TAMU Cybersecurity Club CTF Writeup (9/25/21)

Total Team Points: 6336

My Points: 2792

Challenge: Meow

For this challenge, we are given the flag in a hashed format,

```
gigem{ca0c537de0f92aa666c2513bbb5ae532}
```

And the following information about the plaintext:

- It only has lowercase letters and digits
- It ends with a year
- It is 7 characters long

So with the hash, and given this information, we can assume that it is most likely able to be brute forced using a tool like hashcat. So let's open up hashcat and see if this gives us anything.

Looking at this hash, we see that it could be an MD5 hash based on the syntax. So using the hash passed in with the given information as additional parameters to hashcat, i.e.

```
hashcat -m 0 -a 3 hash.txt ?l?l?l?d?d?d?d
```

We get the following result when executed:

Giving us the flag- Woohoo!

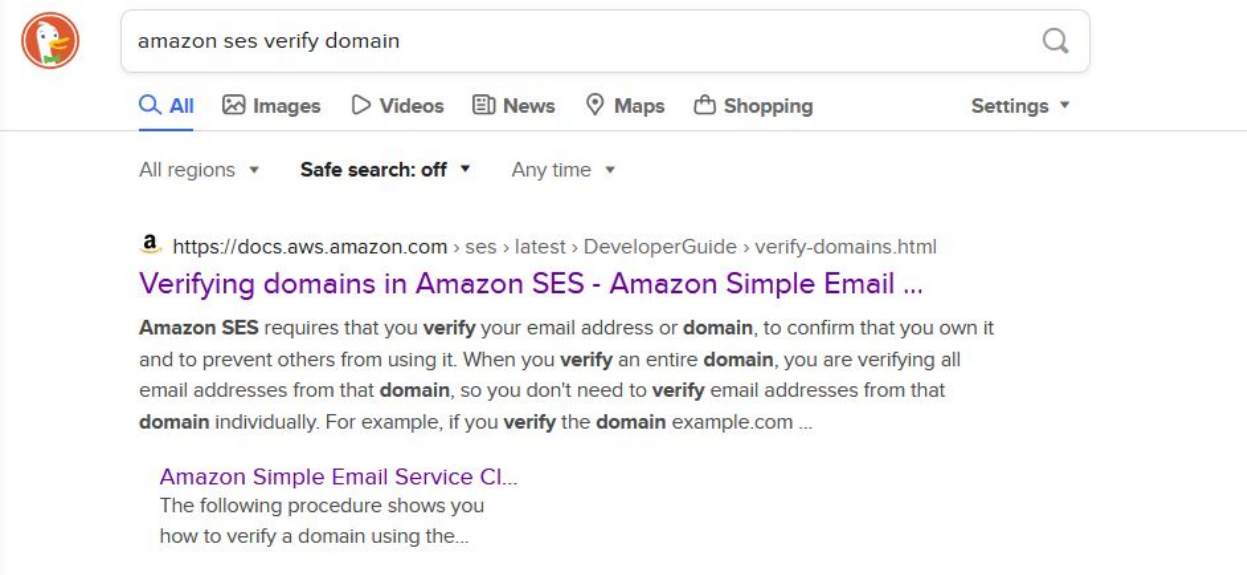
Challenge: Text

This one took me awhile, because I am new to AWS and especially their SES service. However after viewing the hint, "Using AWS documentation would be useful", I realized I would have to do some googling to gain an initial foothold for this challenge.

So for this challenge we are given this,

> Last year's webmaster built cybr.club on AWS. He verified cybr.club domain with Amazon SES service by adding a DNS record. He left a flag in the DNS record using the same format as in the verification process. Can you find it?

The first thing we should do is look up "amazon ses verify domain" in DuckDuckGo and find this link:



Viewing the documentation, we find this information that gives the syntax of the DNS record for an Amazon SES domain:

Amazon SES domain verification TXT records

PDF | Kindle

Your domain is associated with a set of Domain Name System (DNS) records that you manage through your DNS provider. A TXT record is a type of DNS record that provides additional information about your domain. Each TXT record consists of a name and a value.

When you initiate domain verification using the Amazon SES console or API, Amazon SES gives you the name and value to use for the TXT record. For example, if your domain is *example.com*, the TXT record settings that Amazon SES generates will look similar to the following example:

Name	Type	Value
_amazonses.example.com	TXT	pmBGN/7MjnfhTKUZ06En

Add a TXT record to your domain's DNS server using the specified **Name** and **Value**. Amazon SES domain verification is complete when Amazon SES detects the existence of the TXT record in your domain's DNS settings.

So now that we know what the syntax is for the DNS record, let's use NSLOOKUP to find the TXT records for this domain:

```
┌──(kali㉿kali)-[~]
└─$ nslookup -type=TXT  _amazonses.cybr.club dns1.registrar-servers.com
Server:         dns1.registrar-servers.com
Address:        156.154.132.200#53

Non-authoritative answer:
_amazonses.cybr.club    text = "JHDWbZD8pdO+ePSIJ0QLmu5QzlaRx7I1s/vfWxchMM8="
_amazonses.cybr.club    text = "gigem{f0und_cybr_Dn3_TXT_AmaZon3e3}"

Authoritative answers can be found from:
```

And there is our flag!

