**Report: SQL Injection Vulnerability Assessment**

**Author: Atharva Gopal Deshpande**

**Project:** Manual SQL Injection Testing

**Target:** http://testphp.vulnweb.com

**Date:** October 5, 2025

---

## 1. Introduction

This report documents a successful manual SQL injection attack performed on the login page of the designated test website, testphp.vulnweb.com. The objective of this exercise was to identify and exploit an SQL injection vulnerability to bypass the authentication mechanism and to outline effective countermeasures to prevent such attacks. All activities were conducted in a controlled, educational environment.

---

## 2. Vulnerability Details & Demonstration

The login form at

‘testphp.vulnweb.com/login.php’ was found to be vulnerable to authentication bypass via SQL injection. By inputting carefully crafted strings, known as payloads, into the username field, the backend SQL query was manipulated to always return a "true" condition, thus granting unauthorized access without a valid password. A typical, insecure query might look like this:

```
SELECT * FROM users WHERE username = '[USER_INPUT]' AND password
= '[PASSWORD_INPUT]';
```

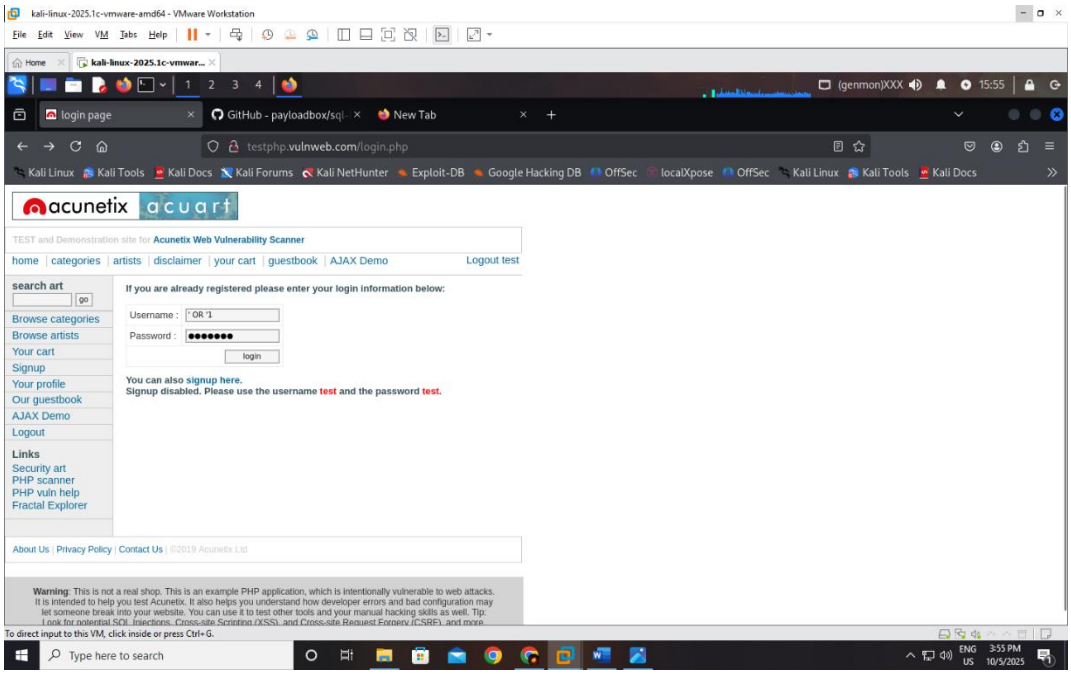
The goal of the attacker is to inject characters into the [USER_INPUT] field to change the logic of this query.

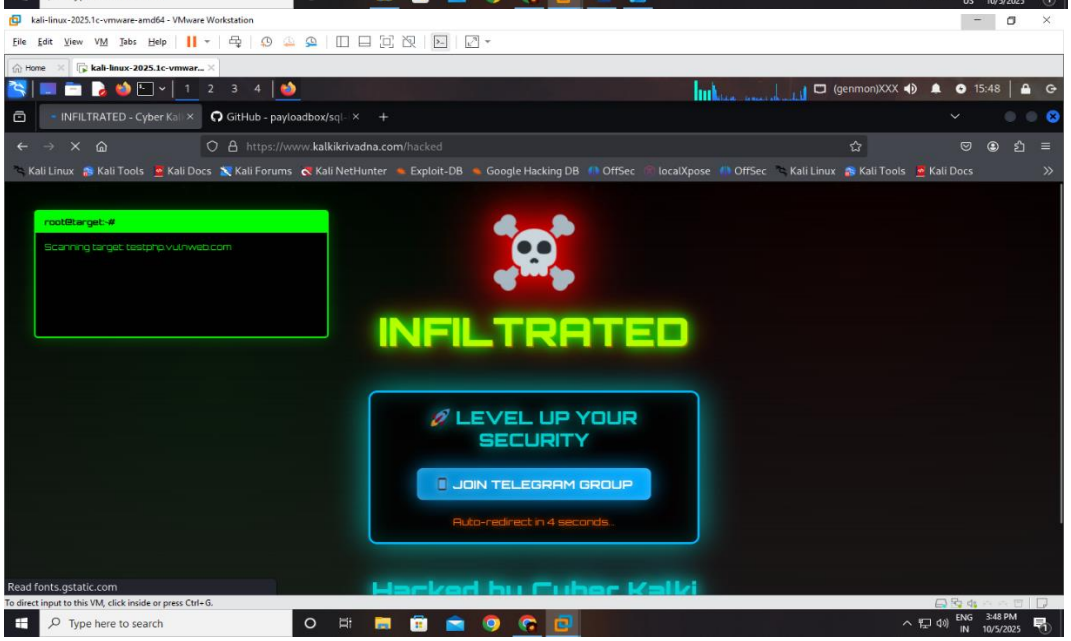
The following payloads were successfully tested, all resulting in a successful login, confirmed by the appearance of the "Logout" link on the subsequent page.

## 2.1 Generic SQL payloads

### ' OR '1

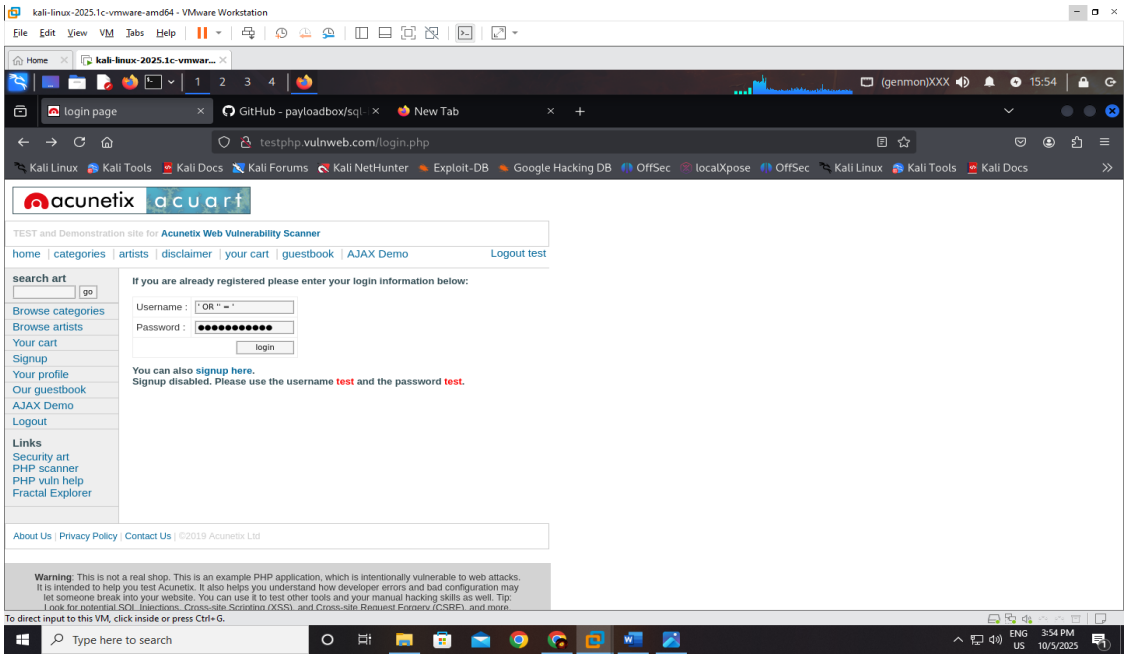
- Tries to break out of a string literal and append a boolean expression (a tautology).
- If concatenated into WHERE col = '…', it can turn the condition into something always true.
- Reveals the app is building SQL with raw string concatenation and not separating data from code.

