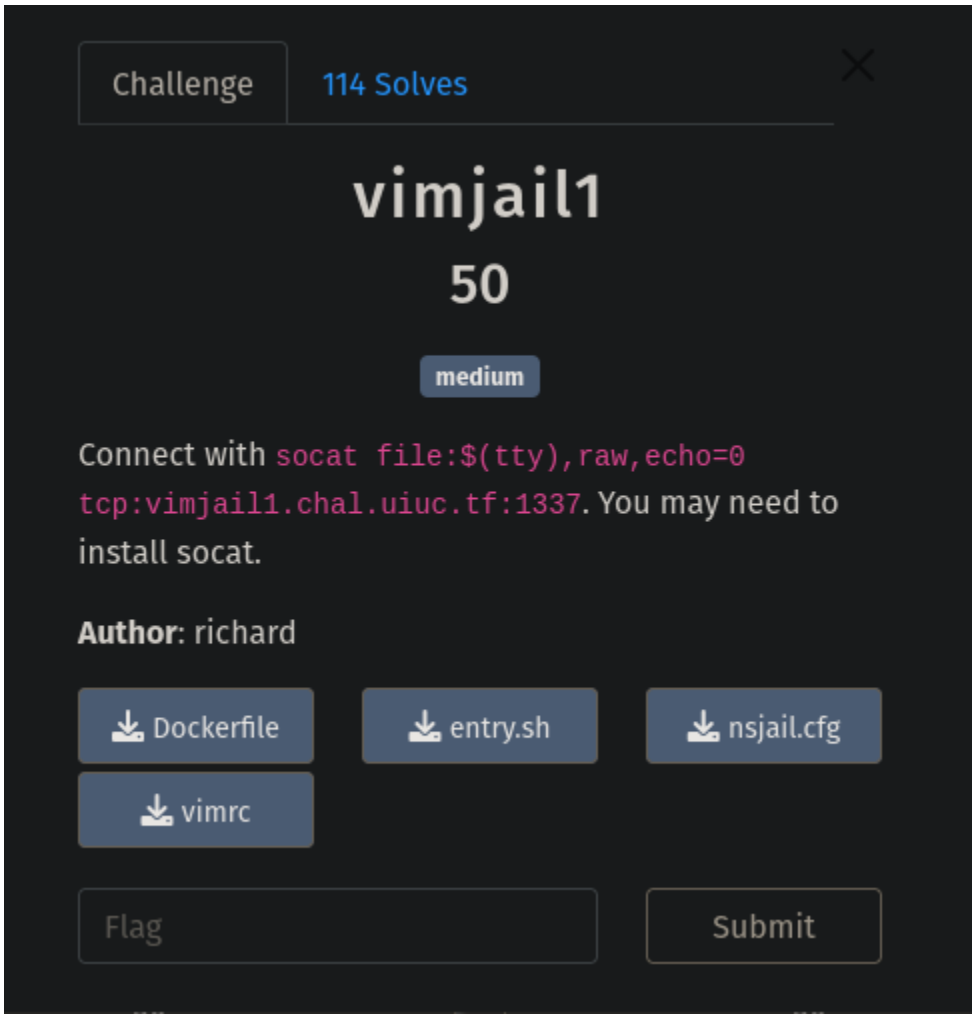
UIUCTF 2023 – Team Hacktoria

MISC

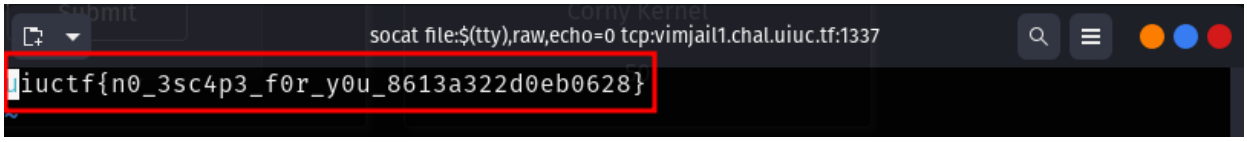
vimjail1



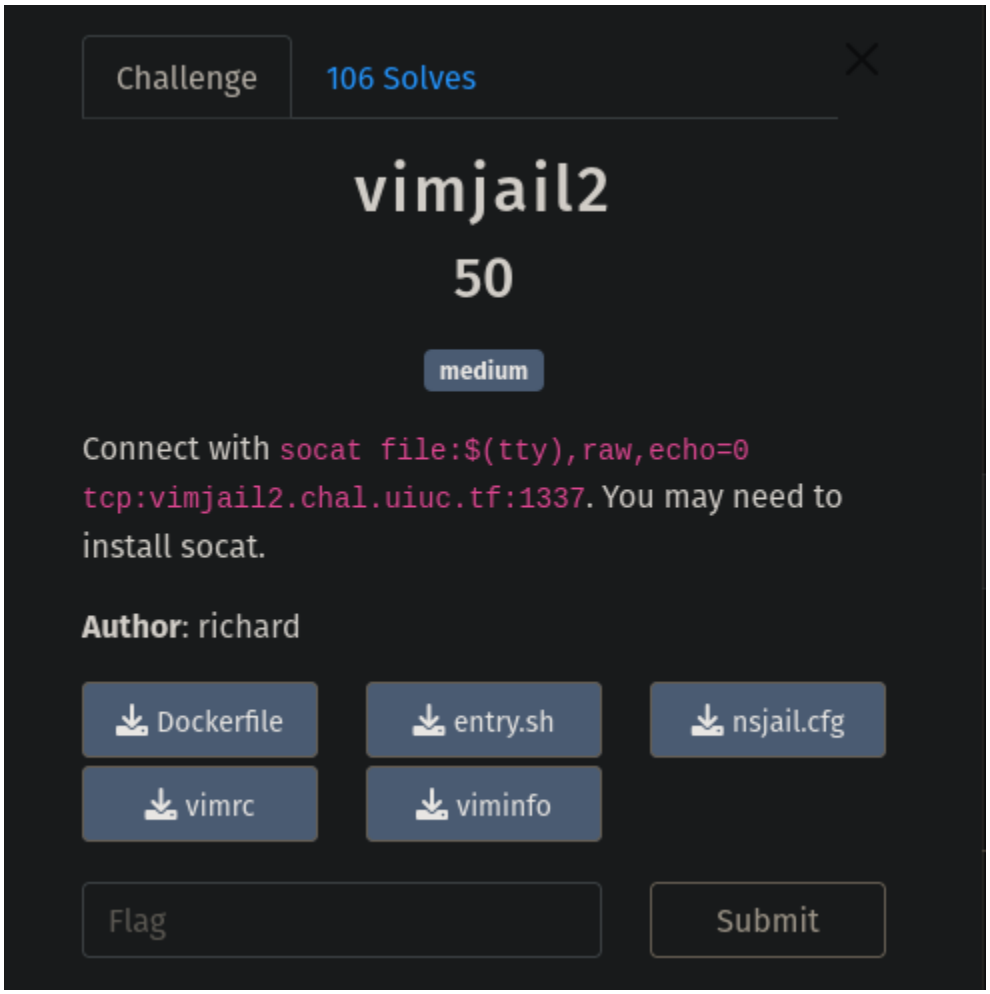
For this problem, we started off by looking at the vimrc file to see what our bounds are:

```
set nocompatible
set insertmode

inoremap <c-o> nope
inoremap <c-l> nope
inoremap <c-z> nope
inoremap <c-\><c-n> nope
```

We can see the remaps of some of the well-known ways to exit the insert mode in vim. After trying multiple things,we also see that it blocks `<c-\><c-n>`, but we can bypass this by typing <c-\> 2 times followed by <c-n> which lets us leave the insert mode. After this, we typed :e /flag.txt which allows us to edit the flag.txt and we get the flag:



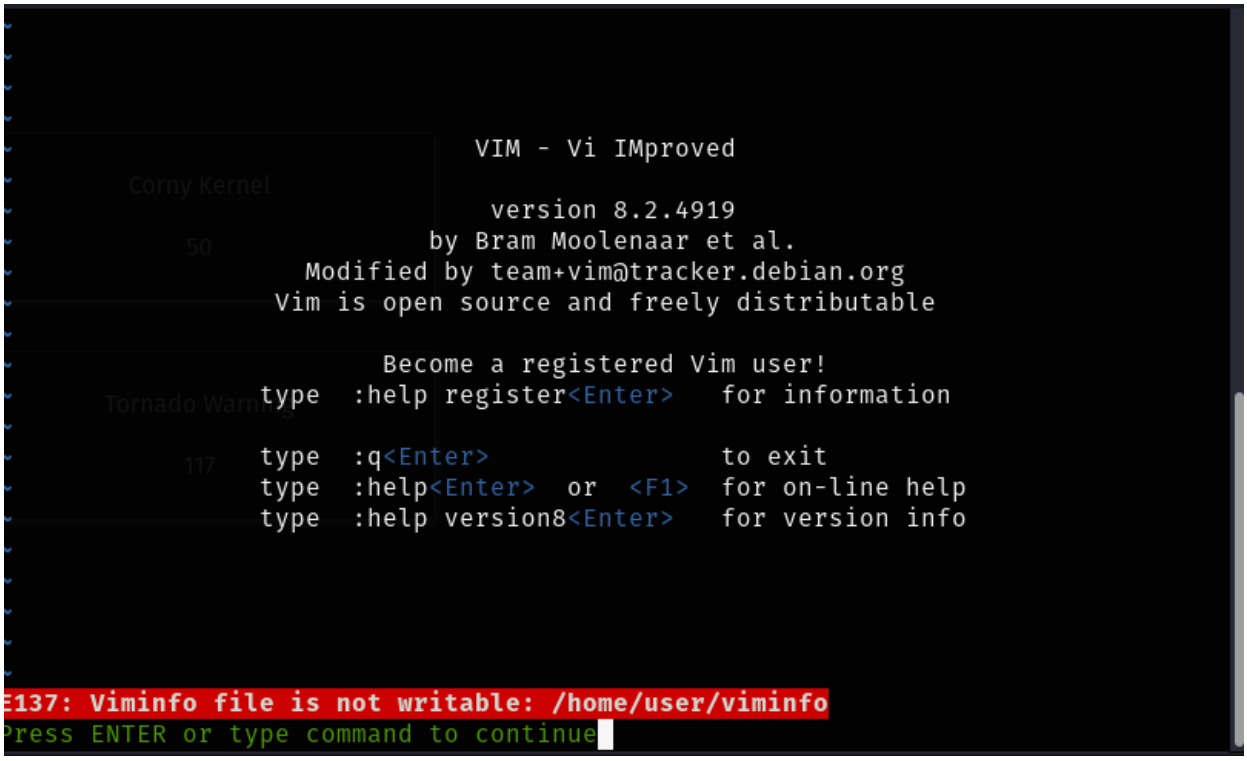
vimjail2



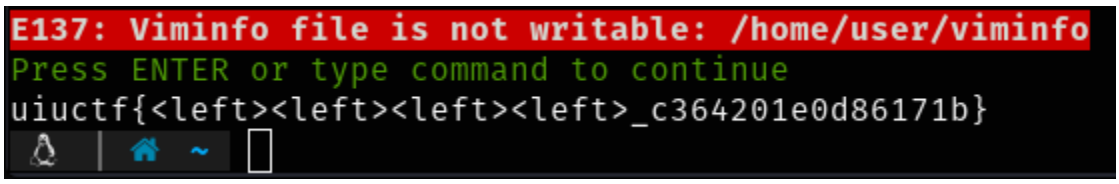
We followed a similar mindset to the previous vimjail question as well. We looked at what the bounds were:

```
set nocompatible
set insertmode

inoremap <c-o> nope
inoremap <c-l> nope
inoremap <c-z> nope
inoremap <c-\><c-n> nope

cnoremap a _
cnoremap b _
cnoremap c _
cnoremap d _
cnoremap e _
cnoremap f _
cnoremap g _
cnoremap h _
cnoremap i _
cnoremap j _
cnoremap k _
cnoremap l _
cnoremap m _
cnoremap n _
cnoremap o _
cnoremap p _
cnoremap r _
cnoremap s _
cnoremap t _
cnoremap u _
cnoremap v _
cnoremap w _
cnoremap x _
cnoremap y _
cnoremap z _
cnoremap ! _
cnoremap @ _
cnoremap # _
cnoremap $ _
cnoremap % _
cnoremap ^ _
cnoremap & _
cnoremap * _
cnoremap - _
cnoremap + _
cnoremap = _
cnoremap ` _
cnoremap ~ _
cnoremap { _
cnoremap } _
cnoremap [ _
cnoremap ] _
cnoremap \| _
cnoremap \ _
cnoremap ; _
```

Every character seemed to be remapped to a `_` except the character q. First we need to leave the insert mode and to do so we need to follow the same approach as vimjail1, which was typing <c-\> 2 times followed by <c-n>. Then after running `:q` to quit from vim, we are able to see this message:

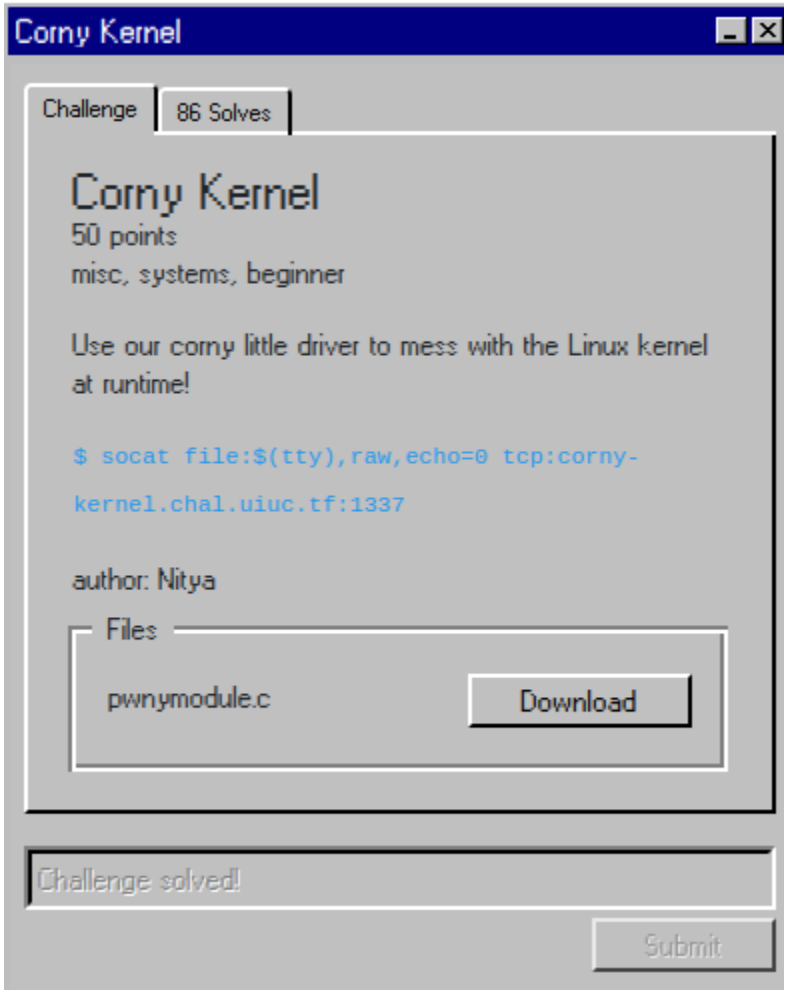


After we hit enter, we are able to see the flag:



Flag: uiuctf{<left><left><left><left>_c364201e0d86171b}

Corny Kernel



I started off by running the command provided:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ socat file:$(tty),raw,echo=0 tcp:corny-kernel.chal.uiuc.tf:1337
== proof-of-work: disabled ==
                            + mount -n -t proc -o nosuid,noexec,nodev proc /proc/
+ mkdir -p /dev /sys /etc
+ mount -n -t devtmpfs -o 'mode=0755,nosuid,noexec' devtmpfs /dev
+ mount -n -t sysfs -o nosuid,noexec,nodev sys /sys
+ cd /root
+ exec setsid cttyhack ash -l
/root #
```

I noticed that I was logged in as root. I see that we are given the kernel module, but it seemed to be zipped. I then ran `*gunzip*` on the file, and was able to see the kernel module:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ socat file:$(tty),raw,echo=0 tcp:corny-kernel.chal.uiuc.tf:1337
== proof-of-work: disabled ==
                              + mount -n -t proc -o nosuid,noexec,nodev proc /proc/
+ mkdir -p /dev /sys /etc
+ mount -n -t devtmpfs -o 'mode=0755,nosuid,noexec' devtmpfs /dev
+ mount -n -t sysfs -o nosuid,noexec,nodev sys /sys
+ cd /root
+ exec setsid cttyhack ash -l
/root # ls
pwnymodule.ko.gz
/root # gunzip pwnymodule.ko.gz
/root # ls
pwnymodule.ko
/root #
```

From there, I loaded the kernel module using `*insmod*`, which let me see a partial flag:

```
/root # insmod pwnymodule.ko
[  202.694291] pwnymodule: uiuctf{m4ster_
```