Challenge: Baguette

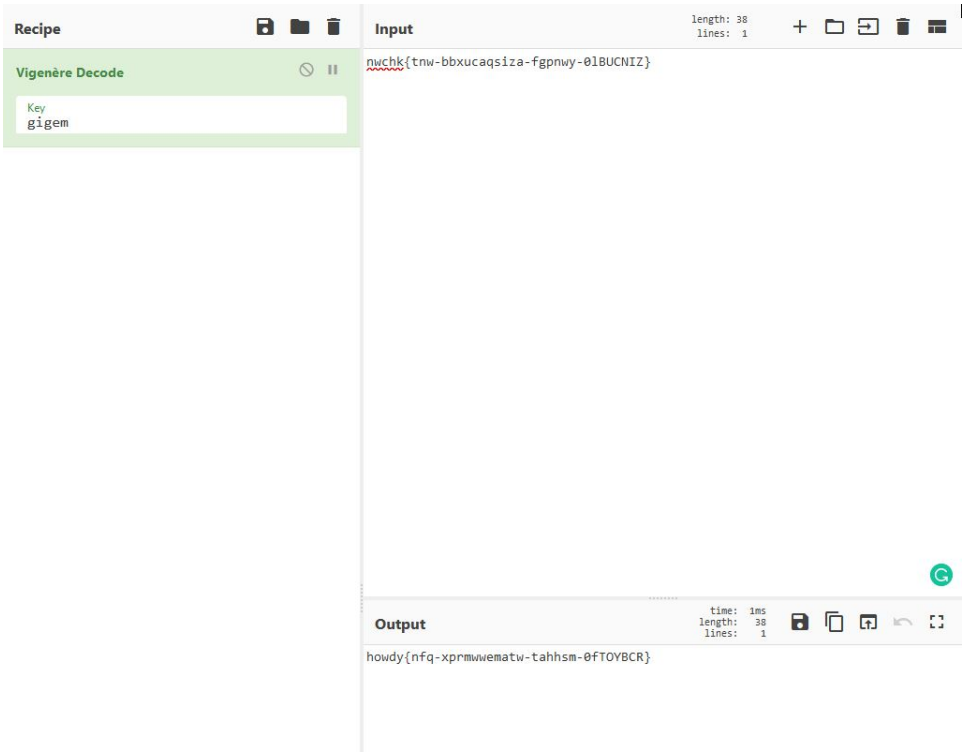
Alright, a cryptography challenge! My favorite. This challenge, we are given the flag in a ciphered form,

```
nwchk{tnw-bbxucaqsiza-fgpnwy-0lBUCNIZ}
```

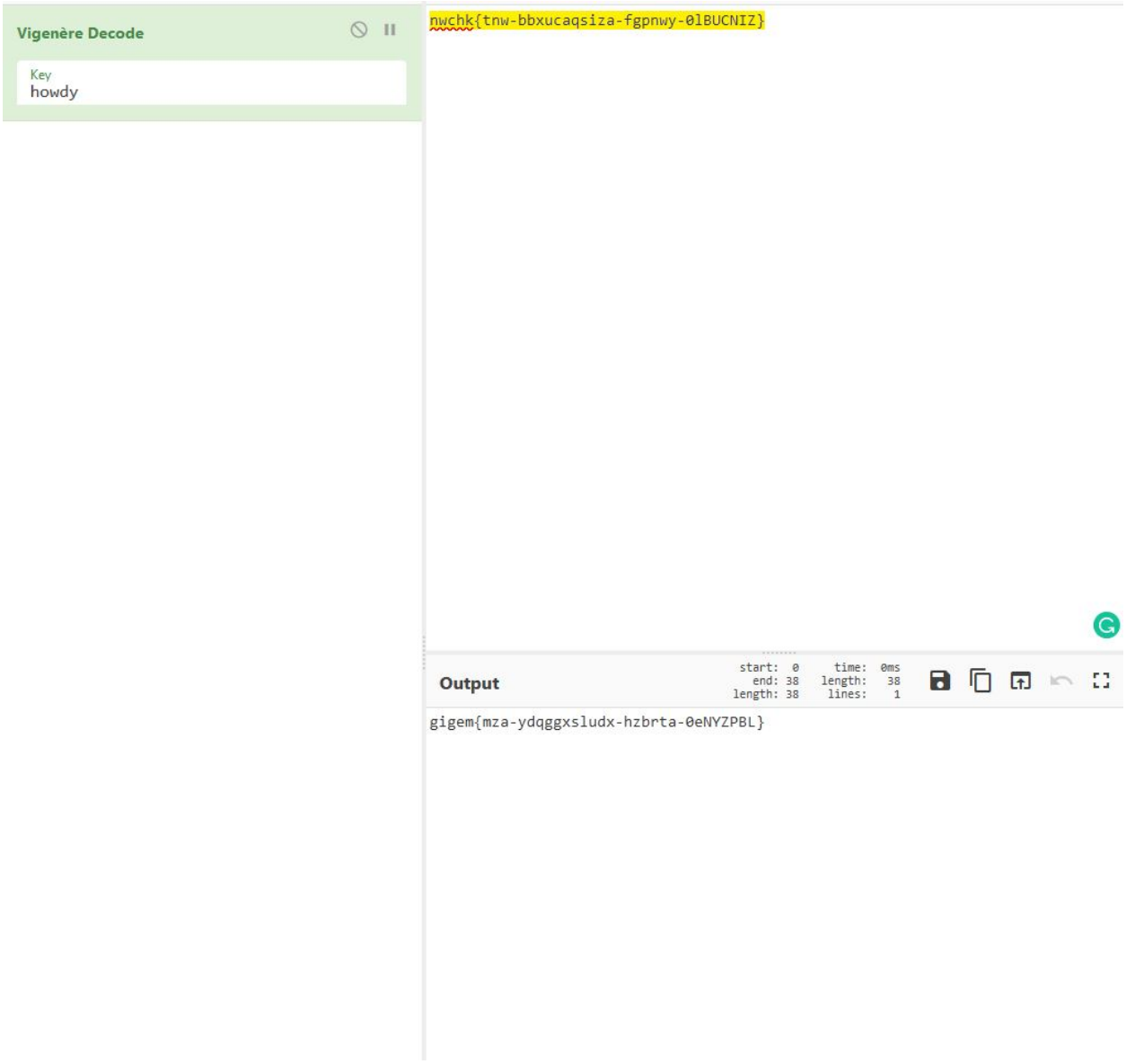
and we are tasked with finding the plaintext.