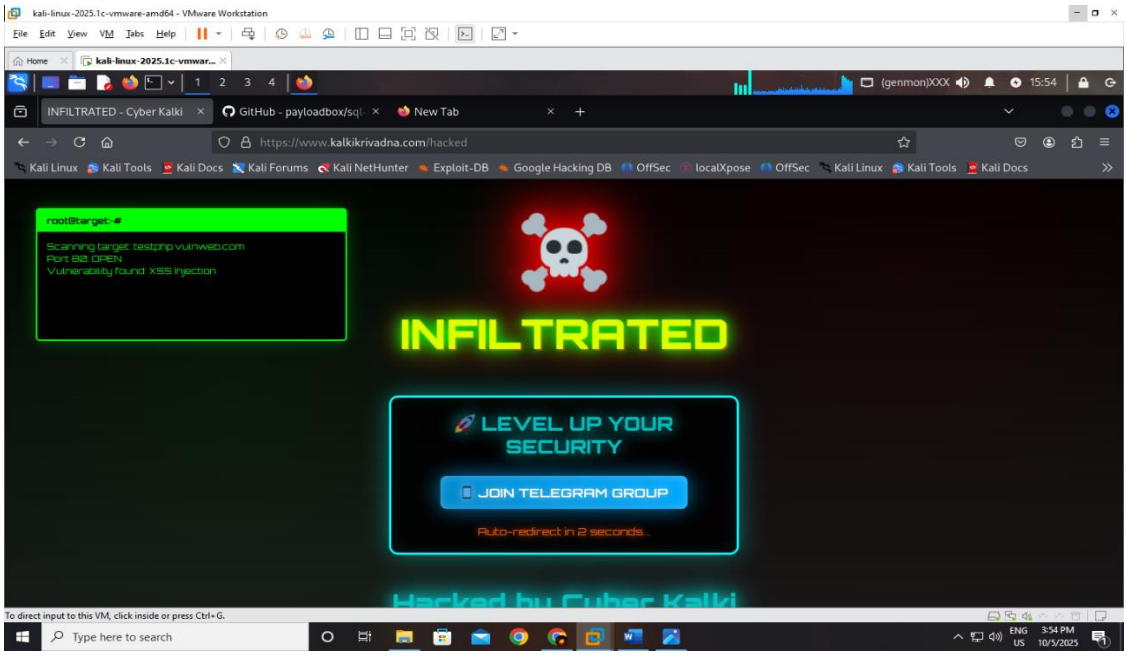


**' OR '' = '**

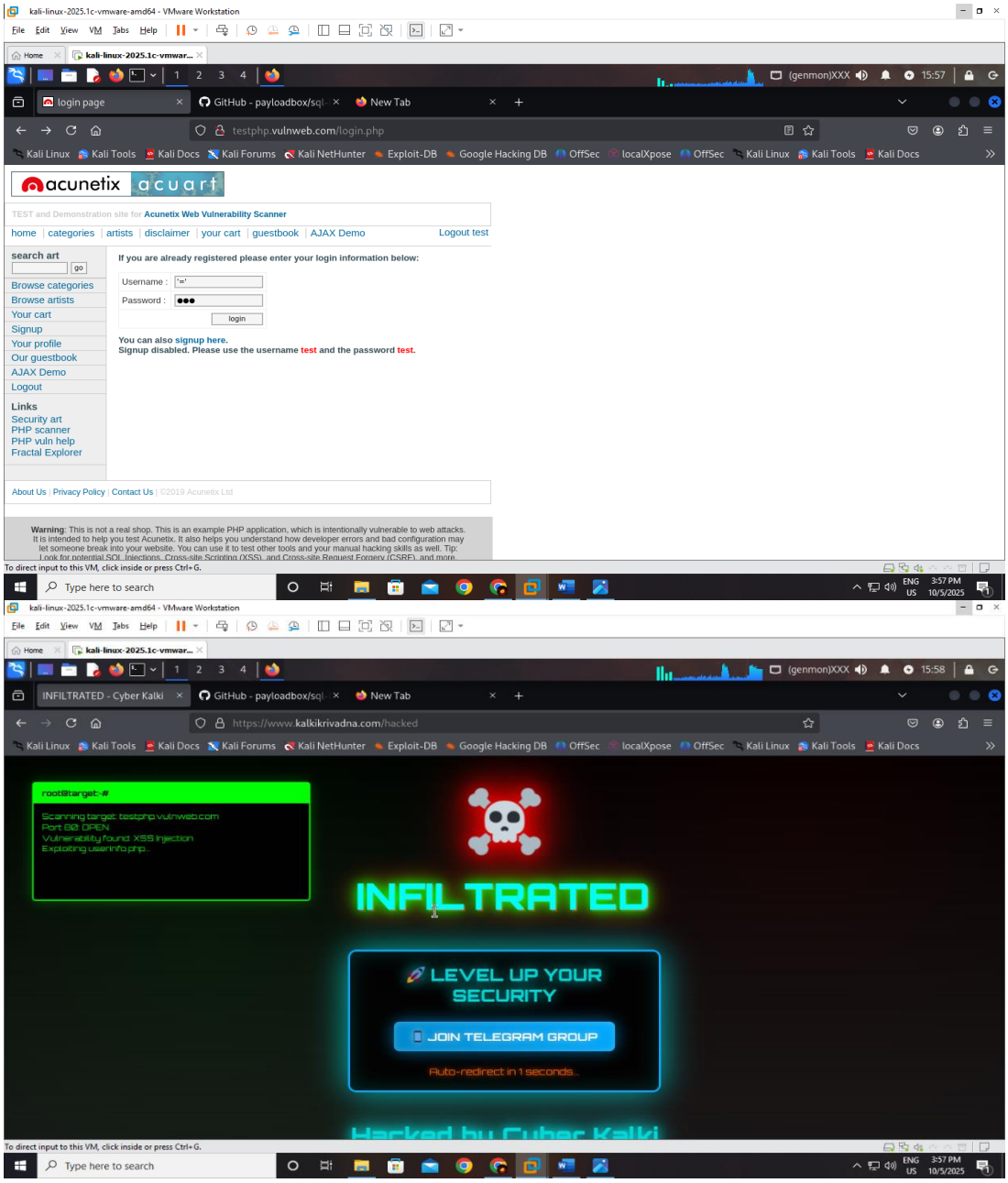
- Closes the original string, injects an equality comparing two empty strings (often a tautology depending on placement).
- Alters the WHERE logic so the condition can evaluate differently than intended.
- Indicates unsanitized input and lack of parameterization.



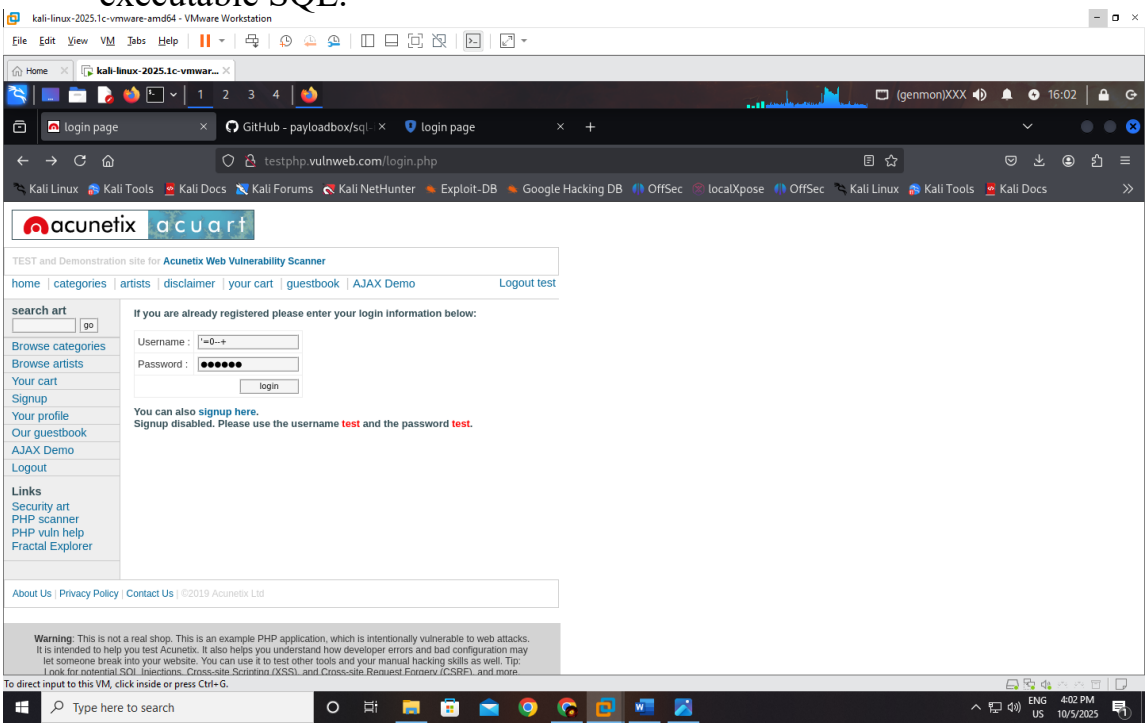
‘ = ‘

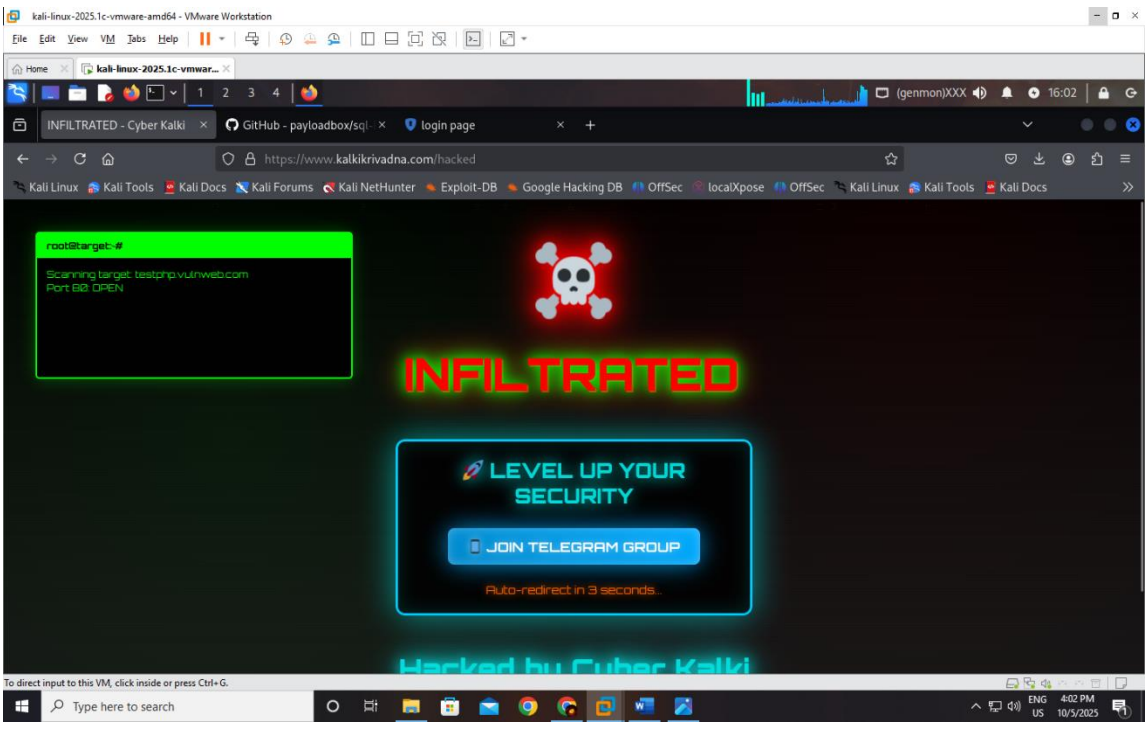
- Uses non-ASCII/typographic quotes to test whether the input is normalized or filters only plain ASCII quotes.
- If accepted, it can bypass naive quote-detection filters or reveal charset/normalization weaknesses.
- Reveals insufficient input normalization and naive filtering that assumes only ASCII characters.