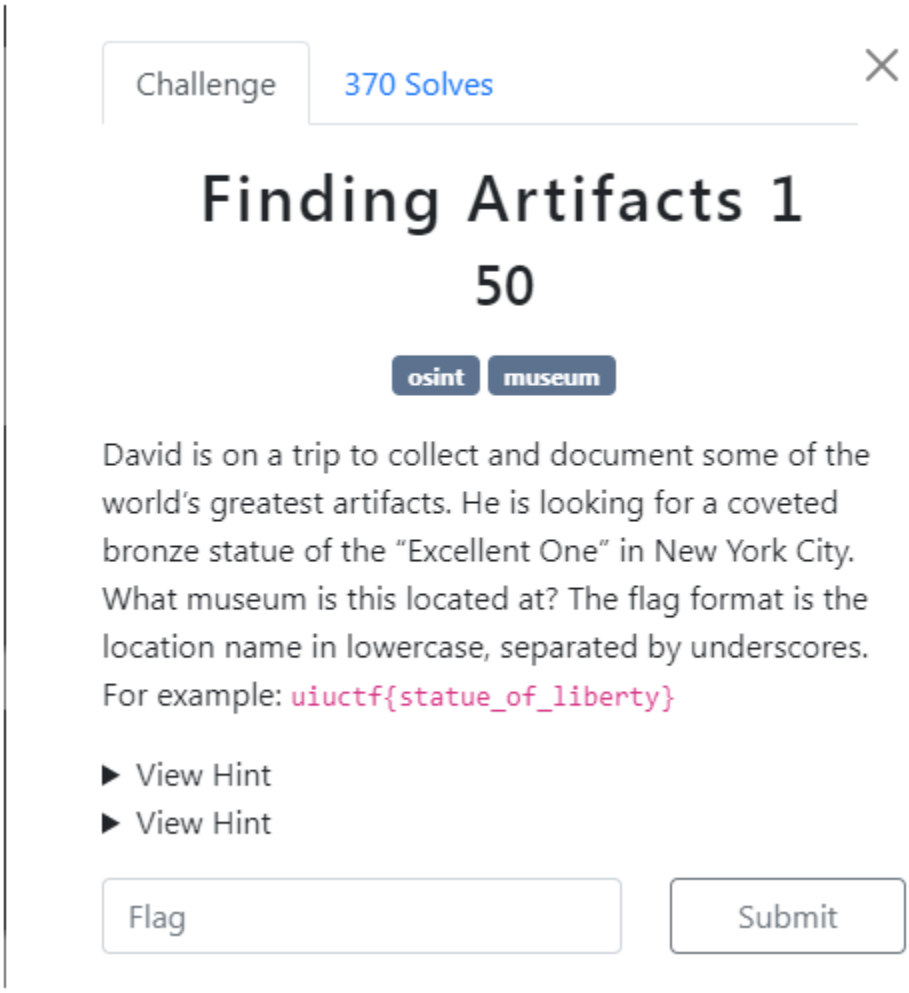
I then removed the module by running `*rmmod*`. I then ran `dmesg` to see the other debugging messages provided, and was able to get the flag:

```
[   11.606739] pwnymodule: uiuctf{m4ster_
[  119.125544] pwnymodule: k3rNE1_haCk3r}
```

Flag: uiuctf{m4ster_k3rNE1_haCk3r}

OSINT

Finding Artifacts 1:



For OSINT challenges, I like to follow the gap analysis ideology which means you ask the following questions:

1. What do I know?
2. What does this mean?
3. So what do I need to know?
4. How do I find out?

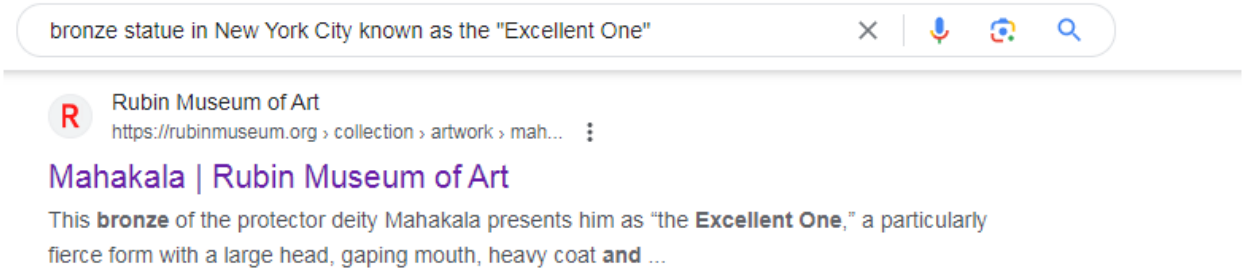
So let's start with **What do I know?:** By reading the instructions, we can directly know that we are looking for the location of one of the world's greatest artifacts, which is a coveted bronze statue of the "excellent one" in New York city.

So what does this mean?: Well, for this challenge it was obvious what that means. It means this is one of the world's greatest artifacts, which is a coveted bronze statue located in New York known as the "excellent one".

So what do I need to know?: Before finding the location, we have to find the statue.

How do I find out? Searching about any bronze statue that is known as the "excellent one" in New York.

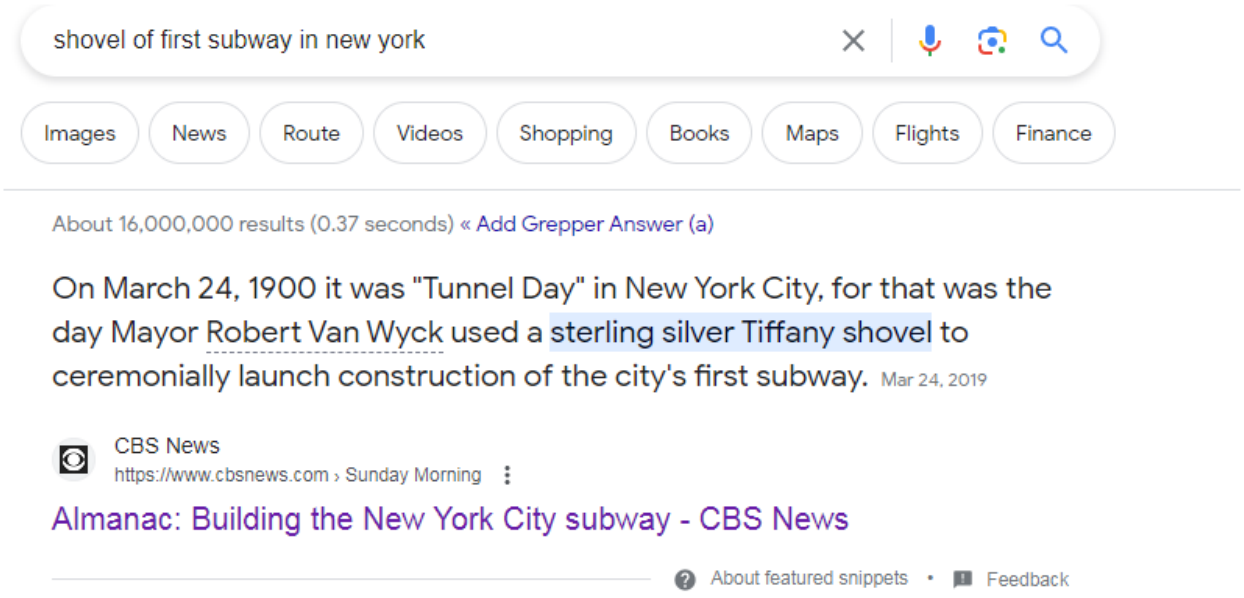
So let's start searching - First, I thought that maybe the statue is named the "excellent one" but of course , that was gonna be too easy. Seeing "excellent one" between quotes made me realize that the name of the statue means "excellent one" in another language or the thing it represents is known as the "excellent one", so without overthinking too much I went to google and searched for "bronze statue in new york city known as "excellent one"" and one of my results was this:



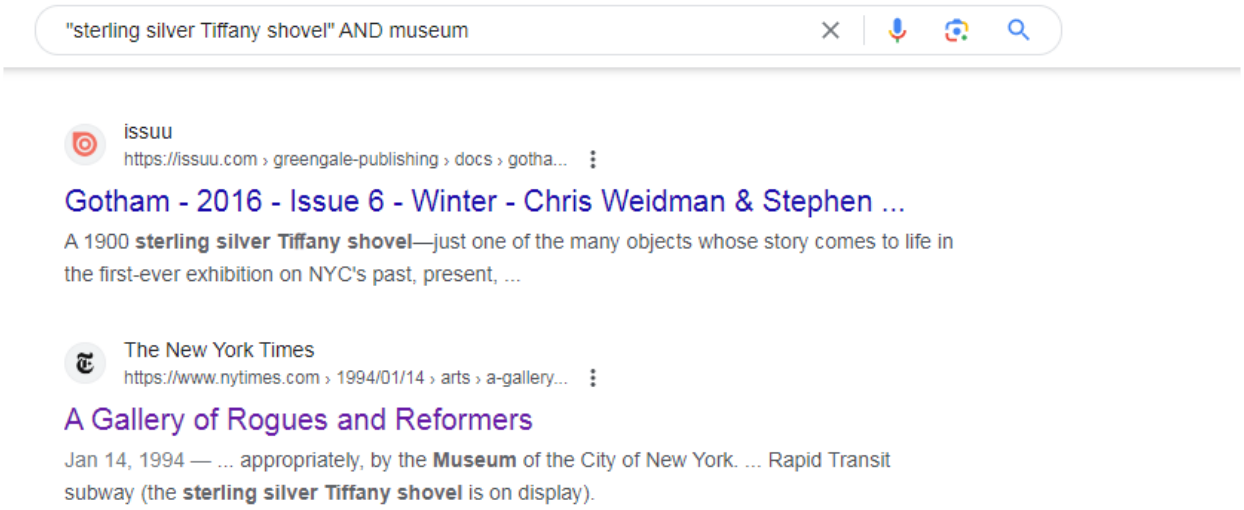
So this is the flag: uiuctf{rubin_museum_of_art}

Finding Artifacts 2:

Let's use a different path here, as for this challenge it is obvious that we are looking for the location of the first ever shovel used to start digging a subway in New York. First we search for the shovel: By typing "shovel of first subway in new york" we get the answer in the first result:

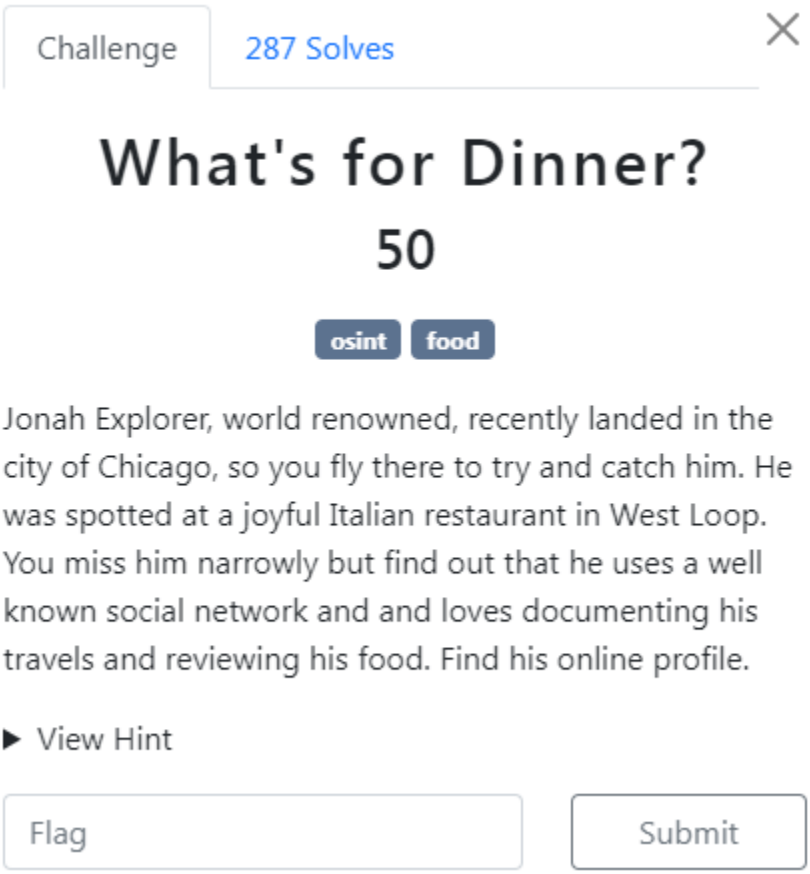


So it was a “sterling silver Tiffany shovel”. Next we search for the museum by searching the following: “"sterling silver Tiffany shovel" AND museum” and here we got the answer:



So the flag is: uiuctf{museum_of_the_city_of_new_york}

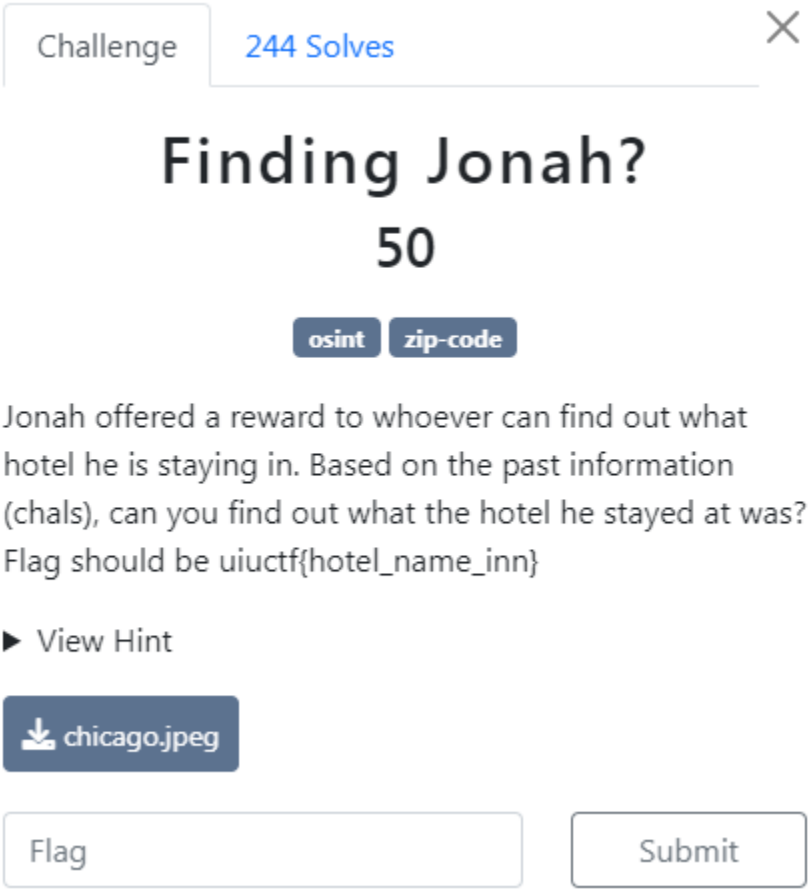
What's for Dinner?



This challenge was the easiest one. You can see in the first line the name “Jonah Explorer” - it is a social media challenge, and as the most used social media in CTFs is Twitter we went to twitter searching for “Jonah Explorer”.
In the first result we find the account and the flag:

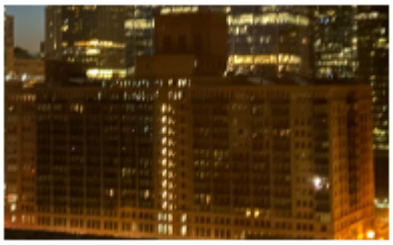
Flag: uiuctf{i_like_spaghetti}

Finding Jonah?

