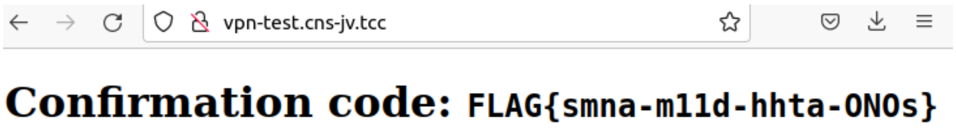
The Catch

CESNET Maritime Communications

Write up by moNNster

VPN access

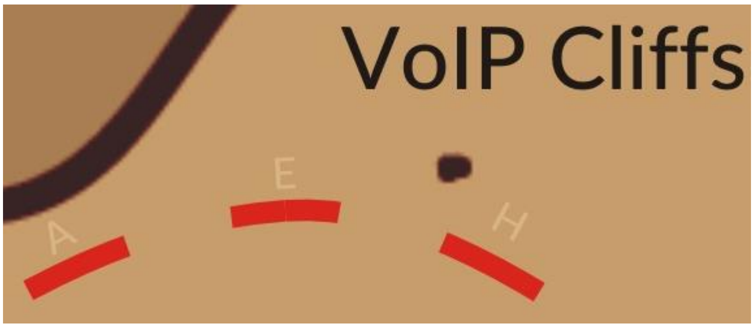
The only step required to complete this first task was to properly configure VPN connection and visit mentioned host.



Confirmation code: `FLAG{smna-m11d-hhta-ONOs}`

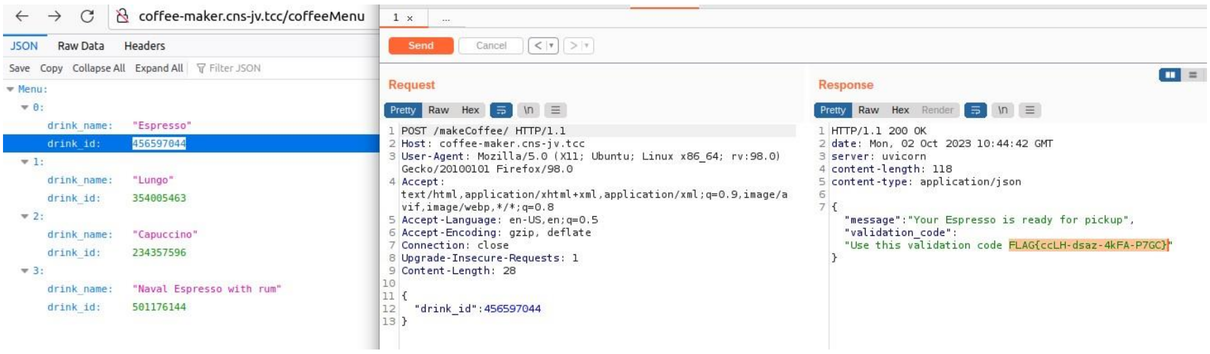
Treasure map

Upon careful inspection of the provided map image, one could notice the flag string is scattered along the red dashed line. It was as easy as following the path and concatenating all the characters.



Captain's coffee

The initial API URL returns a JSON message stating Coffeemaker is ready and suggests visiting /docs. Inspecting the source code there, another 2 URL's were discovered, one of which points to *openapi.json*. There all the API calls required to list the coffee options and make an Espresso were listed. As a further hint, Swagger was also mentioned so online documentation was also handy.



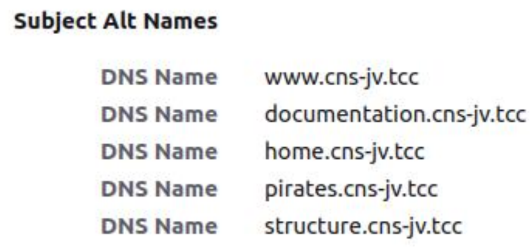
```
Menu:
  0:
    drink_name:   "Espresso"
    drink_id:     456597044
  1:
    drink_name:   "Lungo"
    drink_id:     354005463
  2:
    drink_name:   "Capuccino"
    drink_id:     234357596
  3:
    drink_name:   "Naval Espresso with rum"
    drink_id:     501176144
```

```
POST /makeCoffee/ HTTP/1.1
Host: coffee-maker.cns-jv.tcc
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:98.0)
Gecko/20100101 Firefox/98.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/a
vif,image/webp,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: close
Upgrade-Insecure-Requests: 1
Content-Length: 28

{
  "drink_id":456597044
}
```

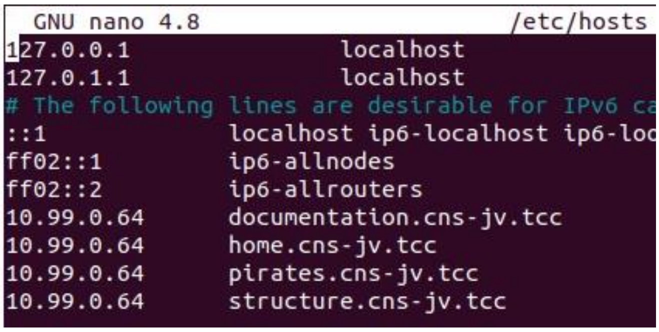
```
HTTP/1.1 200 OK
date: Mon, 02 Oct 2023 10:44:42 GMT
server: uvicorn
content-length: 118
content-type: application/json

{
  "message":"Your Espresso is ready for pickup",
  "validation_code":
  "Use this validation code FLAG{ccLH-dsaz-4kFA-P7GC}"
}
```

Ship web server

Here the first step was to inspect the certificate, where multiple SAN's are mentioned. Pointing those to the same IP address in hosts file was then enough to make them reachable.


