Homework di Sicurezza

SQL Injection

Flavio Corsetti, matricola 1997818

Indice

1. Introduzione

1.1 Traccia

Realizzare un attacco di SQL injection di tipo inband (ovvero stesso canale utilizzato per l'injection della query malevola e per ricevere i risultati), basato su input dell'utente (ovvero l'attaccante inietta i comandi SQL fornendo un input opportunamente costruito) e che utilizzi una o più delle seguenti modalità: Tautologia, commento di fine riga; query piggybacked.
Mediante l'injection di opportuni comandi mostrare che e' sia possibile compromettere almeno due delle proprietà CIA.

2. Descrizione

2.1 Introduzione al progetto

Ho realizzato un attacco di SQL injection di tipo in-band utilizzando un database MySQL e un sito web scritto in PHP. L'obiettivo era dimostrare la possibilità di compromettere almeno due delle proprietà della triade CIA (Confidenzialità, Integrità, Disponibilità) attraverso l'iniezione di comandi SQL opportunamente costruiti.
Sono riuscito a compromettere tutte e tre le proprieta' CIA.

Tipi di Attacco Utilizzati
Ho utilizzato tre tecniche principali di SQL injection:

- ***Tautologia***: Questa tecnica sfrutta condizioni sempre vere per bypassare i controlli di autenticazione.

- ***Commento di Fine Riga***: Utilizza commenti per ignorare il resto della query SQL originale.

- ***Query Piggybacked***: Inietta query aggiuntive che vengono eseguite insieme alla query originale.

2.2 Contesto dato al progetto

Il sito web è progettato per consentire agli utenti di **registrarsi**, effettuare il **login** e **cercare ricette** tramite una barra di ricerca nella homepage. Gli utenti possono inserire un ingrediente e visualizzare le ricette che contengono quell'ingrediente. Questo contesto didattico permette di mostrare chiaramente come un attacco di SQL injection possa compromettere la sicurezza del sistema.

2.3 Struttura progetto

L'attacco sfrutta le vulnerabilità lasciate da un **programmatore inesperto o disattento**, che ha omesso di implementare adeguati controlli di sicurezza.

Struttura Database

Il database **MySQL** utilizzato nel progetto è formato da due tabelle:

- **Utente**: Contiene le colonne '*username*' e '*password*'.
- **Ricette**: Contiene le colonne '*nome*' e '*descrizione*'.

Il file *sqli.sql* include la definizione delle tabelle e alcuni dati dummy, rendendo il database pronto all'uso per dimostrazioni pratiche.
Ho deciso di lasciare le *password* nel database *in chiaro*, senza utilizzare alcuna funzione di hash, per dimostrare in modo più evidente come l'attacco possa rivelare le password degli utenti.

Struttura del Sito Web

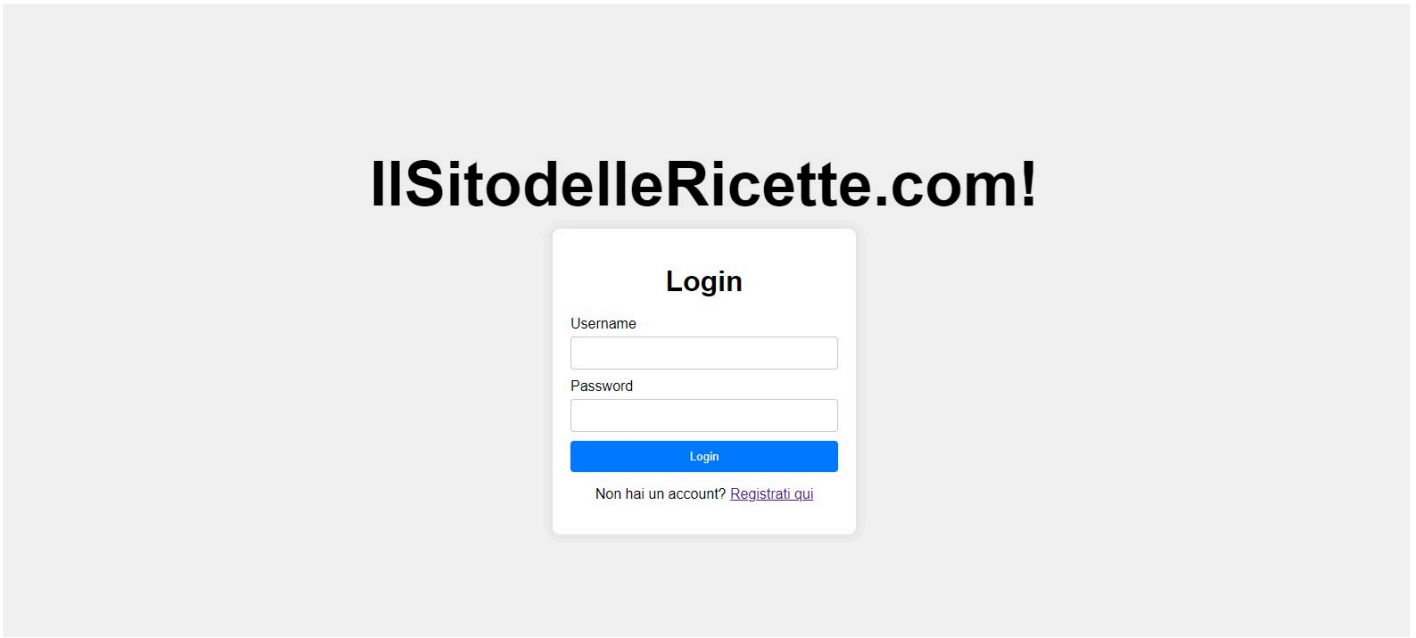
Il sito web è gestito tramite file **PHP** e include le seguenti pagine principali:

- **dbConnection.php**: Contiene il codice per la connessione al database MySQL.
- **login.php**: La pagina di login del sito.
- **homepage.php**: La homepage del sito, dove gli utenti possono cercare ricette.
- **register.php**: La pagina di registrazione per nuovi utenti.
- **logout.php**: Gestisce il logout degli utenti.

I file *logout.php* e *dbConnection.php* sono inclusi negli altri file PHP per implementare le rispettive funzionalità.

Alcuni screen delle pagine web

Homepage (A fine pagina viene mostrata in chiaro la query per motivi didattici)



pagina di login

3. Analisi

3.1 Analisi vulnerabilita’ lato programmatore

Andiamo ad analizzare il codice scritto dal programmatore che, in modo distratto o possibilmente anche voluto, inserisce nel codice vulnerabilita’ gravi e pesanti.

Codice per la gestione della richiesta di login:

```php
if ($_SERVER["REQUEST_METHOD"] == "POST") {

    //Preleviamo in modo diretto, senza controlli, il testo inserito nel form dall'utente.
    //Questo ci rende completamente vulnerabili all'SQLi visto che non controlliamo il testo che ci viene fornito in input
    $username = trim($_POST['username']);
    $password = trim($_POST['password']);

    if(!empty($password) && !empty($username)){ //controllo che siano stati inseriti dei valori nel form.

        $query = "SELECT * FROM user WHERE username = '$username' and password = '$password' ";
        $result = mysqli_query($dbConn, $query);

        //in questo caso  controlliamo solo se ci viene data almeno una riga come risultato
        if (mysqli_num_rows($result) > 0) {

            //non controlliamo i valori che ci vengono dati, creiamo la sessione con
            $_SESSION['username'] = $username;
            header("Location: homepage.php");
            exit();

        } else {

            $error = "Utente non trovato.";

        }

    }
}
```

File: *login.php*

a) Possiamo vedere che ad inizio gestione della richiesta Il **codice preleva direttamente i dati inseriti dall'utente** nel form **senza** alcuna **validazione** o **sanitizzazione**. Questo rende l'applicazione vulnerabile a SQL injection, poiché l'input dell'utente viene inserito direttamente nella query SQL senza alcun controllo. Punto critico:

```
$username = trim($_POST['username']);
$password = trim($_POST['password']);
```

b) Possiamo anche notare che la **query SQL viene costruita concatenando direttamente le variabili** $username e $password all'interno della stringa della query. Questo approccio è estremamente pericoloso:

```
$query = "SELECT * FROM user WHERE username = '$username' and password = '$password'";
```

c) L'ultima vulnerabilità nel codice riguarda la **gestione del risultato della query**. Infatti, il codice **verifica solo se la query restituisce almeno una riga**, senza controllare ulteriori dettagli. Questo approccio non è sufficiente per garantire la sicurezza dell'autenticazione. In particolare, si nota che come per token della sessione viene utilizzato l'username inserito dall'utente, non il risultato della query. Anche se nel progetto attuale non sono presenti, se fosse presente una query che utilizza il token dell'utente, un attaccante potrebbe inserire codice malevolo come nome utente (con l'ausilio di altri comandi SQLi). Ecco il codice:

```
if (mysqli_num_rows($result) > 0) {

        //non controlliamo i valori che ci vengono dati, creiamo la sessione con
        $_SESSION['username'] = $username;
        header("Location: homepage.php");
        exit();
}
```

