SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI INGEGNERIA INFORMATICA AUTOMATICA E GESTIONALE - DIPARTIMENTO DI INFORMATICA

Attacchi ai Database: SQL Injection

CORSO DI SICUREZZA (6 CFU)

Esame:
Sicurezza Informatica

Studente:
Francesco Maura,
2017683

Anno Accademico 2023/2024

Contents

1 Introduzione

La *SQL Injection* è una tecnica di attacco informatico che sfrutta vulnerabilità nelle applicazioni WEB per eseguire comandi SQL non autorizzati su Database sottostanti a livello di BackEnd. Il funzionamento sfrutta l'errata implementazione di validazione degli input di una pagina Web. Scoperta per la prima volta nel anni '90, durante la diffusione *mainstream* dei primi sistemi DBMS, questa tecnica è diventata e rimasta per molto una delle principali minacce alla sicurezza del database. La tecnica di SQL Injection, *brevemente SQLi*, permette all'attaccante di ottenere un accesso autorizzato ai dati, maniplare informazione oppure interrompere i servizi, potenzialmente compromettendo tutte e tre le caratteristiche C.I.A. (*confidentiality, integrity, availability)* che un **sistema informatico sicuro** deve possedere. Gli attacchi SQLi sono stati alla base di numerosi incidenti di sicurezza di alto profilo, evidenziando la necessità di adottare misure di protezione adeguate. In questo breve paper utilizzeremo un attacco di tipo *in-band* ossia il canale di output sarà lo stesso dell'input. In un attacco *out-band* solitamente si utilizza uno sniffer di pacchetti (*wireshark* ad esempio).

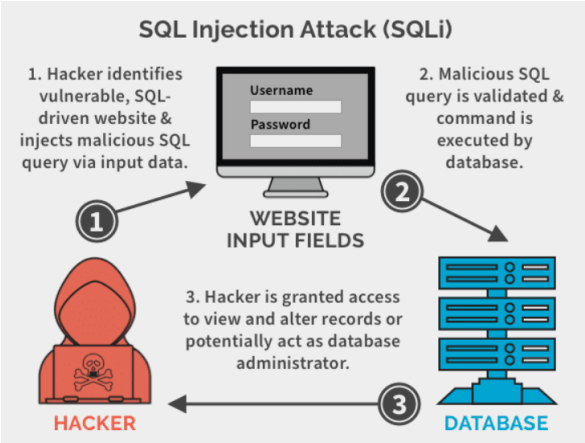


Figure 1: Schema di funzionamento di SQLi

2 Principi teorici dell'attacco

2.1 Funzionamento teorico

L'attacco SQLi, come illustrato nell'introduzione, si basa sull'inserimento nel campo di un input di un'applicazione web o tramite richiesta HTTP al server stesso. Il codice inserito viene poi eseguito dal server del Database presumendo che esso sia legittimo e provocando compromissione del sistema.

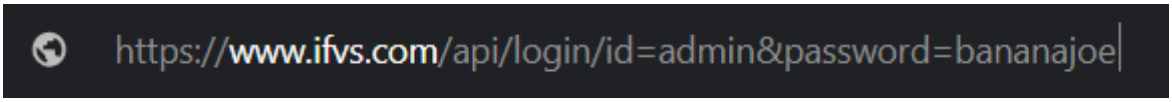


Figure 2: Url con parametri visibili

2.2 Tipo di vulnerabilità e impatto globale connesso

La vulnerabilità sfruttata è la mancanza di *sanitizzazione* degli input utente; infatti è vulnerabile un'applicazione che concatena gli input utente con dei comandi SQL senza l'utilizzo di adeguate misure di sicurezza, come l'uso di *prepared statements.* Secondo il report OWASP Top Ten *(Open Web Application Security Project)* l'attacco di SQLi rappresenta uno dei 10 maggiori attacchi informatici a cui un database può essere sottoposto.