Subject Alt Names

DNS Name	www.cns-jv.tcc
DNS Name	documentation.cns-jv.tcc
DNS Name	home.cns-jv.tcc
DNS Name	pirates.cns-jv.tcc
DNS Name	structure.cns-jv.tcc


```
  GNU nano 4.8                              /etc/hosts
127.0.0.1               localhost
127.0.1.1               localhost
# The following lines are desirable for IPv6 ca
::1             localhost ip6-localhost ip6-loc
ff02::1         ip6-allnodes
ff02::2         ip6-allrouters
10.99.0.64      documentation.cns-jv.tcc
10.99.0.64      home.cns-jv.tcc
10.99.0.64      pirates.cns-jv.tcc
10.99.0.64      structure.cns-jv.tcc
```

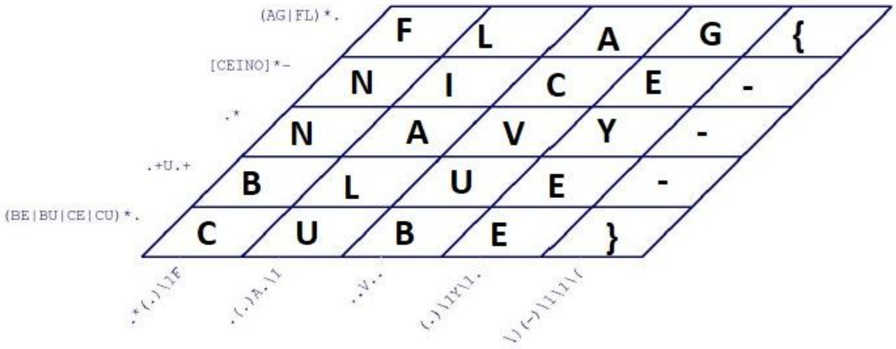
Next, each page contained one part of the flag encoded in base64 more or less in plain sight. *Home* had the first part under each profile page, *structure* and *pirates* showed their bits at the bottom of the page and while *documentation* was seemingly inaccessible, inspecting the source code and viewing */styles.css* revealed the last part.

Sonar logs

Here the first step was to covert all timestamps to one time zone, which was easily achieved with *pytz* python library. Once normalized it was necessary to sort the log entries and filter out events where objects were detected. Converting the hex values then produced what seemed like the flag string, but one character was apparently misplaced. Luckily enough it only took a bit of manual correction to fix that, and seemingly it could also be sorted by downgrading to an older version of the library according to one of the hints.

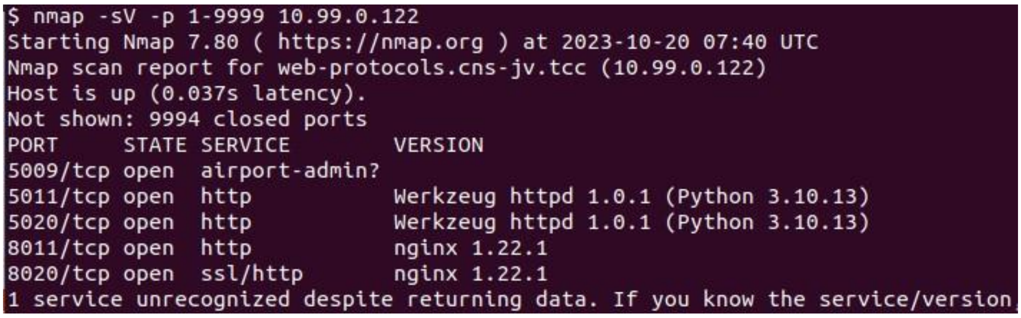
Regular cube

This was a classic regex crossword, just this time it was 3 dimensional:



Web protocols

This task suggests we look at web protocols on a given server, so scanning for available ports is a good first step.

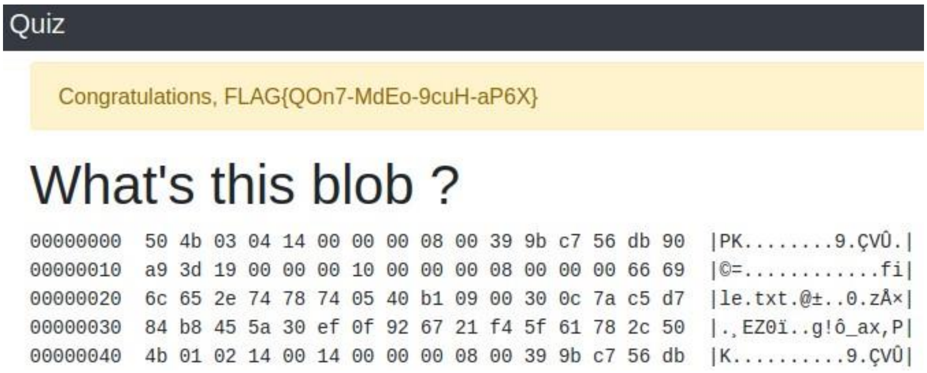


```
$ nmap -sV -p 1-9999 10.99.0.122
Starting Nmap 7.80 ( https://nmap.org ) at 2023-10-20 07:40 UTC
Nmap scan report for web-protocols.cns-jv.tcc (10.99.0.122)
Host is up (0.037s latency).
Not shown: 9994 closed ports
PORT     STATE SERVICE         VERSION
5009/tcp open  airport-admin?
5011/tcp open  http            Werkzeug httpd 1.0.1 (Python 3.10.13)
5020/tcp open  http            Werkzeug httpd 1.0.1 (Python 3.10.13)
8011/tcp open  http            nginx 1.22.1
8020/tcp open  ssl/http        nginx 1.22.1
1 service unrecognized despite returning data. If you know the service/version
```

