Second Order SQL Injections

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

21 giugno 2023

Indice

Capitolo 1

Introduzione

1.1 Le iniezioni sql

La prima iniezione sql fu performata dal ricercatore e hacker James Forristal nel lontano 1998, egli publicò tale scoperta nel rinomato (ormai non più) 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 è altamente problematico, in particolar modo qualora si conoscesse già 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' è quella che si basa su tautologie per bypassare il form di login di una web app. Una tautologia è una proposizione che in un sistema formale, in questo caso la logica proposizionale, ha sempre la stessa assegnazione di verità, 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 di 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 è esaustiva in quanto l'argomento delle iniezioni sql è in costante evoluzione poichè lungo gli anni vengono scoperti altri tipi di attachi.

1.2 Attacco

Tutt'ora la categorida delle sql injections resta tra le più popolari che esistono, come mostrato da OWASP. L'Open Worldwide Application Security Project è un progetto open-source che ha l'obiettivo di realizzare linee guida, strumenti e metodologie per migliorare la sicurezza delle applicazioni.

L'esempio di attacco 'classico' dove si immettono le query direttamente nel database tramite l'input form, basato su tautologie
sfrutta l'interprete di commandi SQL per cui la query

1.2.1 Esempio 1.1

```
1
2     SELECT *
3     FROM Users
4     WHERE Username = 'user_input_username'
5     AND Password = 'user_input_password' = 'something' OR '1' = '1';
```

riporterà sempre il valore di verità True, ovvero un risultato sempre diverso dall'insieme vuoto.
Nell'esempio 1.1, la parte dell'input fornito dall'utente è la stringa " something' OR '1' = '1"
stringa che appena inserita nella query, verrà eseguita.
L'effetto di questa iniezione è il bypass della procedura di login violando, in questo caso, la proprietà di sicurezza dell'autenticazione (basata su conoscenza).

Per rendere meglio l'idea, a seguito viene riportato il codice del backend

1.2.2 Esempio 1.2

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

```
15
16      $conn->close();
```

La vulnerabilità riportata nell'esempio 1.2 risiede nel codice sulla linea 7, dove si accetta input 'raw'.

1.3 Prevenzione

Per prevenire tale attacco, sono state introdotte le interrogazioni parametrizzate, dove associanto dei parametri alla interrogazione ed effettuando automaticamente l'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.

1.3.1 Esempio 1.3

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

Il rimedio proposto nell'esempio 1.3 consta nell'usare i prepared statements, come fatto nelle righe 4 e 5
dunque in questo caso l'attacco 'classico' sarà sventato, lasciando la proprietà di autenticazione integra.

Capitolo 2

Second order sql injections

L'argomento principale del progetto è la iniezione sql di secondo ordine, nota anche come "stored sql injection".
Tale iniezione differisce da quelle del primo ordine in quanto consta in una stringa ottenuta dall'input form per cui è stato fatto sql character escaping e successivamente depositata nella base di dati, per venire usata in query successive.
Le stored sql injections sono molto difficili da rilevare in quanto non vengono effettuate durante la fase di input da parte dell'utente.
Si deduce quindi che le iniezioni di secondo livello sono una diretta conseguenza di quelle di primo livello, tuttavia per poter eseguire tale iniezione con successo bisogna conoscere anticipatamente la struttura della base di dati e la parte del codice della web app che ci interagisce.
Dunque le iniezioni di secondo ordine sono un 'effetto indesiderato' della prevenzione di quelle di primo ordine, mostrando come l'argomento delle iniezioni sql è delicato e altamente polimorfo.

2.1 Attacco

2.1.1 Esempio 2.1

```
1
2               $conn = new mysqli($servername, $username, $password,
                    $database);
3
4               $sessionid = session_id();
5
6               $sql = "select username from Sessions where session_id = '
                    $sessionid'";
7               $result = $conn->query($sql);
8
9               if($result == true) {
10
```

```
11                  $row = $result->fetch_assoc();
12                  $username = (string)$row['username'];
13                  $sql2 = "select username, email, address from Users where
                        username = '$username'";
14                  $result2 = $conn->query($sql2);
15
16                  if($result2 == true) {
17
18                      // ...
19
20                  } else {
21                      // ...
22                  }
23
24              } else {
25                  // ...
26              }
27
28              $conn->close();
```

Nell'esempio 2.1 la vulnerabilita risiede nelle righe 12 e 13, dove per via della negligenza da parte dello sviluppatore, si usa input insicuro per una query, dove ne verrà eseguita un'altra. L'attacco mostrato nell'esempio 2.1 consiste nel registrare un nuovo utente, chiamandolo: (michele' or username = 'admin), con una password qualsiasi
La logica di buisness della web app consente agli utenti di vedere le proprie informazioni e di poterle modificare, e prendendo il nome utente 'iniettato', si effettua una query inattesa.
Dopo essermi registrato con successo con il nome utente malevolo, mi basta richiedre le mie informazioni affinchè l'attacco venga eseguito, mantenendo il controllo dell'esecuzione dell'attacco. In questo caso, credendo sicuro l'input che prende dalla base di dati, in quanto è stato effettuato un sql character query esacping al momento dell'immissione dei dati, verrà prima effettuata una query che richiede il nome utente in base al session id che gli è stato associato al momento di login, successivamente il nome utente ottenuto verrà 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 verrà eseguita una query che otterrà le informazioni di un'altro utente, nonchè l'admin (in questo caso), violando dunque la proprietà di privacy.