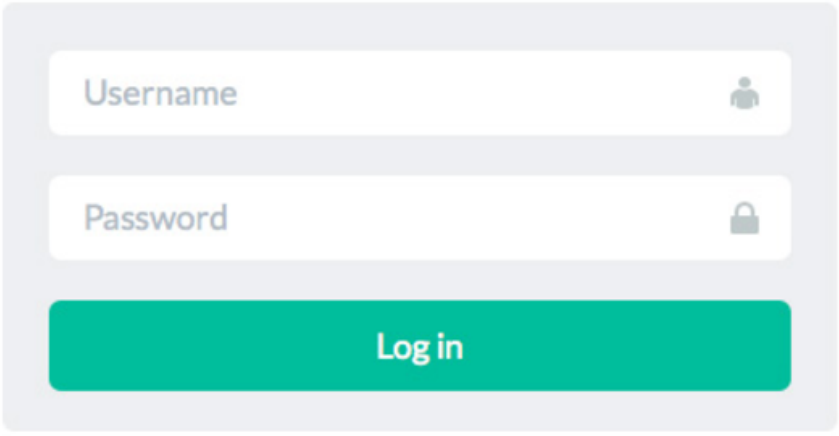


Figure 3: Pagina di login generica

2.3 Proprietà C.I.A. compromesse[1]

- **Confidenzialità:** la confidenzialità in un contesto di attacco SQL Injection (SQLi) si riferisce alla capacità di un attaccante di accedere a dati sensibili non autorizzati attraverso manipolazioni delle query SQL. Gli attaccanti possono

accedere a informazioni riservate (*email, password, numero di carta, residenza, ecc...)*, compromettendo la privacy degli utenti.

- **Integrità:** l'integrità in un contesto di attacco SQLi si riferisce alla capacità di un attaccante di modificare o distruggere dati nel database. Gli attaccanti possono altresì alterare i dati nel database compromettendo l'accuratezza e l'affidabilità delle informazioni. Ad esempio un attaccante sfrutta una vulnerabilità di SQLi per eseguire una query piggybacked che altera i dati nel database

- **Disponibilità:** gli attaccanti possono eseguire comandi che compromettono la disponibilità del servizio, come ad esempio cancellare tabelle o interrompere il funzionamento del database.

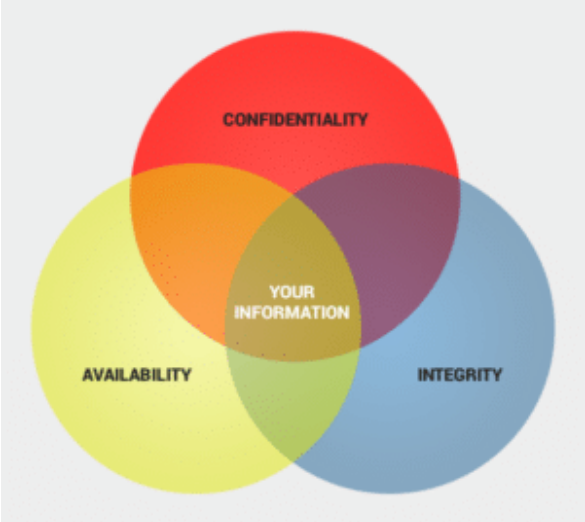


Figure 4: Priprietà C.I.A.

3 Implementazione di un Webserver vulnerabile

3.1 Struttura del WebServer

- **Backend:** PhP
- **Frontend:** HTML, CSS, JS ma in generale qualsiasi linguaggio di markup web
- **DBMS:** Mariadb, MySql, PostGre SQL o altri
- **SQL Engine:** Innodb

3.2 Codice del WebServer vulnerabile

DISCLAIMER: a titolo puramente accademico, e senza intenzione di diffondere codice malevolo, l'autore include il codice sorgente di un server vulnerabile ad attacchi SQLi. Non utilizzare tale codice in macchine in produzione o in webserver aperti in rete e non inserire il codice nella cartella *var/www* di un server.