Here the flag parts were hidden in a session cookie as base64 strings upon hitting on each of the ports using HTTP protocol, the only trick was for 5009 to downgrade to *HTTP/0.9*.

Apha-Zulu quiz

Not sure if there was any trick with this one, but it was possible to simply learn to identify few types of data chunks and pass the test to be provided with the flag. CyberChef could also help a bit, e.g. with its *Analyse hash* function.



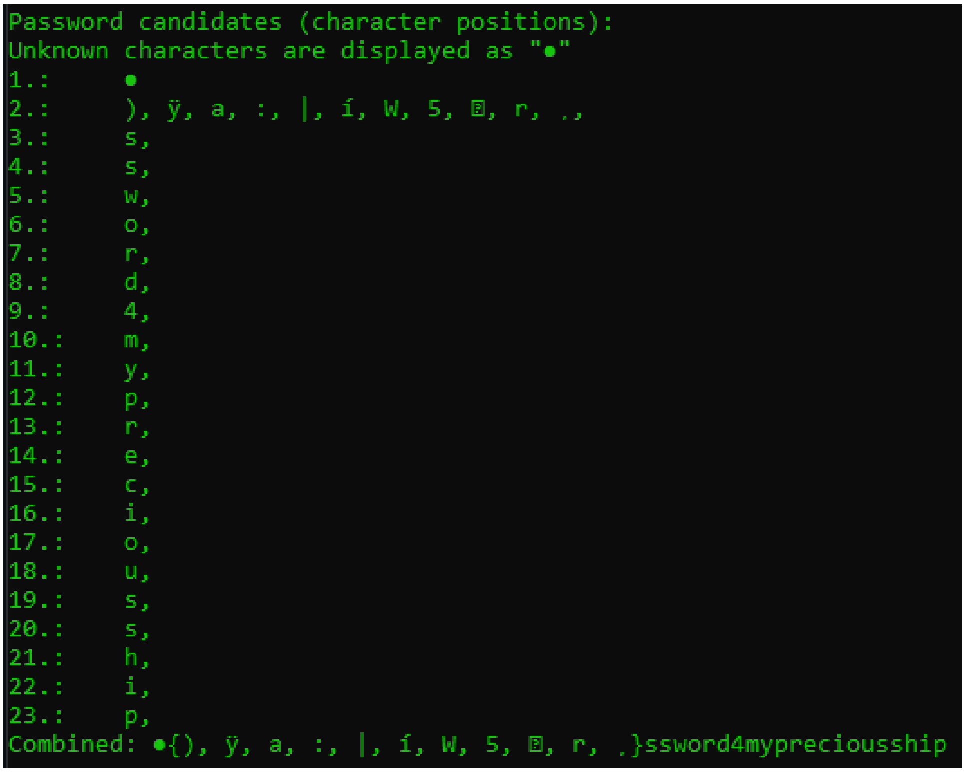
Congratulations, FLAG{QOn7-MdEo-9cuH-aP6X}

What's this blob ?

```
00000000  50 4b 03 04 14 00 00 00 08 00 39 9b c7 56 db 90  |PK........9.ÇVÛ.|
00000010  a9 3d 19 00 00 00 10 00 00 00 08 00 00 00 66 69  |©=............fi|
00000020  6c 65 2e 74 78 74 05 40 b1 09 00 30 0c 7a c5 d7  |le.txt.@±..0.zÅ×|
00000030  84 b8 45 5a 30 ef 0f 92 67 21 f4 5f 61 78 2c 50  |.¸EZ0ï..g!ô_ax,P|
00000040  4b 01 02 14 00 14 00 00 00 08 00 39 9b c7 56 db  |K..........9.ÇVÛ|
```

Captain's password

Provided the memory dump alongside a KeePass database, it was rather straight forward the aim here may be to try and exploit CVE-2023-32784 which allows recovering master key from memory even if KeePass workspace was locked. Quick web search suggested KeePass 2.X Master Password Dumper was designed to do exactly that, and upon running it against the file it quickly revealed the potential password. Looking at the output it wasn't hard to guess the first two characters too.