So let's open up the best tool for this sort of thing, Cyber Chef!

We can tell that this specific ciphertext cannot be a simple caesar cipher, it most likely requires some sort of key, since otherwise we would see repeated letters for the "g" in "gigem", and be able to perform frequency analysis. So in cyber chef, let's try using a Vingenere Decode, which does require a key, and inputting first the most obvious of keys, "gigem" we see that this actually gives us something!



We see in the output "howdy" which indicates that the key should actually be "howdy" and not "gigem". So let's try "howdy" instead as the key:



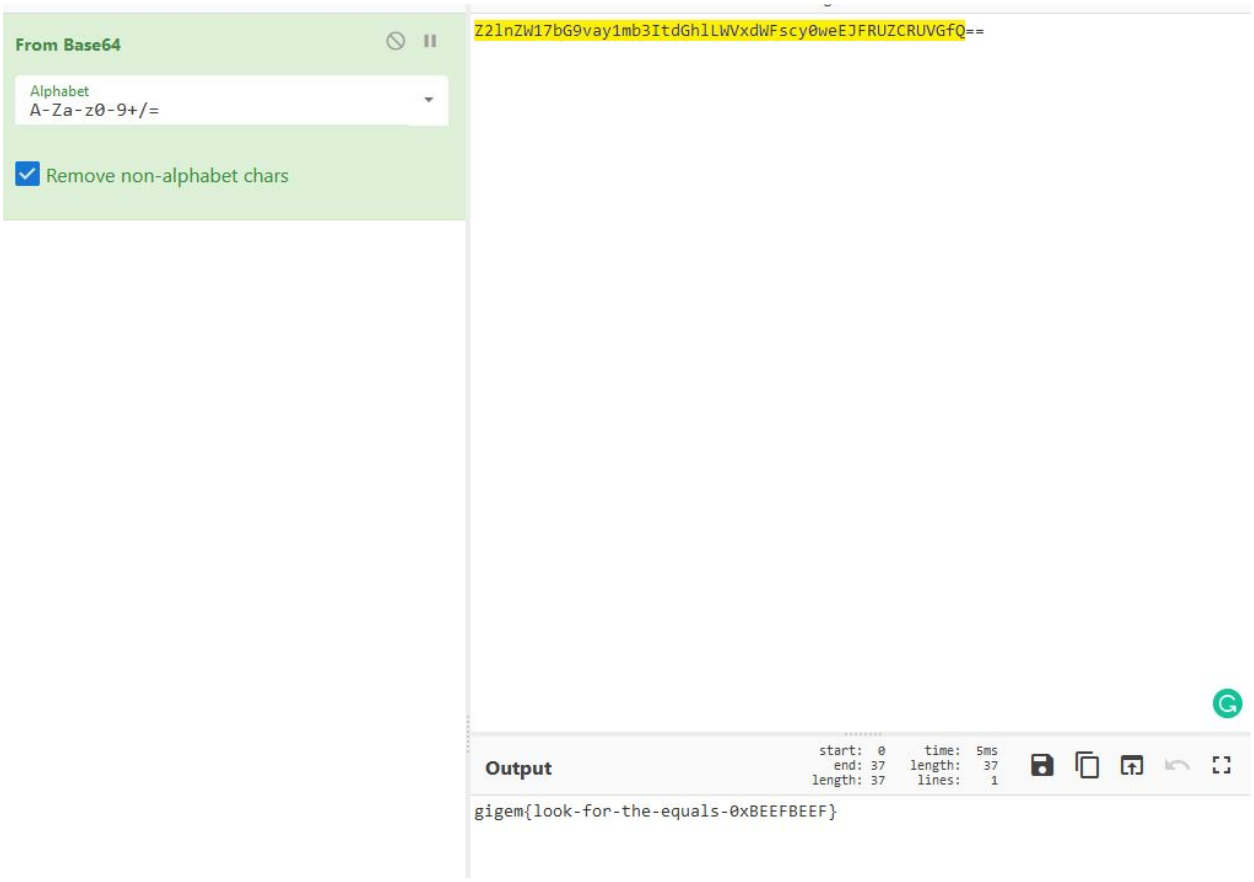
And we have found our flag! gigem{mza-ydqggxsludx-hzbrta-0eNYZPBL}