Codice Html per una semplice pagina di login

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <h2>Login</h2>
    <form method="post" action="login.php">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username"><br>
        <label for="password">Password:</label>
        <input type="password" id="password" name="password"><br>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

Codice PhP per l'autenticazione del database

```
<?php
    $servername = "localhost";
    $username = "root";
    $password = "";
    $dbname = "test_db";

    $conn = new mysqli($servername, $username, $password, $dbname);

    if ($conn->connect_error) {
        die("Connessione fallita: " . $conn->connect_error);
```

```
    }
?>
```

Codice PhP per l'interrogazione della base di dati

```
<?php
    include('db.php');

    $username = $_POST['username'];
    $password = $_POST['password'];

    // Codice vulnerabile: query costruita concatenando
    // input utente senza sanitizzazione
    $sql = "SELECT * FROM utenti
            WHERE nome_utente = '$username'
            AND password = '$password'";
    $result = $conn->query($sql);

    if ($result->num_rows > 0) {
        echo "Login effettuato";
    } else {
        echo "Nome utente o password errati";
    }

    $conn->close();
?>
```

Commento al codice vulnerabile: Nel file login.php, la query SQL è costruita concatenando direttamente gli input utente ($username e $password). Questo rende l'applicazione vulnerabile a SQL injection, poiché un attaccante può manipolare gli input per eseguire comandi SQL arbitrari come vedremo nella sezione "Scenari d'attacco".

4 Scenari d'attacco

4.1 Operazioni preliminari

Per comprendere se un servizio presenta una vulnerabilità di tipo SQLi si hanno due strade:

- **Caratteri di escape**
- **Utilizzo di script**

Per tetare la vulnerabilità con caratteri di escape solitamente viene utilizzato — ' — (apostrofo, codificato %27%20). Inserendo infatti il carattere *apostrofo* all'interno di un input non sanitizzato otteniamo

Fatal error: Uncaught mysqli_sql_exception: You have an error in your SQL syntax; check the manual that corresponds to your MariaDB server version for the right syntax to use near "'" at line 1 in /var/www/html/DVWA/vulnerabilities/sqli/source/low.php:11 Stack trace: #0 /var/www/html/DVWA/vulnerabilities/sqli/source/low.php(11): mysqli_query() #1 /var/www/html/DVWA/vulnerabilities/sqli/index.php(34): require_once('...') #2 {main} thrown in **/var/www/html/DVWA/vulnerabilities/sqli/source/low.php** on line **11**

Questo output ci mostra un comportamento interessante: evidenzia un errore di sintassi in SQL, sintomo che il nostro webserver è stato in grado di gestire l'input senza sanitizzarlo, eseguendo la query che abbiamo passato tramite input.
Un altro possibile modo per verificare o meno la presenza di vulnerabilità SQLi è utilizzare uno script automatizzato. In questo esempio utilizzeremo, solamente per scopi puramente accademici, il tool NMAP.
La distribuzione linux utilizzata è Kali, di Offensive Security. Aprendo un terminale e digitando

```
nmap -sV --script=http-sql-injection <target>
```

inserendo al posto di <target> l'indirizzo web che vogliamo testare. Otterremo un output di questo genere:

```
PORT   STATE SERVICE
80/tcp open  http    syn-ack
| http-sql-injection:
|   Possible sqli for queries:
|     http://foo.pl/forms/page.php?param=13'%20OR%20sqlspider
|   Possible sqli for forms:
|     Form at path: /forms/f1.html, form's action: a1/check1.php
```

```
                    Fields that might be vulnerable:
|       f1text
|     Form at path: /forms/a1/../f2.html, form's action: a1/check2.php
                    Fields that might be vulnerable:
|_      f2text
```

dove f1text e f2text sono i campi vulnerabili.