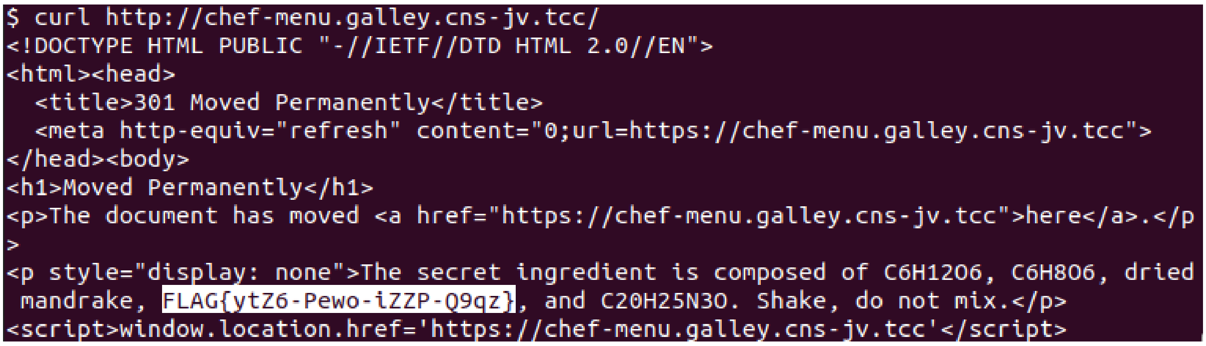


```
Password candidates (character positions):
Unknown characters are displayed as "●"
1.:     ●
2.:     ), ÿ, a, :, |, í, W, 5, □, r, .,
3.:     s,
4.:     s,
5.:     w,
6.:     o,
7.:     r,
8.:     d,
9.:     4,
10.:    m,
11.:    y,
12.:    p,
13.:    r,
14.:    e,
15.:    c,
16.:    i,
17.:    o,
18.:    u,
19.:    s,
20.:    s,
21.:    h,
22.:    i,
23.:    p,
Combined: ●{), ÿ, a, :, |, í, W, 5, □, r, .}ssword4mypreciousship
```

With the database file unlocked, it was easiest to export all to a CSV file and the flag would stick out.

Naval chef's recipe

After some time playing around with different ways to exploit anything on the provided web, I switched to using curl which complained about self-signed certificate while using HTTPS. By simply trying to get a response via HTTP, I've quickly noticed the answer was right there in the 301 redirect to HTTPS version of the page.

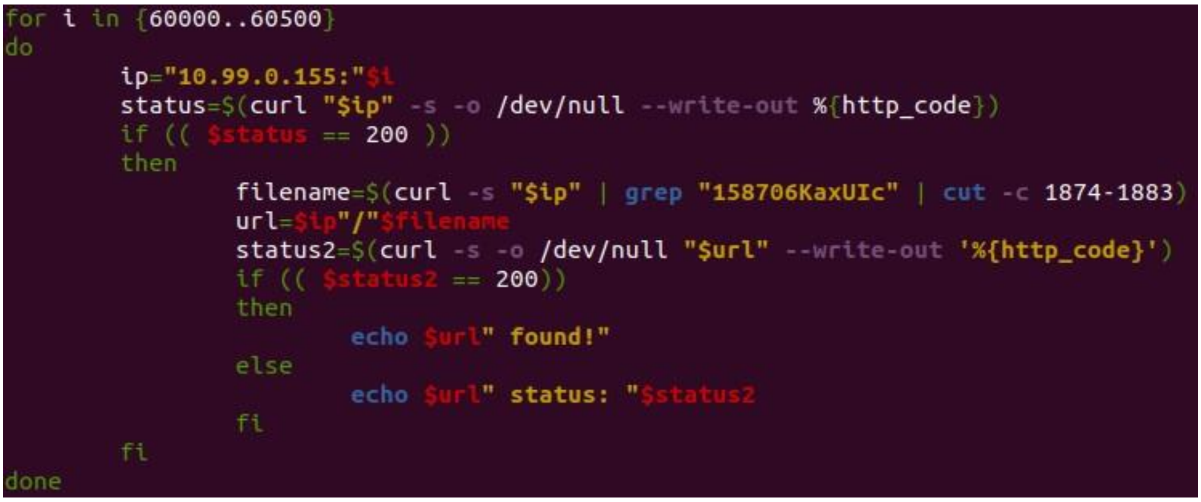


```
$ curl http://chef-menu.galley.cns-jv.tcc/
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html><head>
  <title>301 Moved Permanently</title>
  <meta http-equiv="refresh" content="0;url=https://chef-menu.galley.cns-jv.tcc">
</head><body>
<h1>Moved Permanently</h1>
<p>The document has moved <a href="https://chef-menu.galley.cns-jv.tcc">here</a>.</p
>
<p style="display: none">The secret ingredient is composed of C6H12O6, C6H8O6, dried
 mandrake, FLAG{ytZ6-Pewo-iZZP-Q9qz}, and C20H25N3O. Shake, do not mix.</p>
<script>window.location.href='https://chef-menu.galley.cns-jv.tcc'</script>
```

Keyword of the day

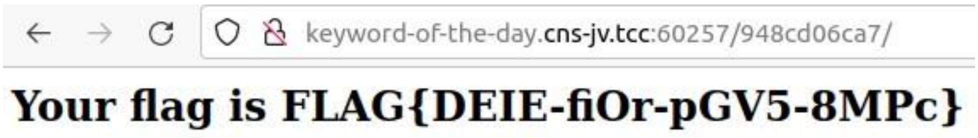
Given the real application is reportedly hidden amongst a lot of fake ones, it's first necessary to figure out where all these are. Simple port scan revealed that they run in the range 60000-60500. By inspecting the source code and resources, it became clear that the fake apps would generate a random jitter before displaying one of 7 emoji images from */img/* directory also at random.