**'=0--+**

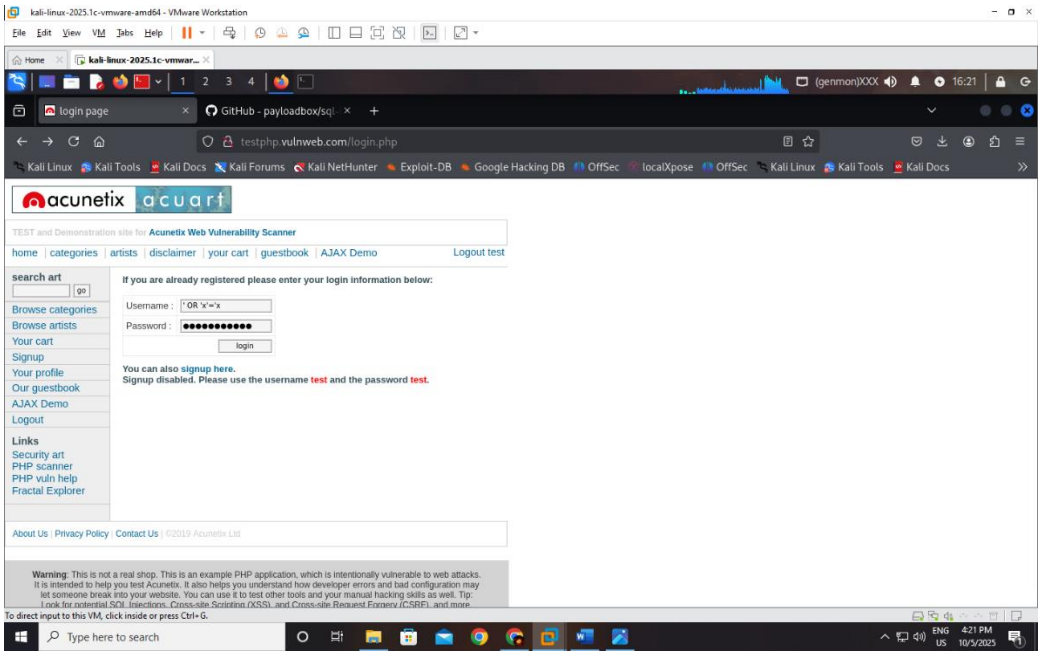
- Closes a string, adds a comparison (=0) and then a SQL comment sequence (-- with optional +) to truncate the remainder of the statement.
- Causes the rest of the intended query (e.g., extra conditions or trailing ') to be ignored, changing logic or preventing syntax errors.
- Reveals the application accepts comment tokens and concatenates input into executable SQL.

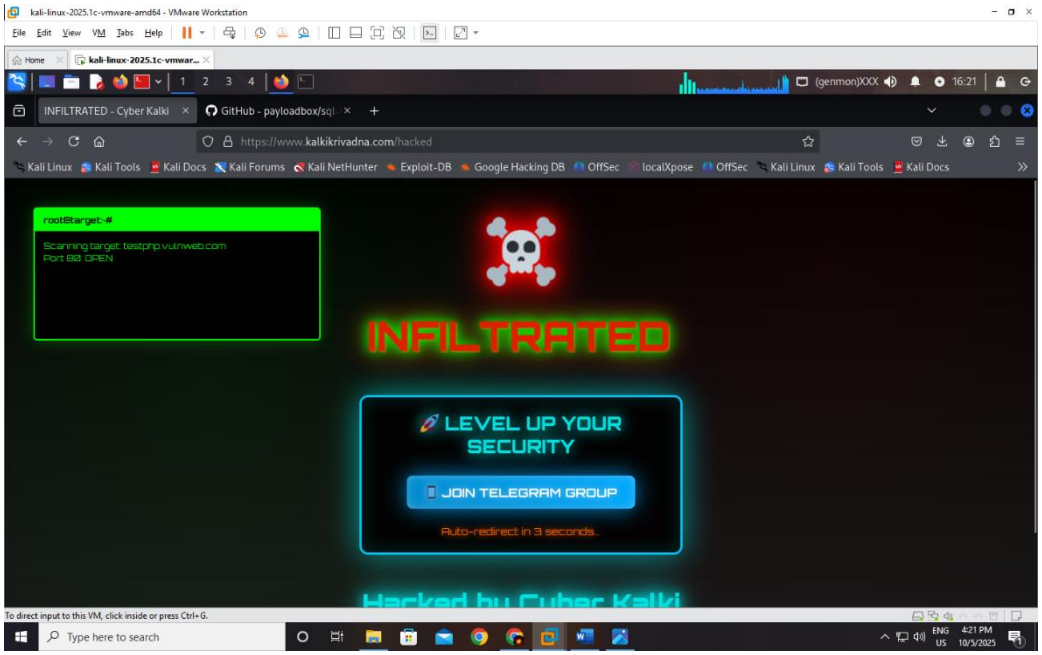


## ' OR 'x'='x

- Classic tautology: closes the literal and injects 'x'='x', which is always true.
- Converts a conditional WHERE into an always-true expression, potentially returning unrestricted rows or bypassing checks.
- Shows direct string concatenation into conditions and lack of prepared statements.

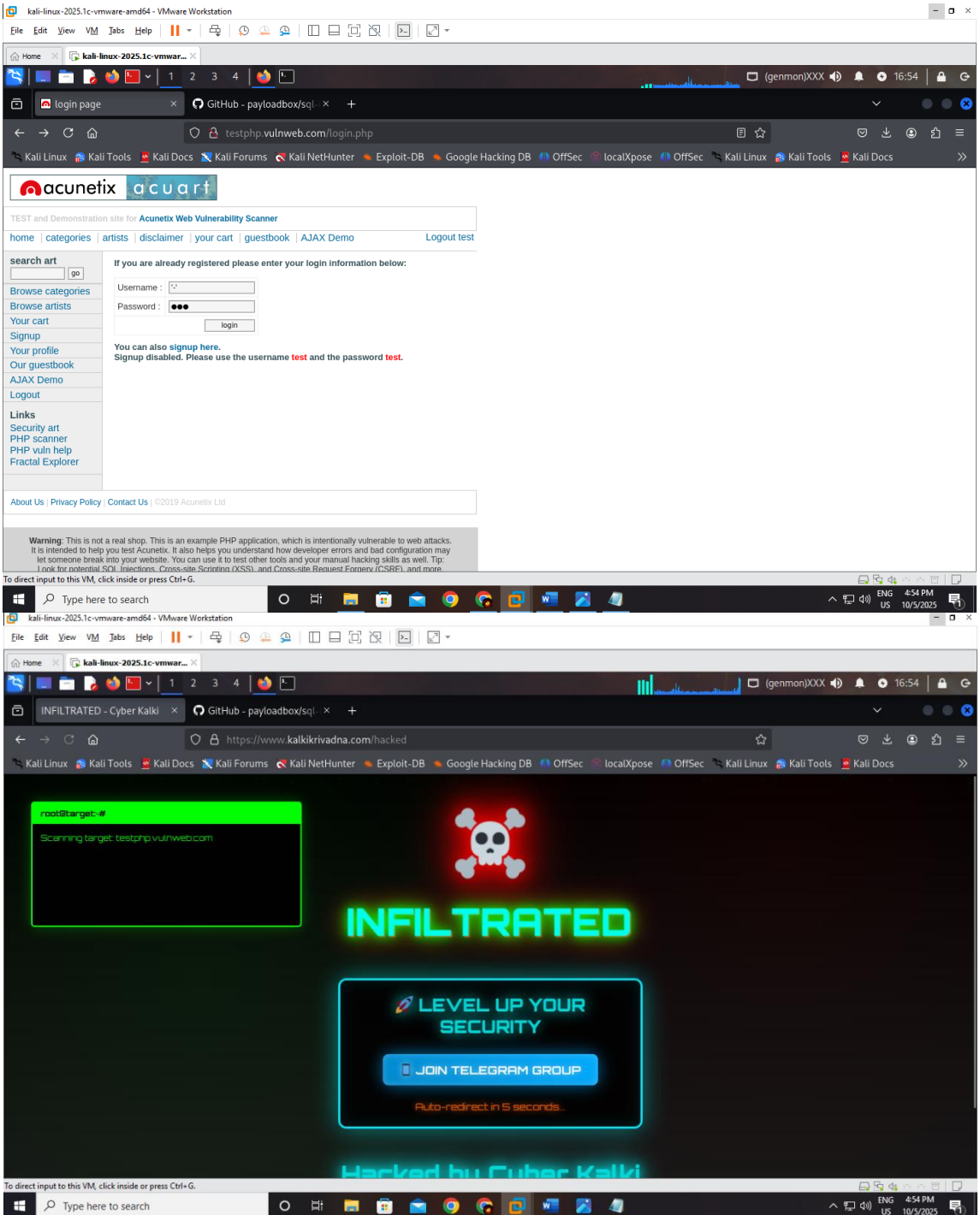


## 2.2 SQL Injection Auth Bypass Payloads

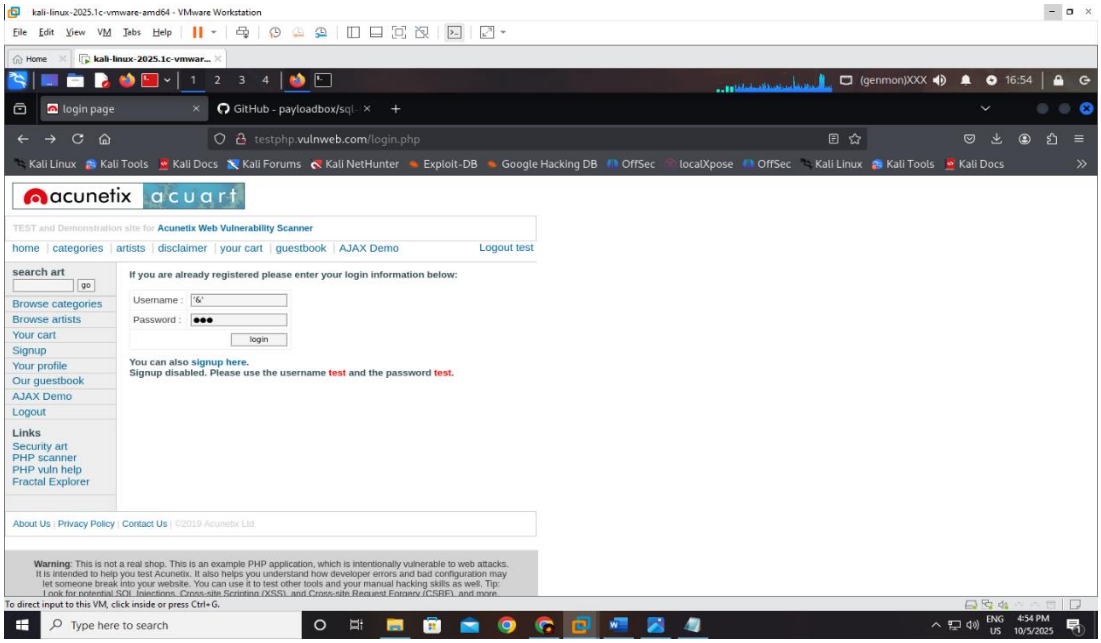
'_'