4.2 Tautologia e commento di fine riga

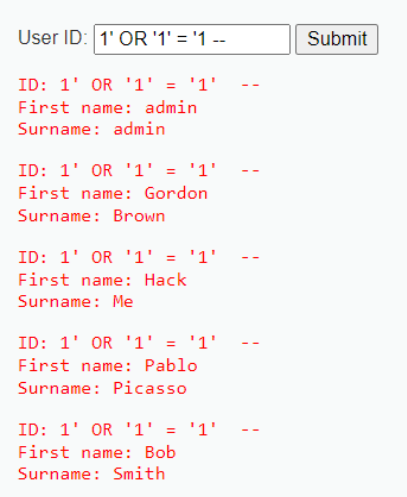
Un attacco basato sulla tautologia utilizza condizioni che sono sempre vere per manipolare la logica della query SQL. L'attaccante inserisce una condizione tautologica (sempre vera) in un campo di input per bypassare le verifiche di autenticazione o ottenere dati non autorizzati. Quando l'applicazione non valida correttamente l'input dell'utente e costruisce dinamicamente le query SQL concatenando gli input, un attaccante può inserire un'espressione che sarà sempre vera (tautologia). Il commento di fine riga (- - o #) viene utilizzato per ignorare il resto della query originale. Supponiamo di avere un form di login vulnerabile, dove l'input dell'utente viene concatenato direttamente nella query SQL:
inserendo

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

come username e qualsiasi cosa come password, la query risultante sarà:

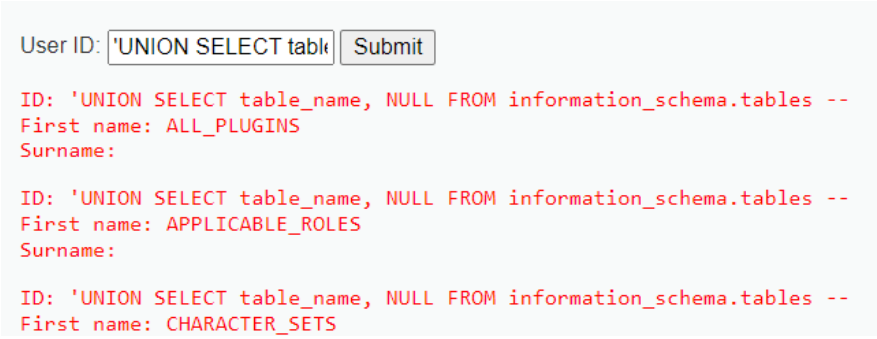
```
SELECT * FROM utenti WHERE nome_utente = '1'
         OR '1'='1' -- '
         AND password = 'minerva'; -- al posto di minerva puo'
                                   -- essere inserito qualsiasi cosa
```

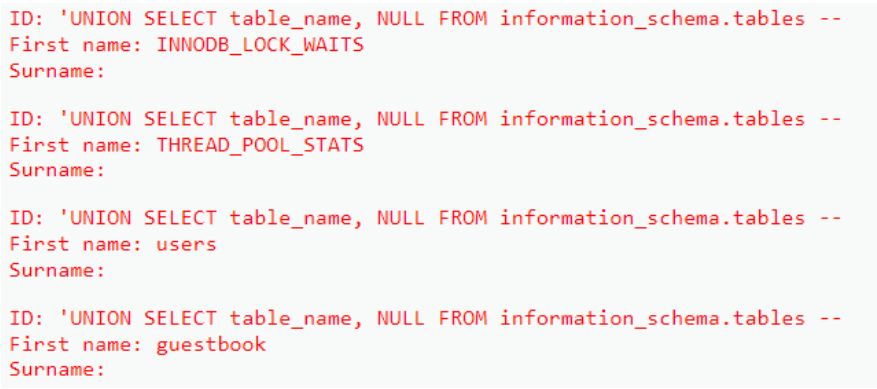
Questo restituirà tutti i record della tabella utenti poiché la condizione '1'='1' è sempre vera. Il commento - - ignora il resto della query, bypassando la verifica della password.



Inserendo opportuni comandi possiamo risalire alla struttura del database, così da effettuare un attacco mirato.

