## **1.Brute Force Attack**

A brute force attack is a hacking method that uses trial and error to crack passwords, [login credentials](https://www.fortinet.com/resources/cyberglossary/login-credentials), and encryption keys. It is a simple yet reliable tactic for gaining unauthorized access to individual accounts and organizations’ systems and networks. The hacker tries multiple usernames and passwords, often using a computer to test a wide range of combinations, until they find the correct login information.

The name "brute force" comes from attackers using excessively forceful attempts to gain access to user accounts. Despite being an old cyberattack method, brute force attacks are tried and tested and remain a popular tactic with hackers.

### ****Tool Used:**** Burp Suite

**URL Tested:** http://localhost/DVWA/vulnerabilities/brute/

**Method:** POST

**Login Parameters:**

username: test input from user.txt

password: test input from pass.txt

**Payloads Used:**

Usernames from user.txt

Passwords from pass.txt

Additional custom payloads can be added using Burp Intruder’s “Payload Positions” tab

**Procedure:**

Captured login POST request via Burp Suite Proxy

Sent the request to **Intruder**.

Set payload positions on username and password fields.

Loaded user.txt and pass.txt as payload sources.

Launched the attack.

Identified successful login based on **response length/status** changes.

**Indicators of Success:**

Status code: 302 Redirect

**Mitigation Recommendations:**

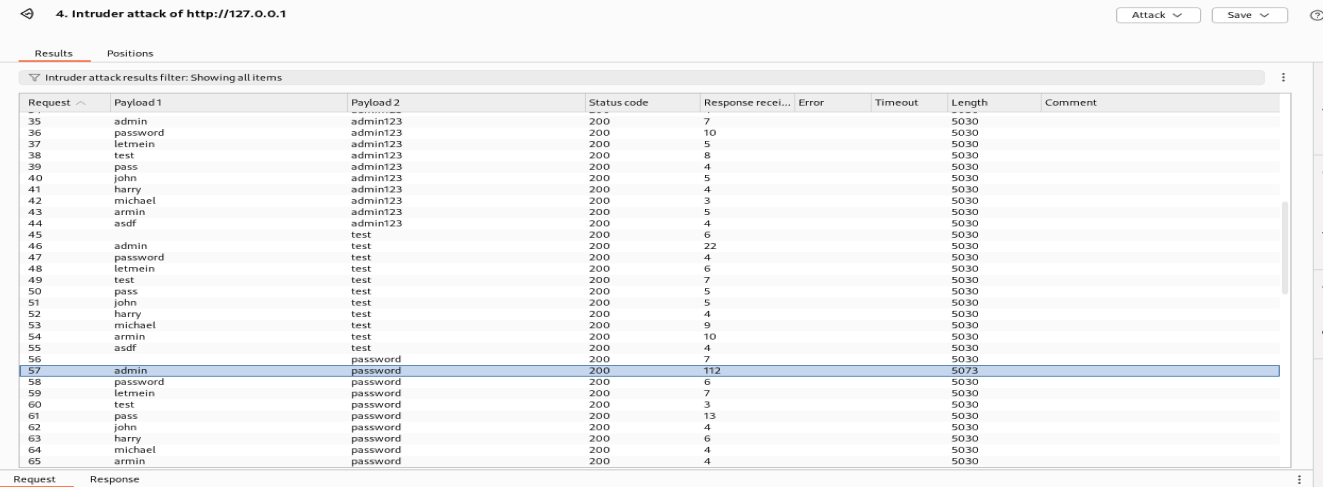
Implement **account lockout** or **rate limiting**

