SQL Injection

Emanuele Storci

Indice

1 Introduzione al progetto

Il progetto rappresenta una simulazione di una pagina di login di una web app. L'obiettivo è dimostrare le possibili violazioni ai principi fondamentali della CIA Triad.

1.1 Tipi di attacco

L'obiettivo è compromettere la Confidenzialità, accedendo alla webapp bypassando l'autenticazione nonostante non si possiedano le credenziali corrette. Successivamente, mostreremo possibili impatti sull'Integrità, agendo sui dati accessibili con la possibilità di modificarli e manipolarli. Infine, andremo a compromettere la Disponibilità eliminando la tabella degli utenti dal database, rendendo di fatto il sistema inutilizzabile. Al termine dell'attacco il server non sarà più operativo.

1.2 Contesto operativo dell'attacco

Nel nostro progetto, la web app è realizzata utilizzando un'architettura composta da:

- un **backend in PHP** che gestisce le API e l'interfaccia web, esposta tramite **Apache**.
- un **database MySQL**, utilizzato per la memorizzazione degli utenti e delle transazioni.

L'intero ambiente è gestito da **Docker Compose**, per garantire semplicità di esecuzione su più macchine. In particolare sono impiegati due container separati: uno dedicato al web server con PHP, sulla porta `80`, e uno per il database MySQL, che è configurato sulla porta `3306`.

L'applicazione presenta vulnerabilità legate a una gestione non sicura delle query SQL lato server, che possono essere sfruttate per condurre un attacco di tipo **SQL Injection**. L'attacco sarà effettuato forzando le n interazioni con l'endpoint PHP di login e con le API di gestione di utenti, bypassando l'autenticazione e accedendo a dati riservati.

2 Analisi del codice

Il progetto è una web app sviluppata in PHP, gestita tramite Docker, con un backend semplice e un database MySQL inizializzato tramite uno script SQL. Lo scopo è mostrare un sistema di login e gestione delle transazioni legato agli utenti registrati, con la possibilità di visualizzare ruoli e operazioni effettuate.

2.1 Struttura del progetto

- **index.php**
  Questo file si occupa delle operazioni principali:
  - Verifica l'esistenza del file del database:
    - Se esiste, apre una connessione
    - Altrimenti, crea il database e lo inizializza con dati di esempio tramite lo script `init.sql`
  - Gestisce la richiesta `POST` all'endpoint `/login`, abilitando CORS per i metodi `POST` e `OPTIONS`
  - Riceve `username` e `password` dalla richiesta e li passa alla funzione di autenticazione
  - Le password non sono hashate, ma si suppone l'uso di una gestione sicura in un contesto reale

- **ruoli.php**
  Permette di visualizzare il ruolo associato all'utente autenticato (come `admin` o `user`)

- **utenti.php**
  Gestisce le operazioni relative agli utenti. La tabella `utenti` è creata tramite lo script `init.sql`:

```
CREATE TABLE utenti (
    id INT AUTO_INCREMENT PRIMARY KEY,
    ruolo VARCHAR(255),
    nome VARCHAR(255),
    pass VARCHAR(255)
);

INSERT INTO utenti (ruolo, nome, pass) VALUES
('admin', 'emanuele', '123'),
```

```
('user', 'alessio', '000'),
('user', 'flaminia', 'ciao'),
('user', 'giuseppe', 'roma');
```

- **transazioni.php**
  Mostra le transazioni associate agli utenti. La tabella `transazioni` è creata con lo script `transazioni.sql`:

```
CREATE TABLE transazioni (
    id INT AUTO_INCREMENT PRIMARY KEY,
    id_utente INT,
    importo DECIMAL(10,2),
    carta VARCHAR(255),
    data_operazione DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (id_utente) REFERENCES utenti(id)
);

INSERT INTO transazioni (id_utente, importo, carta, data_operazione) VALUES
(1, 120.50, '4539 1488 0343 6467', '2025-04-10'),
(2, 89.99, '5500 0000 0000 0004', '2025-02-20'),
(3, 250.00, '3400 0000 0000 009', '2025-01-02'),
(1, 50.00, '4539 1488 0343 6467', '2025-05-08'),
(4, 300.75, '6011 0000 0000 0004', '2025-03-14');
```

- **docker-compose.yml**
  Gestisce l'ambiente di sviluppo tramite il container Docker e avvia i servizi PHP e MySQL, facendo così si garantisce la continuità dei dati e l'accessibilità attraverso il nostro browser locale.

3 Simulazione d'attacco

3.1 Obiettivo dell'attacco

Il nostro obbiettivo è quello di dimostrare come un attacco SQL injection possa compromettere i tre principi fondamentali della sicurezza informatica: **Confidenzialità**, **Integrità** e **Disponibilità** (CIA Triad). Tramite una semplice pagina di login come punto di ingresso all'attacco.

Nel nostro caso, l'applicazione è scritta in PHP e il codice di login crea la query SQL concatenando i valori ricevuti in input:

```
$sql = "SELECT * FROM utenti WHERE nome = '$username' AND pass = '$password'";
```

Questo approccio espone il sistema a vulnerabilità critiche.

3.2 Funzionamento atteso

Il programma esegue correttamente un login se le credenziali corrispondono a un utente nella tabella `utenti`. Ad esempio, accedendo con:

- `username: emanuele`
- `password: 123`

l'utente viene autenticato con successo e può accedere a funzionalità riservate.

3.3 Attacco con payload `tautology`

Possiamo superare il login senza conoscere credenziali valide sfruttando una **tautologia** logica, iniettando il seguente payload nel campo `username`:

```
' OR 1=1 --
```

La query risultante sarà:

```
SELECT * FROM utenti WHERE nome = '' OR 1=1 --' AND pass = ''
```

La condizione `1=1` è sempre vera, e il commento `--` tronca il resto della query, rendendo inutile il controllo della password. Il risultato è un accesso non autorizzato. Viene compromessa la **Confidenzialità** del sistema.

3.4 Attacco con `EOL comment`

Se conosciamo un `username` valido (es. `flaminia`), possiamo bypassare anche il controllo della password usando il seguente payload:

```
flaminia' --
```

La query risultante:

```
SELECT * FROM utenti WHERE nome = 'flaminia' --' AND pass = ''
```

Questo permette l'accesso diretto come un utente esistente. Vengono compromessi sia la **Confidenzialità** che l'**Integrità**, visto che l'attaccante può accedere a dati riservati e potrebbe modificarli.

3.5 Attacco `Piggyback query`

Infine, possiamo lanciare un attacco distruttiv concatenando una seconda istruzione SQL:

```
'; DROP TABLE utenti --
```

La query diventa:

```
SELECT * FROM utenti WHERE nome = ''; DROP TABLE utenti --' AND pass = ''
```

In questo modo si elimina l'intera tabella `utenti`, rendendo impossibile qualsiasi login futuro. Viene compromessa la **Disponibilità** del sistema.

4 Conclusioni finali

Questi esempi dimostrano come una concatenazione non sicura di stringhe in una query SQL possa avere effetti devastanti per il sistema. È importante usare sempre `prepared statements` o `query parametrizzate` per prevenire questo tipo di attacchi SQLi. Questo esempio può essere utilizzato per mostrare in maniera semplice l'impatto della mancanza di protezioni basilari.