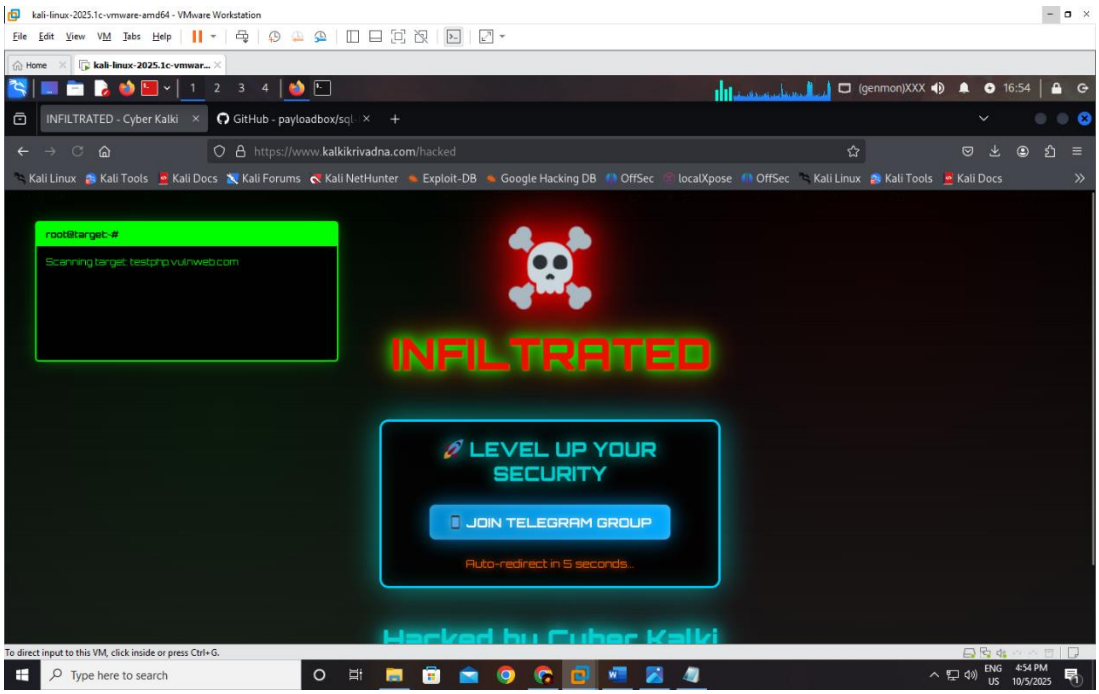
- A minimal attempt to close/alter a quoted string then add a leftover single-character condition; often used to probe how quotes are handled.
- Depending on placement it may create a syntax error or alter the query's string parsing.
- Useful diagnostically: indicates whether the app rejects, escapes, or accepts lone quote characters.



'&'

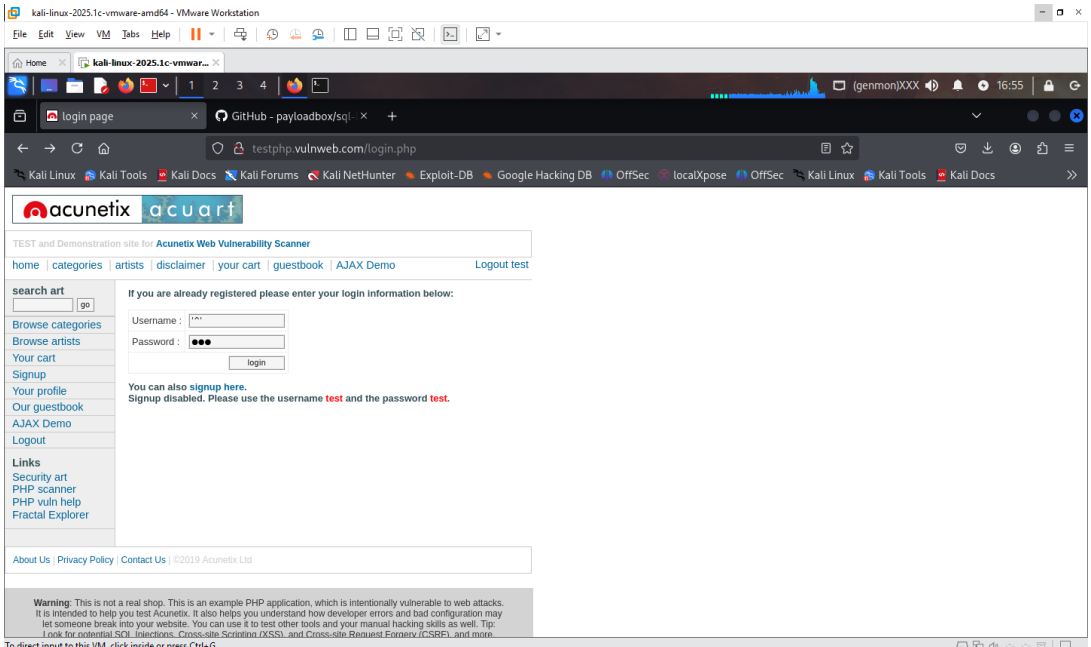
- Inserts an ampersand inside a string context to test how the DB or application treats special chars (sometimes used in URL contexts).
- Can reveal weaknesses in input encoding/escaping or how the app forwards input to the DB.
- Shows whether the app properly encodes or normalizes special characters from user input.

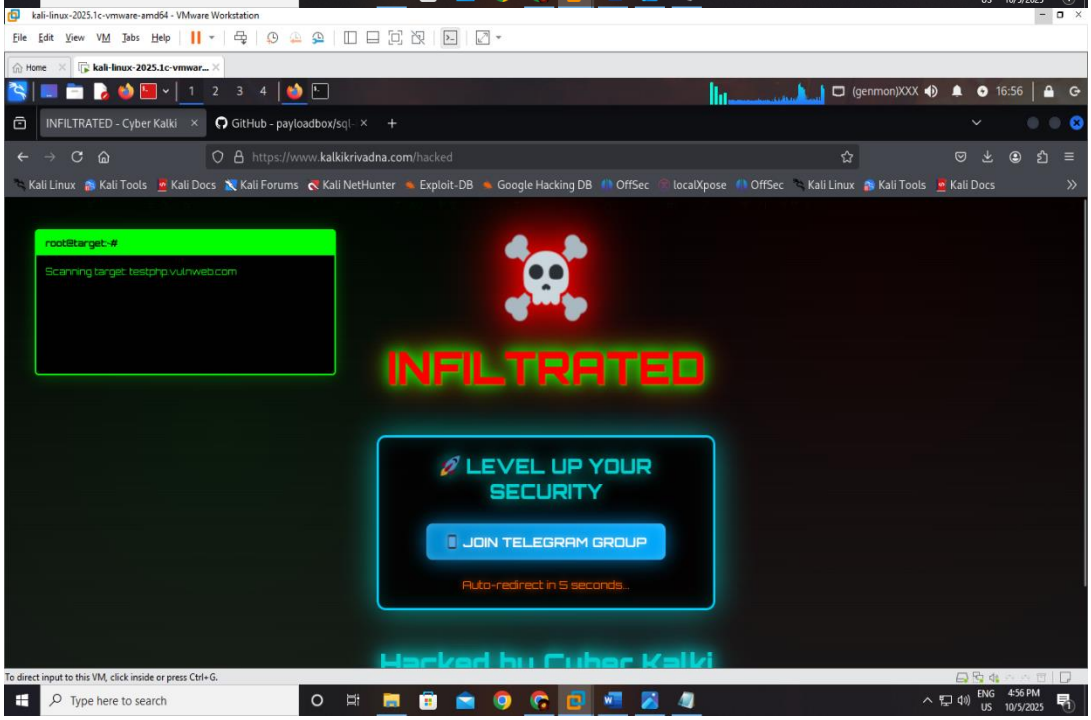


'^'

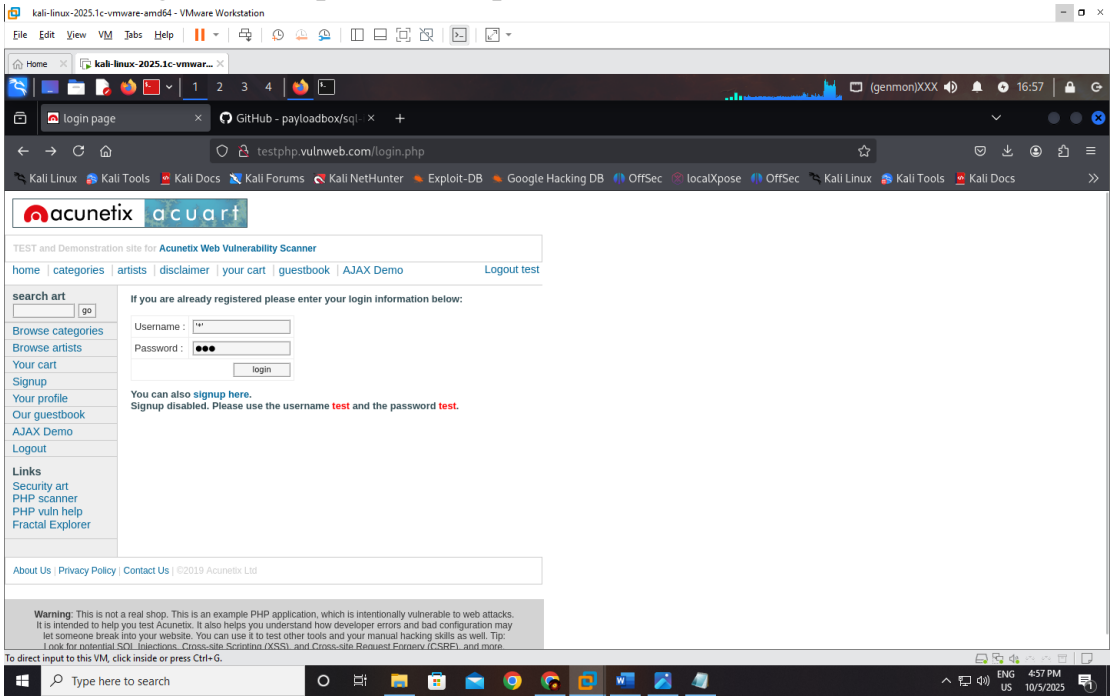
- A single caret tests parsing/escaping behavior; in some DB dialects it’s an operator or just a character.
- If it changes behavior, it signals insufficient escaping or assumptions about allowed character sets.
- Reveals brittle input handling or incorrect assumptions about allowed tokens.

