

## Software Security project

Emanuele d'Ajello

M63 001435

Federico II università di Napoli

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.

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>

## Vulnerabilità: Session fixation

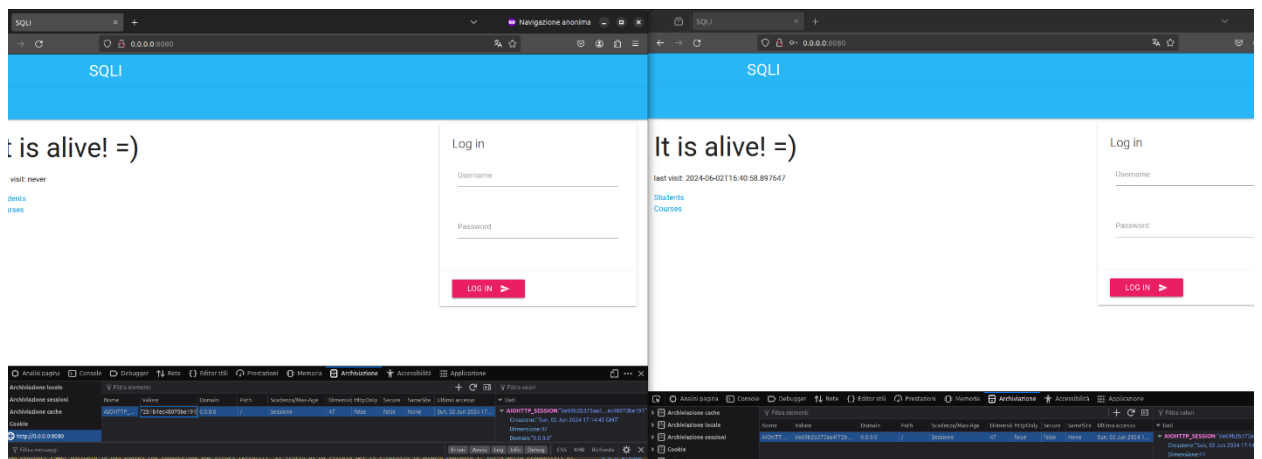
Tale vulnerabilità sfrutta la Session hijacking. Ciò avviene poiché per ogni sessione è rilasciato un cookie AIOHTTP\_SESSION il quale non viene mai rinnovato finché tale sessione è attiva.

Esempio di attacco:

Durante una sessione aperta su una scheda, possiamo aprire una nuova scheda incognita e utilizzare lo stesso cookie.

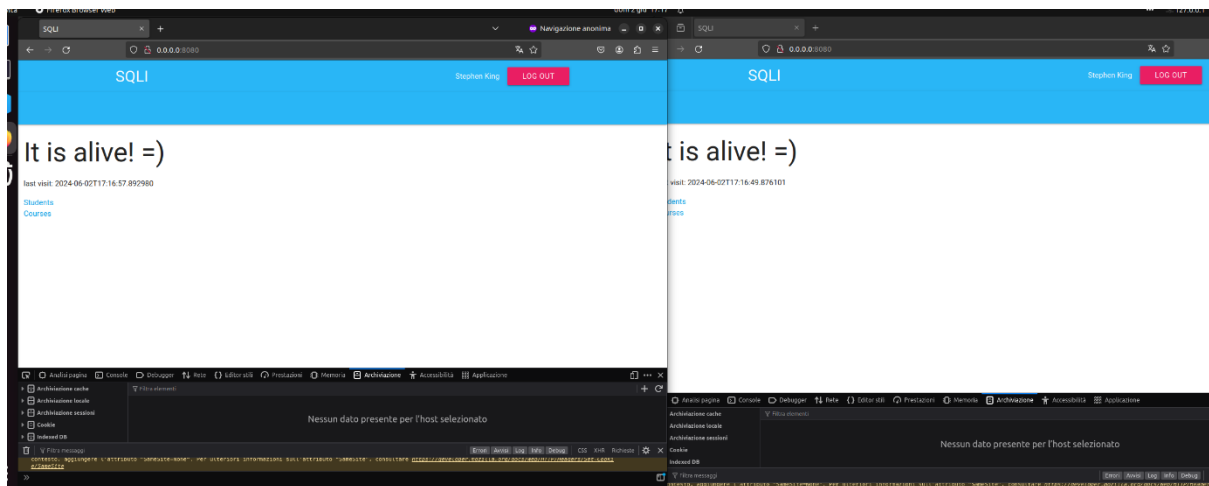
A questo punto se si effettua il login nella sessione originale questo sarà effettuato anche nella sessione incognita.

