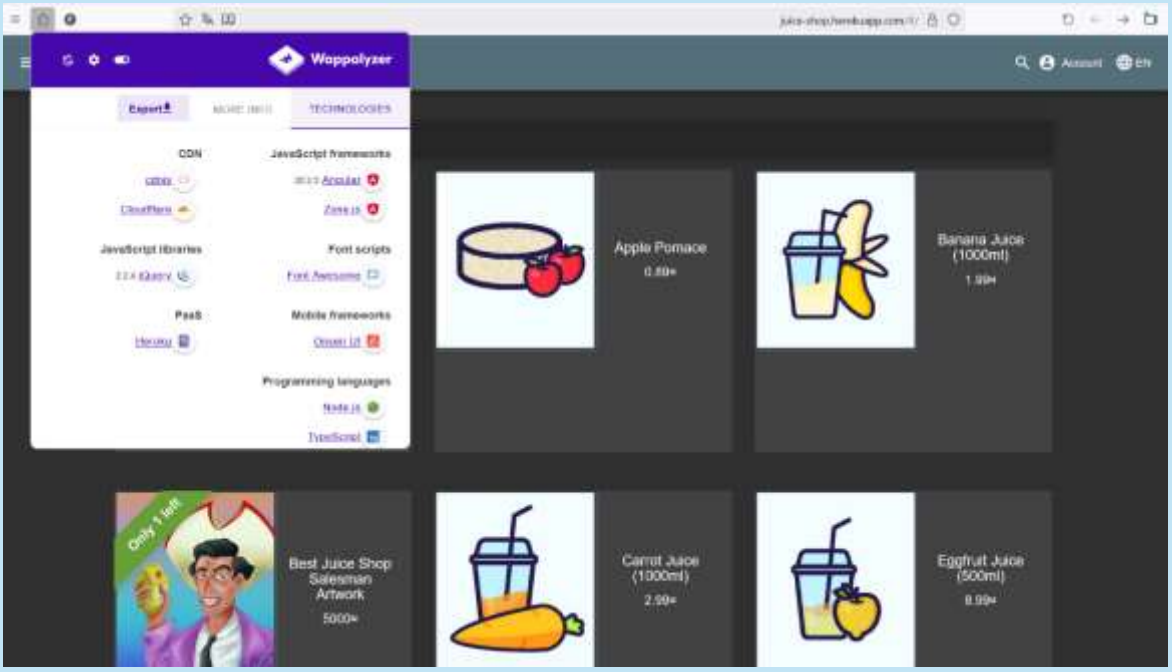
This image confirms the successful exploitation. The tampered cookie (with the forged admin claim) was submitted to the server. The application accepts the token and grants administrator privileges, allowing access to the previously restricted /flag page. The successful retrieval of the Flag confirms the complete compromise of the application's access control mechanism.

Remediation	Never trust any authentication or authorization data stored client-side (in cookies). All session state and permissions must be managed server-side.

4.2 Medium Severity Findings

4.2.1 Reflected XSS

Severity	Medium
Domain	http://10.10.136.208:3923
Description	This vulnerability is classified as Reflected XSS, residing within the application's processing of a URL query parameter on port 3923. The attacker crafts a specific URL containing a malicious JavaScript payload (such as an <img> tag with an onerror function). When the victim clicks this link, the server fails to sanitize the input, reflecting the code directly into the HTTP response. The victim's browser then executes the embedded script immediately. The attack was successfully proven by executing an alert(1) command, confirming the feasibility of injecting any malicious code aimed at stealing session cookies or redirecting the victim to phishing sites.
Impact	The impact of this Reflected XSS vulnerability is classified as Medium to High due to its potential to severely compromise client-side security. While the immediate proof of concept only triggered an alert, the vulnerability enables an attacker to inject any client-side script into the victim's browser. This capability directly leads to critical issues such as Session Hijacking, where the attacker can steal the victim's session cookies and impersonate them. Furthermore, the vulnerability can be leveraged for highly effective phishing attacks by displaying malicious login forms within the context of the trusted website, or for coercive redirection, automatically sending the victim to a site containing malware or exploits. Ultimately, this flaw grants the attacker unauthorized access to the user's current session data and client-side functionality.
Evidence	**4.2.1.1 Executing JavaScript Code in the User's Browser as Part of a Reflected XSS Test.**  This visual evidence confirms the successful execution of the Reflected XSS attack's Proof of Concept (PoC). The activation of the injected payload is illustrated by a browser Alert Box displaying the number 1, which proves that JavaScript code execution via the URL was possible. After dismissing the alert, the screen shows the fixed text go to /?h#cc, thereby confirming that the malicious code was successfully passed through and reflected directly into the page's content.
Remediation	To patch this Reflected XSS vulnerability, Output Encoding must be implemented as the primary countermeasure. This requires processing all data taken from the URL query parameter and encoding it using a function like htmlspecialchars() in PHP before rendering it on the HTML page. This ensures that dangerous characters (< and >) are displayed as inert text rather than being executed as code. Additionally, it is recommended to enforce Input Validation to reject any unexpected or malicious characters from the input fields.

4.2.2 Vulnerable and Outdated Components: Use of End-of-Life JavaScript Libraries	
Severity	Medium
Domain	https://juice-shop.herokuapp.com/#/
Description	The application utilizes outdated, third-party software components or libraries that are known to contain documented security vulnerabilities. Using these components exposes the system to direct exploitation of publicly known flaws.
Impact	An attacker can leverage publicly known vulnerabilities (CVEs) in these outdated components to execute various attacks, including: 1. Cross-Site Scripting (XSS): If the library flaw allows client-side code injection. 2. Remote Code Execution (RCE): If the library is used server-side or a critical client-side vulnerability is exploited. 3. Data Exfiltration: Stealing sensitive user or session data. Successful exploitation can lead to full system compromise or breach of user data integrity.
Evidence	**4.2.2.1 Identification of Outdated Components via Wappalyzer**  This image displays the results from the Wappalyzer tool, confirming the application's use of vulnerable, outdated libraries and frameworks. Specifically, the tool identified JavaScript libraries such as jQuery 2.2.4 and Angular 20.2.3. These versions are legacy and known to have publicly documented security vulnerabilities (CVEs).
Remediation	Conduct a regular inventory of all third-party components and immediately update or replace any software that is outdated or contains known security issues. Required Actions: 1. Immediate Update: Update all third-party libraries to the latest stable and secure version. 2. Dependency Management: Implement an automated system (Dependency Scanning/Tracking) to continuously monitor components for new vulnerabilities (CVEs). 3. Removal: Remove any legacy or unused components to reduce the overall attack surface.

5. Summary and Environmental Assurance

This assessment confirmed that the overall security posture of the targeted applications is currently weak. A total of six findings were identified, including four High-Severity issues that led to full system compromise and administrative privilege escalation. Immediate remediation of all High-Severity findings is mandatory within 3 days to mitigate the severe risks of data breach and unauthorized access. Important Note: All assets tested are secure, intentionally vulnerable learning environments (Capture The Flag - CTF)

used exclusively for ethical hacking and personal development, and the test was performed strictly on dedicated, non-production training infrastructure.