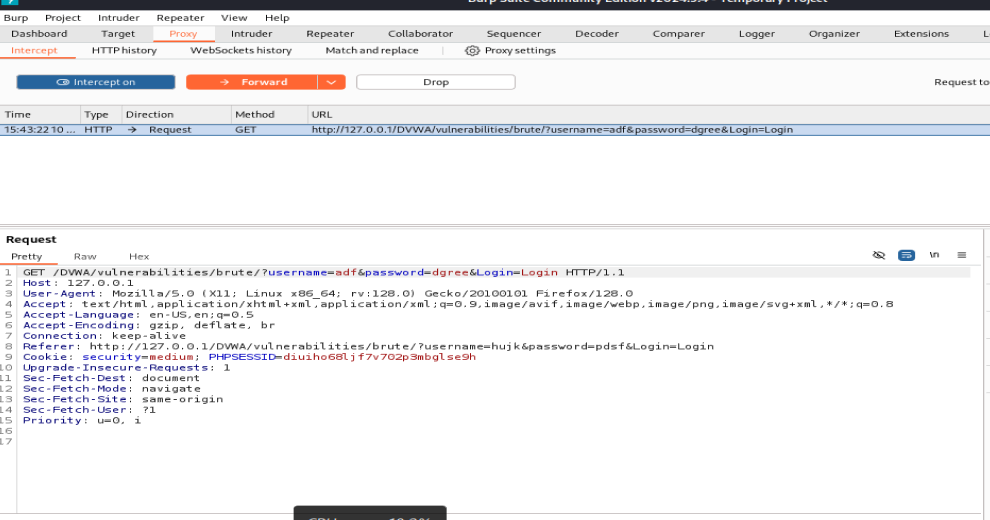
Use **CAPTCHAs** to prevent automation

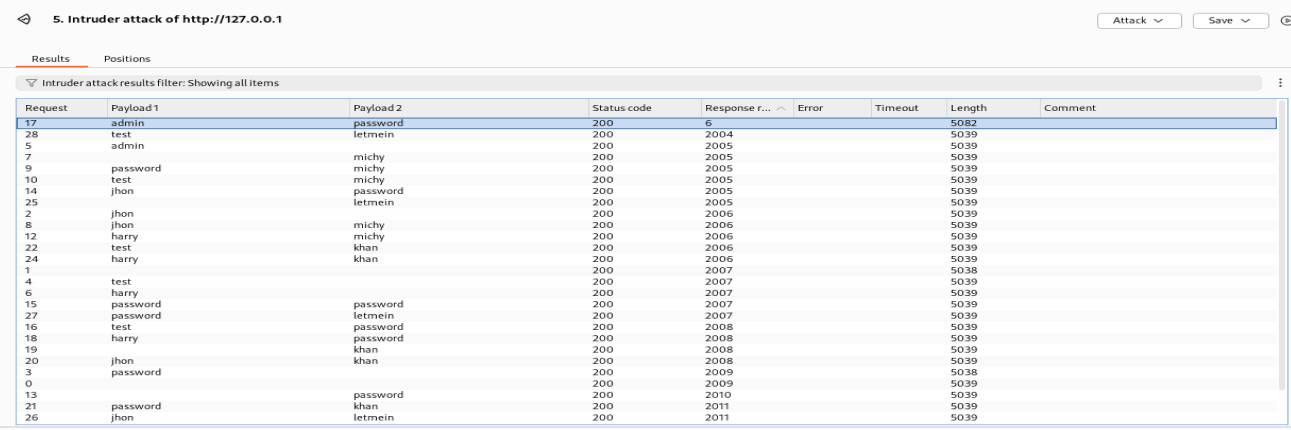
Monitor and log authentication attempts

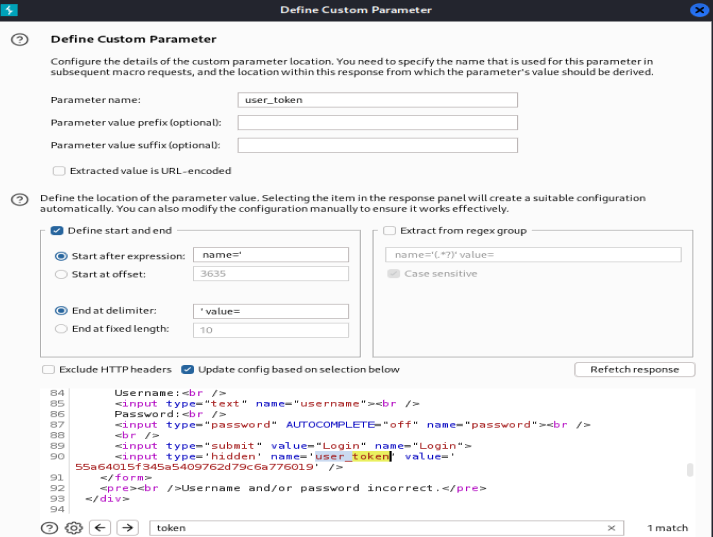
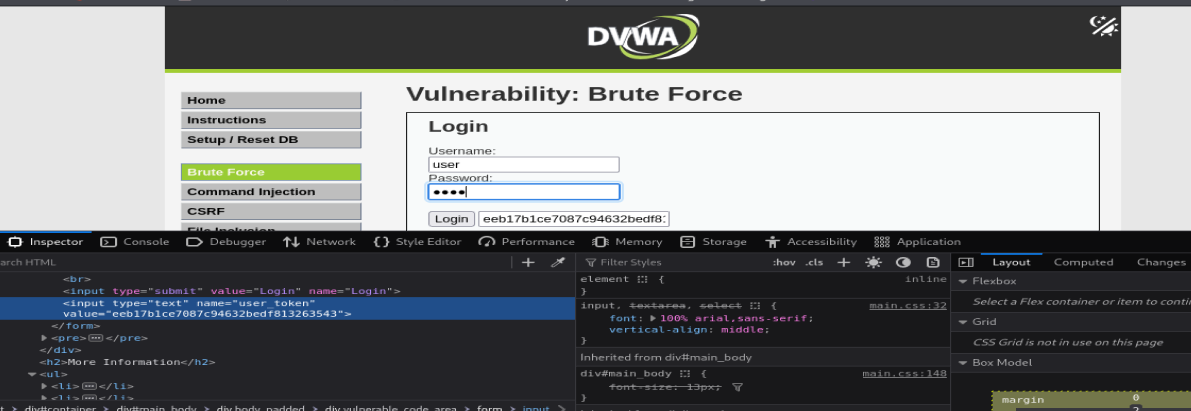
Enforce **strong password policies**

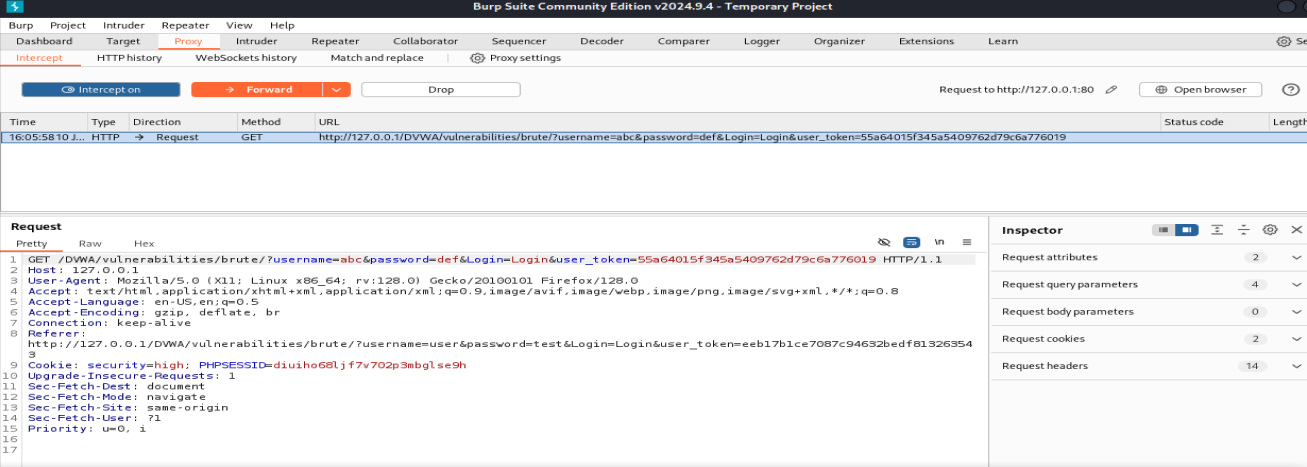


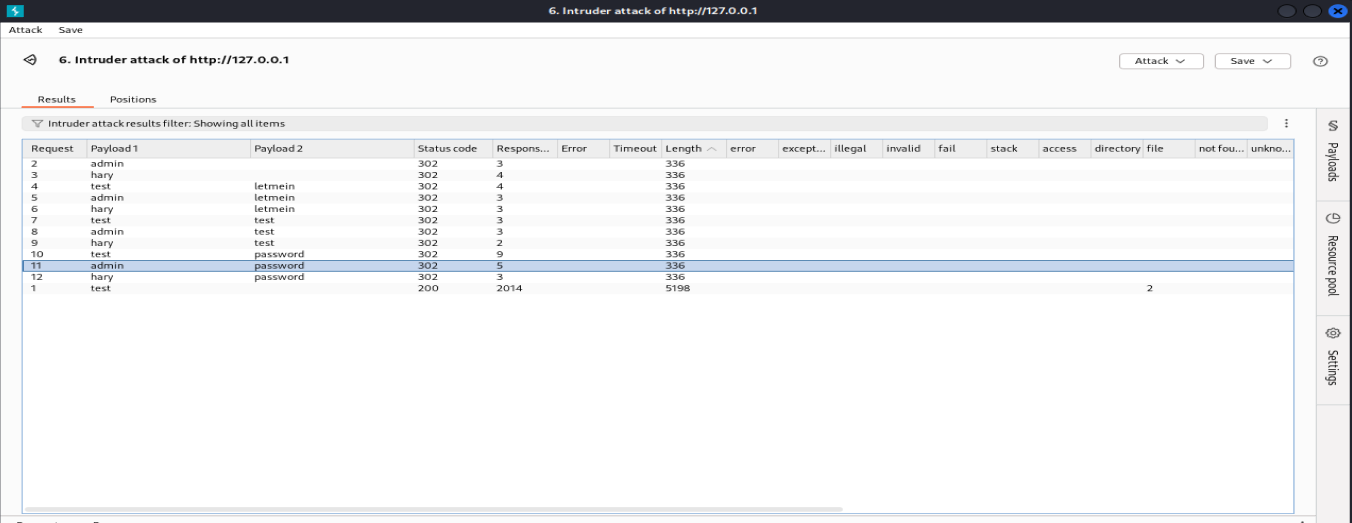












## **What Is SQL Injection?**

SQL injection is a type of attack that happens when someone finds a way to mess with the SQL queries your app sends to the database. Normally, those queries are supposed to do things like fetch a user’s profile or update a product listing. But with SQLi, an attacker can inject malicious bits of SQL code into your input fields (like search bars or login forms), and suddenly the database is doing exactly what *they* want instead.

## **Types of SQL Injection**

Depending on how the attacker interacts with your application and what kind of feedback they get, SQLi comes in a few different flavours. There are 3 main types you might run into:

### In-band SQLi

This is the most straightforward type. The attacker sends a malicious SQL query and gets the results right back through the application. It’s fast, and often very effective.

**Error-based SQLi**:

This technique relies on the database helpfully returning error messages. These errors can reveal a ton of useful info, like table names or the structure of queries, which makes it easier for the attacker to plan their next move.

**Union-based SQLi**

Here, the attacker uses the UNION operator to combine their own query with the one your app is running. It’s a clever way to pull extra data out of the database and sneak it into the response.

### Inferential SQLi (aka Blind SQLi)

This one’s sneakier. Instead of seeing the results of their query directly, the attacker watches how the application behaves to figure out what’s happening under the hood.

**Boolean-based (content-based) SQLi**

The attacker tweaks the query with conditions that are either true or false (like 1=1 or 1=2) and observes how the page changes. Does it load normally? Show an error? Act weird?

**Time-based SQLi**

This technique adds time delays into the query (e.g. WAITFOR DELAY '00:00:05') and uses the response time to infer whether a condition is true.

### **Out-of-band SQLi (aka when things get fancy)**

This one’s less common, but still worth knowing about. Out-of-band SQLi doesn’t rely on immediate responses from the app. Instead, the attacker uses alternate channels like DNS or HTTP requests to extract data. It's usually reserved for situations where direct feedback isn’t possible but the database server has internet access (and in 90% of cases, it probably shouldn't).

**How It Works:**

**Identify Vulnerable Input** (e.g., forms, URLs, cookies).

**Inject SQL Payload** to manipulate the query.

**Bypass Filters/Validation** (e.g., using ' OR 1=1--).

**Execute Malicious SQL** like reading or deleting data.

**Extract or Modify Data**, gain unauthorized access

**Advanced SQLi** may execute OS-level commands.

### Impact of Successful SQLi:

Steal credentials and personal data.

Access, modify, or delete sensitive records.

Gain database admin rights.

Move laterally into other systems.

