Second Order SQL Injections

Andrew R. Darnall
A.A. 2022-2023
Matricola: 1000026223

June 20, 2023

Contents

Chapter 1

Intruduzione

1.1 Le iniezioni sql

La prima iniezione sql fu performata dal ricercatore e hacker James Forristal nel lontano 1998, egli publico' tale scoperta nel rinomato (ormai non piu') Phrack magazine notevole rivista elettronica del mondo della CyberSicurezza.
L'attacco consisteva nello sfruttare i campi del form di input della web app, input che essa manda direttamente al backend senza alcun controllo, mediante metodo HTTP POST.
Dunque in tal modo si possono direttamente mandare query sql al backend, senza alcuna protezione il che e' altamente problematico, in particolar modo qualora si conoscesse gia' la struttura della base di dati, si potrebbe ad esmpio cancellare una intera tabella o peggio, l'intera base di dati.
Un altro esempio basilare di questa iniezzione sql 'classica' e' quella che si basa su tautologie per bypassare il form di login di una web app. Una tautologia e' una proposizione che in un sistema formale, in questo caso la logica proposizionale, ha sempre la stessa assegnazione di verita', ovvero True.
Dal momento della sua scoperta il numero di attacchi basati sulla iniezione sql, ha mantenuto una posizione persistente nella lista 'OWASP Top 10 attacks'.
Nel corso degli anni sono emersi altre tipologie di attachi sql injection tra cui:

- Blind sql injection
- Time Based sql injection
- Union based sql injection
- Error based sql injection
- Out of bounds sql injection
- Second order sql injection

La lista non e' esaustiva in quanto l'argomento delle iniezioni sql e' in costante evoluzione poich lungo gli anni vengono scoperti altri tipi di attachi.

1.2 Attacco

Tutt'ora l'sql injection resta tra gli attacchi pi popolari che esistono, come mostrato da OWASP.

Esempio di una tautologia in SQL:

```
1
2     -- Questa interrogazione sara' sempre vera, a
          prescindere dalla presenza del record all'interno del
           db
3     SELECT *
4     FROM Users
5     WHERE Username = 'user_input_username' AND Password = '
          user_input_password' = '' OR '1' = '1';
```

Esempio di codice backend vulnerabile all'exploit

```
1
2     // Connessione al database
3     $conn = mysqli($server, $username, $password, $db);
4
5     $uname = $_POST['username'];
6     $passwd = $_POST['password'];
7
8     // Interrogazione vulnerabile a iniezione sql 'classica'
9     $sql = "SELECT * FROM Users WHERE Username = '$uname'
          AND Password = '$passwd'";
10    $result = $conn->query($sql);
11
12    if($result == true) {
13        // ... Login code
14    } else {
15        // Error handling code
16    }
17
18    $conn->close();
```

Nel momento in cui il codice del backend, php in questo caso, crea la stringa che usera' per l'interrogazione SQL, usando la tautologia '1' = '1', la stringa di interrogazione assumera' la forma della interrogazione del primo esempio.

Una volta che cio' avviene l'utente malintenzionato otterra' l'accesso alla web app, avendo dunque violato la proprieta' di autenticazione.

1.3 Prevenzione

Per prevenire tale attacco, sono state introdotte le interrogazioni parametrizzate, dove associanto dei parametri alla interrogazione ed effettuando automaticamente sql character escaping, ovvero tramutando una stringa presa in input dall'utente direttamente in testo, quest'ultima cambiera' il risultato atteso dall'attaccante, restituendo un empty set come risultato della query.

Di seguito viene riportato il codice 'hardened':

```
1
2     // Connessione al database
3     $conn = mysqli($server, $username, $password, $db);
4
5     // Interrogazione harderdened
6     $stmt = $conn->prepare("SELECT * FROM Users where
          Username = ? and Password = ?");
7     $stmt->bind_param("ss",$_POST['username'],$_POST['
          password']);
8     $stmt->execute();
9
10    $result = $stmt->get_result();
11
12    $stmt->close();
13
14    if($result == true) {
15        // ... Login code
16    } else {
17        // Error handling code
18    }
19
20    $conn->close();
```

Con questo codice 'irrobustito' mediante l'uso di interrogazioni parametrizzate, implementate con i prepared statements, l'utente malevolo non sara' piu' in grado di eseguire, tramite la superfice di attacco dell'input form, l'attacco di sql injection dunque lasciando integra la proprieta' di autenticazione (per conoscenza).

Chapter 2

Second order sql injections

L'argomento principale del mio progetto e' la iniezione sql di secondo ordine, nota anche come "stored sql injection".
Tale iniezione differisce da quelle del primo ordine in quanto quest'ultime vengono prontamente eseguite alla conversione della stringa in interrogazione sql, tuttavia le stored sql injections, come intuibile dal nome, vengono 'ingerite' dalla base di dati per venire eseguite in un secondo momento.
Questa particolare iniezione e' una diretta conseguenza della prevenzione della iniezione 'classica', ovvero, facendo l'escaping dei caratteri sql, e' possibile iniettare una query che verra' successivamente eseguita.
Cio' accade in quanto gli sviluppatori della web app hanno riposto parzialmente fiducia nell'input ottenuto dall'utente, ovvero non parametrizzando tutte le interrogazioni successive effettuate con i dati presi dalla base di dati.