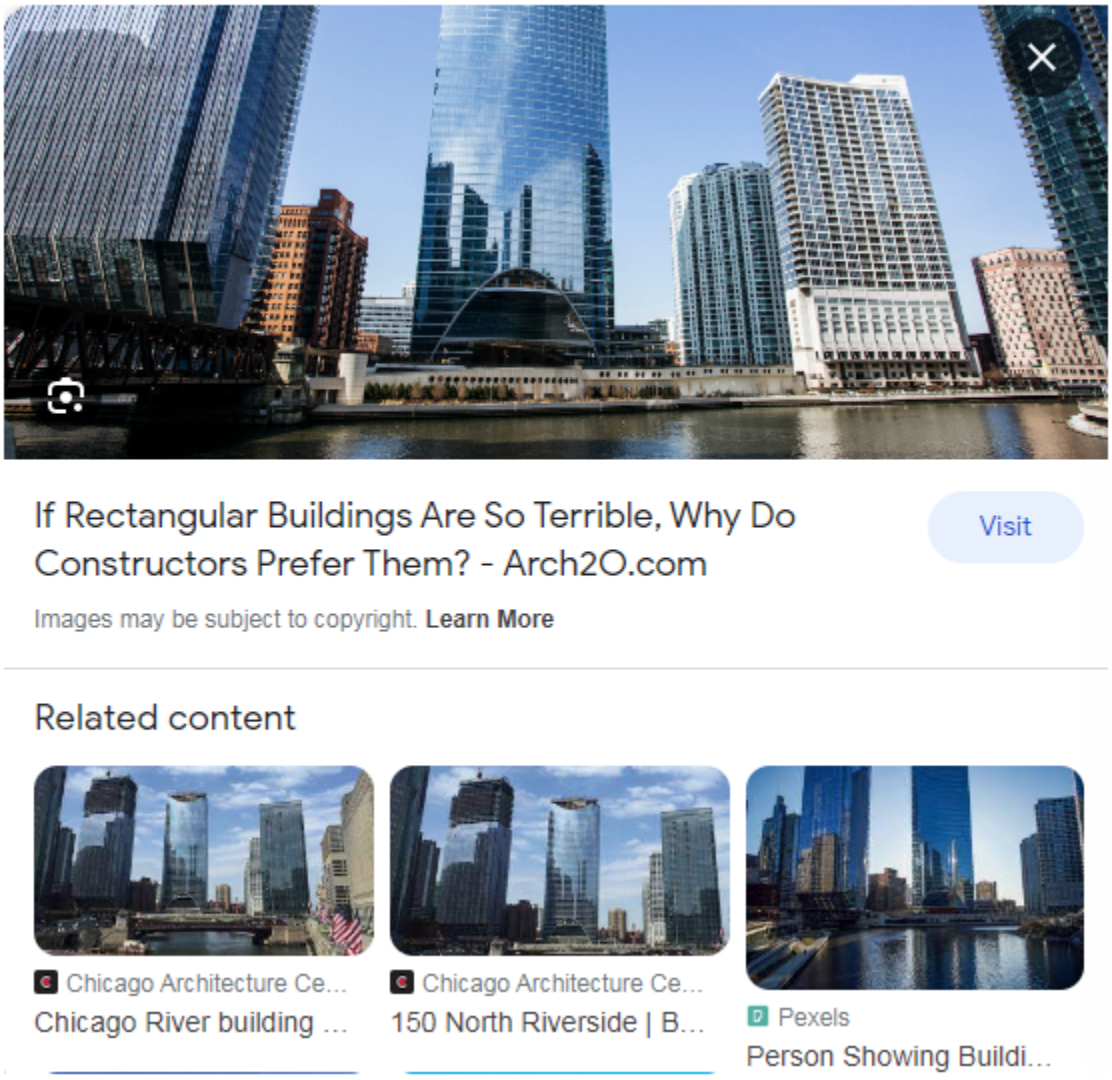


This challenge is a geolocation challenge which gave us this picture:

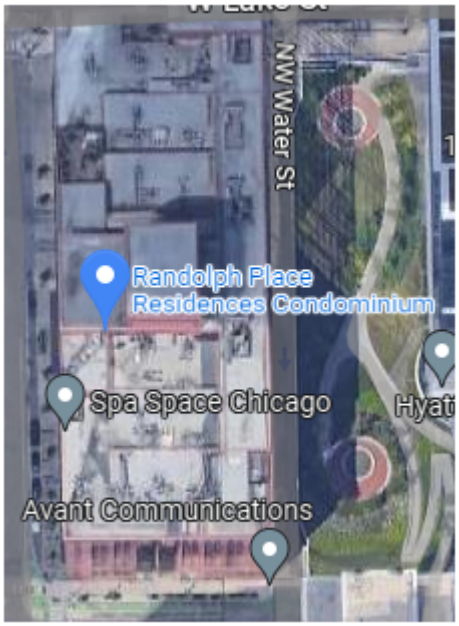
In this challenge the name of the picture says it is in Chicago, which made it easier. By analyzing the picture, we can see a rectangular building:



So I went to google and searched “rectangular building in chicago”. This search didn’t give me the picture of the building I was looking for, but it gave me another building in the picture:



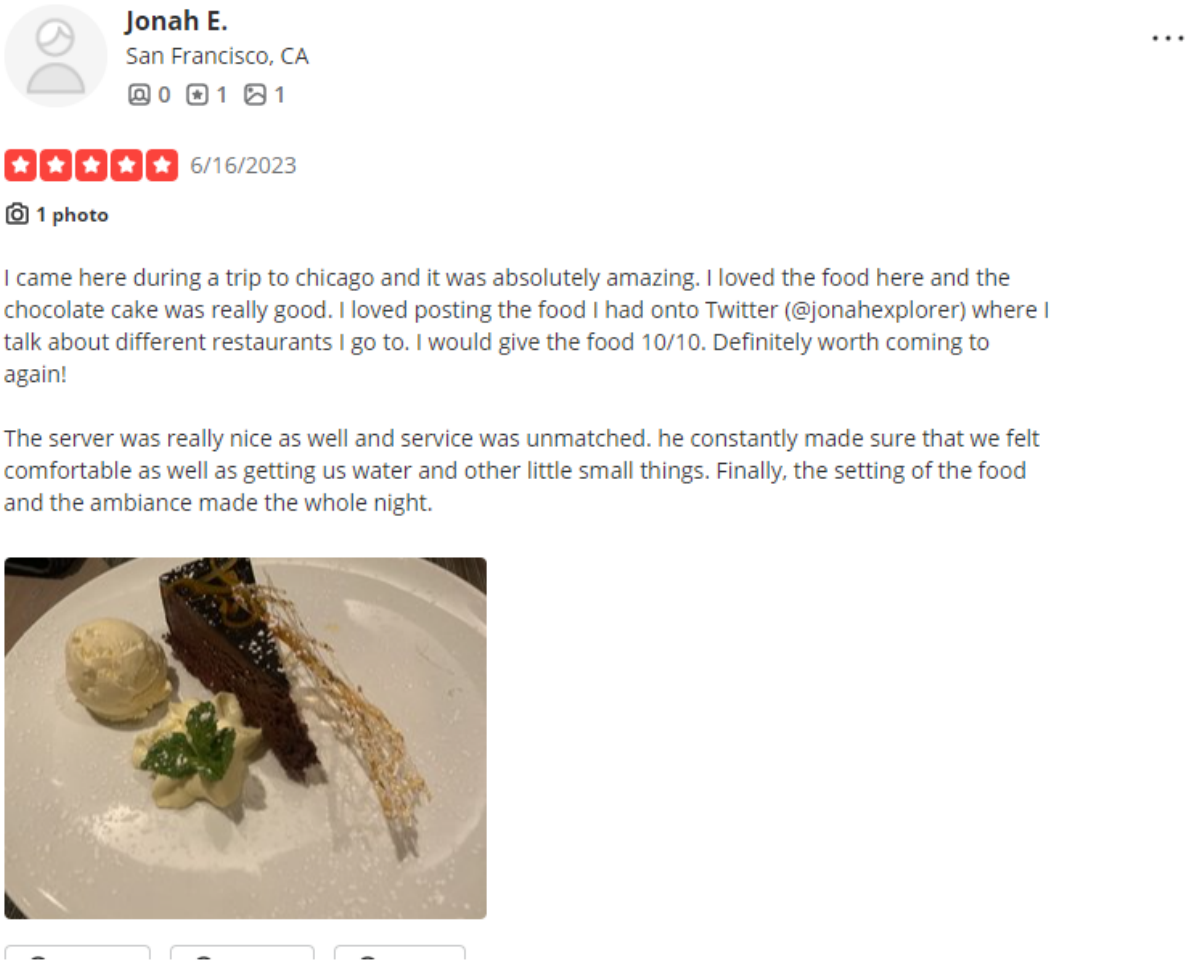
So now I know this building is called River Point in Chicago. For this challenge google earth pro would be the best choice, but for my connection it is slow, so I decided to go out with google maps. It was harder, but I got the same result. So this was the building i was searching for:



And by searching for hotels nearby, I was able to identify the hotel which is hampton inn:
So the flag: uiuctf{hampton_inn}