### Examples of SQL Injection:

#### Example 1: Admin Login Bypass

#### ' OR 1=1--

#### Example 2: Access Data

#### ' UNION SELECT username, password FROM users--

#### Example 3: Inject in Form Field

anything' OR 'x'='x

### Real-Life SQLi Attacks:

**GhostShell:** Stole 36,000 records from 53 universities.

**7-Eleven breach:** 130 million credit cards stolen

**Tesla, Fortnite, Cisco:** Major vulnerabilities exposed.

Types of SQLi:

**Union-Based SQLi** – uses UNION to combine queries.

**Error-Based SQLi** – forces error messages to reveal data.

**Blind SQLi** – no visible output, data inferred via behavior.

Boolean-Based

Time-Based

#### Based on Injection Method:

**User Input**

**Cookies**

**HTTP Headers**

**Second-Order SQLi**

### Prevention Cheat Sheet (Key Tips):

Use **Prepared Statements / Parameterized Queries**.

**Validate & Sanitize** all user inputs.

Us **ORMs** (Object Relational Mappers).

Apply **Least Privilege Principle**.

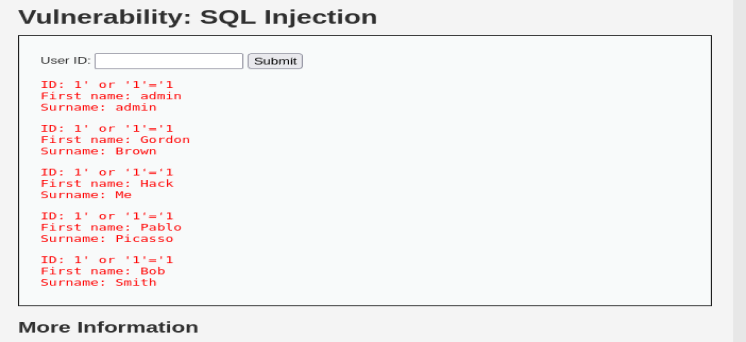
Employ **Web Application Firewalls (WAFs)**.

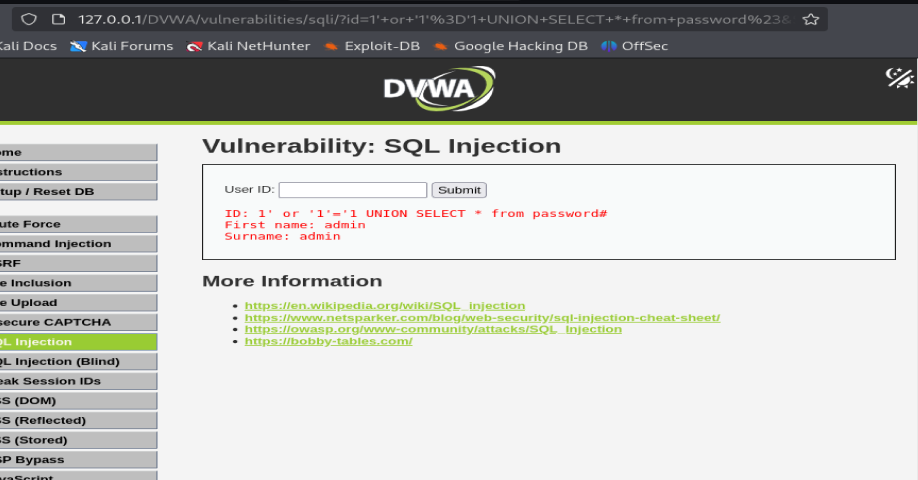
Keep software and databases **up to date**.

**TOOL USED :SQLPMAP**

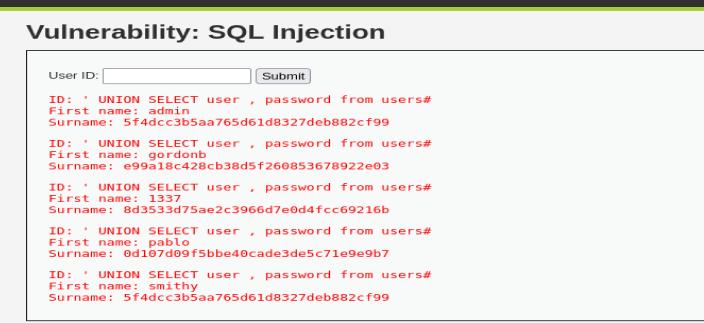
**1.SQL INJECTION EASY:**





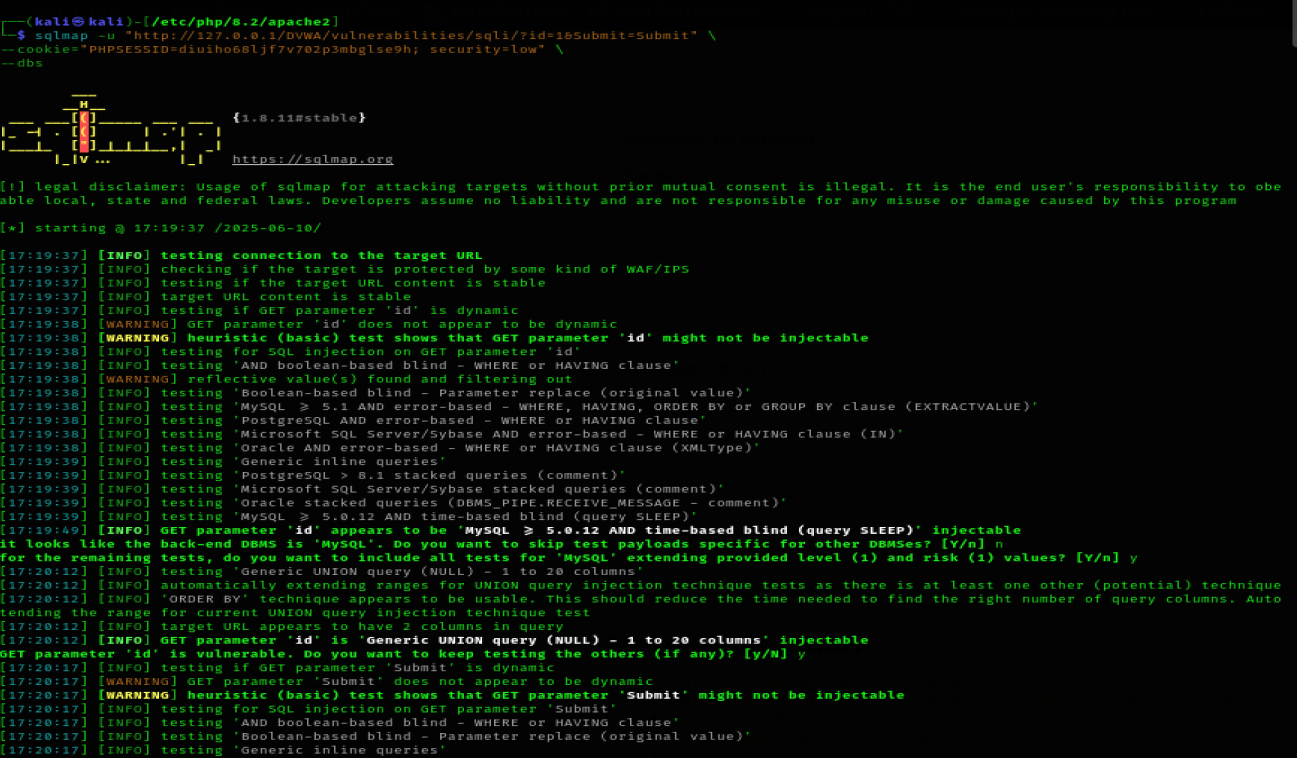


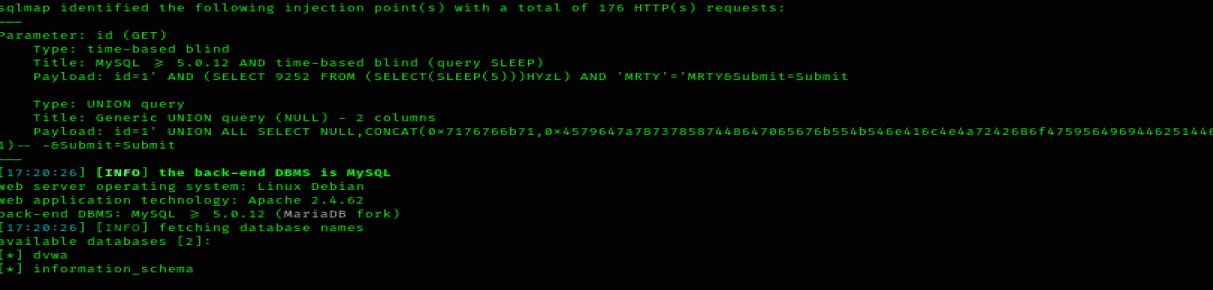
**Find username and password by manually trying SQL queries.**