Listing all resources in that folder was not allowed and the delay made it particularly tricky to enumerate all applications and resources by a script. First, I tried checking if all ports served all those images and if any one of them would differ. To my disappointment, they all matched. Next, after some experiments I've compared few apps to realize that the source code of each app differs by only a single string present always at the same position in the code and of a same length. This made it possible to extract this string easily and after hitting few dead ends, try the same approach to see if any app would respond differently.



```
for i in {60000..60500}
do
        ip="10.99.0.155:"$i
        status=$(curl "$ip" -s -o /dev/null --write-out %{http_code})
        if (( $status == 200 ))
        then
                filename=$(curl -s "$ip" | grep "158706KaxUIc" | cut -c 1874-1883)
                url=$ip"/"$filename
                status2=$(curl -s -o /dev/null "$url" --write-out '%{http_code}')
                if (( $status2 == 200))
                then
                        echo $url" found!"
                else
                        echo $url" status: "$status2
                fi
        fi
done
```

And, bingo! While most responded with 404, one app gave a 301 pointing to the flag.



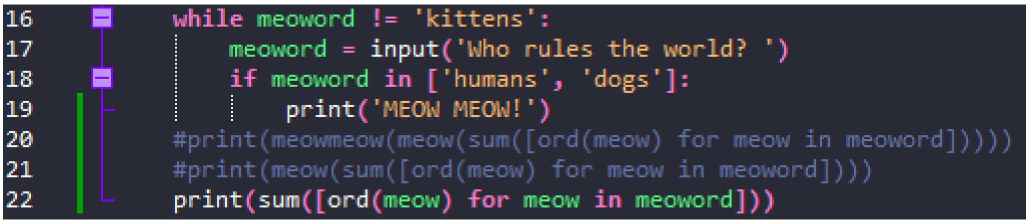
keyword-of-the-day.cns-jv.tcc:60257/948cd06ca7/

Your flag is FLAG{DEIE-fiOr-pGV5-8MPc}

Cat code

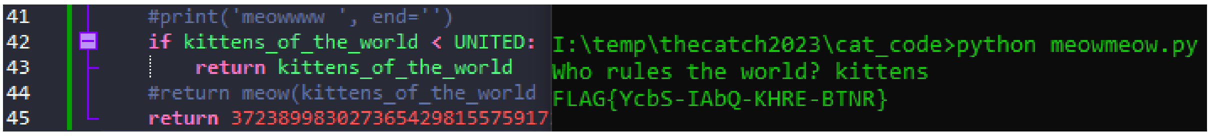
The code consisted of two python scripts, *meow* and *meowmeow* which imported from the former. *Meow* contained a constant named *meeow*, which appeared to possibly be obfuscated version of the flag – length matched, positions where we'd expect the brackets differed slightly from other and were preceded by what could well be a representation of the keyword "FLAG". Inspecting the routines, one apparently takes the *meeow* constant as an input and somehow decodes it by calling the other function. The other one seemed to implement some recursion, which turned out to be implementation of the Fibonacci sequence calculating the n-th integer based on the input.

Looking at *meowmeow*, it become clear one must answer the question *“Who rules the world?”* by entering *“kittens”*. This keyword would then be used as an input to the functions imported from *meow*. Since I suspected already that the input should be a number, I tried printing all the partial products of the final print to quickly figure out we’re going to need the 770-th of the Fibonacci sequence, even without fully understanding all the code.



```
while meoword != 'kittens':
    meoword = input('Who rules the world? ')
    if meoword in ['humans', 'dogs']:
        print('MEOW MEOW!')
#print(meowmeow(meow(sum([ord(meow) for meow in meoword]))))
#print(meow(sum([ord(meow) for meow in meoword])))
print(sum([ord(meow) for meow in meoword]))
```

If the code is triggered as is, it should eventually produce the flag but will loop in the recursive for very long time since the complexity of such algorithm is exponential, only meowing a lot during that. To avoid that the easiest solution appeared to be finding the 770-th integer some other way and returning that straight away. Luckily enough, the internet knew the answer, so it only took a bit of editing and the flag was indeed revealed.

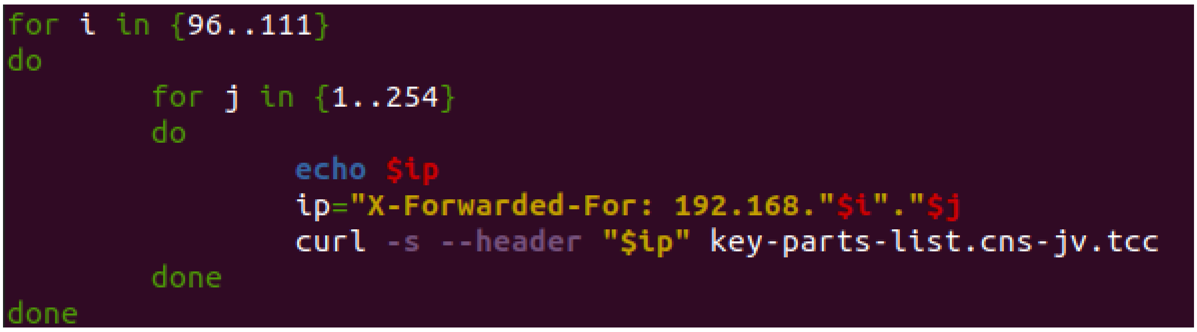


```
#print('meowwww ', end='')
if kittens_of_the_world < UNITED:
    return kittens_of_the_world
#return meow(kittens_of_the_world
return 372389983027365429815575917
```

```
I:\temp\thecatch2023\cat_code>python meowmeow.py
Who rules the world? kittens
FLAG{YcbS-IAbQ-KHRE-BTNR}
```