Il progetto è stato progettato per **dimostrare le vulnerabilità di SQL injection**, quindi ho deciso di **lasciare le password nel database in chiaro**, senza utilizzare alcuna funzione di hash. Questa scelta è stata fatta intenzionalmente per *evidenziare in modo più chiaro come un attacco di SQL injection possa rivelare le password degli utenti*.
In un contesto reale, le password vengono sempre protette utilizzando funzioni di hashing sicure.

Codice per la gestione delle richieste nella homepage:

```
105     <div class="results">
106         <?php
107         if (mysqli_multi_query($dbConn, $query)) {
108             do {
109                 // Memorizza il primo set di risultati
110                 if ($result = mysqli_store_result($dbConn)) {
111                     while ($row = mysqli_fetch_row($result)) {
112                         echo '<div class="recipe">';
113                         echo '<div class="recipe-name">' . htmlspecialchars($row[0]) . '</div>'; // Prima colonna
114                         echo '<div class="recipe-description">' . htmlspecialchars($row[1]) . '</div>'; // Seconda colonna
115                         echo '</div>';
116                     }
117                     mysqli_free_result($result);
118                 }
119                 // Se ci sono più set di risultati, continua
120                 if (mysqli_more_results($dbConn)) {
121                     printf("-----------------\n");
122                 }
123             } while (mysqli_next_result($dbConn));
124         } else {
125             echo "Errore: " . mysqli_error($dbConn);
126         }
127         mysqli_close($dbConn);
128
129         die($query);
130
131         ?>
```

$query = "SELECT * FROM ricette WHERE descrizione LIKE '%$search_term%'";

a) Come prima il **codice preleva direttamente i dati inseriti dall'utente** nel form **senza** alcuna **validazione** o **sanitizzazione**.

b) Il codice utilizza la funzione **mysqli_multi_query** per eseguire query SQL. Questa funzione permette l'esecuzione di query multiple concatenate da un punto e virgola (;). Questo può essere **sfruttato per attacchi di tipo piggybacked**, dove un attaccante inietta una query aggiuntiva che viene eseguita insieme alla query originale.
La scelta di utilizzare **mysqli_multi_query** invece di **mysqli_query** non è chiara e potrebbe derivare da una svista del programmatore o dal riutilizzo di codice non adattato correttamente, o anche per funzionalita' non piu' aggiunte o rimosse.
[Ovviamente, in questo caso, l'uso di **mysqli_multi_query** è stato intenzionalmente aggiunto per poter mostrare un attacco piggybacked]

3.2 Analisi vulnerabilita’ lato attaccante

L'attaccante può individuare come punti di ingresso principali i form di input per il login e la pagina principale per la ricerca delle ricette per ingredienti. Sebbene il sito non mostri inizialmente alcuna vulnerabilità evidente, l'inserimento di un input con SQLi rivela la sua natura completamente insicura.

4. Attacco

4.1 Attacco alla pagina di Login

Ecco due possibili tipi di attacco alla *pagina di login*:

1) **Attacco con Tautologia**

   Inserendo come nome utente (o anche password) la seguente stringa:

   ' OR 1 = 1 -- ' (tutti gli apici e spazi compresi)

   andiamo ad inserire una condizione sempre vera all'interno della query, che quindi tornera' tutte le stringhe della tabella user. Questa e' la **query finale eseguita sul database:**

   SELECT * FROM user WHERE username = " OR 1 = 1 -- " and password = 'testo casuale'

   La parte **–** commenta tutto il codice dopo la condizione inserita dall'attaccante, quindi ***and password = 'testo casuale'*** non viene eseguita.

2) **Attacco con simboli di commento a fine riga**

   Inserendo come nome utente la seguente stringa:

   Flavio' -- ' (apici e spazi compresi)

   Utilizziamo il simbolo **--** per annullare la parte della query che richiede la password. L'attaccante può accedere a un account conoscendo solamente lo username. A differenza di un attacco con tautologia ***qui si accede a un account esistente di un altro utente***. Il malintenzionato può così accedere ai dati sensibili dell'utente e impersonarlo.