Challenge: Change of Base

Alright, this one is quick and easy. We are given the flag in this form,

```
Z2lnZW17bG9vay1mb3ItdGhlLWVxdWFscy0weEJFRUZCRUVGfQ==
```

And just giving it a once over we notice the "==", which is the general indicator of the end of a Base64 encoding. So let's take a leap of faith and just throw this into Cyber Chef,



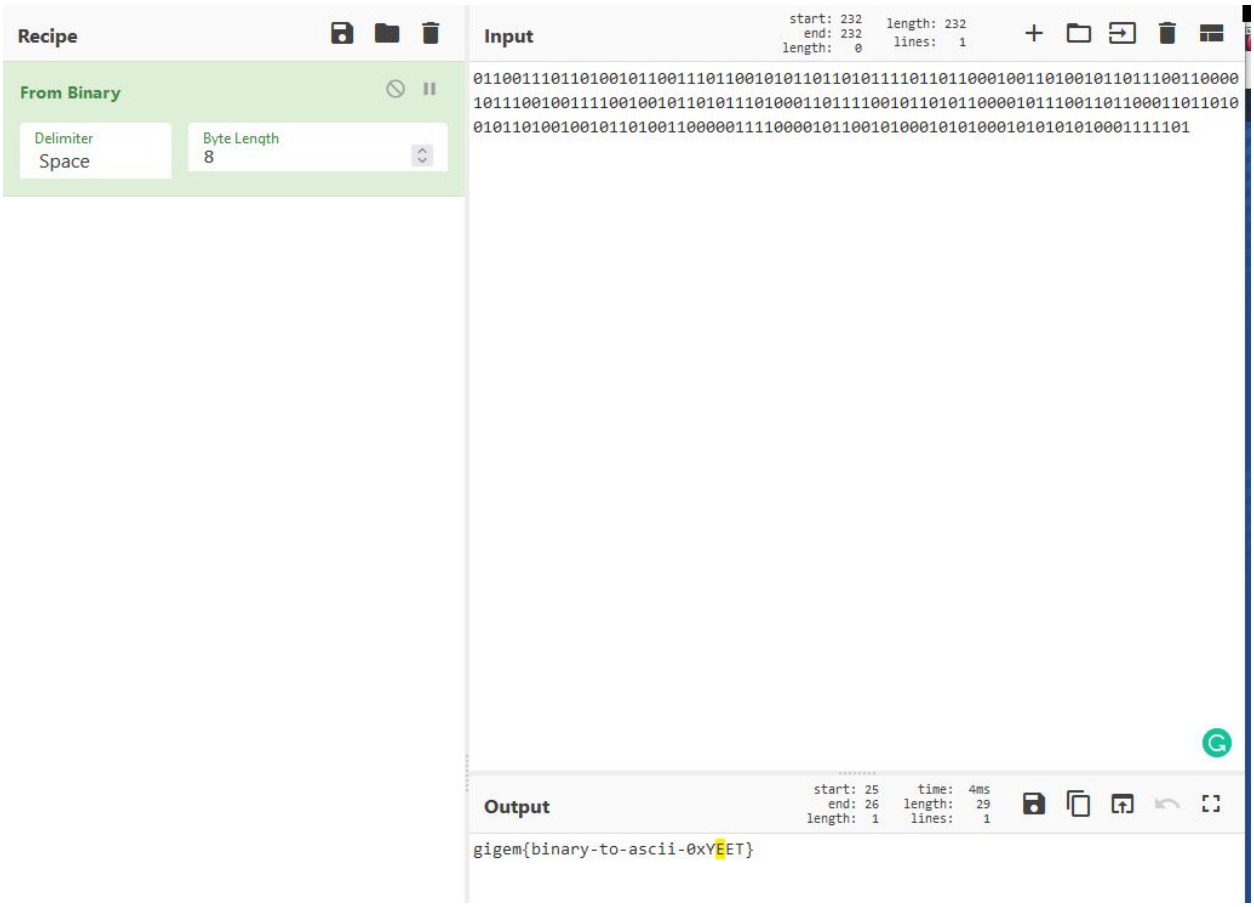
And awesome, we are given the flag as the output! gigem{look-for-the-equals-0xBEEFBEEF}

Challenge: The Base of It All

Another very quick challenge. We are given these beautiful looking binary values,

```
01100111011010010110011101100101011011010111011011000100110100101101110011000
10111001001111001001011010111010001101111001011010110000101110011011000110111010
01011010010010110100110000011110000101100101000101010001010101010001111101
```

And let's just change these binary into its ASCII representation using Cyber Chef to see if it gives us anything:



There we see the flag in all of its glory, easy 50 points!