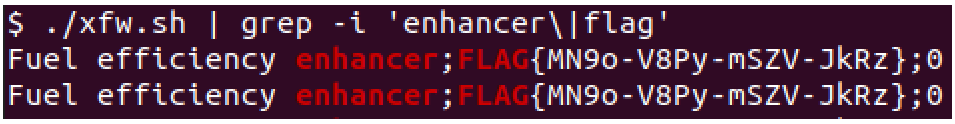
Component replacement

Upon visiting the website, it complained about the IP we’re visiting from. The obvious solution was to try and spoof this using the *X-Forwarded-For* header and since the given range was quite broad, it was apt to script this.



```
for i in {96..111}
do
        for j in {1..254}
        do
                echo $ip
                ip="X-Forwarded-For: 192.168."$i"."$j
                curl -s --header "$ip" key-parts-list.cns-jv.tcc
        done
done
```

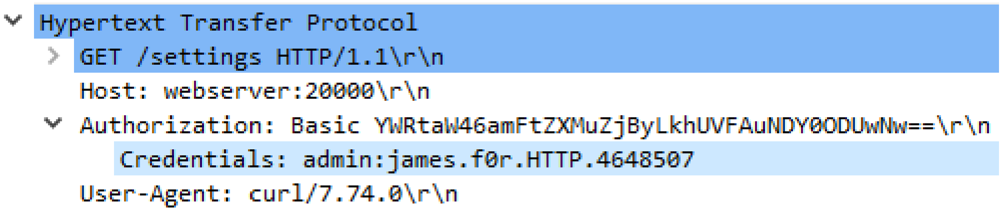
As soon as responses started to contain some data, it was easy to just filter for required keywords.



```
$ ./xfw.sh | grep -i 'enhancer\|flag'
Fuel efficiency enhancer;FLAG{MN9o-V8Py-mSZV-JkRz};0
Fuel efficiency enhancer;FLAG{MN9o-V8Py-mSZV-JkRz};0
```

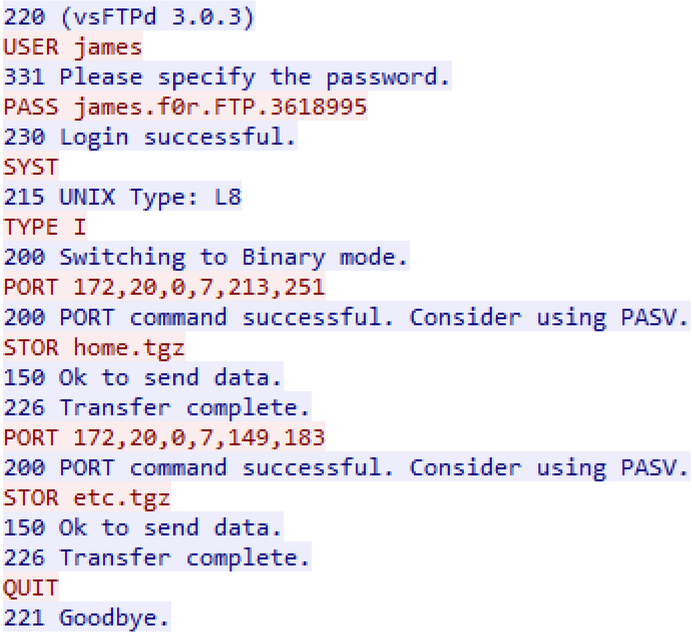
Suspicious traffic

To look for any files exchanged over the network, I first tried exporting any objects from the packet using Wireshark. Although some db files seem to be transferred over SMB, the one we're looking for was not amongst them. My next step was to filter HTTP traffic to see if anything useful would be in there. There was again nothing really pointing to the exfiltrated file, but some pieces of information that turned out to be useful later – credentials for accessing a webserver.



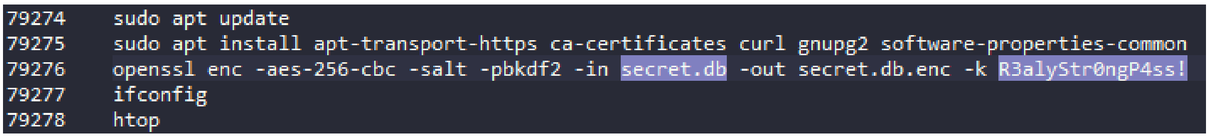
```
Hypertext Transfer Protocol
  GET /settings HTTP/1.1\r\n
  Host: webserver:20000\r\n
  Authorization: Basic YWRtaW46amFtZXMuZjByLkhUVFAuNDY0ODUwNw==\r\n
    Credentials: admin:james.f0r.HTTP.4648507
  User-Agent: curl/7.74.0\r\n
```

Further inspecting the traffic, I also discovered some FTP credentials and files transferred. After some struggling I've learned from the *PORT* command it was possible to calculate the port used as 213*256+251=54779 and filter respective traffic, which appeared to correspond to tcp.stream 6.