4.2 Attacco alla homepage

Ecco alcuni possibili attacchi alla *Homepage*:

1) **Attacco piggybacked**

   Inserendo nel form di input la stringa:

   xxx%' ; Select * FROM user; -- '
   (tutti gli apici e spazi compresi)

   Andiamo ad **eseguire una query illegittima dopo una query legittima**, questo grazie alla vulnerabilita' lasciata nel form diinput che ci permette di esegure due query una dopo l'altra.
   Le query sono divise dal simbolo ***;*** e la query che lo segue ci **permette di prelevare tutte le informazioni di ogni utente** nel sistema.

(Le password sono state lasciate volutamente in chiaro sul database)

2) **Secondo attacco piggybacked**

   Inserendo nel form di input la stringa:

   x%' ; DROP TABLE ricette -- '
   (tutti gli apici e spazi compresi)

   Andiamo ad **eseguire una query illegittima dopo una query legittima**. La seconda query permette all'attaccante di eliminare dal database la tabella ricette, **tabella fondamentale per il funzionamento del sito**.

3) **Terzo attacco piggybacked**

   Inserendo nel form di input la stringa:

   x%' ; UPDATE ricette SET descrizione = 'Un classico della cucina Francese con lumache, panna, tonno e paprika.' WHERE nome = 'Spaghetti alla Carbonara' -- '

   Andiamo ad **eseguire una query illegittima dopo una query legittima**. La seconda query permette all'attaccante di modificare i dati all'interno della tabella ricette, **il dato che l'utente ritiene fidato e' stato alterato dall'attaccante**.

4.3 Capire come e cosa attaccare

Fino ad ora abbiamo dato per scontato che l'attaccante conoscesse già i nomi delle tabelle all'interno del database. Tuttavia, in attacchi reali, l'attaccante non conosce la struttura e i nomi degli attributi e delle tabelle nel database. In questi casi, **una serie di comandi può essere eseguita tramite query per ottenere dal database** le **informazioni** necessarie a conoscere la sua struttura.

Questi comandi variano a seconda del tipo di DBMS utilizzato. Pertanto, partiamo dal presupposto che l'attaccante conosca il tipo di database usato nel servizio.

1) **Capire come viene gestito il risultato di un input specifico**

   L'attaccante deve capire quante colonne vengono gestite dalla parte di codice php che riceve e formatta il risultato della query. Questo serve per non ricevere errore dalla pagina web:

   mozzarella%' ; SELECT 1,2 FROM dual -- '

   Questa query, non ritornando errore, ci fa capire che il codice gestisce due colonne dal risultato della query.
   [Provando con ***1*** *o* ***1,2,3*** il sito non torna nulla, quindi questo ci fa capire il funzionamento del codice php]

2) **Ricevere i nomi delle tabelle**

   Dopo aver capito con quale formattazione il codice gestisce il risultato possiamo passare alla parte per trovare i nomi delle tabelle del database:

   mozzarella%' ; SELECT TABLE_NAME, TABLE_SCHEMA FROM information_schema.tables -- '

   Questa query tornera' il nome di tutte le tabelle nel database.

5. Risultati

Abbiamo analizzato la struttura del codice e le sue vulnerabilità, identificando gli attacchi che un attaccante può eseguire sui dati del database. I tipi di **attacchi utilizzati** sono stati **tautologia, commento di fine riga e piggybacking**. Abbiamo constatato che è **possibile** *prelevare illegittimamente tutti i dati degli utenti dal database*, *accedere con l'account di un altro utente* e anche *accedere senza utilizzare un account*. Questi tre attacchi violano la **Confidenzialità** e anche l'Autenticità, poiché permettiamo di impersonare un altro utente. Inoltre, abbiamo visto che è **possibile** *eliminare le tabelle dal database* e *modificare i dati in esse contenuti*. Questi attacchi compromettono la **Disponibilità** (eliminando tabelle fondamentali per il funzionamento del sito) e **l'Integrità** (modificando dati che l'utente considera affidabili).

In sintesi, abbiamo violato tutte e tre le proprietà del modello CIA: **Confidenzialità, Integrità e Disponibilità.**