Challenge: Unshadow

This challenge falls under the forensics category, another favorite type of mine! So we are given the information

> A lazy administrator gave permissions to a new hire to change his password. He inadvertently gave the new employee permission to read /etc/passwd and /etc/shadow files and the new employee leaked those files. Can you figure out the kali user's password?

As well as two .txt files, "passwd.txt" and "shadow.txt" containing some information about the system. Looking through the passwd.txt file,

```
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
```

```
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
systemd-timesync:x:101:101:systemd Time Synchronization,,,:/run/systemd:/usr/sbin/nologin
systemd-network:x:102:103:systemd Network Management,,,:/run/systemd:/usr/sbin/nologin
systemd-resolve:x:103:104:systemd Resolver,,,:/run/systemd:/usr/sbin/nologin
mysql:x:104:110:MySQL Server,,,:/nonexistent:/bin/false
tss:x:105:111:TPM software stack,,,:/var/lib/tpm:/bin/false
strongswan:x:106:65534::/var/lib/strongswan:/usr/sbin/nologin
ntp:x:107:112::/nonexistent:/usr/sbin/nologin
messagebus:x:108:113::/nonexistent:/usr/sbin/nologin
redsocks:x:109:114::/var/run/redsocks:/usr/sbin/nologin
rwhod:x:110:65534::/var/spool/rwho:/usr/sbin/nologin
iodine:x:111:65534::/run/iodine:/usr/sbin/nologin
miredo:x:112:65534::/var/run/miredo:/usr/sbin/nologin
_rpc:x:113:65534::/run/rpcbind:/usr/sbin/nologin
usbmux:x:114:46:usbmux daemon,,,:/var/lib/usbmux:/usr/sbin/nologin
tcpdump:x:115:120::/nonexistent:/usr/sbin/nologin
rtkit:x:116:121:RealtimeKit,,,:/proc:/usr/sbin/nologin
sshd:x:117:65534::/run/sshd:/usr/sbin/nologin
statd:x:118:65534::/var/lib/nfs:/usr/sbin/nologin
postgres:x:119:123:PostgreSQL administrator,,,:/var/lib/postgresql:/bin/bash
avahi:x:120:125:Avahi mDNS daemon,,,:/run/avahi-daemon:/usr/sbin/nologin
stunnel4:x:121:126::/var/run/stunnel4:/usr/sbin/nologin
Debian-snmp:x:122:127::/var/lib/snmp:/bin/false
sslh:x:123:128::/nonexistent:/usr/sbin/nologin
nm-openvpn:x:124:129:NetworkManager OpenVPN,,,:/var/lib/openvpn/chroot:/usr/sbin/nologin
nm-openconnect:x:125:130:NetworkManager OpenConnect plugin,,,:/var/lib/NetworkManager:/usr/sbin/nologin
pulse:x:126:131:PulseAudio daemon,,,:/var/run/pulse:/usr/sbin/nologin
saned:x:127:134::/var/lib/saned:/usr/sbin/nologin
inetsim:x:128:136::/var/lib/inetsim:/usr/sbin/nologin
colord:x:129:137:colord colour management daemon,,,:/var/lib/colord:/usr/sbin/nologin
geoclue:x:130:138::/var/lib/geoclue:/usr/sbin/nologin
lightdm:x:131:139:Light Display Manager:/var/lib/lightdm:/bin/false
king-phisher:x:132:140::/var/lib/king-phisher:/usr/sbin/nologin
kali:x:1000:1000:Kali,,,:/home/kali:/usr/bin/zsh
systemd-coredump:x:999:999:systemd Core Dumper:/:/usr/sbin/nologin
newhire:x:1001:1001::/home/newhire:/bin/sh
```

We don't really see anything of interest to us. However when we look through the "shadow.txt" file and scroll to the bottom, something peculiar stands out to us:

```
kali:$6$JOTveDYM47UKYe3X$XZUYmo8B4MCvtSbOIKsFVYLkkDQIlCuP3sJmf8tZDs/4p1sGCl7rpGj1CCgzR4nROYILdVheXzmslu6W7QKD3/:18770:0:99999:7:::
```

Alright! Looks like a hashed password! Let's see if we can use JohnTheRipper and a wordlist (rockyou.txt) that contains the most common passwords used to try and find a plaintext that matches this hash.

```
john --wordlist=rockyou.txt --rules shadow.txt
```

We run this and then show the value of the cracked hash,

```
┌──(kali㉿kali)-[~/Downloads]
└─$ john -show shadow.txt
kali:monkey123:18770:0:99999:7:::
newhire:NO PASSWORD:18770:0:99999:7:::

2 password hashes cracked, 0 left
```

And there we have our password (which is the flag) "monkey123"!

Side note: this password literally took ~2 seconds to crack, if your password is around the complexity of this, please change it or better yet, use a password manager! Trying to crack a password of that complexity would make my computer melt.