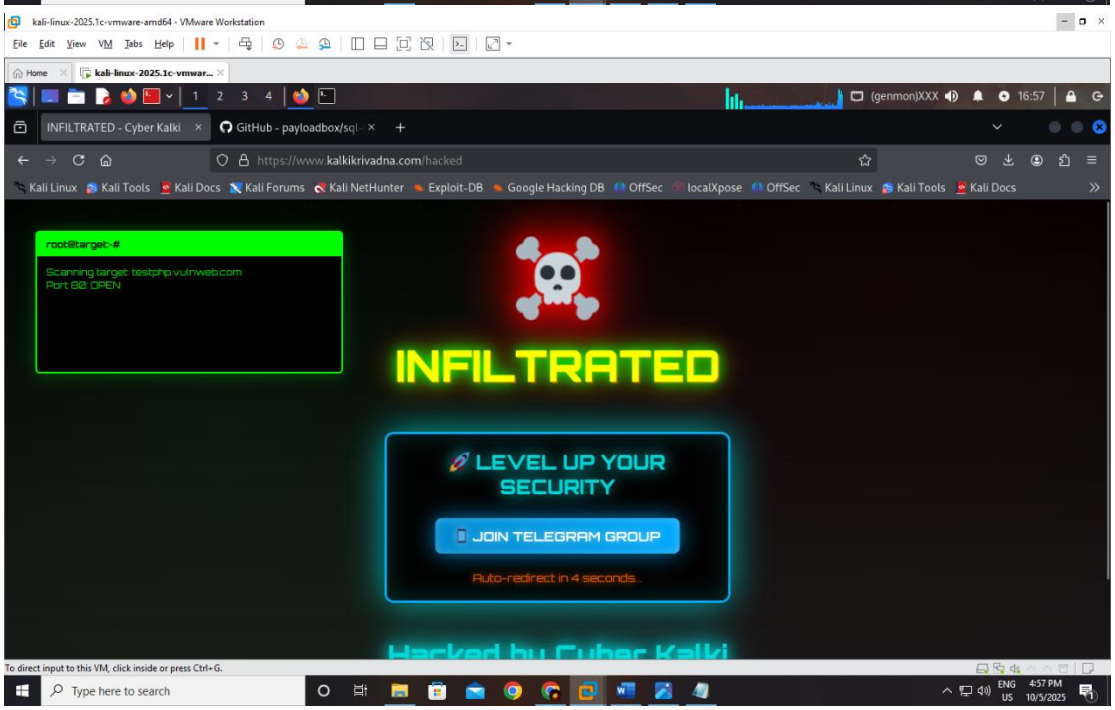


'*'

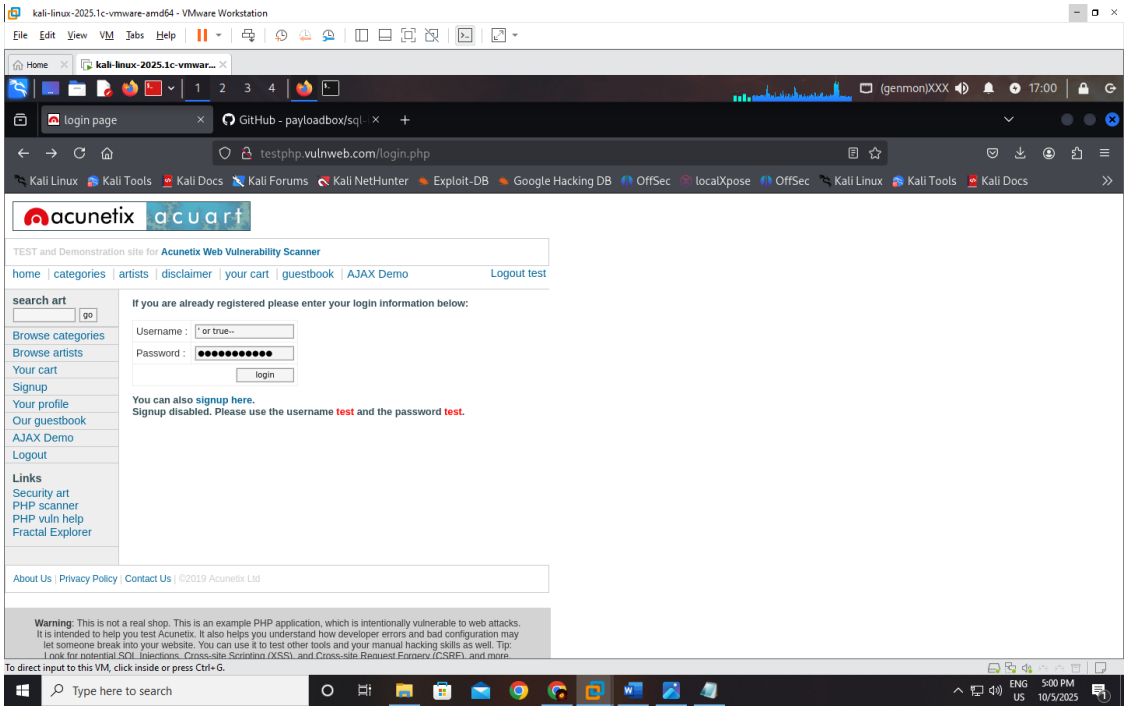
- Asterisk inside a string or expression probes whether wildcard or operator characters are being interpreted or filtered.
- May show whether input gets used in contexts that treat * specially (like SELECT *) or whether it's sanitized.
- Diagnostic: exposes lax input validation for meta-characters.

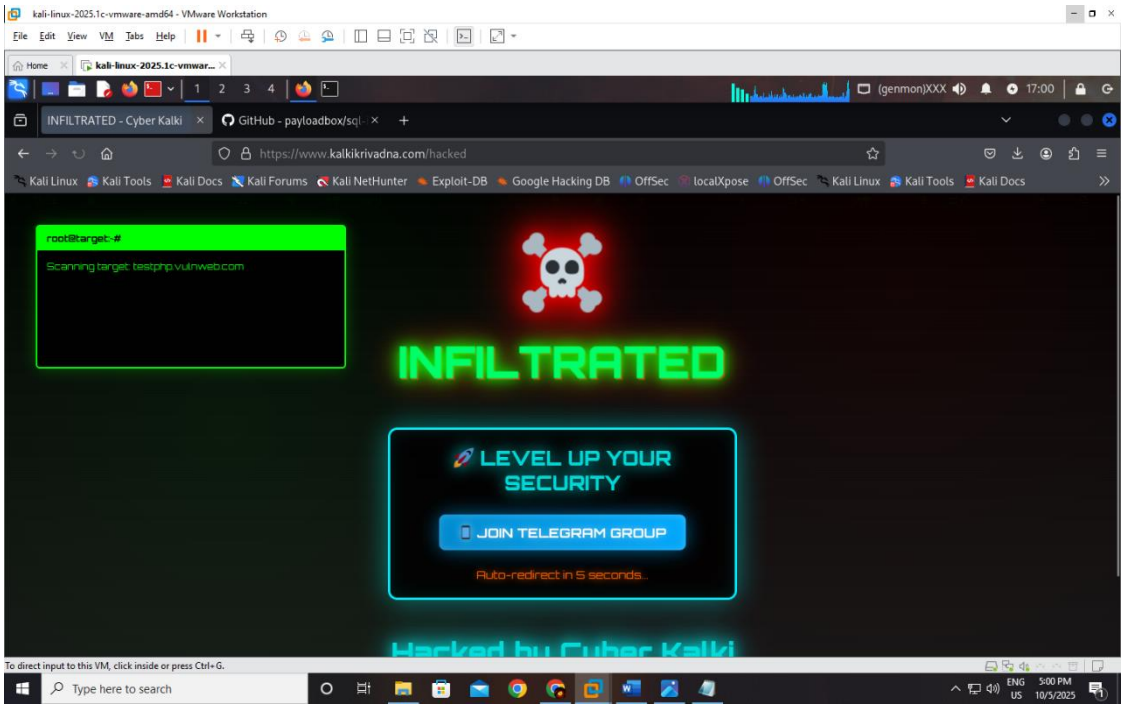


**' or true--**

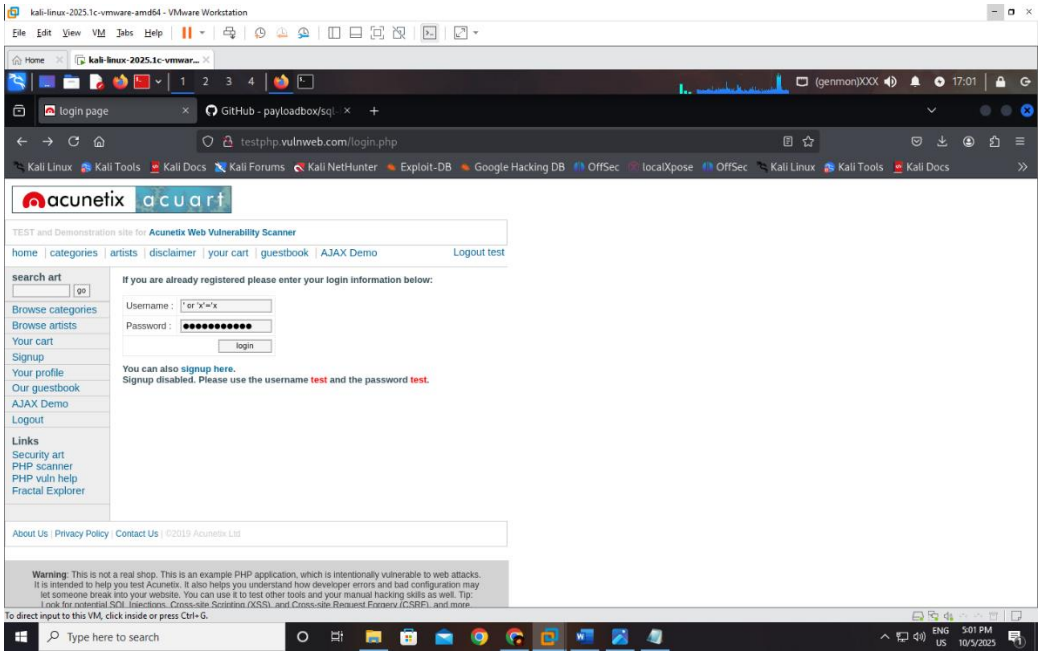
- Attempts to inject a boolean true after closing a string, often followed by an em-dash or long dash used to terminate/comment the rest.
- Changes the condition to always-true and attempts to truncate trailing syntax.
- Reveals both concatenation into boolean logic and tolerance for comment/terminator tokens.