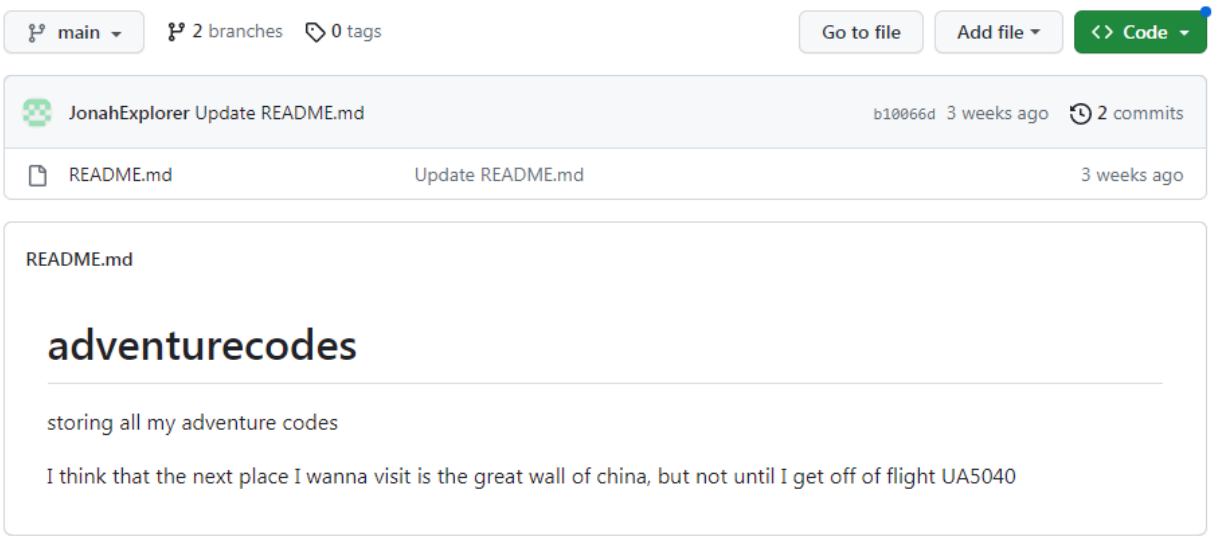
Jonah’s journal

So for this challenge, I was able to find answers in 2 ways. The first way is a bit of guessing and the second way I believe is the right one. So let’s start with the guessy way:
First i searched about his username “jonahexpolrer” in duckduckgo which gave me this link:
https://www.yelp.com/biz/gioia-ristorante-pastificio-chicago
It is an italian restaurant in chicago:

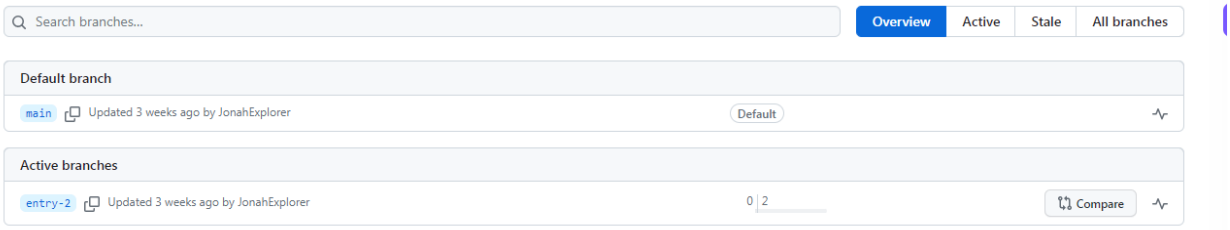


So after that, I just guessed that since it's an italian restaurant, the next destination is Italy, lol, and it was the right answer.

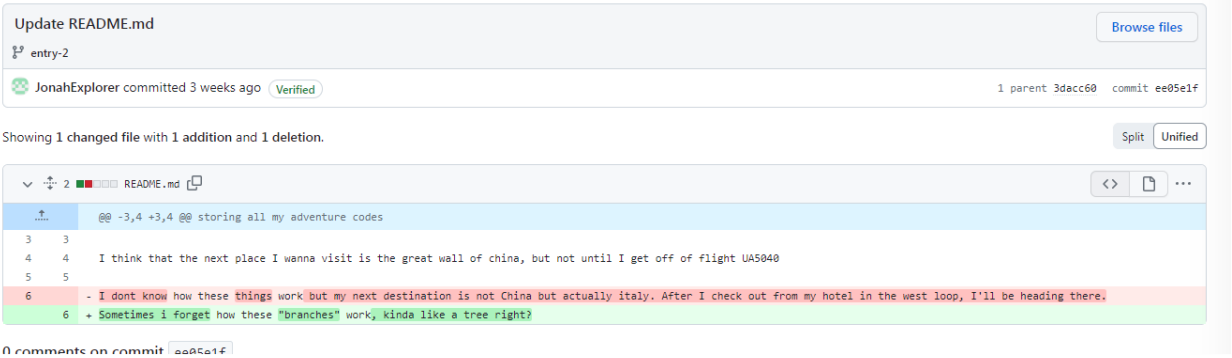
The intended solution was after I searched for his username in duckduckgo I also got his github account https://github.com/JonahExplorer. We can see on repository called adventurecodes:



He says that he wants to visit China but that is not the answer. Then we can see in the picture there are 2 branches. Navigating to the second branch led to the following:

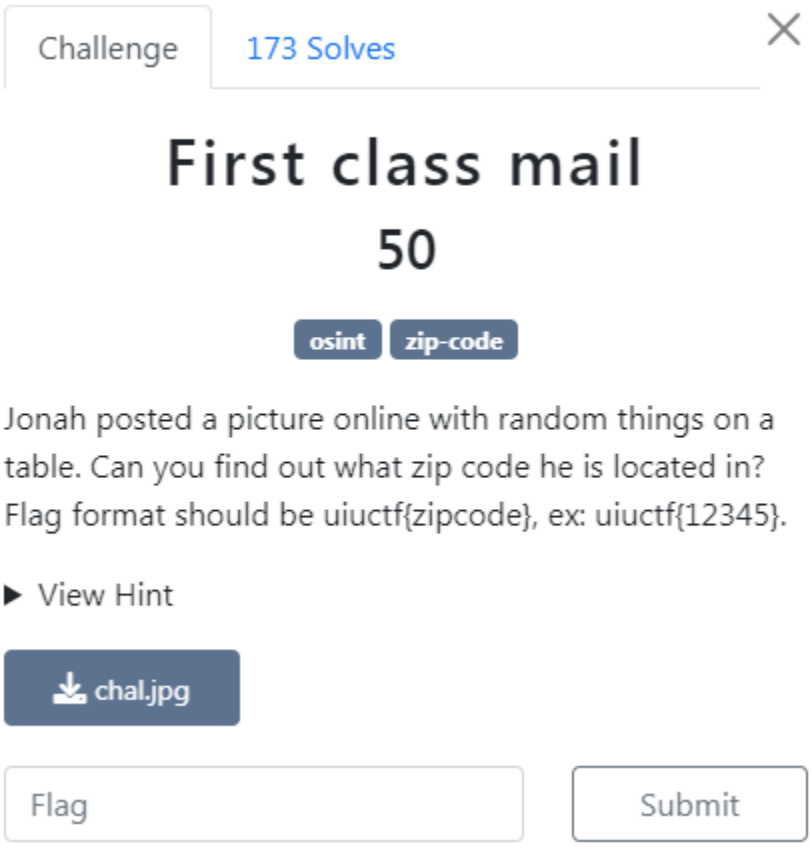


Press compare, then the hash ee05e1f and here the answer:

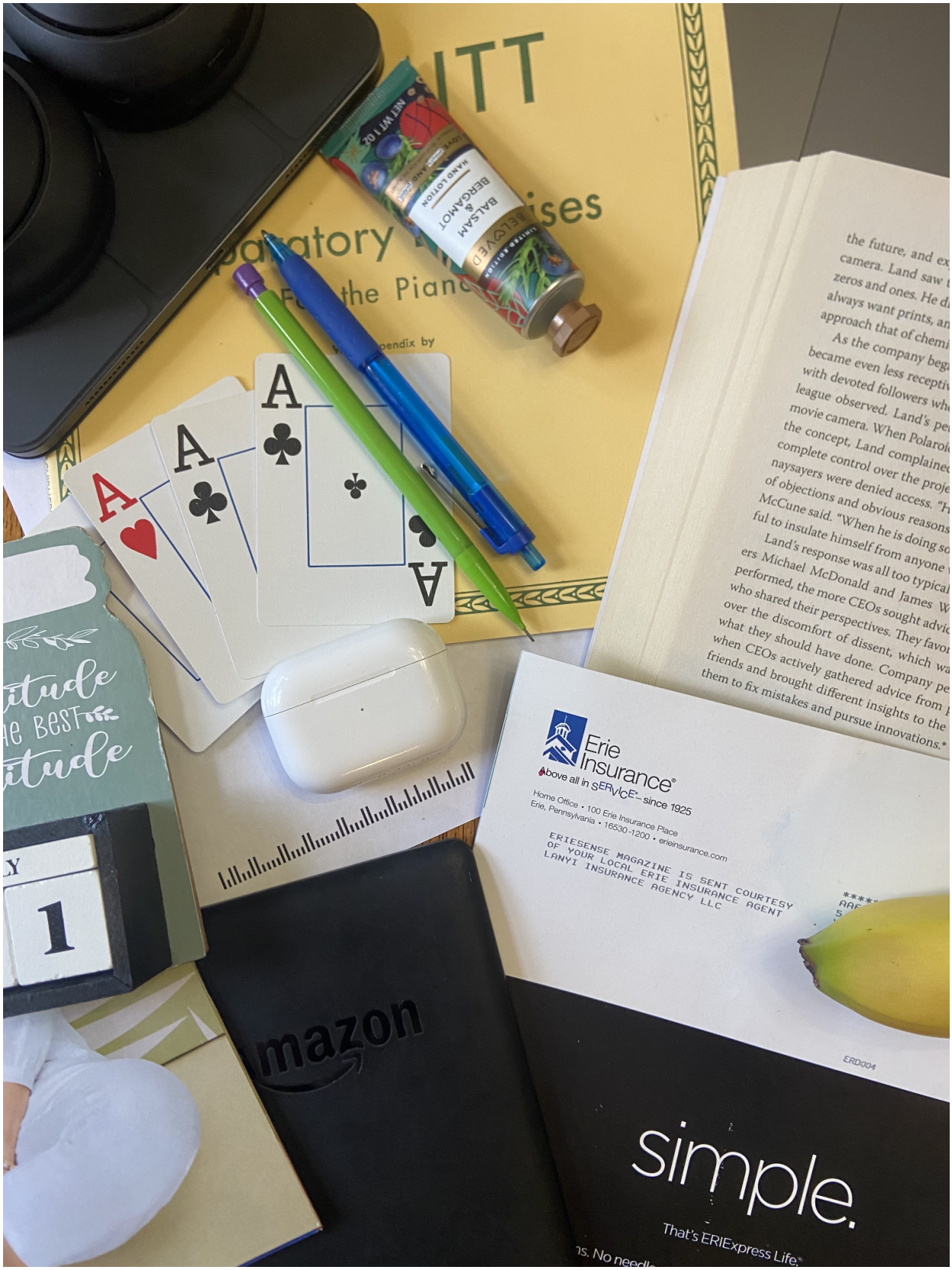


So the flag: uiuctf{italy}

First class mail

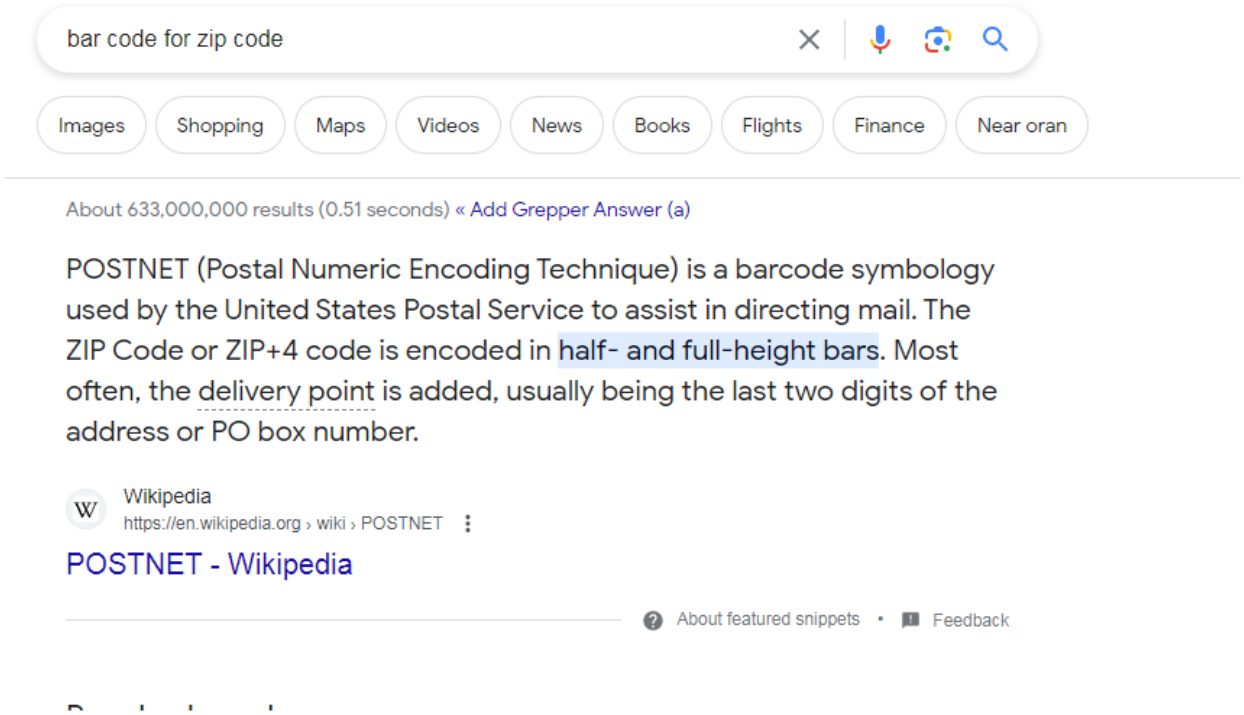


For this challenge we also have a given picture which is this:


Erie Insurance
above all in SERVICE since 1925
Home Office • 100 Erie Insurance Place
Erie, Pennsylvania • 16530-1200 • erieinsurance.com
ERIESENSE MAGAZINE IS SENT COURTESY OF YOUR LOCAL ERIE INSURANCE AGENT
LANYI INSURANCE AGENCY LLC
ERD004
simple.
That's ERIExpress Life.
amazon
BALSAM & BERGAMOT
HAND LOTION
NET WT 1 OZ

By analyzing the picture nothing seemed useful other than this:

It looked like a barcode to me, and the instruction of challenge says we are looking for a zip code so I searched in google “bar code for zip code” and this was the result:



So using a POSTNET online decoder https://www.dynamsoft.com/barcode-reader/barcode-types/postnet/ gave us this result: 6066111234

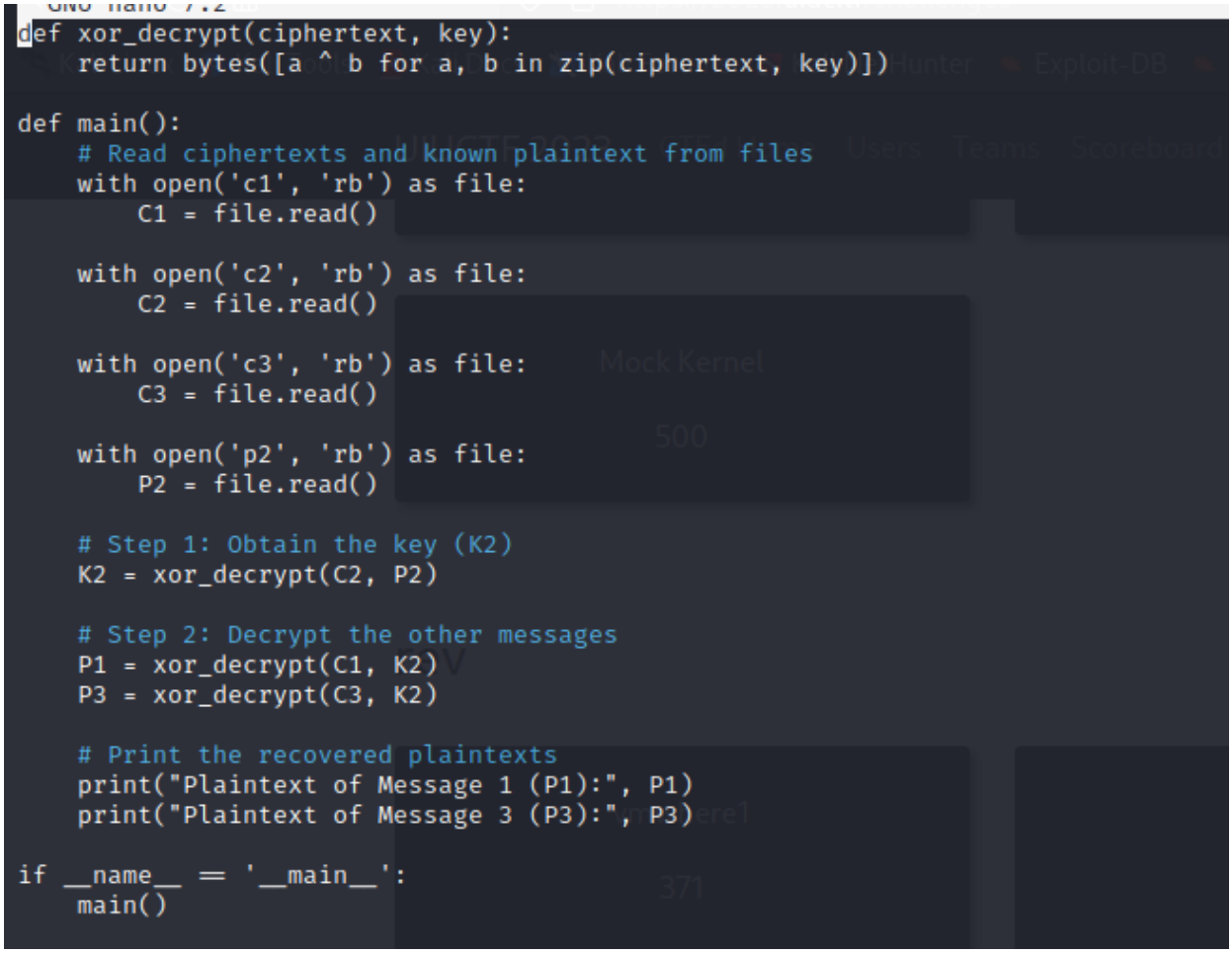
By reading a bit of the wikipedia page we conclude that the sum of all digits should be divisible by 10, so 6+0+6+6+1+1+1+2+3+4=30 means it is written in this format: 60661-1123 which lets us know that the zip code is 60661.