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.



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:



## Mitigation:

Tale vulnerabilità è dovuta al mancato rinnovamento del cookie di sessione durante l'accesso.

Il modo più semplice per mitigare tale vulnerabilità è quella di aggiornare i cookie di sessione ad ogni login e logout, in questo modo anche se un attaccante dovesse riuscire a recuperare tale cookie questo sarà rinnovato al seguito delle operazioni suddette.

Un ulteriore metodo di mitigazione è l'aggiornamento dei cookie di sessione scandito tramite un timer, in questo modo per un attaccante sarà molto difficile stabilire una connessione duratura tramite la sessione della vittima.

## Vulnerabilità: SQL-Injection

Nell'intro è stato già descritto il database che alla base dell'architettura di DVPWA.

La vulnerabilità è chiara: tutte le query fatte al database non sono sanitizzate.

Questo implica che per un attaccante è possibile modificare l'input di quest'ultime per fare query malevoli.

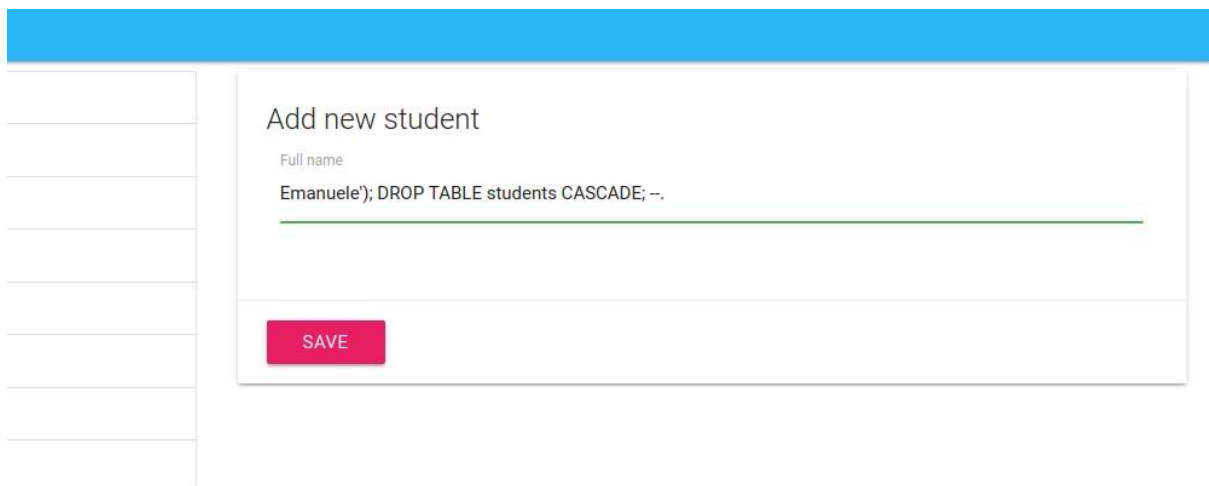
Qui è riportato nell'immagine seguente una query vulnerabile.

```
25 class Student(Base):
26     async def get_many(conn: Connection, limit: Optional[int] = None,
27                          offset: Optional[int] = None):
28         params = {}
29         if limit is not None:
30             q += ' LIMIT %(limit)s '
31             params['limit'] = limit
32         if offset is not None:
33             q += ' OFFSET %(offset)s '
34             params['offset'] = offset
35         async with conn.cursor() as cur:
36             await cur.execute(q, params)
37             results = await cur.fetchall()
38             return [Student.from_raw(r) for r in results]
39
40     @staticmethod
41     async def create(conn: Connection, name: str):
42         q = ("INSERT INTO students (name) "
43             "VALUES ('%(name)s') % {'name': name})")
44         async with conn.cursor() as cur:
45             await cur.execute(q)
46
47
```

