Command Injection | TryHackMe (THM)

Aircon Follow 8 min read · May 13, 2022

53 2

Lab Access: https://tryhackme.com/room/oscommandinjection

Command Injection — It is an abuse of an application's behavior to execute commands on the operating system by using the same privileges as the program executing on a device.

- It remains one of the top ten vulnerabilities in the OWASP Framework.

```
It is also known as a “Remote Code Execution” (RCE) because an
attacker can mislead the application into executing a sequence of
payloads provided by the attacker without gaining direct access to
the system itself (i.e. an interactive shell).
```

Having said that, the webserver will continue to process and execute the code under the “**privileges**” and “**access controls**” of the person who is running the application.

- Furthermore, because of the ability to remotely execute code within an application, these vulnerabilities are frequently the most rewarding to an attacker because they **allow the attacker to actively communicate with the vulnerable system, allowing the attacker to read system or user files, data, and other similar things**.

```
command: whoami
- It will list the victim's account under which the application is
operating as an example of command injection.
```

[Question 1.1] Read me!

Answer: No answer is needed.

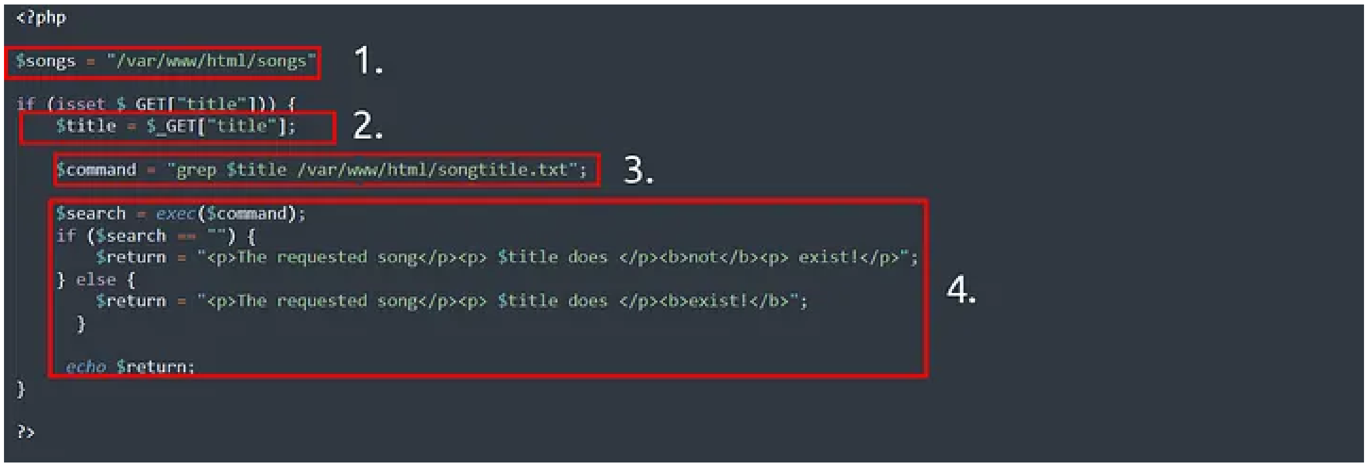
Task 2 Discovering Command Injection

This vulnerability exists because apps frequently use functions in programming languages such as **PHP**, **Python**, and **NodeJS** to transfer data to and make system calls on the machine's operating system.

```
**Example:** Taking input from a field and searching for an entry in a
file.
```

Code Snippet Example:

- This type of data is often saved in a database, and it is where an application collects user input to interact with the application's operating system.



```php
<?php

$songs = "/var/www/html/songs"

if (isset $_GET["title"])) {
    $title = $_GET["title"];

    $command = "grep $title /var/www/html/songtitle.txt";

    $search = exec($command);
    if ($search == "") {
        $return = "<p>The requested song</p><p> $title does </p><b>not</b><p> exist!</p>";
    } else {
        $return = "<p>The requested song</p><p> $title does </p><b>exist!</b>";
      }

     echo $return;
}

?>
```