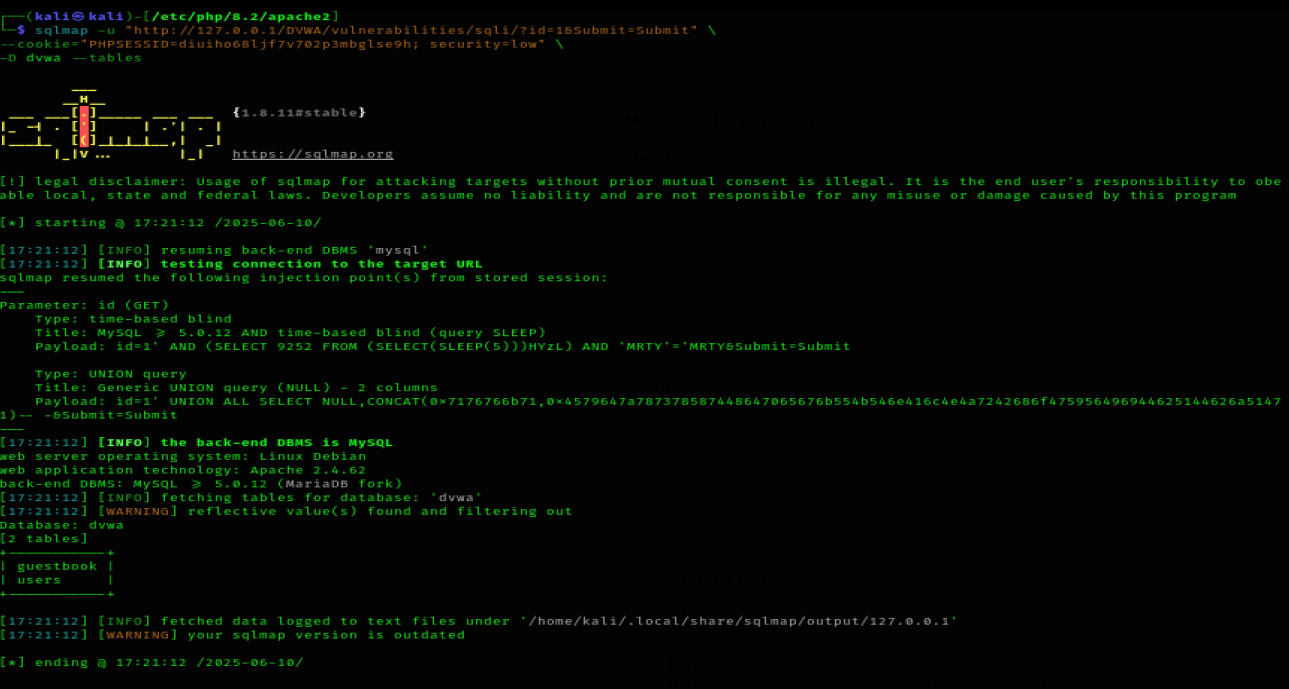
**Screenshot Of SQLMAP:**

**Find databases:**

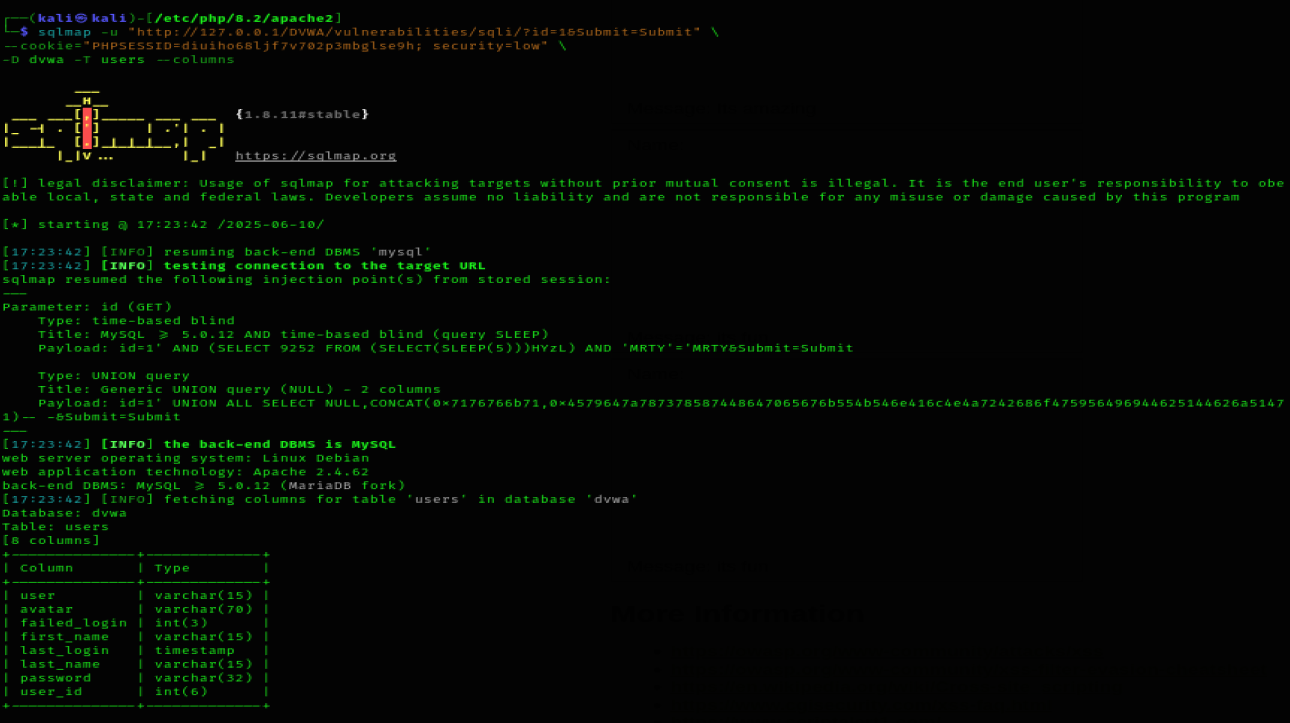


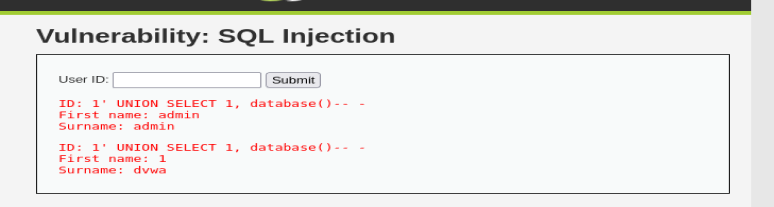


**Find tables in the DVWA Database.**



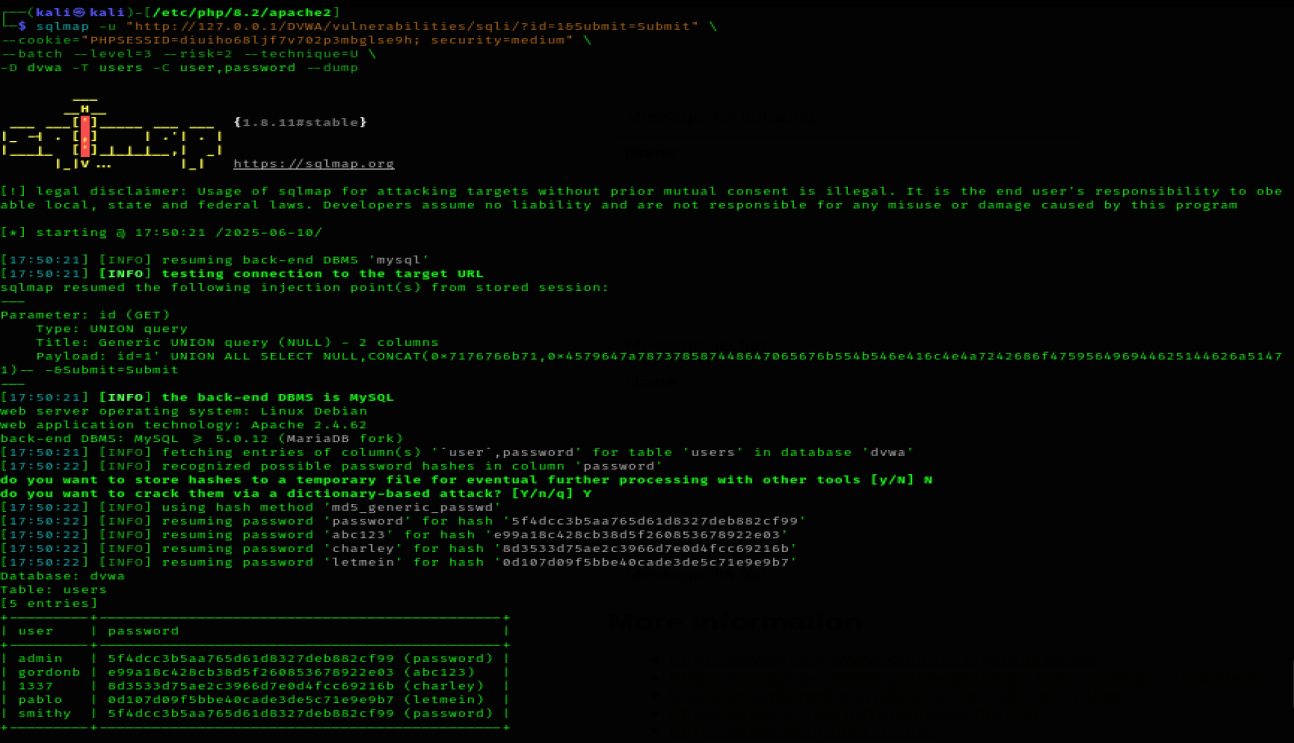