Notiamo subito che la suddetta query si trova all'endpoint <http://0.0.0.0:8080/students/> qui è possibile verificare la presenza del campo “Add new student” la query a questo collegato è:

(“INSERT INTO students (name) VALUES ('%(name)s')” da cui si evince che la manipolazione della query avviene tramite il campo “name” passato in ingresso alla funzione. Tale parametro è una stringa non sanitizzata quindi inserendo per esempio nel campo studenti l'input : “A’); DROP TABLE students CASCADE; --” il risultato sarà l'eliminazione di tutta la tabella Students con conseguente alterazione e malfunzionamento di tutto il database.

L'immagine seguente mostra l'esempio di attacco:



The screenshot shows a web application interface with a blue header. On the left, there is a sidebar with a list of items. The main content area displays a form titled "Add new student". The form has a label "Full name" and a text input field. The input field contains the text "Emanuele"); DROP TABLE students CASCADE; --". Below the input field is a red button labeled "SAVE".

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

```

File "/app/sql/dao/student.py", line 30, in get_many
    await cur.execute(q, params)
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/asyncio/tasks.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

```

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.

## Vulnerabilità: XSS

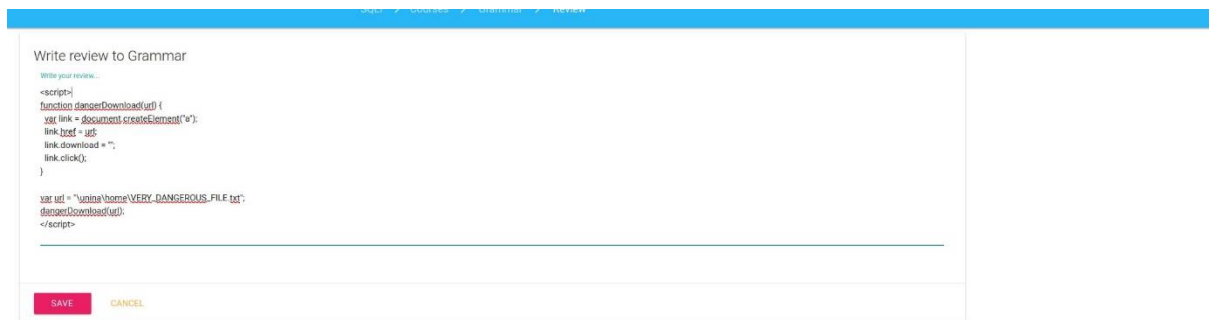
La vulnerabilità in questione è la stored XSS. Cioè l'applicazione memorizza uno script malevolo che verrà eseguito dai browser di tutti gli utenti "vittima" che visiteranno il sito.

In particolare nel nostro caso l'end-point vulnerabile si trova all'url

"http://0.0.0.0:8080/courses/1/review", inserendo un nostro script HTML/Javascript è possibile farlo eseguire dal browser. L'attacco è inoltre detto "stored" in quanto il server dell'applicazione salva lo script.

L'esempio di attacco eseguito è l'utilizzo di uno script per il download di un file malevolo,

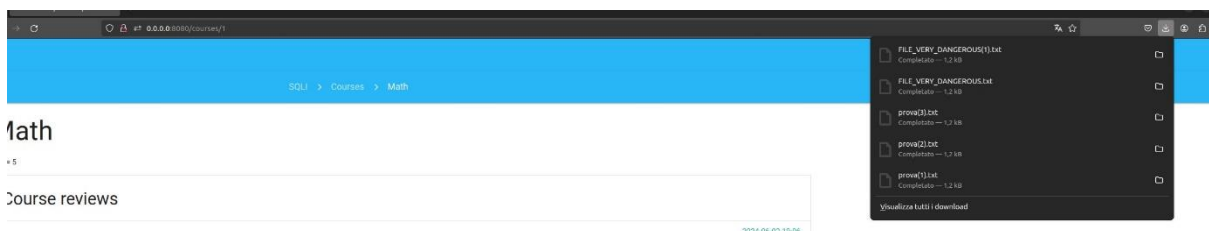
ogni volta che un utente (vittima ) seguira l'ulr sopra citato, il browser eseguirà lo script nell'immagine qui di seguito:



Questo è solo uno dei tanti script malevoli inseribili nella web-app.