Challenge: Salad

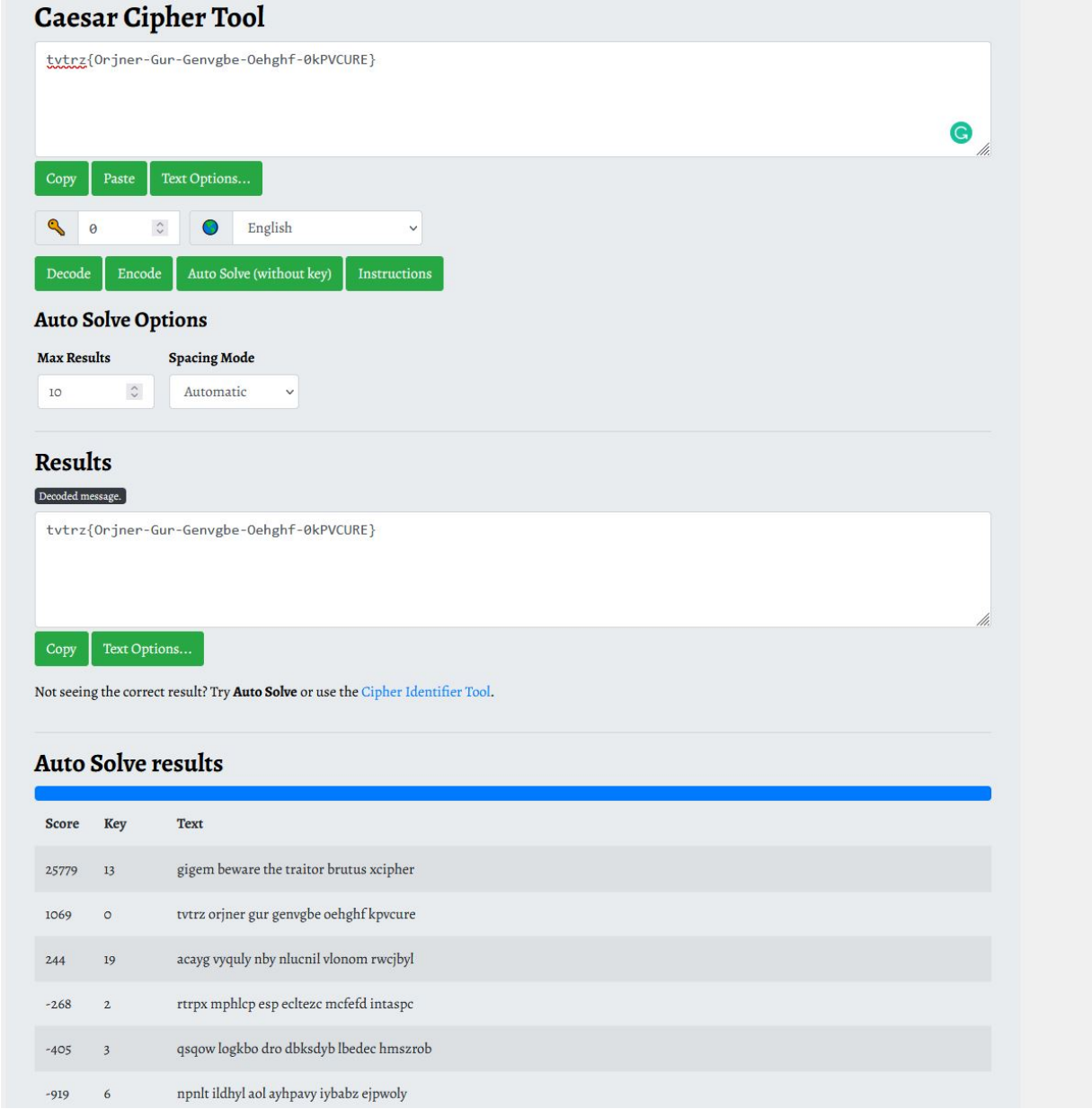
This one is like the Baguette Challenge, except even easier because we notice that the cipher text,

```
tvtrz{Orjner-Gur-Genvgbe-Oehghf-0kPVCURE}
```

Has the same value (t) for the two g's in gigem. So now we can most likely guess that this is just a basic shift cipher, or more traditionally known as the Caesar Cipher (Which is also delicious salad). Rather than performing frequency analysis (which is a valid option for reversing this!) we will take the path of least resistance and put this into an online tool.