**Find column in the Users Table.**

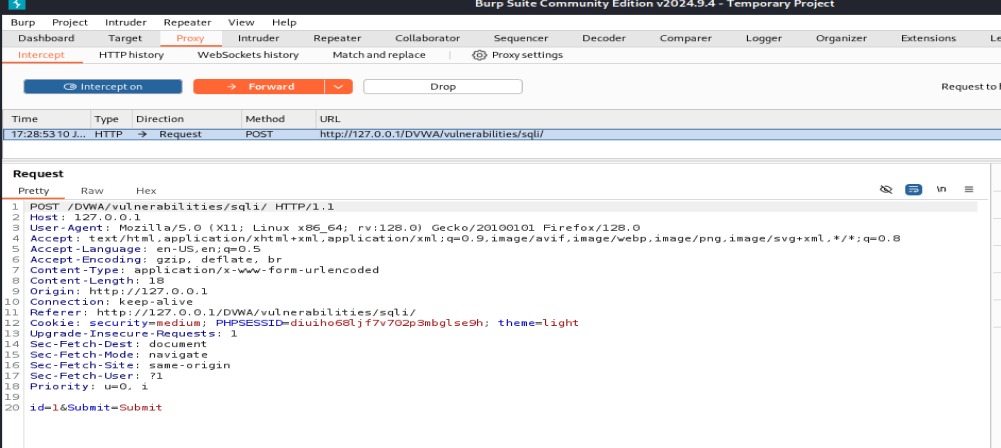


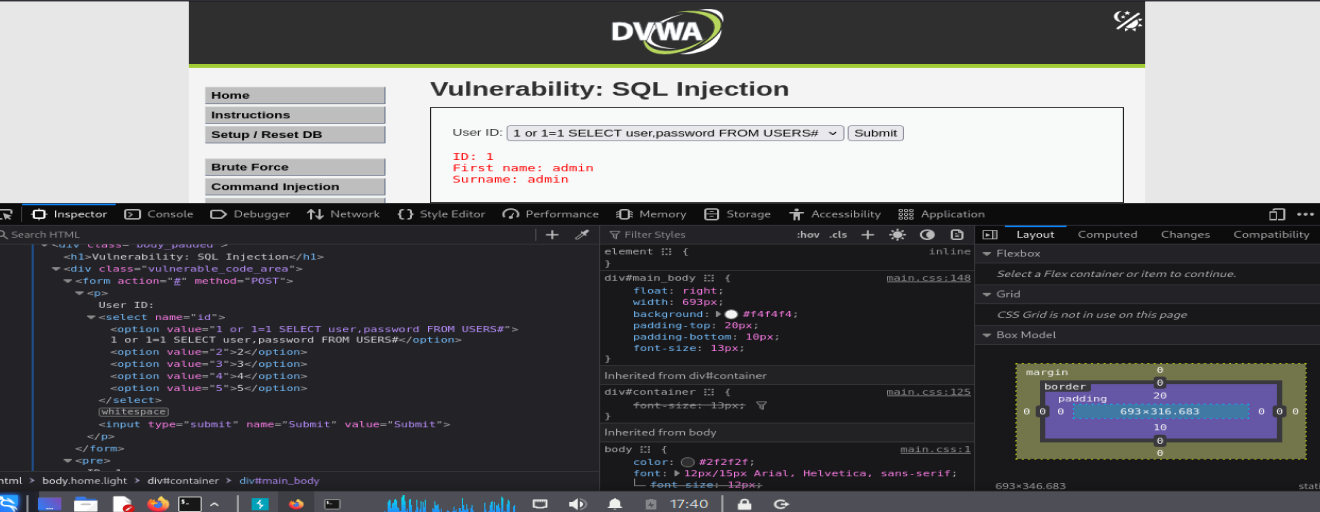


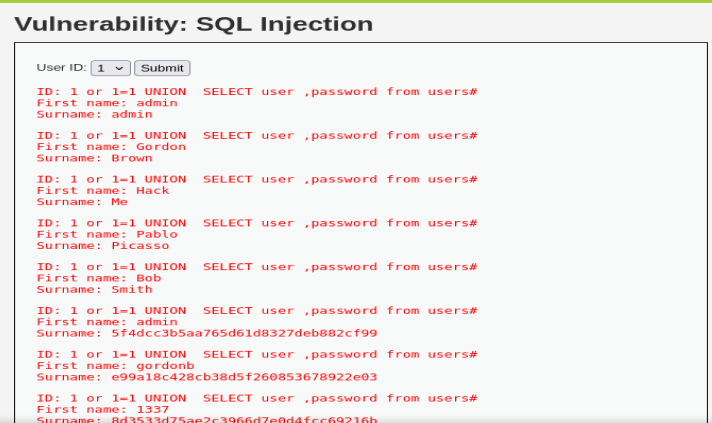
**2.SQL INJECTION MEDIUM:**

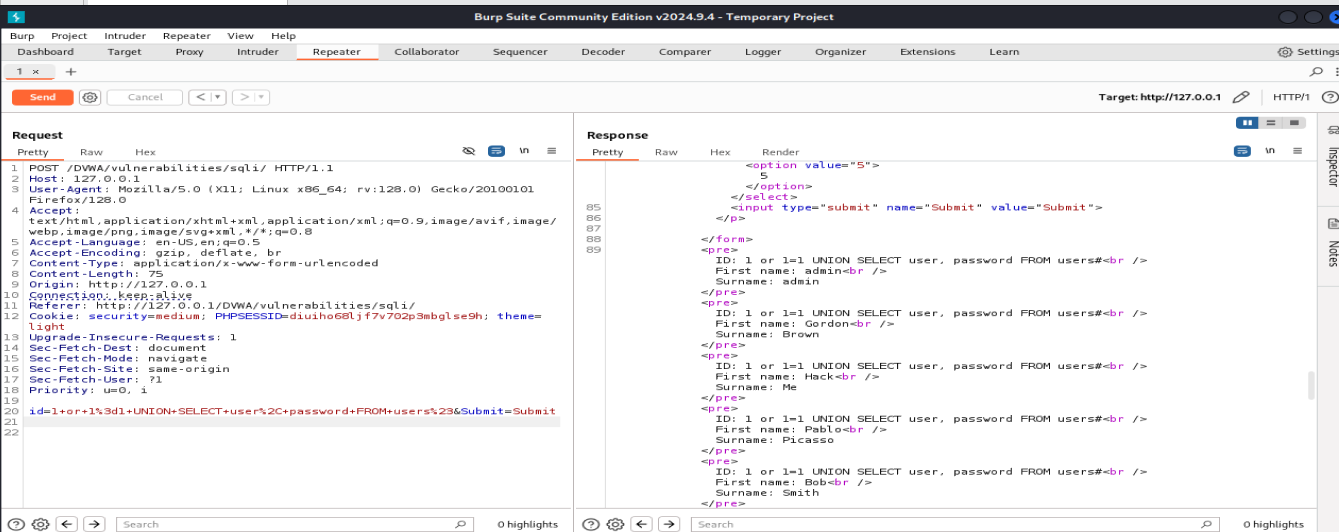
**Finding username and password in the table users.**