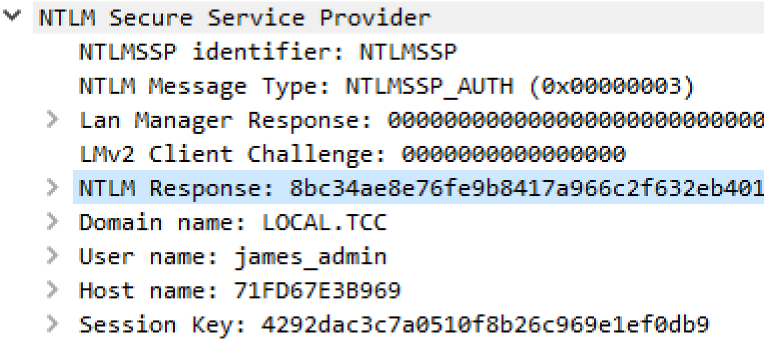
```
220 (vsFTPd 3.0.3)
USER james
331 Please specify the password.
PASS james.f0r.FTP.3618995
230 Login successful.
SYST
215 UNIX Type: L8
TYPE I
200 Switching to Binary mode.
PORT 172,20,0,7,213,251
200 PORT command successful. Consider using PASV.
STOR home.tgz
150 Ok to send data.
226 Transfer complete.
PORT 172,20,0,7,149,183
200 PORT command successful. Consider using PASV.
STOR etc.tgz
150 Ok to send data.
226 Transfer complete.
QUIT
221 Goodbye.
```

Extracting and inflating the file *home.tgz*, I already found some breadcrumbs pointing to the stolen database.



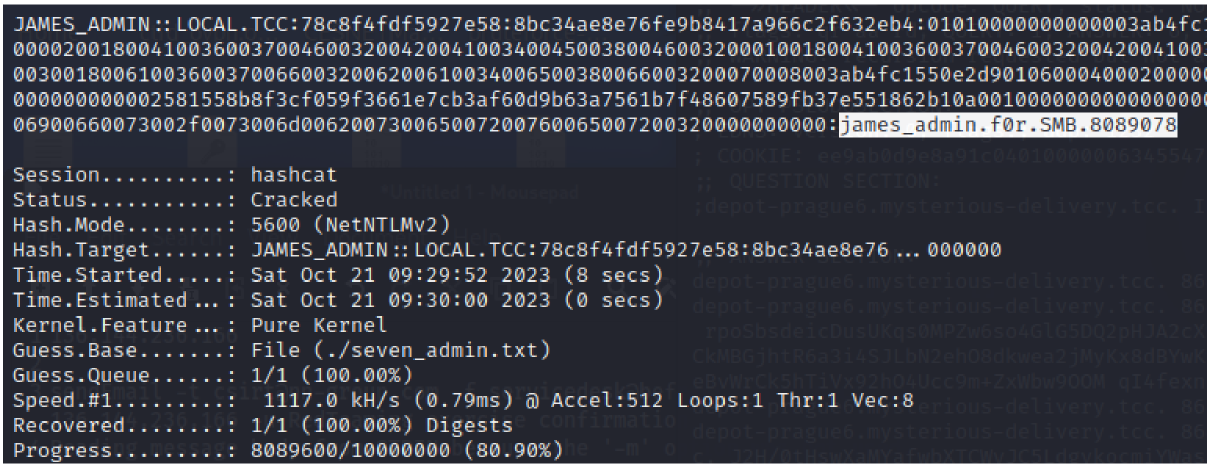
```
79274    sudo apt update
79275    sudo apt install apt-transport-https ca-certificates curl gnupg2 software-properties-common
79276    openssl enc -aes-256-cbc -salt -pbkdf2 -in secret.db -out secret.db.enc -k R3alyStr0ngP4ss!
79277    ifconfig
79278    htop
```

Not much else could be found in any of the files, but what caught my attention was some encrypted SMB3 traffic in tcp.stream 11. I shortly found an article called [Decrypting SMB3 Traffic with just a PCAP? Absolutely (maybe.)]() which was the key to solving this challenge. Following the instructions here, I grabbed some of the NTLM details required to calculate the random session key.



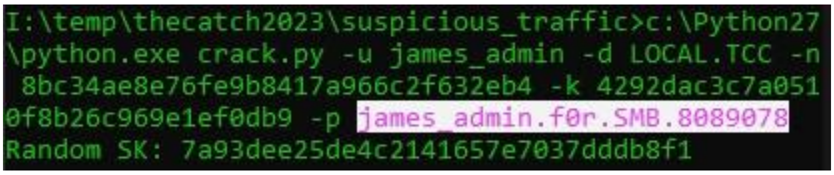
```
NTLM Secure Service Provider
    NTLMSSP identifier: NTLMSSP
    NTLM Message Type: NTLMSSP_AUTH (0x00000003)
  > Lan Manager Response: 000000000000000000000000000
    LMv2 Client Challenge: 0000000000000000
  > NTLM Response: 8bc34ae8e76fe9b8417a966c2f632eb401
  > Domain name: LOCAL.TCC
  > User name: james_admin
  > Host name: 71FD67E3B969
  > Session Key: 4292dac3c7a0510f8b26c969e1ef0db9
```

For the next step, it was necessary to figure out James's password. It took me a while to realize it may be following the same format as the 2 already known sets of credentials. Still following the article, I've created a wordlist using mp64 as *james_admin.f0r.SMB.* followed by 7 digits and quickly found the valid one with *hashcat*.