Un altro possibile utilizzo è quello di inserire implementare uno snippet in grado di salvare il cookie di sessione in una variabile poi visualizzabile dall'attaccante eseguendo quindi successivamente un attacco di session fixation (citato come prima vulnerabilità).

Di seguito l'immagine della riuscita dell'attacco con il download del file “malevolo” involontario da parte dell'utente.



## Mitigation:

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 inseriti 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.

### **Vulnerabilità: Brute-Force Password.**

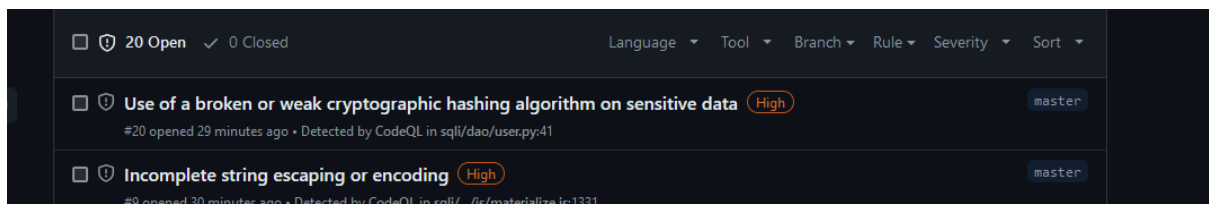
L'ultima vulnerabilità è osservabile notando che le password sono hashate con la funzione md5 e salvate direttamente sul database. Questa tecnica è vulnerabile ad attacchi di brute force in quando md5 è un noto algoritmo di hash abbastanza debole con probabili collisioni (stesso output per input diverso) . Inoltre essendo salvate sul database tramite un attacco di SQL-Injection è possibile leggere le hash sul database e tramite un tool di hash-crack risalire alla password originale.

Mitigation:

Per evitare la vulnerabilità sopra citata bisognerebbe non salvare mai le password direttamente sul database e utilizzare algoritmi di hash moderni che offrono maggior sicurezza contro attacchi di tipo Brute-Force.

### **Static Analysis:**

In particolare quest'ultima vulnerabilità descritta è analizzata anche tramite una Static Analysis guidata da GitHub-CodeQL. Infatti dopo aver effettuato una scansione del codice con query default l'analisi ci riporta quanto mostrato nelle seguenti immagini:



Qui notiamo quindi due vulnerabilità riscontrate nel codice segnalate di tipo “high” cioè un uso di una debole crittografia di hashing per le password utente ed inoltre anche un escape



incompleto delle stringe di testo date in input causanti le vulnerabilità di tipo xss ed sql-injection descritte nelle sezioni precedenti.

Nell'immagine seguente invece mostrato più nel dettaglio la vulnerabilità riscontrata nell'hashing delle password ed anche delle “suggestion” per la mitigation.

**Use of a broken or weak cryptographic hashing algorithm on sensitive data** Dismiss alert

Open in `master` 23 minutes ago

```
sqli/dao/user.py:41
38         return User.from_raw(await cur.fetchone())
39
40     def check_password(self, password: str):
41         return self.pwd_hash == md5(password.encode('utf-8')).hexdigest()
```

Sensitive data (`password`) is used in a hashing algorithm (MD5) that is insecure for password hashing, since it is not a computationally expensive hash function.

[CodeQL](#) [Show paths](#)

Tool	Rule ID	Query
CodeQL	py/weak-sensitive-data-hashing	<a href="#">View source</a>

Using a broken or weak cryptographic hash function can leave data vulnerable, and should not be used in security related code.

[Show more](#)

**Severity**  
High

**Affected branches**  
master

**Tags**  
security

**Weaknesses**  
[CWE-327](#)  
[CWE-328](#)  
[CWE-916](#)