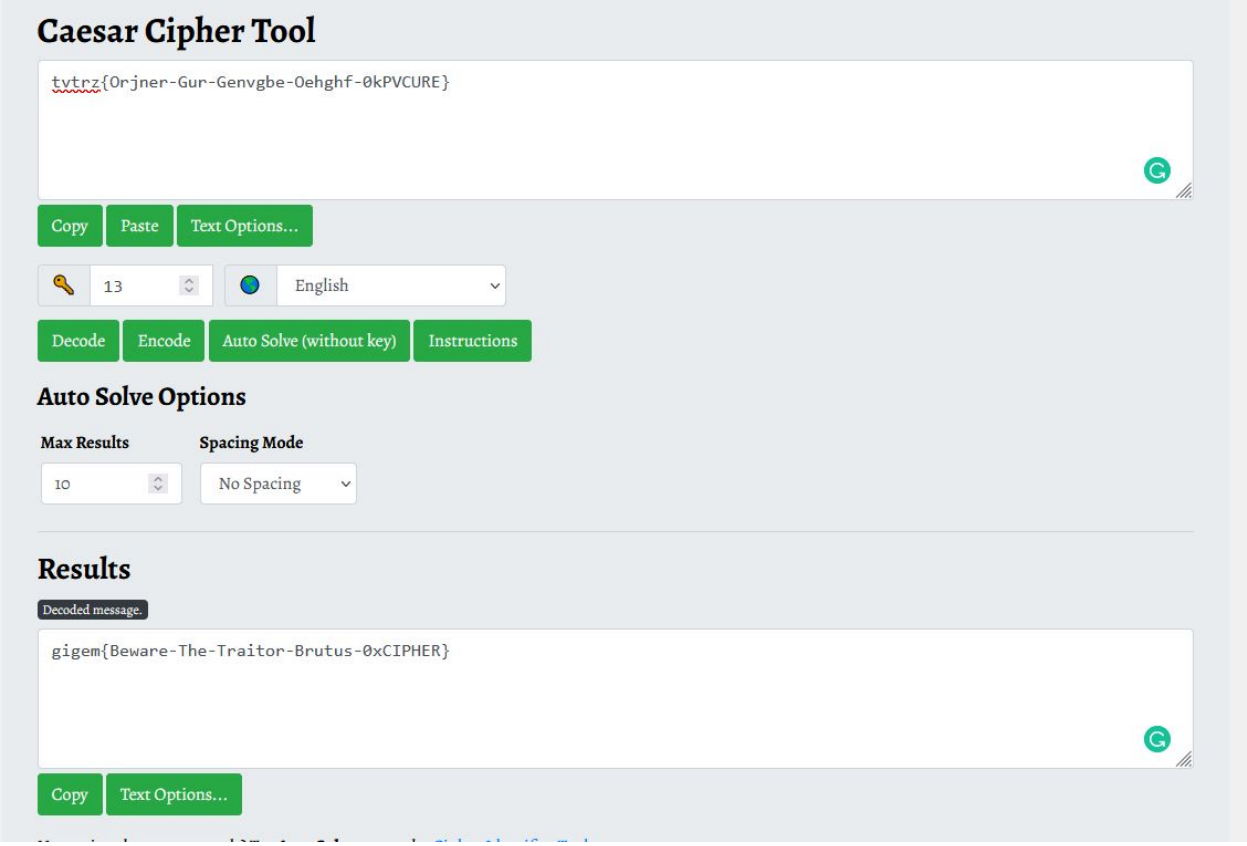
https://www.boxentriq.com/code-breaking/caesar-cipher

We use the Caesar Cipher Tool provided to us by Boxentriq, and hit Auto solve so it can find the shift value for us!



And we see the first result there has a key value of 13, indicating that all of the letters in the cipher are shifted by 13 letters in the alphabet, i.e. g → t, i→v, a→n etc.

Rerunning this with the key value to get the correct format,



We have our flag!

gigem{Beware-The-Traitor-Brutus-0xCIPHER}

Challenge: Basic RSA

```
ciphertext = [318282133112158,1053347067461556,318282133112158,127579114061798,925498245919157,1238489031973435,43104005545708
1,1499700252695404,1090770538737549,1689979174887063,1053347067461556,163674852996679,1689979174887063,1090770538737549,1636748
52996679,469972058981107,925498245919157,925498245919157,63019771348952,209173781555113,1408493700962500,1053347067461556,57832
6965779290,865903205215761]

N = 1745089977867487
d = 183693671819227
e = 19
```

Here we are given a sequence of numbers which are the encrypted versions of the ASCII letters of the flag, and tasked with decrypting them to get the flag and solve the challenge.

So looking at this, we are actually given everything that we need to decrypt the ciphertext, since we have the decryption exponent, "d". For learning purposes let's go through all of the steps to see how the ciphertext was calculated in the first place.

So the N value that we see is called the "Public Modulus", and it turns out that this number has exactly 2 (large) prime factors, which are the "private keys" (these are very difficult to find from just N).

The "e" value is the second part of the public key, along with N, and e is called the encryption exponent.

The "d" value is called the decryption exponent, and is supposed to always be kept secret. The creators of this challenge essentially gave us the answer, they just wanted us to understand the basics of RSA by going through the (tedious)

calculations.

The Ciphertexts (c's) that we see there are the original message (that we are trying to find) raised to the power e, so

$$ciphertext = c = m^e \bmod N$$

So in order to reverse the encryption, we calculate the message (m) by doing

$$m = c^d \bmod N$$

for each ciphertext ("c") value that we have, and then once we have the decimal values of the plaintext, we need to change them in ASCII values to obtain our flag!