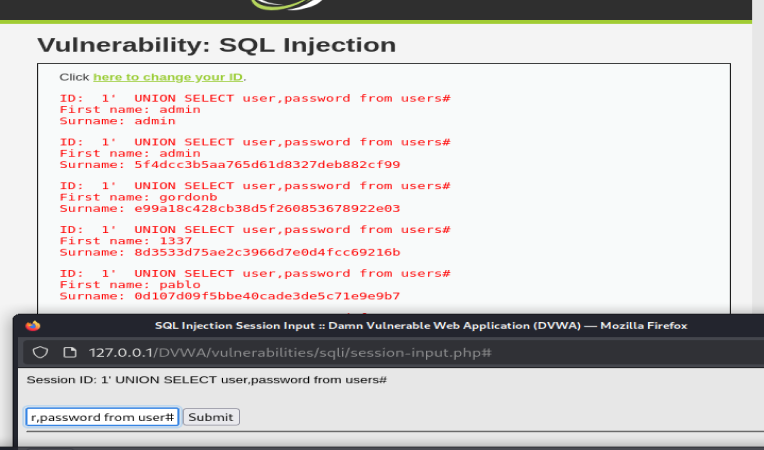








3.SQL INJECTION HARD:



**3.Cross Site Scripting(XSS)**

**What is XSS attack??**

OWASP (Open Web Application Security Project an online community that produces freely-available articles, methodologies, documentation, tools, and technologies in the field of web app security) defines XSS as an injection attack in which an attacker tries to inject some malicious code in the website’s input field. This code is merely one or two-line script written in any language that a web browser executes. Most importantly, in an XSS attack, an attacker tries to inject some HTML code or some JavaScript code in the input field of the website. The input field can be anywhere in the website’s page. The user gives his input using this input field. The input field can be an HTML form, a Signup button, or an HTTP header like HOST, COOKIE, USER-AGENT, etc.

Even if an attacker can inject any client-side code in the input field, but his main focus is on injecting such code that will give a popup on the user’s browser screen. The popup is an alert box that proves that the given website is vulnerable to an XSS attack. The universal and most basic method to get a pop-up window on the browser screen is to include alert() function inside the script tag (<script>…. </script>). So, whenever an attacker injects payload (a group of words & characters) <script> alert() </script> in any of the input fields of the website, it will pop up an alert box on the browser screen which confirms that the website is vulnerable to XSS attack. Let’s move towards our second question which is, what are the types of XSS attacks?

What are the types of XSS attacks?

There are three types of XSS attacks namely:-

Reflected XSS

Stored XSS

DOM Based XSS

Let us understand the concept behind each type of attack

**1. Reflected XSS**

Reflected XSS occurs when the input supplied by the user reflects back in the browser window or inside page source of the web page. What does it mean? Let us understand it with an example, suppose I have entered some value let’s say thisisreflecting in the input field of the website, now open the source of the page by pressing CTRL+U and search for the string thisisreflecting in the page source. If this word (thisisreflecting) is reflected or present in the page source then that parameter which is accepting the input may be vulnerable to reflected XSS. Now, you can try the payload <script> alert() </script> in place of thisisreflecting in the same input field. If it is vulnerable it will give a popup.

**2. Stored XSS**

Stored XSS occurs when the input supplied by the user is stored on the server side without performing proper sanitization or HTML encoding. The storage place can be a database, a message forum, a visitor log, a comment field, etc. Instead of reflecting back immediately, it may reflect back when you login into the website the next time. When you visit the vulnerable web page you will get a pop-up as alert window. Stored XSS is more dangerous than reflected XSS because it will harm the whole community by popping an alert box on every user’s browser who visits the vulnerable page. The payload used in stored XSS is same as reflected XSS.

For more info on Stored XSS and its exploitation on the DVWA app check this article.

**3. DOM-Based XSS**

DOM XSS stands for Document Object Model based Cross-Site Scripting. Here, the vulnerability appears in the DOM instead of part of the HTML. It is different from the other two types of XSS in the way that in DOM-based XSS user-supplied input doesn’t reflect back in the source code but the user can see the data in the HTML DOM tree using CTRL+SHFT+C and taking mouse pointer to the input field where he has entered his data and observing the DOM tree.

According to W3C – “The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document.“

To exploit this vulnerability, an attacker tries to inject some JavaScript code into the Document Object Model or HTML DOM. It has the capability to dynamically modify html tags, CSS properties, and content of HTML documents. And hence, can be used to inject JavaScript code to get a popup on the screen.

For more info on DOM-Based XSS and its exploitation on the DVWA app check this article.

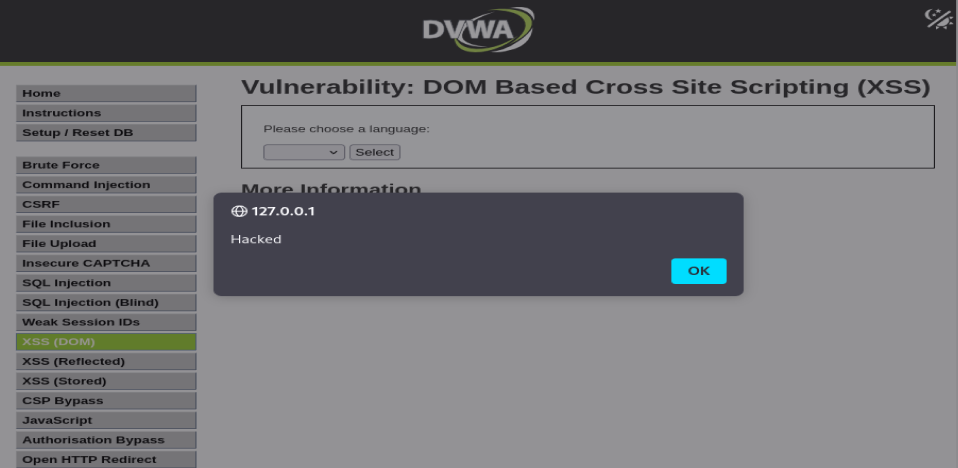
**Why XSS vulnerability arises?**

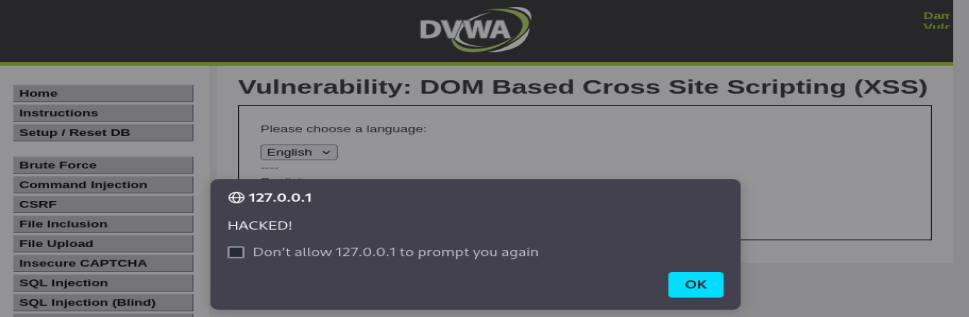
XSS vulnerability occurs in those websites which take input from the user and while reflecting back in the browser they don’t perform any sanitization or content encoding. Sanitization means removing some unwanted characters like >,<,“,”,%, etc. from the input string. And encoding means changing bad characters ( >,<,“,”,% ) to some other similar meaning characters including numbers and letters so that they will not reflect back in the same format as they were entered.