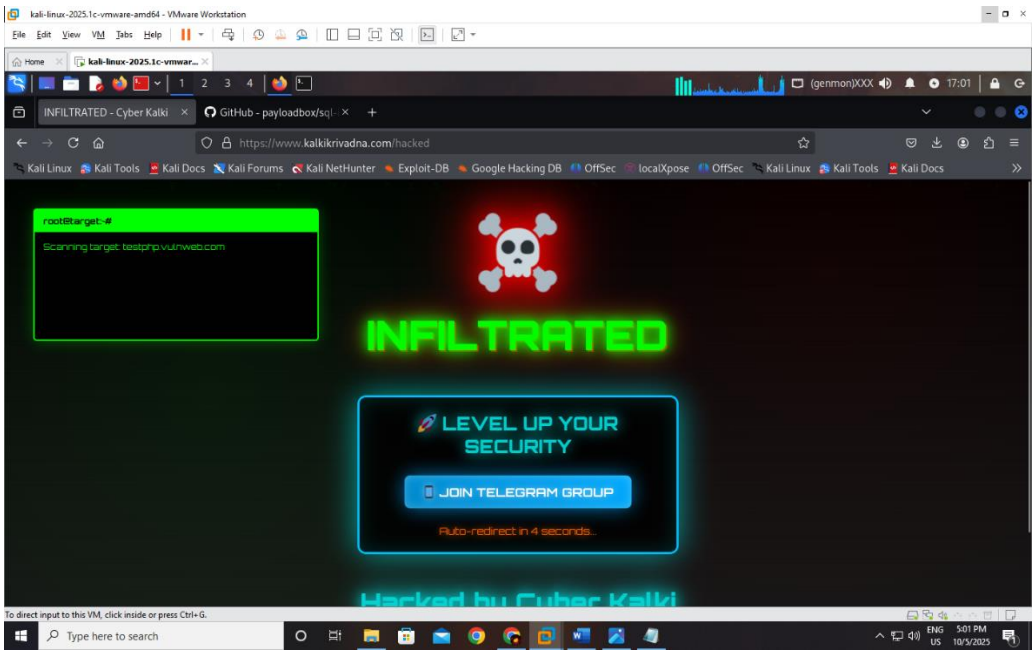


**' or 'x'='x**

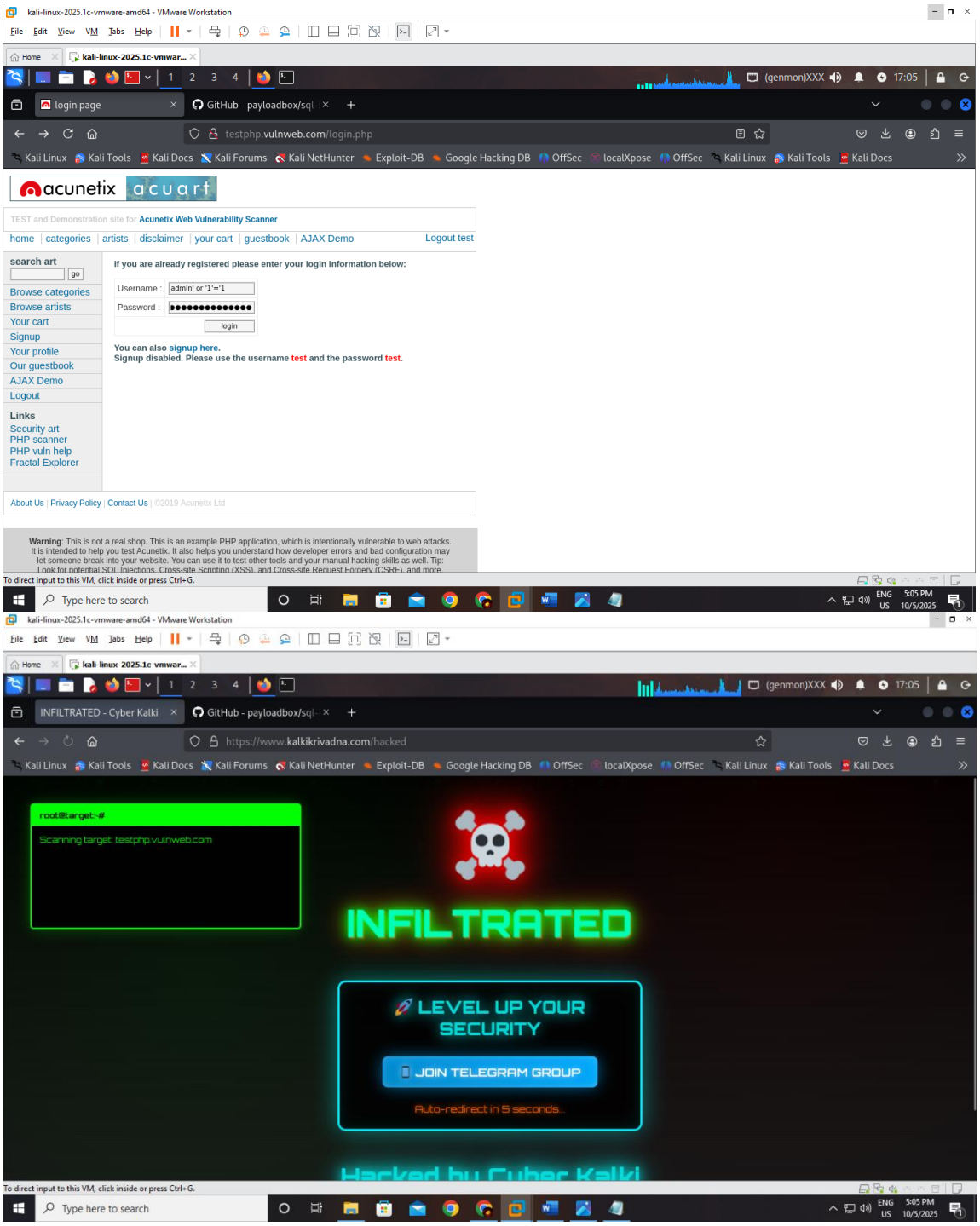
- Same as earlier tautology pattern targeted at authentication checks (username = '…').
- If accepted, it can make authentication WHERE conditions evaluate true regardless of supplied credentials.
- Indicates the authentication logic directly injects raw input into SQL.



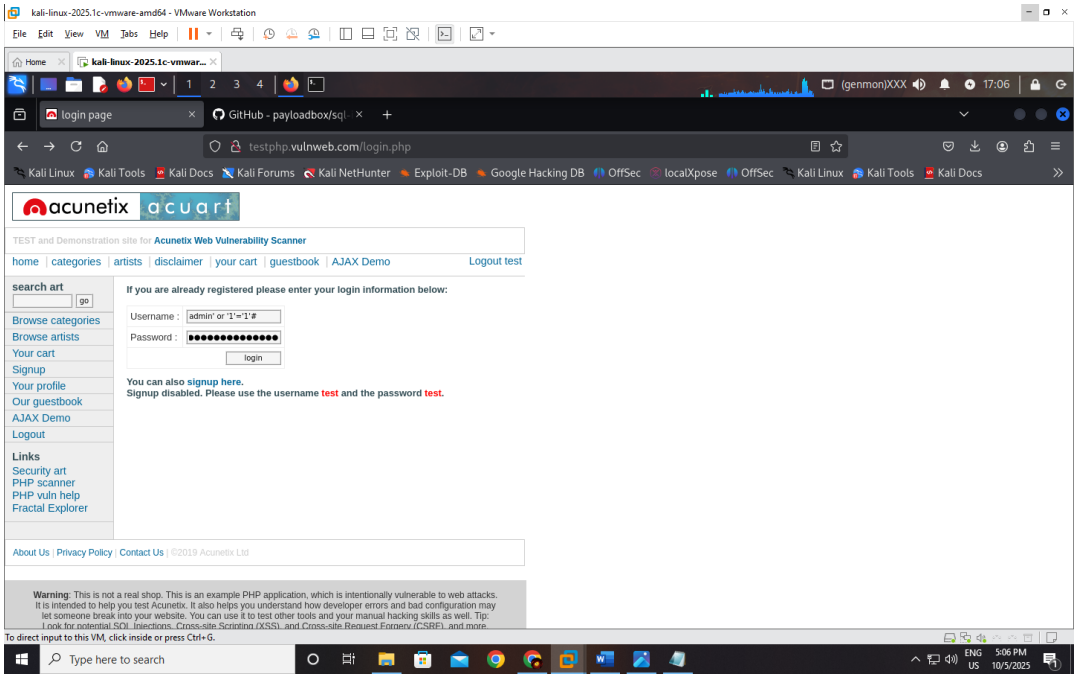
**admin' or '1'='1**

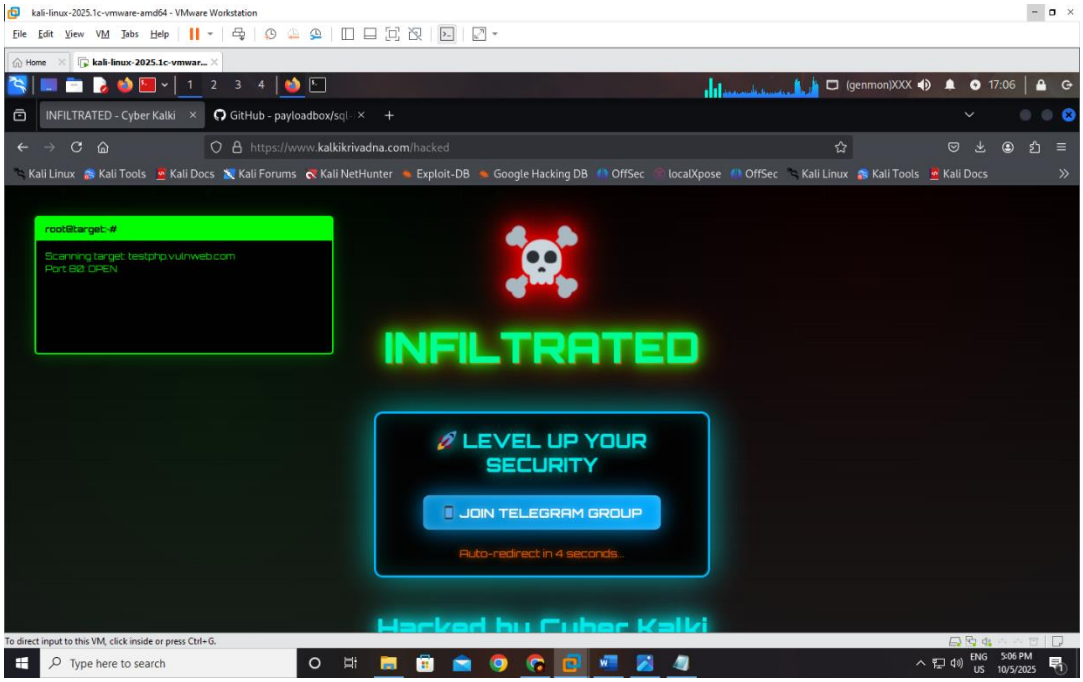
- Uses a known username (admin) then injects a tautology to bypass password checks when input is concatenated into something like username = '…' AND password = '…'.
- Alters the combined condition so the username check passes or the overall expression becomes true.
- Reveals that login fields are concatenated into SQL and not parameterized.