The flag: uiuctf{60661}

CRYPTO

Three-Time Pad

You are given 3 cipher texts and 1 plain text (for the second cipher text), so you have to XOR c2 with p2 to get the key. After getting the key you can use the key to XOR it again with cipher message 1 and cipher message 3. After this, you will get the flag.
I used this script:



```python
def xor_decrypt(ciphertext, key):
    return bytes([a ^ b for a, b in zip(ciphertext, key)])

def main():
    # Read ciphertexts and known plaintext from files
    with open('c1', 'rb') as file:
        C1 = file.read()

    with open('c2', 'rb') as file:
        C2 = file.read()

    with open('c3', 'rb') as file:
        C3 = file.read()

    with open('p2', 'rb') as file:
        P2 = file.read()

    # Step 1: Obtain the key (K2)
    K2 = xor_decrypt(C2, P2)

    # Step 2: Decrypt the other messages
    P1 = xor_decrypt(C1, K2)
    P3 = xor_decrypt(C3, K2)

    # Print the recovered plaintexts
    print("Plaintext of Message 1 (P1):", P1)
    print("Plaintext of Message 3 (P3):", P3)

if __name__ == '__main__':
    main()
```

```
┌──(root㉿qaisqupti)-[/home/qaisqupti/Downloads]
└─# python ez
Plaintext of Message 1 (P1): b'before computers, one-time pads were sometimes'
Plaintext of Message 3 (P3): b'uiuctf{burn_3ach_k3y_aft3r_us1ng_1t}'
```

Flag: uiuctf{burn_3ach_k3y_aft3r_us1ng_1t}

At Home

Writeup by voxelbugged

I started by downloading the files. I noticed that chal.txt contained three large integer values - e, n, c. It had nothing else, so without the python script I could not do anything with this file just yet. Of course, I opened the script next. It detailed the exact steps of how the values in chal.txt were calculated, as expected.
However, the most interesting part was that the flag was converted into an int and then used in this equation:

```
c = (flag * e) % n
```

This is a modular equation, and the value of flag could be calculated with a simple python script to solve for x as I already knew the values of the other numbers:

```python
import math

def modular_inverse(e, n):
    g, x, _ = extended_gcd(e, n)
    if g == 1:
        return x % n

def extended_gcd(a, b):
    if a == 0:
        return b, 0, 1
    else:
        g, y, x = extended_gcd(b % a, a)
        return g, x - (b // a) * y, y

def solve_for_x(e, n, c):
    d = modular_inverse(e, n)
    if d is not None:
        x = (c * d) % n
        return x

e =
3590503891528215534161395815035053470579252085604518644266341003331165604223136392602839814968249200897894978185201051896843118232507955200581117633104282026544393519223617227315577436407
n =
2686611247680500440660820998667333729621683371086008990123843295238481171468440400188535405203911234020955722625665066118684372692595812533497441211147124446241957729405174414181741151229
c =
6774337446244858210744016851368752043459452933182174073739611640792811104381508466500210419675402053046936053925332373893570841436300537345878204195545027895434830640154237430978893872065

x = solve_for_x(e, n, c)
byte_x = int.to_bytes(x, 69, "big")
print(byte_x)
```

Once I got the int value, I converted it back to bytes and received the flag:

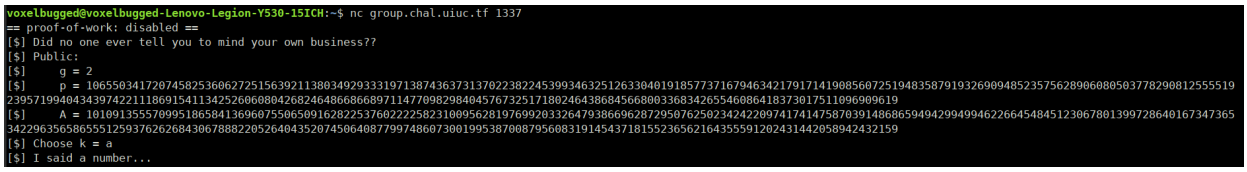
```
voxelbugged@voxelbugged-Lenovo-Legion-Y530-15ICH:~/Downloads$ python solve.py
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00uiuctf{W3_hav3_R5A_@_h0m3}'
```

Flag: uiuctf{W3_hav3_R5A_@_h0m3}

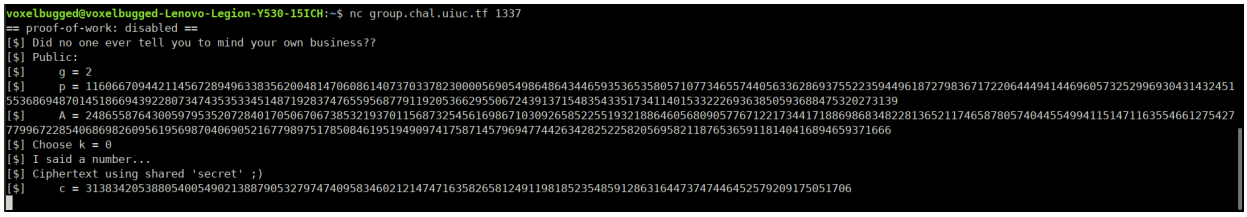
Group Project

Writeup by voxelbugged

I started by simply connecting to the challenge. I saw a message containing three numbers: “g” , “p”, “A”, along with a prompt asking me to “Choose k”. The first thing I did was type in a letter to test what would happen. I got met with a message saying “I said a number…” and the program hung up.



I connected to the challenge again, this time putting in k as 0. A few interesting things happened:



The first thing is that the “p” and “A” numbers have changed, suggesting random number generation is involved. The second and more important part is that the challenge actually let it through, giving me both the error message AND the actual cipher!
At this point I downloaded the python script for the challenge to check how it works, and this is when I noticed the exploitable part:

```
Ak = pow(A, k, p)
b = randint(2, p - 1)
B = pow(g, b, p)
Bk = pow(B, k, p)
S = pow(Bk, a, p)

key = hashlib.md5(long_to_bytes(S)).digest()
cipher = AES.new(key, AES.MODE_ECB)
c = int.from_bytes(cipher.encrypt(pad(flag, 16)), "big")
```

It turns out that setting “k” to 0 forces “S” to always equal 1, which means the resulting cipher will always equal the same value, no matter what numbers are generated for “p” and “A”!
After learning this, I wrote a quick python script to reverse this process and get the flag based on the “S” and “c” values:

```python
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from Crypto.Util.number import bytes_to_long

def reverse_process(S, c):

    key = hashlib.md5(S.to_bytes((S.bit_length() + 7) // 8, "big")).digest()

    cipher = AES.new(key, AES.MODE_ECB)

    decrypted_data = unpad(cipher.decrypt(c.to_bytes((c.bit_length() + 7) // 8, "big")), 16)

    flag = decrypted_data.decode()

    return flag

S = 1
c = 313834205388054005490213887905327974740958346021214747163582658124911981852354859128631644737474464525792091750517006

flag = reverse_process(S, c)
print("Original flag:", flag)
```

After running this, the original flag is outputted:

```
voxelbugged@voxelbugged-Lenovo-Legion-Y530-15ICH:~/Downloads$ python reverser.py
Original flag: uiuctf{brut3f0rc3_a1n't_s0_b4d_aft3r_all!!11!!}
```

Flag: uiuctf{brut3f0rc3_a1n't_s0_b4d_aft3r_all!!11!!}

PWN

REV

WEB