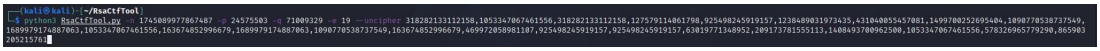
So now onto the actual solving of the challenge:

Let's use the Rsactftool from github, https://github.com/Ganapati/RsaCtfTool

And since the value of N is small, we can easily factor this giving us the values of the private key using Wolfram Alpha

$$1745089977867487 = 24575503 \times 71009329 \ (2 \text{ distinct prime factors})$$

So with all of this information, we can solve for the flag using our tool:



```
python3 RsaCtfTool.py -n 1745089977867487 -p 24575503 -q 71009329 -e 19 --uncipher 318282133112158,10533470674615 56,31828213311
2158,127579114061798,925498245919157,1238489031973435,431040055457081,1499700252695404,1090770538737549,1689979174887063,105334
7067461556,163674852996679,1689979174887063,1090770538737549,163674852996679,469972058981107,925498245919157,925498245919157,63
019771348952,209173781555113,1408493700962500,10533470674615 56,578326965779290,865903205215761 -output plaintext.txt
```

Squinting really hard at the output above, we can see that the flag is "gigem{RSA_is_AsYmmEtriC}" (repeated a bunch of times)

Challenge: RSAjr

This one actually introduced the RsaCtfTool to me, as I did not know about it before this CTF! By the way, this tool is fantastic for these types of challenges (and I really wish I knew about it when I was taking my Crypto Classes). But anyways, for this challenge we are not given any information about our private key(s), and the value for N is massive, so attempting to factor it is a waste of time and energy.

```
N = 213681069641007179601208741450037729586376793837279335231506862036319655235788370940854350009517009433738383219972205641663
024883215901280615312850106368571638978998117122840139210685346167726847173232244364004850978371121744321827034365483575406101 7
503137136489303437996367224915212044704472299799616089259112992421843 7
e = 3
c = 136027186122673360164165605186479668391597362556595460218001327359141646051717311764973659340728226526827151813506566402650
```

```
340114198494070783092124519478251276777032121812630571999542698760728546126894496085844727740682250592252205295575885468532400675673806657089125
75673806657089125
```

However, look at the value of encryption exponent e; it is tiny! So tiny in fact, that we can abuse this and actually completely break the encryption and recover the original message using what is called "the cube root attack", which I think is a really cool attack on RSA, since it elegantly takes advantage of the mathematics that goes into RSA. Let's look at how it works:

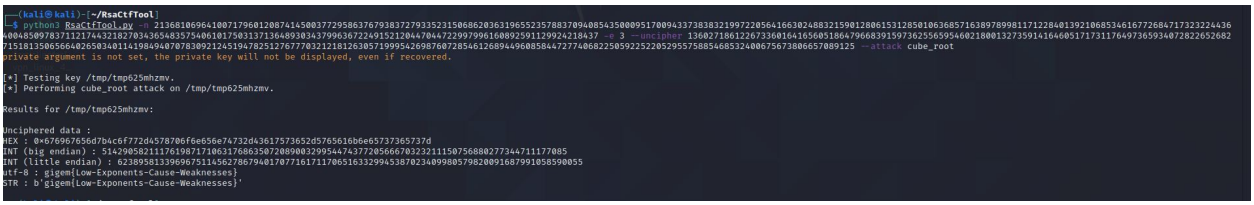
Because the numerical value of the ciphertext is less than the modulus after encrypting, i.e.

$$c = m^3 \bmod \mathrm{N} \; < N$$

We can recover m by just taking the cube root of c!;

$$\implies m = \sqrt[3]{c} \bmod \mathrm{N}$$

So let's do this using our handy RsaCtfTool:



Squinting again, the output is

```
python3 RsaCtfTool.py -n 21368106964100717960120874145003772958637679383727933523150686203631965523578837094085435000951700943373
738383219972205641663024883215901280615312850106368571638978998117122840139210685346167726847173232244364004850978371121744321 8
270343654835754061017503137136489303437996367224915212044704472299799616089259112992421843 7 -e 3 --uncipher 1360271861226733601
641656051864796683915973625565954602180013273591416460517173117649736593407282265268271518135065664026503401141984940707830921 2
451947825127677703212181263057199954269876072854612689449608584472774068225059225220529557588546853240067567380665708912 5 --att
ack cube_root
private argument is not set, the private key will not be displayed, even if recovered.

[*] Testing key /tmp/tmp625mhzmv.
[*] Performing cube_root attack on /tmp/tmp625mhzmv.

Results for /tmp/tmp625mhzmv:

Unciphered data :
HEX : 0x676967656d7b4c6f772d4578706f6e656e74732d43617573652d5765616b6e657373657737d
INT (big endian) : 514290582111761987171063176863507208900329954474377205666703232111507568802773447111770085
INT (little endian) : 623895813396967511456278679401707716171170651633299453870234099805798200916879910585900055
utf-8 : gigem{Low-Exponents-Cause-Weaknesses}
STR : b'gigem{Low-Exponents-Cause-Weaknesses}'
```

Giving us our flag, gigem{Low-Exponents-Cause-Weaknesses}!