Now I will show you how you can exploit Reflected XSS vulnerability in the DVWA application. Although, there are various ways to exploit Reflected XSS in Dvwa but we are going to cover the most basic one. So let’s begin,

**DOM-Based XSS – Easy (Low Security)**

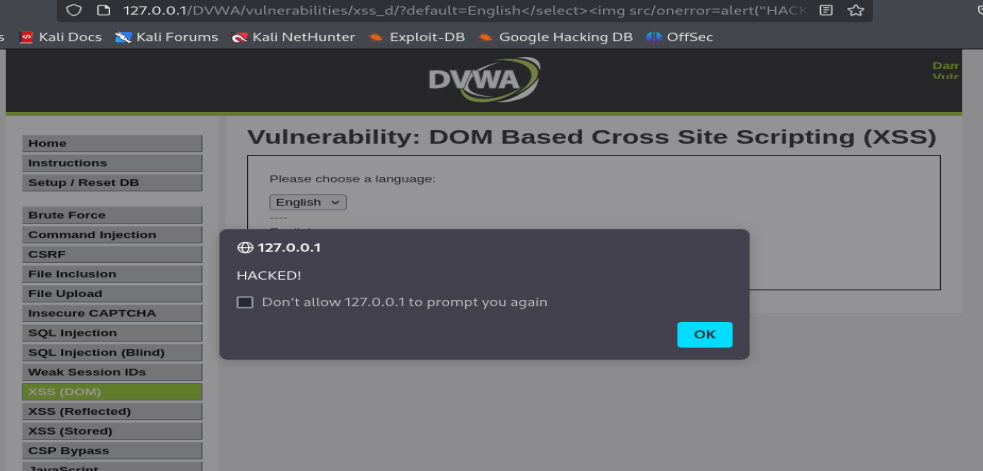
**Payload :**<script>alert('Hacked')</script>

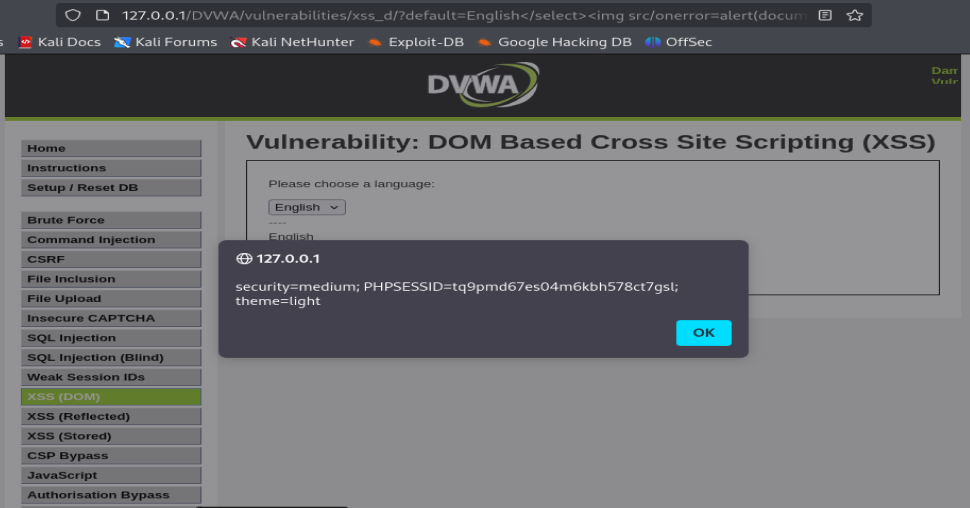




**2.DOM-Based XSS – Medium Security**

**Payload:**</select><img src/onerror=alert("HACKED")>

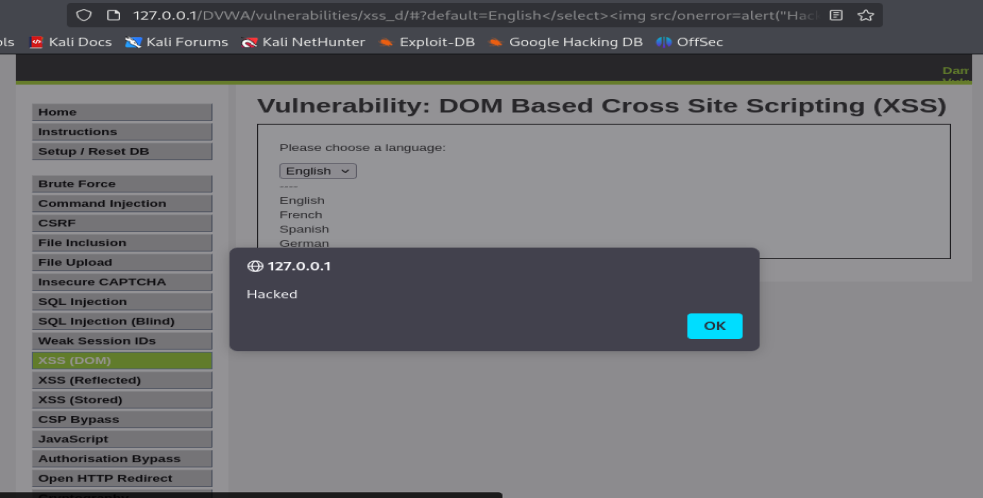




**3. DOM-Based XSS – Hard Security**

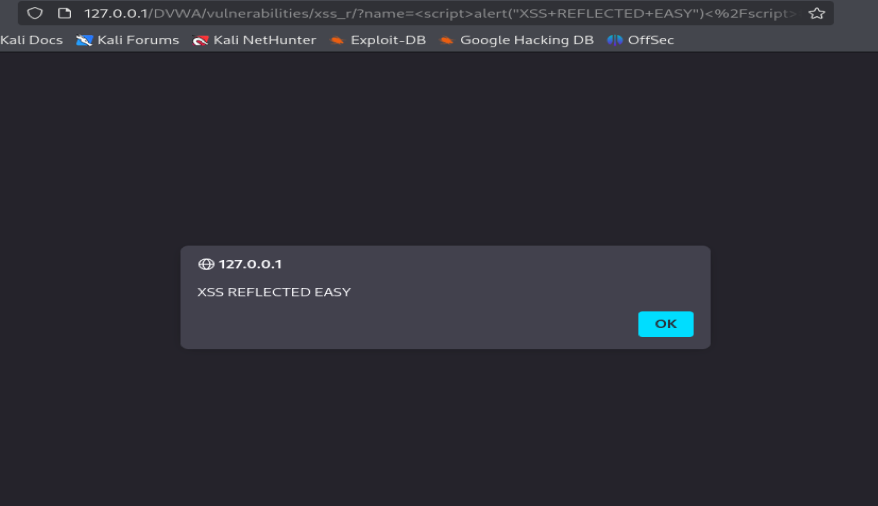
**Hard mode sanitizes common tags more strictly, but you can still try creative bypasses.**

**Payload :</select><svg onload=alert('XSS')>**



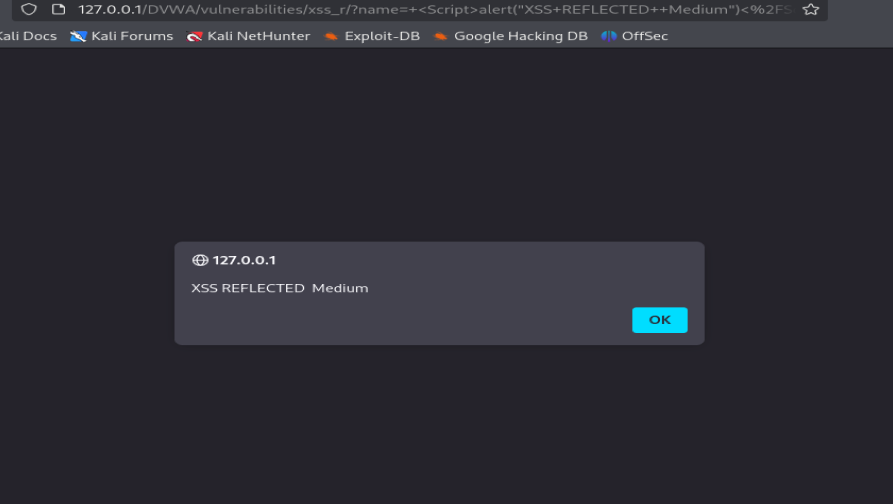
## **1.Reflected XSS – Easy**

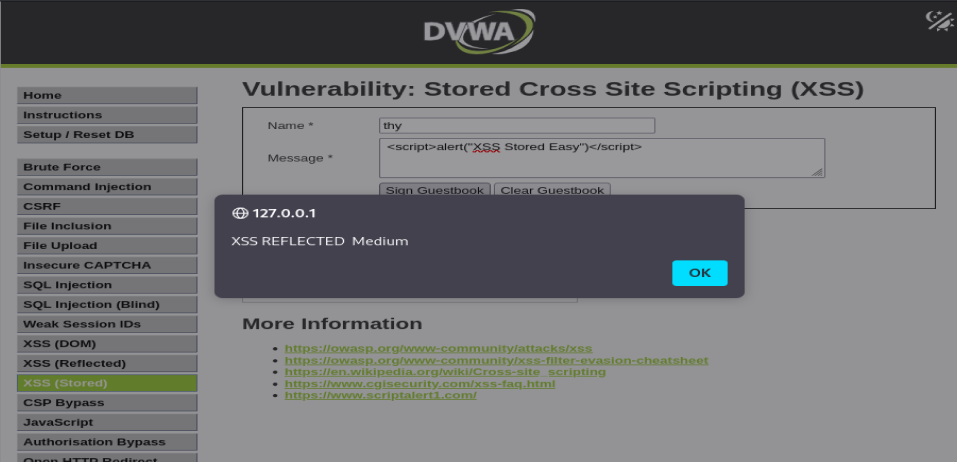
**Payload:**<script>alert("XSS REFLECTED EASY")</script>