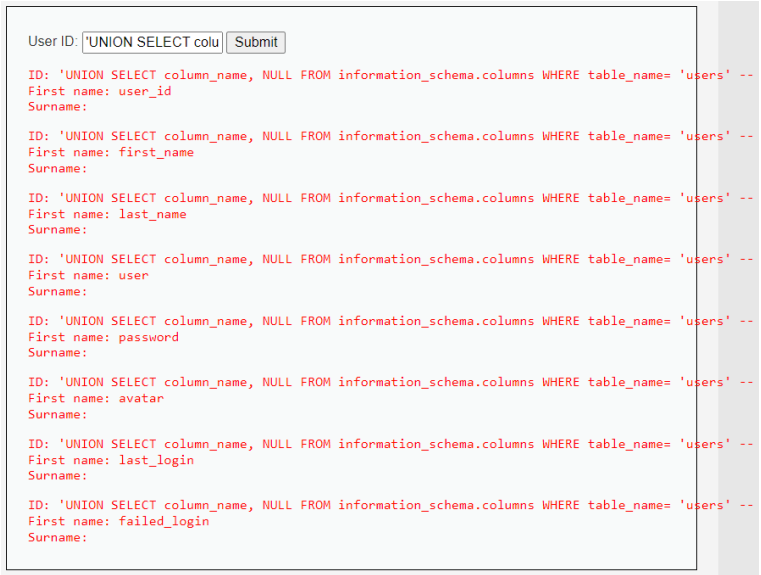
```
1 1'UNION SELECT table_name, NULL FROM information_schema.tables --
```





Con questo sistema possiamo inoltre vedere quali sono tutte le colonne della tabella users nel Database:

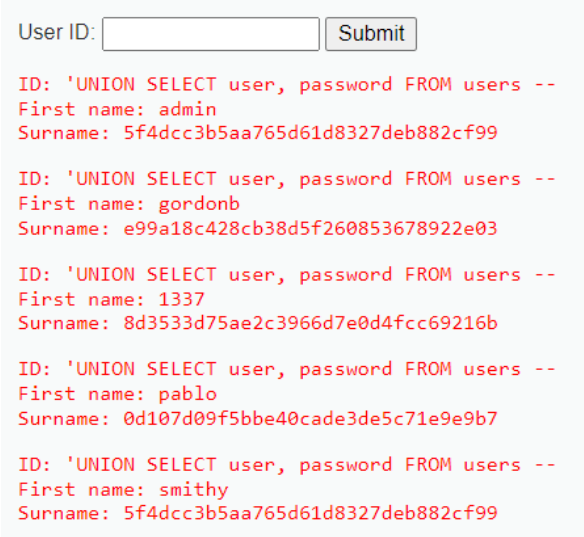
```
1 1'UNION SELECT column_name, NULL FROM information_schema.columns
2                             WHERE table_name= 'users' --
```



ed infine scoprire le password (in questo caso criptate in MD5, algoritmo attualmente classificato come non sicuro e soggetto a rainbow tables e collisioni). Inserendo infatti

```
1 'UNION SELECT user, password FROM users --
```

otteniamo il seguente output.



4.3 Query Piggybacked

Le query piggybacked si riferiscono all'inserimento di comandi SQL aggiuntivi oltre alla query prevista. Questo tipo di attacco permette all'attaccante di eseguire più comandi SQL in una singola richiesta. L'attaccante sfrutta la mancanza di sanitizzazione degli input per aggiungere ulteriori comandi SQL alla query esistente. Questo è possibile quando l'applicazione esegue concatenazione di stringhe per costruire query SQL. Supponiamo di avere lo stesso form di login vulnerabile, inserendo

```
1 admin'; DROP TABLE utenti; --
```

nel campo **username** e qualsiasi cosa nel campo **password**, la query risultante sarà:

```
1 SELECT * FROM utenti
2 WHERE nome_utente = '1';
3 DROP TABLE utenti;-- ' AND password = 'minerva';
```

Questa query esegue due comandi: la prima parte

```
1 SELECT * FROM utenti WHERE nome_utente = 'admin'
```

restituisce i risultati della query, mentre la seconda

```
1 DROP TABLE utenti;
```

elimina la tabella utenti.

5 Misure di sicurezza

5.1 Perpared statements

Utilizzare *prepared statements* per evitare la concatenazione diretta di input utente nelle query SQL. Un *prepared statement* è un modo di eseguire query SQL in modo sicuro ed efficiente. Si tratta di una funzionalità fornita dai moderni sistemi di gestione di database (DBMS) che permette di separare la logica SQL dai dati inseriti dall'utente. Questo approccio previene le vulnerabilità di SQL injection e migliora le prestazioni delle query SQL.