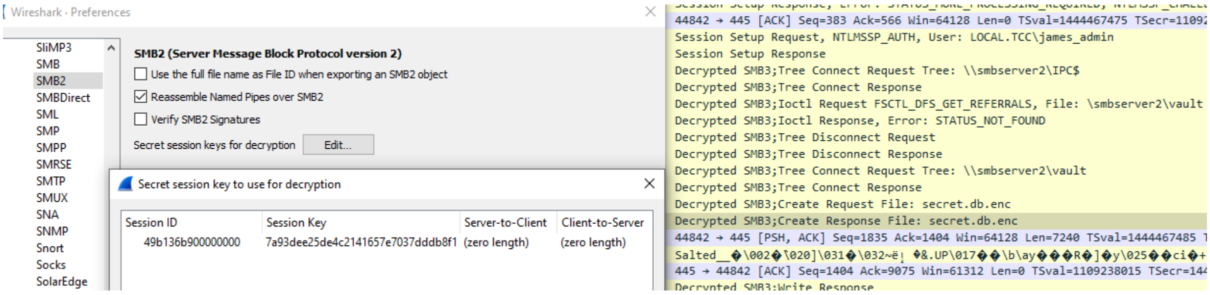
```
JAMES_ADMIN::LOCAL.TCC:78c8f4fdf5927e58:8bc34ae8e76fe9b8417a966c2f632eb4:0101000000000000003ab4fc
000020018004100360037004600320042004100340045003800460032000100180041003600370046003200420041003
0030018006100360037006600320062006100340065003800660032000700080003ab4fc1550e2d9010600040002000
000000000002581558b8f3cf059f3661e7cb3af60d9b63a7561b7f48607589fb37e551862b10a0010000000000000000
069006600730002f0073006d006200730065007200760065007200320000000000:james_admin.f0r.SMB.8089078

Session..........: hashcat
Status...........: Cracked
Hash.Mode........: 5600 (NetNTLMv2)
Hash.Target......: JAMES_ADMIN::LOCAL.TCC:78c8f4fdf5927e58:8bc34ae8e76...000000
Time.Started.....: Sat Oct 21 09:29:52 2023 (8 secs)
Time.Estimated...: Sat Oct 21 09:30:00 2023 (0 secs)
Kernel.Feature...: Pure Kernel
Guess.Base.......: File (./seven_admin.txt)
Guess.Queue......: 1/1 (100.00%)
Speed.#1.........:  1117.0 kH/s (0.79ms) @ Accel:512 Loops:1 Thr:1 Vec:8
Recovered........: 1/1 (100.00%) Digests
Progress.........: 8089600/10000000 (80.90%)
```

Using the provided python script and password, I was now able to calculate the session random key and configure Wireshark to decrypt it.



```
I:\temp\thecatch2023\suspicious_traffic>c:\Python27
\python.exe crack.py -u james_admin -d LOCAL.TCC -n
 8bc34ae8e76fe9b8417a966c2f632eb4 -k 4292dac3c7a051
0f8b26c969e1ef0db9 -p james_admin.f0r.SMB.8089078
Random SK: 7a93dee25de4c2141657e7037dddb8f1
```

Sure enough, the session was now easy to read, and more traces of the lost file surfaced. It was even possible to extract the file now and the last step was to decrypt it with the previously recovered password using *openssl* to get the flag.



SMB2 (Server Message Block Protocol version 2)

☐ Use the full file name as File ID when exporting an SMB2 object

☑ Reassemble Named Pipes over SMB2

☐ Verify SMB2 Signatures

Secret session keys for decryption

Secret session key to use for decryption

Session ID	Session Key	Server-to-Client	Client-to-Server
49b136b900000000	7a93dee25de4c2141657e7037dddb8f1	(zero length)	(zero length)

```
Session Setup Response, Error: STATUS_MORE_PROCESSING_REQUIRED, NTLMSSP_CHALLE
44842 → 445 [ACK] Seq=383 Ack=566 Win=64128 Len=0 TSval=1444467475 TSecr=1109
Session Setup Request, NTLMSSP_AUTH, User: LOCAL.TCC\james_admin
Session Setup Response
Decrypted SMB3;Tree Connect Request Tree: \\smbserver2\IPC$
Decrypted SMB3;Tree Connect Response
Decrypted SMB3;Ioctl Request FSCTL_DFS_GET_REFERRALS, File: \smbserver2\vault
Decrypted SMB3;Ioctl Response, Error: STATUS_NOT_FOUND
Decrypted SMB3;Tree Disconnect Request
Decrypted SMB3;Tree Disconnect Response
Decrypted SMB3;Tree Connect Request Tree: \\smbserver2\vault
Decrypted SMB3;Tree Connect Response
Decrypted SMB3;Create Request File: secret.db.enc
Decrypted SMB3;Create Response File: secret.db.enc
44842 → 445 [PSH, ACK] Seq=1835 Ack=1404 Win=64128 Len=7240 TSval=1444467485
Salted__�\002�\020]\031�\032~ë¦ �&.UP\017��\b\ay���R�]�y\025��ci�+
445 → 44842 [ACK] Seq=1404 Ack=9075 Win=61312 Len=0 TSval=1109238015 TSecr=144
Decrypted SMB3;Write Response
```