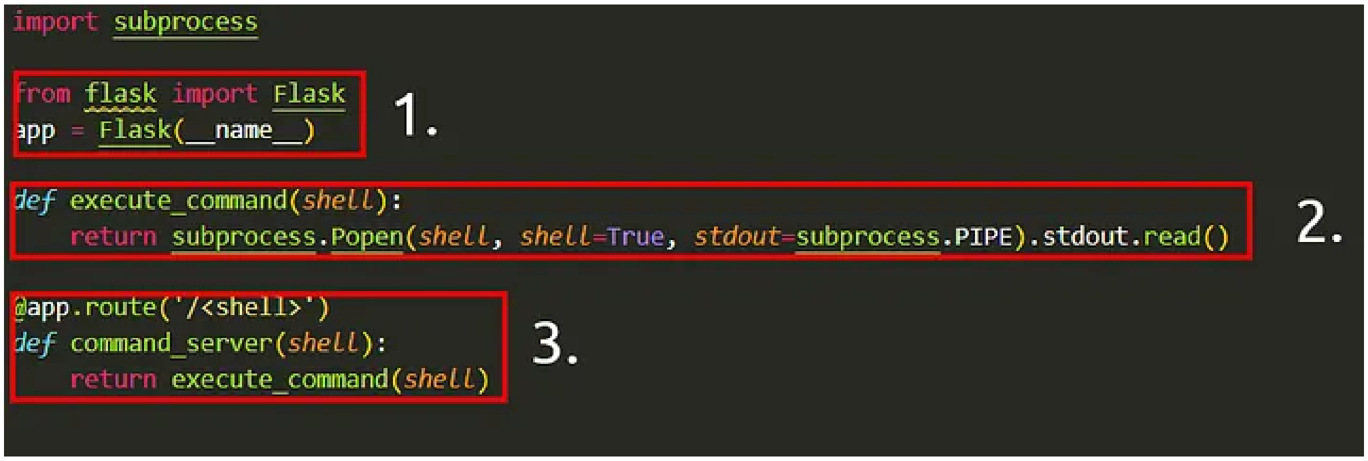
```
1)  The application stores MP3 files in a directory contained on the
operating system.
```

2) The user inputs the song title they wish to search for. The application stores this input into the **$title** variable.

3) The data within this **$title** variable is passed to the command **grep** to search a text file named *songtitle.txt* for the entry of whatever the user wishes to search for.

4) The output of this search of *songtitle.txt* will determine whether the application informs the user that the song exists or not.

An attacker could take advantage of this program by introducing their own commands for it to execute. Instead of using **grep** to search for an entry in songtitle.txt, the attacker could request that the application read data from a more sensitive file.



```
import subprocess

from flask import Flask
app = Flask(__name__)

def execute_command(shell):
    return subprocess.Popen(shell, shell=True, stdout=subprocess.PIPE).stdout.read()

@app.route('/<shell>')
def command_server(shell):
    return execute_command(shell)
```

1) The “flask” package is used to set up a web server

2) A function that uses the “subprocess” package to execute a command on the device

3) We use a route in the webserver that will execute whatever is provided. For example, to execute **whoami**, we'd need to visit http://flaskapp.thm/whoami

[Question 2.1] What variable stores the user’s input in the PHP code snippet in this task?

Answer: title

[Question 2.2] What HTTP method is used to retrieve data submitted by a user in the PHP code snippet?

Answer: GET

[Question 2.3] If I wanted to execute the `id` command in the Python code snippet, what route would I need to visit?

Answer: /id

Task 3 Exploiting Command Injection

Applications that use input from the user to supply system instructions with data can frequently result in unexpected behaviour.

Shell Operators:

1) **;** = functions similarly to "&&," in that it does not require the first command to be executed successfully, which means that **if the first command fails, the second command will still run**.