Un altro esempio di attacco di iniezione sql di secondo ordine è lo sfruttare il meccanismo (query) di cambio della password

2.1.2 Esempio 2.2

```
1
2       UPDATE users SET password = 'password_eve' WHERE username = ' ' OR 1 = 1
            -- ' AND password = 'actual_password_eve';
```

Nell'esempio 2.2 la stringa che è stata iniettata dall'utente malevolo è anch'essa quella dello username, ovvero " ' OR 1 = 1 – ", una tautologia.
In questo caso la password che avrà immesso l'utente malevolo (eve) diventerà

la password di tutti gli utenti presenti nella tabella della base di dati, vilando anche in questo caso la proprietà di autenticazione.

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 così la possibilità di un attacco sql injection sulla web app.

2.2.1 Esempio 2.3

```
1
2
3                   $conn = new mysqli($servername, $username, $password,
                        $database);
4
5                   $sessionid = session_id();
6
7                   $sql = "select username from Sessions where session_id = '
                        $sessionid'";
8                   $result = $conn->query($sql);
9
10                  if($result == true) {
11
12                      $row = $result->fetch_assoc();
13
14                      $username = (string)$row['username'];
15
16                      $stmt = $conn->prepare("select username, email, address
                            from Users where username = ?");
17                      $stmt->bind_param("s",$username);
18                      $stmt->execute();
19
20                      $result2 = $stmt->get_result();
21
22                      $stmt->close();
23
24                      if($result2 == true) {
25
26                          // ...
27
28                      } else {
29                          // ...
30                      }
31
32                  } else {
33                      // ...
34                  }
35
36                  $conn->close();
```

La sistemazione del codice dell'esempio 2.2, nell'esempio 2.3 è nelle righe 16 e 17. In questa versione del codice, invece, quando l'utente malevolo tenterà di effettuare la query per attivare il codice sql iniettato precedentemente, non

avrà successo e le informazioni dell'utente vittama (in questo esempio l'admin) resteranno private e dunque non riuscirà a violare la proprietà di sicurezza della privacy.

La prevenzione ideale contro le iniezioni di secondo livello è ancora motivo di dibattito nella comunità scientifica, tuttavia tra i vari metodi propsti negli anni, esiste un metodo molto promettente che potrebbe togliere l'onere della gestione appropriata dell'input agli ingegneri del software potenzialmente negligenti.
Tale metodo, riportato in questo articolo dell'IEEE, propone l'utilizzo di un middleware facilmente configurabile e portabili con un overhead non troppo gravante e con risultati promettenti.
Il middleware si basa su Instruction Set Randomization, tale middleware verrà posto tra il server, un proxy server e il database.
L'ISR è ottenuta creando una definizione di inisemi di codewords fidate del linguaggio SQL in base all'obbiettivo della interrogazione, ad esempio una Query ha come codewords fidate: SELECT, FROM, WHERE, ORDER BY, GROUP BY, HAVING, UNION, OR, AND tuttavia non avrà come codeword fidata: ALTER. Il sistema genererà un numero pseudo-randomico di 3 cifre, r, e calcolerà R = F(r,K), K è la chiave condivisa tra il proxy server (MySQL nel caso del paper) e il middleware randomizzante ed F() è la funzione randomizzante. Per evitare che l'attaccante impersoni una query leggittima, il numero r verrà pre-posto alla codeword mentre il numero R verrà anteposto alla codeword, ottenendo così r+CODEWORD+R, usando r come salt e R come un checksum. Dopo che il middleware randomizzante avrà inoltrato la query al proxy server e quest'ultimo avrà separato il salt r e ricalcolato R' = F(r,k), comparerà R con R' e qualora fossero diverse, il proxy avrà la certezza che è una codeword non autorizzata e mandata dall'attaccante.

Sebbene questo paper sia brillante, esso non specifica come gli autori abbiano sintetizzato gli insiemi di codewords fidate.

Capitolo 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 avuto il permesso di effettuare attacchi, concludo che per prevenire le iniezioni sql 'classiche' e di secondo ordine, l'euristica da adottare è il controllo e la sanitizzazione di tutto l'input preso dagli utenti, anche quando quest'ultimo viene ripreso dalla base di dati stessa, è dunque buona implementare in maniera esaustiva query parametrizzate. 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 proprietà 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 proprietà 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 autorità esperte in materia, nonchè OWASP.