## **2.Reflected XSS – Medium**

**Payload:**<Script>alert("XSS REFLECTED EASY")</Script>

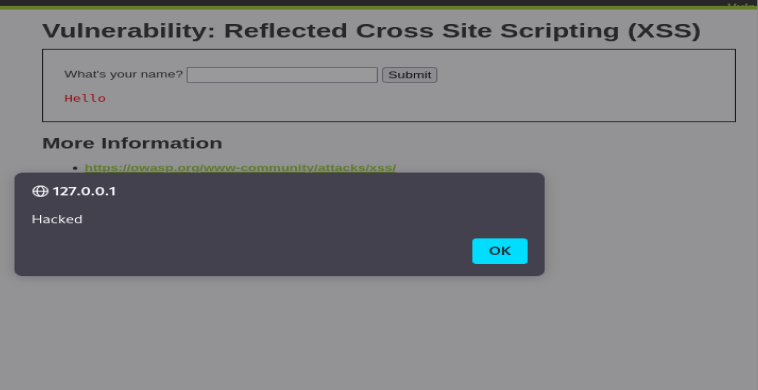




## 3.Reflected XSS – Hard Security

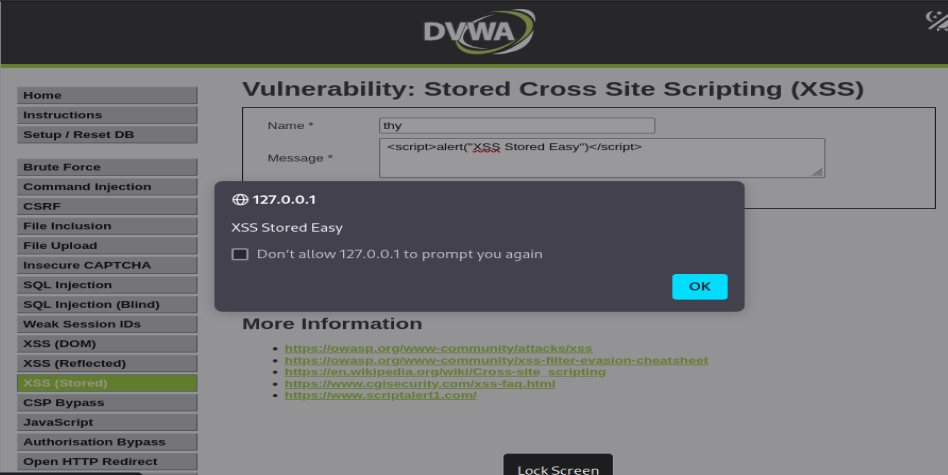
**Hard mode blocks** <script> **and other common tag injections. Try creative or encoded payloads.**

**Payload:<img src/onerror=alert("Hacked")>**



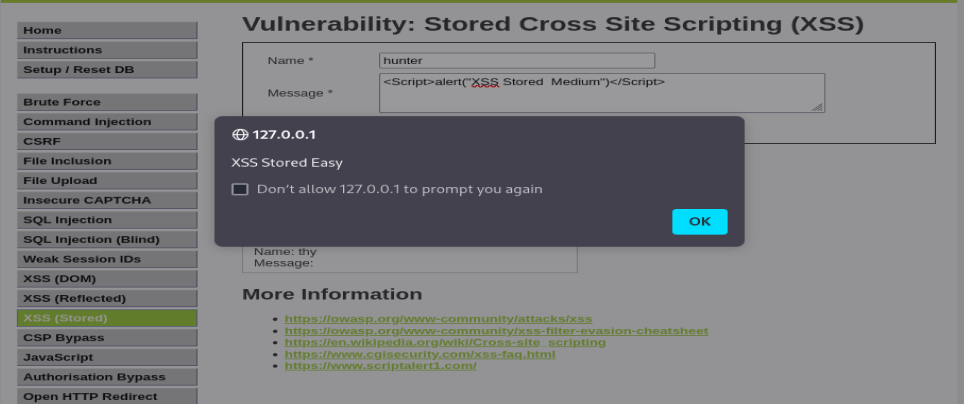
## **1.Stored XSS – Easy**

**Payload:**<script>alert("XSS Stored Eeasy")</script>

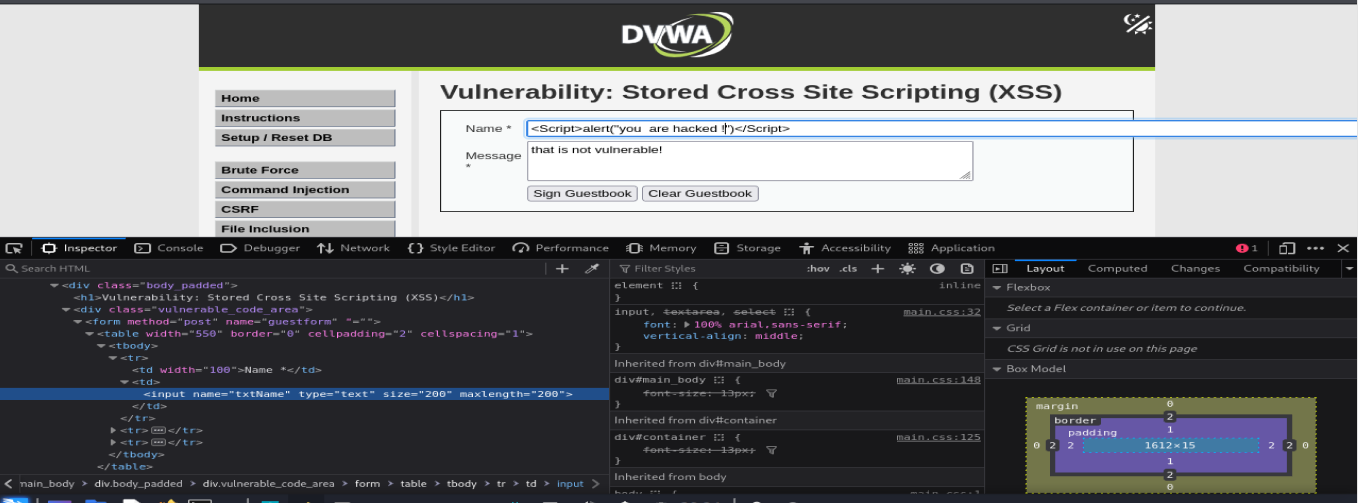


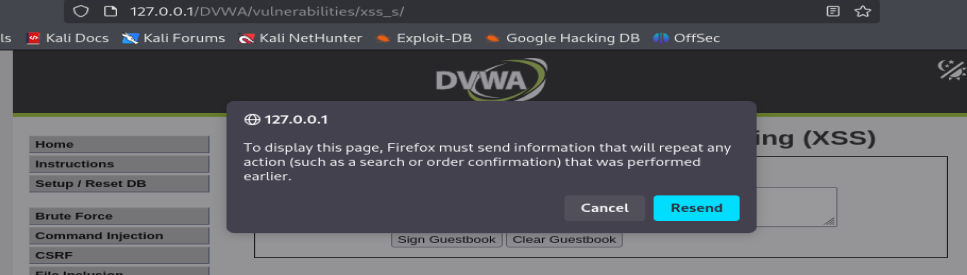
## **2.Stored XSS – Medium**

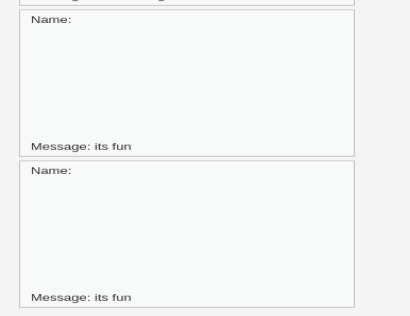
**Payload:**<Script>alert("XSS Stored Medium")</Script>



## **3.Stored XSS – Hard**

**Payload:<svg/onload=alert(“Hacked”)>**





The data are stored in the database .