2) **&** = Because it is a background operator, the **command will continue to run and you will be able to perform other tasks** (means **able to run concurrently with the first command**)

3) **&&** = It signifies "**and**," that it will execute a second command after the first one has been successfully completed, implying that

```
the first command MUST BE SUCCESSFUL in order to run the second
command.
```

In most cases, Command Injection can be detected in one of two ways:

1. **Blind Command Injection —** When testing the payload, there is **no direct output from the application**, and the attack must analyze the application's behaviour to discover whether or not your payload was successful.
2. **Verbose Command Injection —** After you have tested a payload, **the attacker will receive an immediate response from the program**. For example, you can use the "**whoami**" command to see what user the application is running as, and the username will be displayed directly on the page.

Detecting Blind Command Injection:

- It must employ payloads, which will cause some time delay.
- If at all possible, force some output.

```
1) ping = The application will hang for x seconds, depending on how
many pings you specify.

2) > = It's a "redirector," which means we can take the output of a
command (such as "cat" to output a file) and redirect it somewhere
else. Once redirected, we can use "cat" to read the contents of the
newly generated file.

3) whoami = Print the user name that corresponds to the current
effective user id.

4) curl = It's a great approach to test because it sends and receives
data from and from an application in your payload.
```

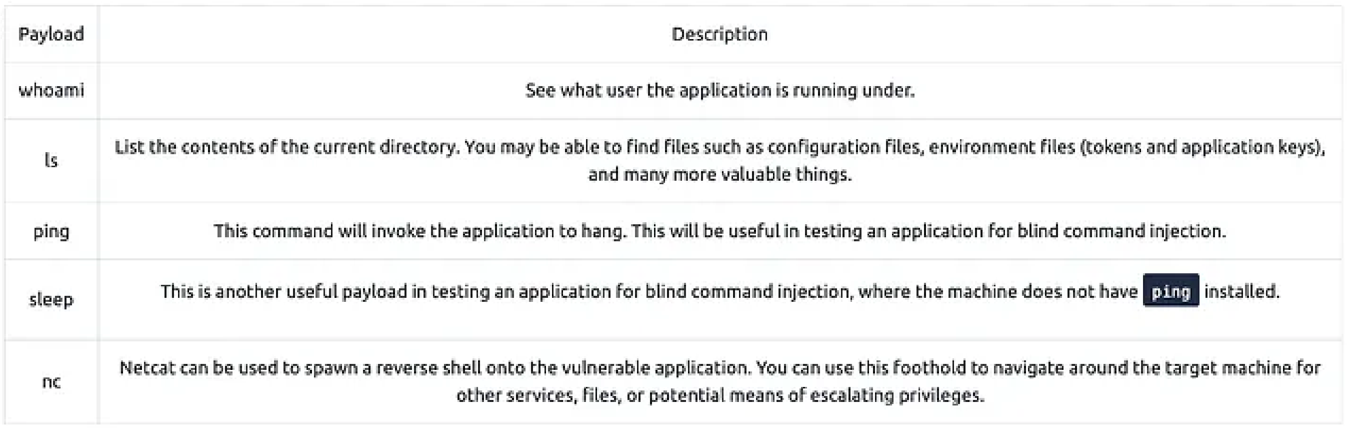
Detecting Verbose Command Injection:

- One of the simplest methods is because the application provides feedback or output on what is happening or being run.

```
ping or whoami is directly displayed on the web application
```