2.1 Attacco

Di seguito viene riportato il codice esempio di una web app vulnerabile ad una iniezione di secondo ordine:

```
1
2     require_once('db_connect.php');
3
4                 $conn = new mysqli($servername, $username,
                      $password, $database);
5
6                 // I know that this works only temporairly,
                      it would need to be updated at every
                      login once the session
7                 // expires, but building a fully developed
                      web app goes beyond the scope of the
```

```
    project
// Thus this will suffice for a live demo
    where the session id doesn't expire
$sessionid = session_id();

$sql = "select username from Sessions where
    session_id = '$sessionid'";
$result = $conn->query($sql);

if($result == true) {

    $row = $result->fetch_assoc();
    $username = (string)$row['username'];
    // Questo e' il passo in cui lo
        sviluppatore e' stato negligente e
        dunque ha preso input potenzialmente
        nocivo
    // Immettendolo direttamente in una
        query, che verra' eseguita a favore
        dell'attaccante
    $sql2 = "select username, email, address
         from Users where username = '
        $username'";
    $result2 = $conn->query($sql2);

    if($result2 == true) {

        $row2 = $result2->fetch_assoc();
        echo "<tr><td>" . $row2['username']
            . "</td><td>" . $row2['email'] .
            "</td><td>" . $row2['address'] .
            "</td></tr>";

    } else {
        echo "<p style=\"color: red;\">
            Internal query error </p>";
    }

} else {
    echo "<p style=\"color: red;\"> External
         query error </p>";
}

$conn->close();
```

L'attacco che ho eseguito, basandomi su questo codice, consiste nel registrare un nuovo utente, chiamandolo: (michele' or username = 'admin), con una password qualsiasi
Dopo essermi registrato con successo, mi basta richiedre le mie informazioni

affinche l'attacco venga eseguito, duqnue ho il controllo dell'esecuzione dell'attacco. In questo caso, credendo sicuro l'input che prende dalla base di dati, in quanto e' stato effettuato un sql character query esacping al momento dell'immissione dei dati, verra' prima effettuata una query che richiede il nome utente in base al session id che gli e' stato associato al momento di login, successivamente il nome utente ottenuto verra' inserito (senza sql character escaping) nella seconda interrogazione dove interrogo il database per ottenere le informazioni dell'utente, tuttavia, non trovando il nome utente 'michele' la query che verra' eseguita un'altra, ovvero verra' eseguita una query che otterra' le informazioni di un'altro utente, nonche' l'admin (in questo caso), violando dunque la proprita' di privacy.

Nel caso di questo tipo di attacco, la proprieta' di sicurezza violata e' la privacy dell'utente, in questo caso l'admin ma in base alla iniezione puo'essere anche quella di qualsiasi altro utente.
Questo e' solo un esempio, ma la stessa vulnerabilita' sfruttata con una iniezione diversa avrebbe potuto recare danni seri ad una azienda, possibilmente provocando un drop della tabella o peggio, dell'intero database.

2.2 Prevenzione

Per prevenire gli attacchi di second order sql injection occorre usare nuovamente le query parametriche e in generale, usarle ovunque nel proprio codie insieme a input saitization, riducendo cosi' la possibilita' di un attacco sql injection sulla web app.

Di seguito viene riportato il codice ulteriormente 'irrobustito':

```
1
2                   require_once('db_connect.php');
3
4                   $conn = new mysqli($servername, $username,
                        $password, $database);
5
6                   $sessionid = session_id();
7
8                   $sql = "select username from Sessions where
                        session_id = '$sessionid'";
9                   $result = $conn->query($sql);
10
11                  if($result == true) {
12
13                      $row = $result->fetch_assoc();
14
15                      $username = (string)$row['username'];
16
```

```
17                  // Pathced up code
18                  $stmt = $conn->prepare("select username,
                        email, address from Users where
                        username = ?");
19                  $stmt->bind_param("s",$username);
20                  $stmt->execute();
21
22                  $result2 = $stmt->get_result();
23
24                  $stmt->close();
25
26                  if($result2 == true) {
27
28                      $row2 = $result2->fetch_assoc();
29                      echo "<tr><td>" . $row2['username']
                            . "</td><td>" . $row2['email'] .
                            "</td><td>" . $row2['address'] .
                            "</td></tr>";
30
31                  } else {
32                      echo "<p style=\"color: red;\">
                            Internal query error </p>";
33                  }
34
35              } else {
36                  echo "<p style=\"color: red;\"> External
                        query error </p>";
37              }
38
39              $conn->close();
```

In questa versione del codice, invece, quando l'utente malevolo tentera' di effettuare la query per attivare il codice sql iniettato precedentemente, non acra' successo e le informazioni dell'utente vittama (in questo esempio l'admin) resteranno private e dunque non riuscira' a violare la proprieta' della privacy.

Chapter 3

Conclusione

3.1 Errare umanum est, perseverare autem diabolicum

Dopo i svariati esperimenti, tutti condotti su 'software' da me creato, per cui ho sempre abuto il permesso di effettuare attacchi, concludo che per prevenire le iniezioni sql 'classiche' e di secondo ordine, e' sufficiente implementare in maniera esaustiva query parametrizzate insieme a input sanitization dove si rimuovono tutti i caratteri potenzialmente nocivi alla buisness logic della web app. Tuttavia questa soluzione non risolve tutte le tipologie di attachi sql injection ne tanto meno altri tipi di attachi sulle web app, come ad esempio il Cross Side Scripting o la Code Injection, tali attacchi possono infatti violare le stesse proprieta' di sicurezza che verrebbero violate da parte degli attachi sql sopracitati.
Tuttavia essendo l'ingegneria del software un settore altamente complesso, risulta facile farsi sfuggire il trattamento insicuro dell'input e delle query usate da parte del software, dunque come spesso ripetuto durante il corso di Internet Security, un fattore altamente discriminante sulla violazione o meno di una proprieta' di sicurezza, dipende dal fattore umano.
Risulta fondamentale (per gli ingegneri del software) attenersi sempre a tutte le pratiche standard per la prevenzione di tali attacchi e tenersi costantemente aggiornati in merito seguendo le linee guida di autorita' esperte in materia, nonche' OWASP.