**admin' or '1'='1'#**

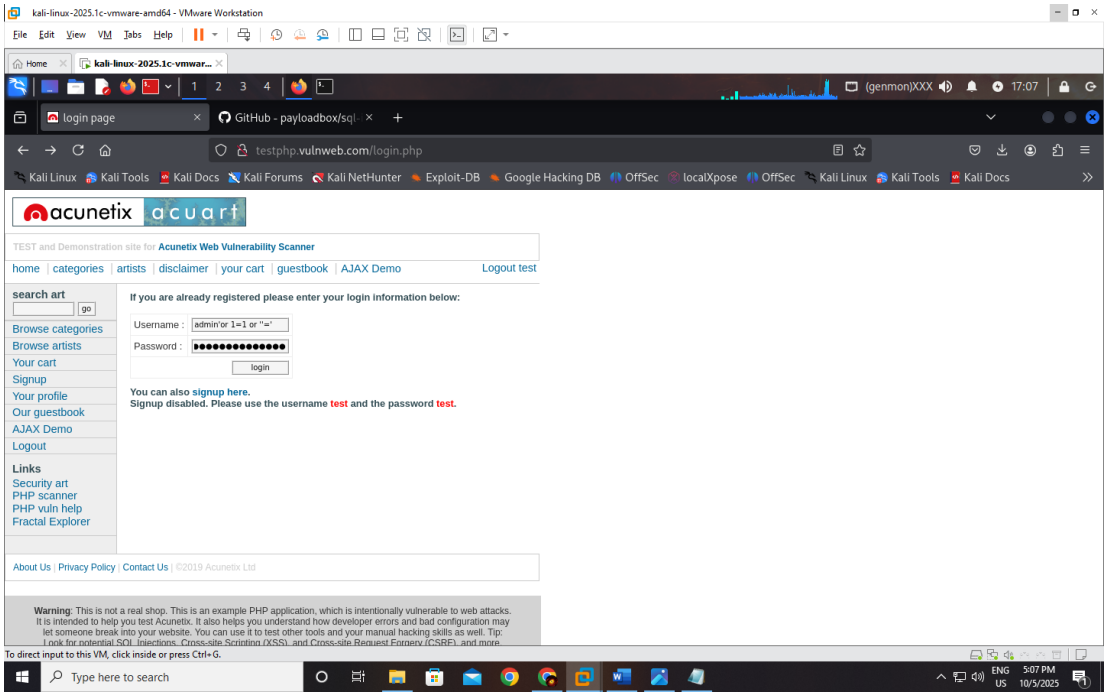
- Same as above but appends #, a comment marker in some SQL dialects, to discard the rest of the query (for example, the password condition).
- Truncates remainder of the statement so only the injected condition is evaluated.
- Indicates acceptance of comment characters and raw concatenation of credentials into SQL.

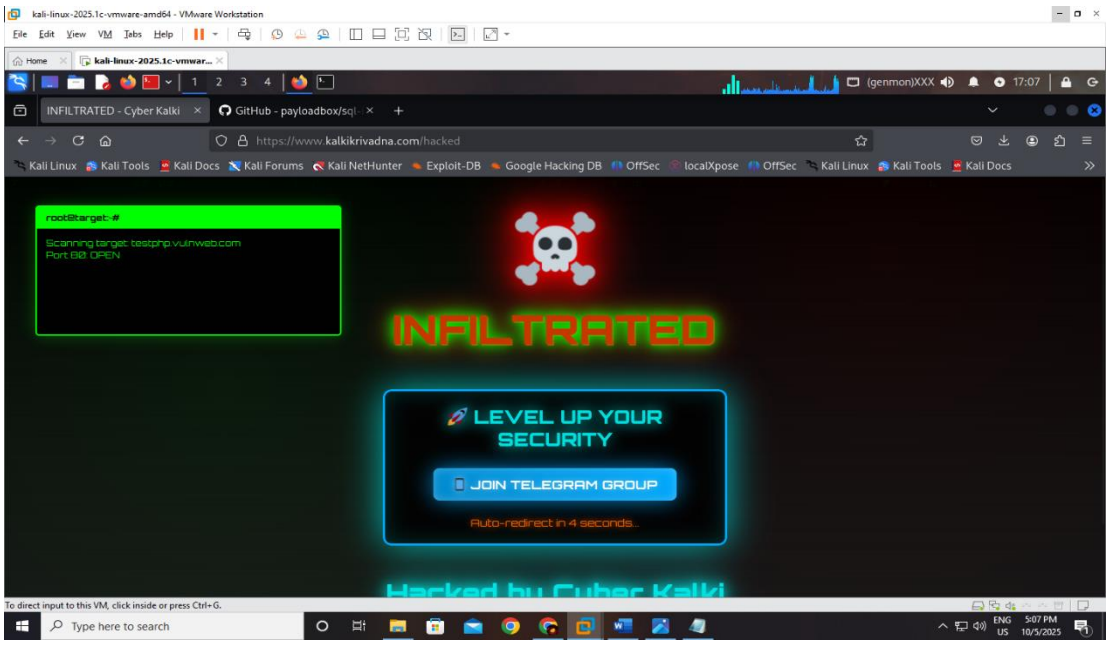


**admin'or 1=1 or ''='**

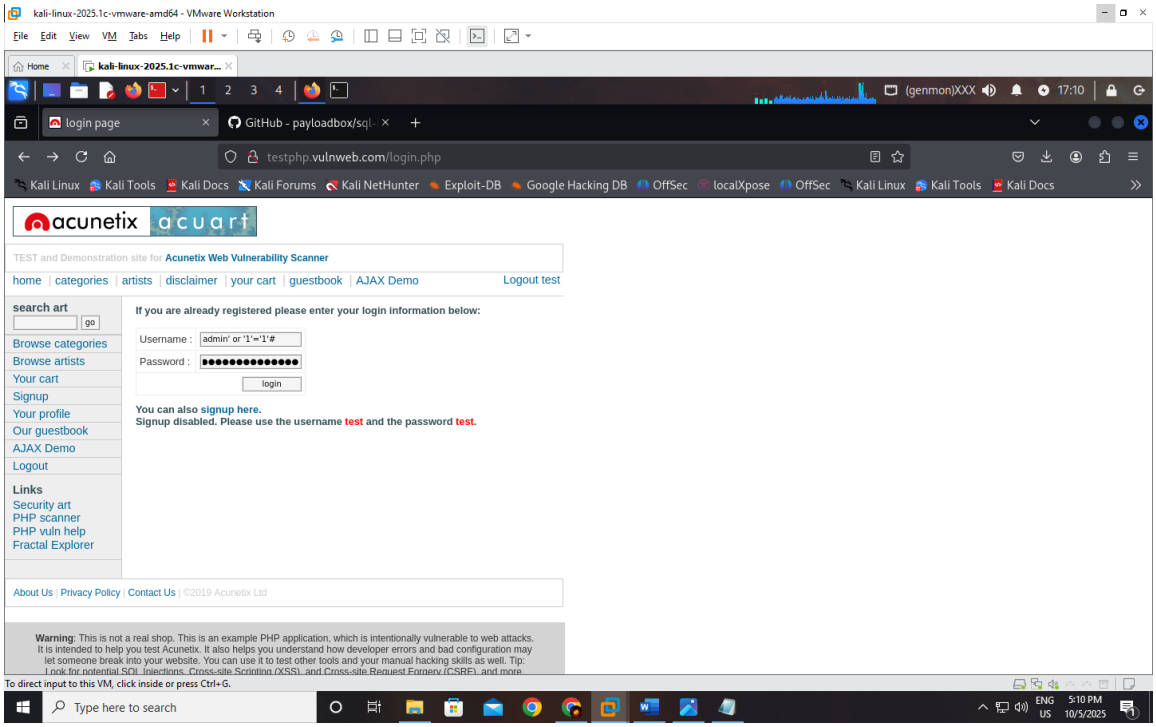
- Combines an identifier (admin) with numeric tautology 1=1 and an empty-string equality — multiple attempts to force a true condition across different parsing contexts.
- If any of the appended conditions succeed, the WHERE may become true and bypass checks.
- Reveals the app does not reliably sanitize or use typed parameters for credential checks.