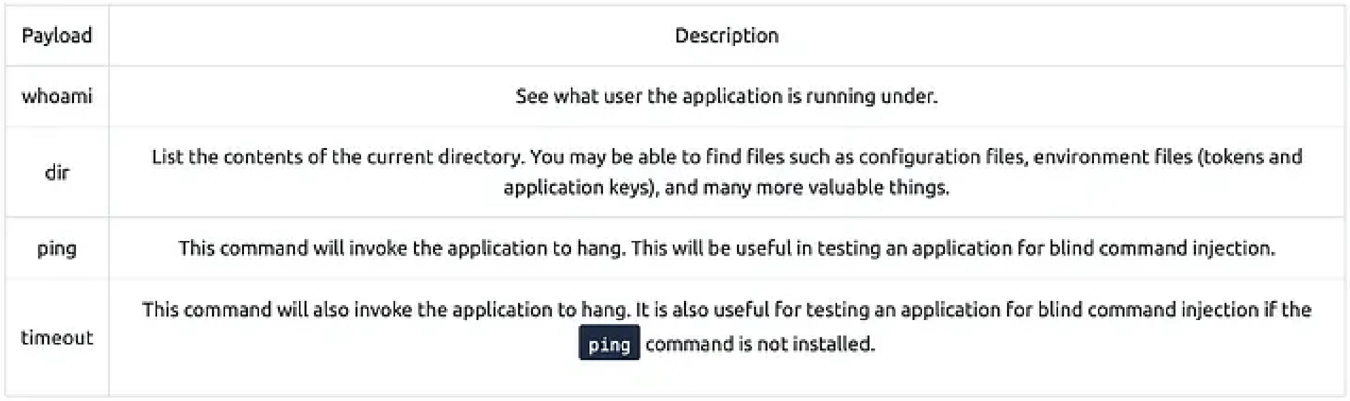
Useful Payloads:

- Linux



Payload	Description
whoami	See what user the application is running under.
ls	List the contents of the current directory. You may be able to find files such as configuration files, environment files (tokens and application keys), and many more valuable things.
ping	This command will invoke the application to hang. This will be useful in testing an application for blind command injection.
sleep	This is another useful payload in testing an application for blind command injection, where the machine does not have `ping` installed.
nc	Netcat can be used to spawn a reverse shell onto the vulnerable application. You can use this foothold to navigate around the target machine for other services, files, or potential means of escalating privileges.

- Windows



Payload	Description
whoami	See what user the application is running under.
dir	List the contents of the current directory. You may be able to find files such as configuration files, environment files (tokens and application keys), and many more valuable things.
ping	This command will invoke the application to hang. This will be useful in testing an application for blind command injection.
timeout	This command will also invoke the application to hang. It is also useful for testing an application for blind command injection if the `ping` command is not installed.

[Question 3.1] What payload would I use if I wanted to determine what user the application is running as?

Answer: whoami

[Question 3.2] What popular network tool would I use to test for blind command injection on a Linux machine?

Answer: ping

[Question 3.3] What payload would I use to test a Windows machine for blind command injection?

Answer: timeout

Task 4 Remediating Command Injection

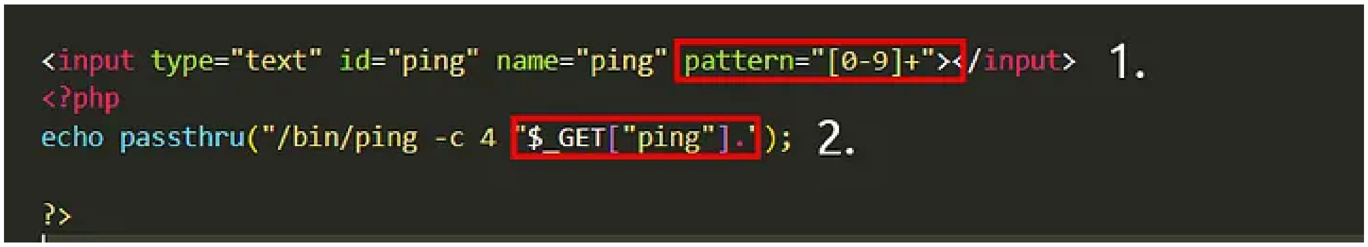
It can be avoided in a number of methods, ranging from using potentially **harmful functions or libraries** in a programming language to **filtering input without relying on user input.**

Vulnerable Functions:

- `Exec`
- `Passthru`
- `System`

In the example below, the program will **only accept and process numbers** entered into the form.