5.1.1 Preparazione della Query

Il client (ad esempio, un'applicazione PHP) invia la struttura della query SQL al server di database, dove viene compilata e ottimizzata. In questa fase, la query contiene dei segnaposto (placeholder) al posto dei valori effettivi.

```
SELECT * FROM utenti WHERE nome_utente = ? AND password = ?
```

Il client invia i valori effettivi per i segnaposto al server di database, che li sostituisce nei punti appropriati e esegue la query compilata. Poiché i valori sono trattati come dati e non come codice, non possono alterare la struttura della query.

5.1.2 Esempio in PhP dei prepared statements

```
<?php
include('db.php'); // Connessione al database

$username = $_POST['username'];
$password = $_POST['password'];

// Preparazione della query
$stmt = $conn->prepare("SELECT * FROM utenti
                        WHERE nome_utente = ? AND password = ?");
$stmt->bind_param("ss", $username, $password); // Collegamento parametri
$stmt->execute(); // Esecuzione della query
$result = $stmt->get_result(); // Ottenimento del risultato

if ($result->num_rows > 0) {
    echo "Login effettuato";
} else {
    echo "Nome utente o password errati";
}

$stmt->close(); // Chiusura del prepared statement
$conn->close(); // Chiusura della connessione al database
?>
```

5.1.3 Vantaggi dei Prepared Statements

1. **Sicurezza:** i prepared statements proteggono dalle SQL injection perché i valori degli input vengono trattati come dati e non come parte del codice SQL. Questo significa che anche se un attaccante cerca di inserire codice SQL malevolo, esso verrà trattato come un semplice valore.

2. **Efficienza:** le query preparate possono migliorare le prestazioni del database, soprattutto quando vengono eseguite ripetutamente con parametri diversi. La fase di preparazione e ottimizzazione della query viene eseguita una sola volta, mentre la fase di esecuzione può essere ripetuta con diversi valori.

3. **Manutenzione:** separare la logica SQL dai dati rende il codice più leggibile e manutenibile. È più facile identificare e correggere errori nelle query e aggiornare la logica SQL senza dover modificare l'intero codice dell'applicazione.

I prepared statements sono una best practice essenziale per la sicurezza e l'efficienza delle applicazioni che interagiscono con un database. Implementarli non solo protegge le applicazioni dagli attacchi di SQL injection, ma migliora anche le prestazioni e la manutenibilità del codice.

6 Conclusioni

La SQL injection rappresenta una delle principali minacce alla sicurezza delle applicazioni web, compromettendo le proprietà di confidenzialità, integrità e disponibilità dei dati. Implementare misure di sicurezza come i prepared statements, la validazione degli input e una corretta configurazione del database è essenziale per proteggere i sistemi da questi attacchi. La consapevolezza e la vigilanza continua sono fondamentali per mantenere un ambiente sicuro.

References

[1] Lawrie Brown William Stallings. *1.3 Security Functional Requirements*, chapter 5.4 SQL Injection Attacks. Pearson, 2015.

[2] Lawrie Brown William Stallings. *5.1 Computer Security: principles and Practice (3rd Edition)*, chapter 5.4 SQL Injection Attacks. Pearson, 2015.

[3] OWASP. Owasp top 10, 2021.
`https://owasp.org/Top10/A03_2021-Injection/`.

[4] Piotr Olma Eddie Bell. Script http-sql-injection, 2024.
`https://nmap.org/nsedoc/scripts/http-sql-injection.html`
Accesso: 05/07/2024. Ultimo aggiornamento 13/07/2024.

[5] Various. Sql injection, 2024.
`https://en.wikipedia.org/wiki/SQL_injection`
Accesso: 06/07/2024. Ultimo aggiornamento 13/07/2024.

[6] Kingthorin. Sql injection, 2023.
`https://owasp.org/www-community/attacks/SQL_Injection`
Accesso: 08/07/2024 . Ultimo aggiornamento 13/07/2024.

[7] Invicti. Sql injection: overview and attacks, 2022.
`https://www.invicti.com/blog/web-security/sql-injection-cheat-sheet/`
Accesso: 11/07/2024. Ultimo aggiornamento 13/07/2024.

Lovely written in LaTeX ♥