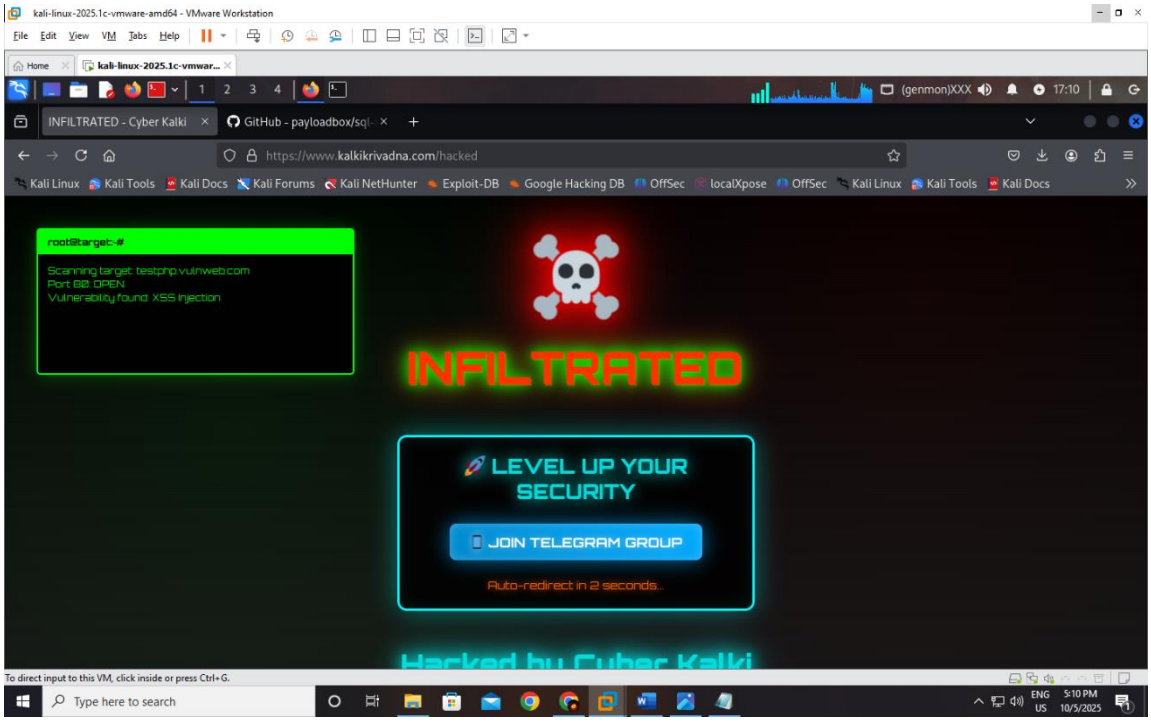


**admin' or '1'='1'#**

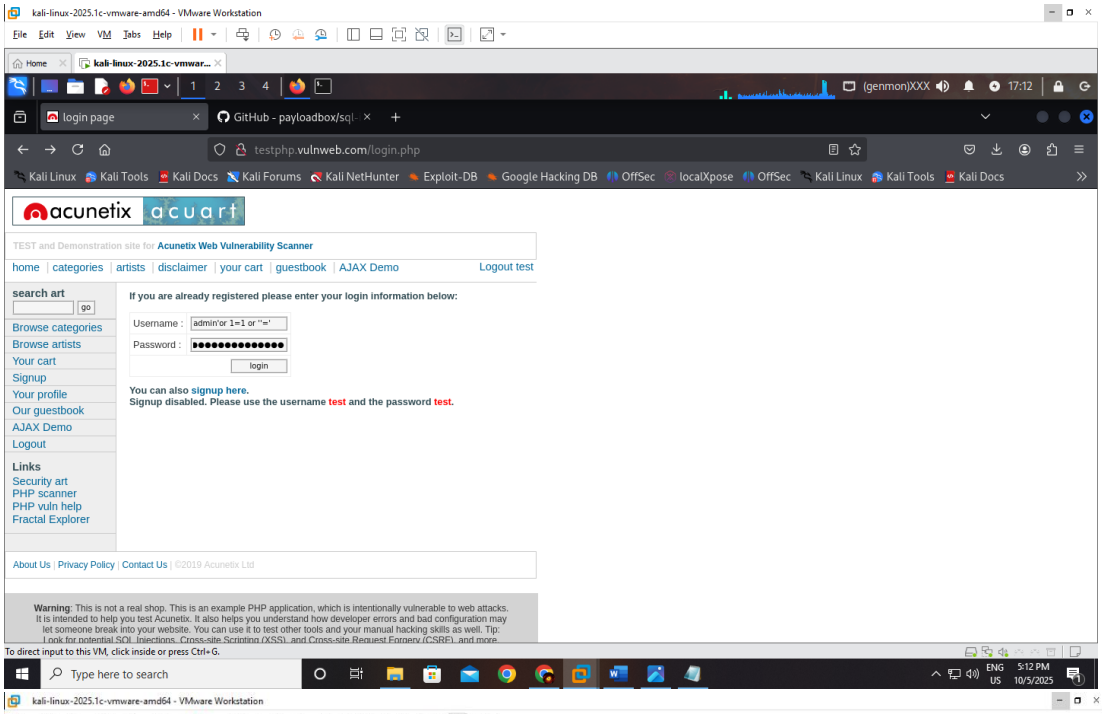
- Closes the username, injects 1=1 (tautology) and uses # to comment out the rest.
- Attempts to make the authentication conditional true while removing subsequent constraints.
- Diagnostic signal that concatenation and comment tokens are not blocked.

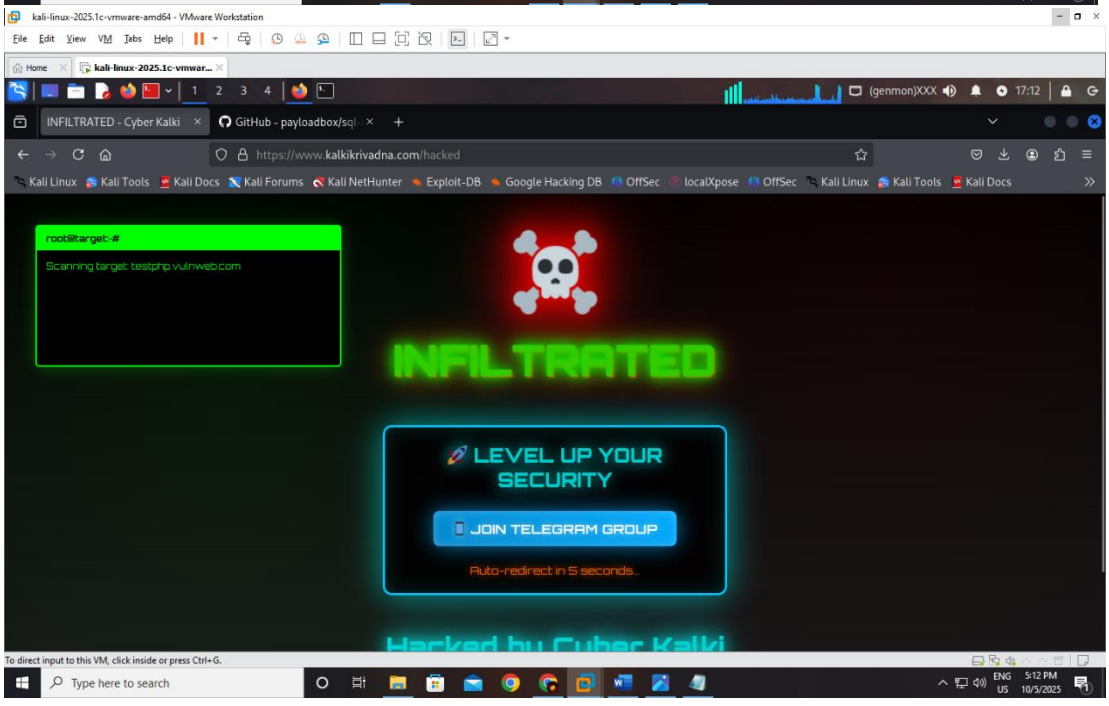


**admin'or 1=1 or ''='**

- Variation mixing numeric tautology and empty-string equality, sometimes included to evade simple filters or to adjust whitespace sensitivity.
- If accepted, it again tries to force an always-true condition or confuse naive sanitizers.
- Reveals fragile or pattern-based filtering instead of true parameterization.



---

## 3. Impact Analysis

An SQL Injection (SQLi) vulnerability is a critical security flaw that acts as a gateway for escalating attacks, leading to severe and widespread damage. The impact extends far beyond the initial breach, often resulting in a full compromise of the application, its data, and the underlying server. The consequences can be broken down into three primary stages.

### Unauthorized Access and Privilege Escalation

The most immediate outcome of a successful SQLi attack is gaining unauthorized access to the application, bypassing all authentication controls.

- **Access to Sensitive Data:** An attacker can impersonate legitimate users, gaining access to their personal and confidential information, including names, emails, financial details, and private messages. This leads directly to privacy violations and potential identity theft.
- **Administrative Control:** If an administrator account is compromised, the attacker gains full control over the application. They can manage users, alter content, disable security features, and effectively operate as a legitimate administrator, locking out the actual owners.

### Database Compromise: Theft, Tampering, and Destruction

The vulnerability provides a direct line to the database, allowing an attacker to read, modify, or delete any information stored within it.

- **Data Theft (Exfiltration):** Using advanced injection techniques, an attacker can steal entire databases. This includes sensitive customer lists, user credentials (password hashes), intellectual property, and financial records, which are often sold or leaked online.
- **Data Tampering (Modification):** The attacker can maliciously alter existing data. This could involve changing prices in an e-commerce store, modifying user records to commit fraud, or tampering with logs to hide their activity.

- **Data Destruction:** In the most damaging scenario, an attacker can execute commands to permanently delete data (e.g., DROP TABLE). This results in catastrophic, irreversible data loss, which can instantly halt all business operations.

**Full Server and System Takeover**

The attack can escalate beyond the application and database to a complete takeover of the web server itself.

- **Remote Code Execution:** Many database services can be exploited to run commands on the server's operating system. This allows an attacker to upload malicious files, such as a web shell, to gain persistent remote control.
- **System-Wide Compromise:** With control of the server, an attacker can install ransomware, use the server to mine cryptocurrency, or deploy other malware. They can also use the compromised server as a pivot point to attack other secure systems within the organization’s internal network, deepening the breach.

---

## 4. Prevention and Mitigation Strategies

Preventing SQL injection is crucial for application security. The following methods are highly effective and should be implemented.

### 4.1. Use Prepared Statements (Parameterized Queries)

This is the **most effective** way to prevent SQL injection. Instead of mixing user input with the SQL code, you send the query template to the database first and then send the user's input as separate parameters. The database treats the user input as pure data, never as executable code.

- **Vulnerable Code (Example in PHP):**

PHP

```
// User input is directly stitched into the SQL query string
$username = $_POST['username'];
$sql = "SELECT * FROM users WHERE username = '$username'";
```

- **Secure Code (Example in PHP using PDO):**

PHP

```
// 1. PREPARE: Send the command template with a placeholder
$stmt = $pdo->prepare('SELECT * FROM users WHERE username = :username');
// 2. EXECUTE: Send the user's input as a separate parameter
$stmt->execute(['username' => $_POST['username']]);
```

### 4.2. Input Validation and Sanitization

Always treat user input as untrusted.

- **Validation:** Check that the input matches the expected format (e.g., an email address should look like an email address, a date should be a valid date). If it doesn't, reject it.

- **Sanitization:** If you absolutely cannot use prepared statements, you must "escape" special characters in the user input. This process adds a backslash (\) before characters like ', ", and \, telling the database to treat them as literal characters, not as special SQL operators.

### 4.3. Implement the Principle of Least Privilege

The database user account that the web application uses should only have the absolute minimum permissions it needs to function. For example, a user account for a login page only needs SELECT permissions on the users table. It should not have permission to DROP tables or modify data in other parts of the database. This limits the damage an attacker can do if they manage to successfully exploit a vulnerability.

---

## 5. Conclusion

The successful penetration of testphp.vulnweb.com serves as an unambiguous demonstration of a **critical security failure**. The ease and speed with which the login mechanism was completely bypassed underscores the catastrophic potential inherent in SQL injection vulnerabilities. What was a straightforward exercise in this test environment would equate to a devastating, large-scale data breach on a live application.

This assessment makes it clear that failing to implement secure coding practices leads to fundamentally insecure software. Therefore, the adoption of **prepared statements** is not merely a suggestion—it is the **non-negotiable standard** for modern web development. Protecting an application from this type of attack is essential, and it begins with treating secure coding as a foundational requirement, not an afterthought.

---

## 6. Attachment

https://drive.google.com/file/d/1mp0oKSWbm4wRg9PLvA4TwWIl2oSuMU73/view?usp=sharing