- This means that commands like "**whoami**" will not be executed.



```
<input type="text" id="ping" name="ping" pattern="[0-9]+"></input>   1.
<?php
echo passthru("/bin/ping -c 4 "$_GET["ping"].');   2.

?>
```

```
1. The application will only accept a specific pattern of characters
(the digits 0-9)

2. The application will then only proceed to execute this data which
is all numerical.
```

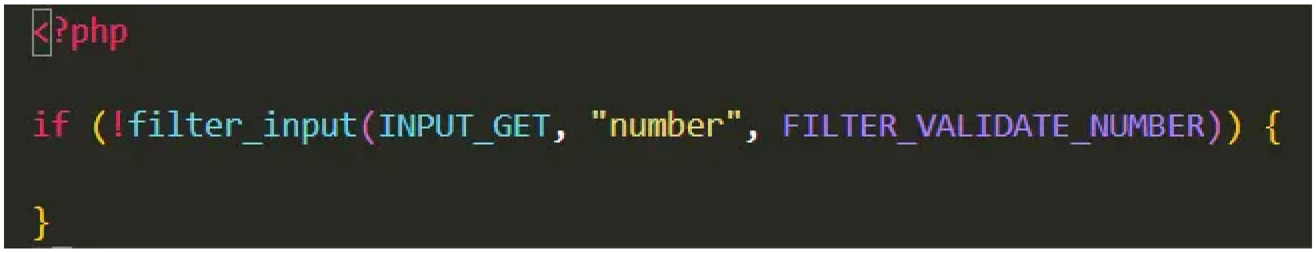
It should be noted that **any program that uses these functions without performing sufficient checks is vulnerable to command injection.**

Input Sanitisation:

- It is one of the most effective methods of prevention; it is a process of regulating the formats or types of data that a user can submit, which implies that by restricting the input character that anyone is allowed to input.

```
Example:
1) Only accept numerical data
2) Removes any special characters such as >, &, and /.
```

The following image demonstrates how the “**filter_input**” PHP function is used to determine whether or not data supplied via an input form is a number. **It must be invalid input if it is not a number.**



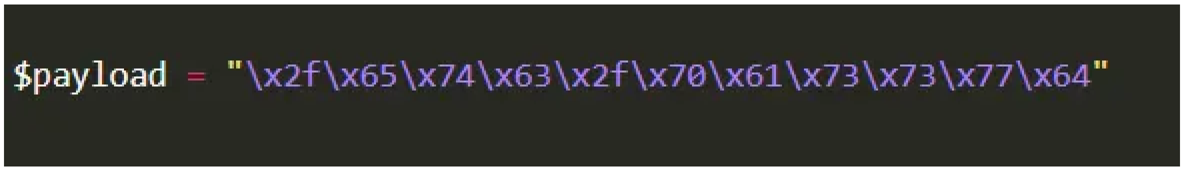
```php
<?php

if (!filter_input(INPUT_GET, "number", FILTER_VALIDATE_NUMBER)) {

}
```

Bypassing Filters:

These filters will limit you to specified payloads; however, we can overcome these filters by exploiting an application’s logic.

- For example, a program may remove quotation marks; we can achieve the same behaviour by using the hexadecimal value of this.

The example below demonstrates that when performed, even if the data provided is in a different format than expected, it can still be processed and produce the same result.



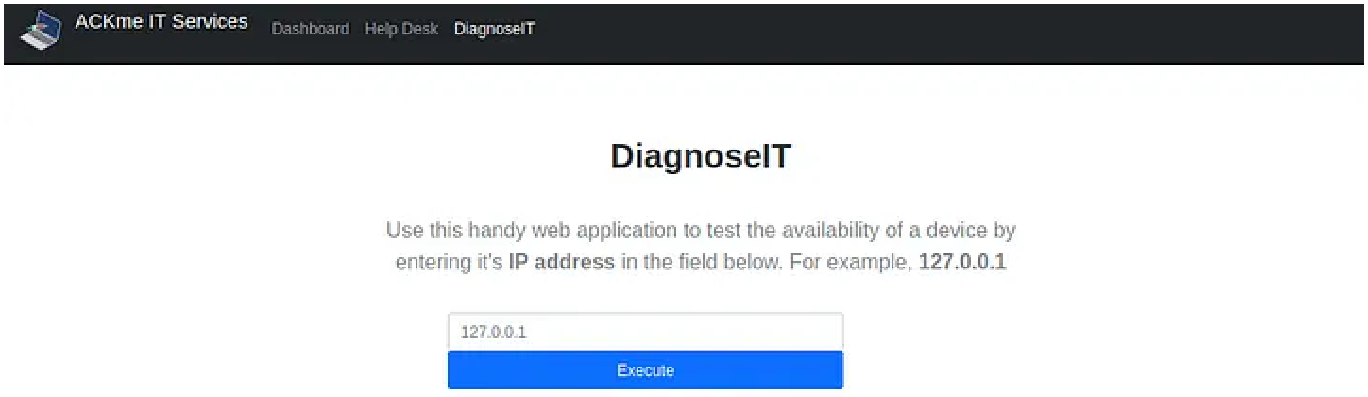
```php
$payload = "\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
```

[Question 4.1] What is the term for the process of "cleaning" user input that is provided to an application?

Answer: sanitisation

Task 5 Practical: Command Injection (Deploy)

It's interesting that the tests recommended "127.0.0.1" for testing localhost.

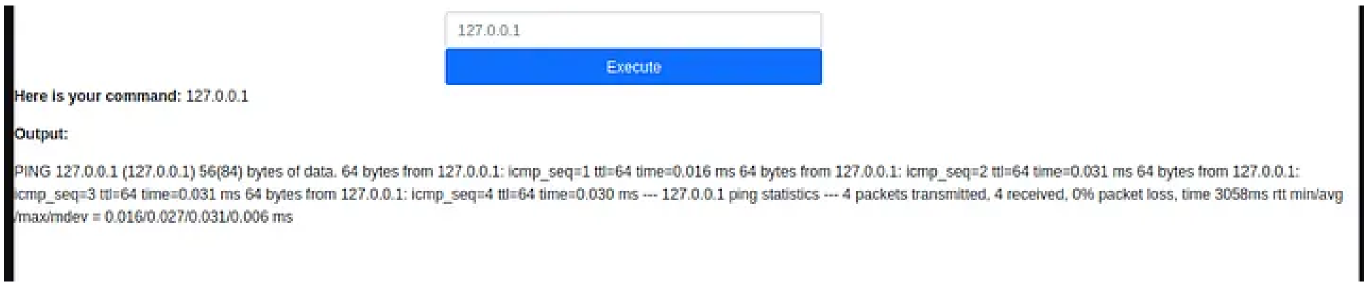


DiagnoseIT

Use this handy web application to test the availability of a device by entering it's **IP address** in the field below. For example, **127.0.0.1**

127.0.0.1

With the "localhost" test, certain findings were achieved.

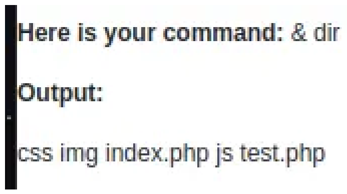

127.0.0.1

Here is your command: 127.0.0.1

Output:

```
PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data. 64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.016 ms 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.031 ms 64 bytes from 127.0.0.1:
icmp_seq=3 ttl=64 time=0.031 ms 64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.030 ms --- 127.0.0.1 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3058ms rtt min/avg
/max/mdev = 0.016/0.027/0.031/0.006 ms
```

“& dir” worked, indicating that the victim is using “**Windows**” rather than Linux.



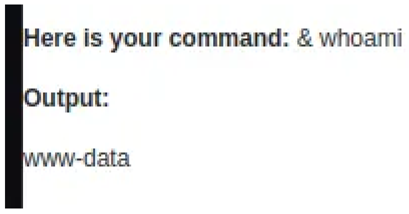
```
Here is your command: & dir

Output:

css img index.php js test.php
```

& dir

```
Command Injection Cheatsheet:
- https://github.com/payloadbox/command-injection-payload-list
```

[Question 5.1] What user is this application running as?

Since we know the victim is using “**Windows**,” we can use the “**&**” shell operator to find out what application is running.



```
Here is your command: & whoami

Output:

www-data
```

Answer: www-data

[Question 5.2] What are the contents of the flag located in /home/tryhackme/flag.txt?

Because the “**&**” operator is the vulnerable function and allows us to navigate around with it, and because we can access the “**dir**,” we can simply