Lab 02

디컴파일러의 일종인 IDA Freeware를 이용해 바이너리 분석을 진행했습니다.

IDA View-A | Pseudocode-A | Hex View-1

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3   init(argc, argv, envp);
4   puts("Let's spice it up! Trigger print_flag() to obtain your flag.");
5   printf("- Address of main(): %p\n", main);
6   vulnerable();
7   return 0;
8 }
```

- main()의 디컴파일 결과

vulnerable() 함수가 수상합니다.

IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums

```
1  __int64 vulnerable()
2  {
3    unsigned int v0; // eax
4    int v1; // eax
5    __int64 result; // rax
6    char v3[24]; // [rsp+0h] [rbp-20h] BYREF
7    __int64 v4; // [rsp+18h] [rbp-8h]
8
9    v0 = time(0LL);
10   srand(v0);
11   v1 = rand();
12   canary = v1;
13   v4 = v1;
14   printf("Enter input: ");
15   read(0, v3, 0x40uLL);
16   for ( i = 0; i <= 63; ++i )
17   {
18     if ( v3[i] > 32 && v3[i] <= 84 )
19     {
20       printf("[-] Error: Prohibited byte detected at buf[%d]: 0x%02X\n", (unsigned int)i, (unsigned int)v3[i]);
21       exit(-1);
22     }
23   }
24   result = canary;
25   if ( v4 != canary )
26   {
27     puts("[-] Error: Canary has been modified");
28     exit(-1);
29   }
30   return result;
31 }
```

- vulnerable()의 디컴파일 결과

time(0)을 이용해 srand를 초기화하고, 이를 이용해 custom canary를 만듭니다. read(0, v3, 0x40)을 통해 v3의 위치에 0x40bytes 만큼을 읽습니다. 그 후, 읽은 byte들에 대해 32보다 크고, 84보다 같거나 작은지 확인합니다. 만약 금지된 byte가 존재한다면 프로그램을 종료합니다.

1. vulnerable()의 stack 메모리 맵

vulnerable()의 stack상태를 보기 위해 gdb로 실행된 파일에 breakpoint를 걸고, "aaaaaaaaaaaaaaaa"를 입력해 동적 디버깅을 해보자.

```
   0x5555555553a0 <vulnerable+0052> mov    edx, 0x40
   0x5555555553a5 <vulnerable+0057> mov    rsi, rax
   0x5555555553a8 <vulnerable+005a> mov    edi, 0x0
 → 0x5555555553ad <vulnerable+005f> call   0x555555555120 <read@plt>
   ↳  0x555555555120 <read@plt+0000>  endbr64
      0x555555555124 <read@plt+0004>  bnd    jmp QWORD PTR [rip+0x2e6d]        # 0x555555557f98 <read@got.plt>
      0x55555555512b <read@plt+000b>  nop    DWORD PTR [rax+rax*1+0x0]
      0x555555555130 <srand@plt+0000> endbr64
      0x555555555134 <srand@plt+0004> bnd    jmp QWORD PTR [rip+0x2e65]        # 0x555555557fa0 <srand@got.plt>
      0x55555555513b <srand@plt+000b> nop    DWORD PTR [rax+rax*1+0x0]
──────────────────────────────────────────────────────────────────────────────
read@plt (
   $rdi = 0x0000000000000000,
   $rsi = 0x00007fffffffc8b0 → 0x0000000000000000,
   $rdx = 0x0000000000000040
)
──────────────────────────────────────────────────────────────────────────────
[#0] Id 1, Name: "target", stopped 0x5555555553ad in vulnerable (), reason: SINGLE STEP
──────────────────────────────────────────────────────────────────────────────
[#0] 0x5555555553ad → vulnerable()
[#1] 0x5555555554a6 → main()
──────────────────────────────────────────────────────────────────────────────
gef➤
aaaaaaaaaaaaaaaa
```

read를 통해 입력을 받는 곳에서 "aaaaaaaaaaaaaaaa"를 입력했다.

입력 한 후, `x/10gx $rsp`를 통해 16byte를 rsp부터 10개 읽습니다.

```
gef➤  x/10gx $rsp
0x7fffffffc8b0: 0x6161616161616161      0x6161616161616161
0x7fffffffc8c0: 0x00007fffffffca0a      0x000000004d14e732
0x7fffffffc8d0: 0x00007fffffffc8f0      0x000055555555554a6
0x7fffffffc8e0: 0x00007fffffffca08      0x0000000100000000
0x7fffffffc8f0: 0x0000000000000001      0x00007ffff7dafd90
gef➤  disass main
Dump of assembler code for function main:
   0x0000555555555457 <+0>:     endbr64
   0x000055555555545b <+4>:     push   rbp
   0x000055555555545c <+5>:     mov    rbp,rsp
   0x000055555555545f <+8>:     sub    rsp,0x10
   0x0000555555555463 <+12>:    mov    DWORD PTR [rbp-0x4],edi
   0x0000555555555466 <+15>:    mov    QWORD PTR [rbp-0x10],rsi
   0x000055555555546a <+19>:    call   0x555555555289 <init>
   0x000055555555546f <+24>:    lea    rax,[rip+0xc2a]        # 0x5555555560a0
   0x0000555555555476 <+31>:    mov    rdi,rax
   0x0000555555555479 <+34>:    call   0x5555555550f0 <puts@plt>
   0x000055555555547e <+39>:    lea    rax,[rip+0xfffffffffffffdd2]        # 0x555555555457 <main>
   0x0000555555555485 <+46>:    mov    rsi,rax
   0x0000555555555488 <+49>:    lea    rax,[rip+0xc4e]        # 0x5555555560dd
   0x000055555555548f <+56>:    mov    rdi,rax
   0x0000555555555492 <+59>:    mov    eax,0x0
   0x0000555555555497 <+64>:    call   0x555555555100 <printf@plt>
   0x000055555555549c <+69>:    mov    eax,0x0
   0x00005555555554a1 <+74>:    call   0x55555555534e <vulnerable>
   0x00005555555554a6 <+79>:    mov    eax,0x0
   0x00005555555554ab <+84>:    leave
   0x00005555555554ac <+85>:    ret
End of assembler dump.
```

메모리를 확인하면, "a"의 byte 값인 `0x61`이 16개 들어간 것을 확인 할 수 있습니다. 그리고, `$rsp`에서 `0x24`만큼 떨어진 곳에는 4 bytes canary와 4 bytes `\x00` padding이 들어 있고, `$rsp`로부터 `0x40`만큼 떨어진 곳에는 다시 `main()`으로 돌아가는 return address가 들어있습니다.

```
$rsp+0x00: v3
$rsp+0x08: v3
$rsp+0x10: v3
$rsp+0x18: canary + \x00 padding
$rsp+0x20: ??
$rsp+0x28: return main()
```

vulnerable()의 메모리 맵은 다음과 같습니다. `$rsp`로부터 24bytes는 v3에 해당하고, 그 다음은 8bytes로 패딩된 4bytes canary가 들어 있습니다. 8bytes 떨어진 곳에는 다시 main() 함수로 돌아가는 return address가 들어있습니다.

2. vulnerable()의 취약점

v3을 보면 char 타입으로 24 bytes의 크기를 가지고 있습니다. read를 통해 64 bytes를 입력받으므로, 입력 범위를 벗어나, 다른 메모리에 접근할 가능성이 있습니다.
위에서 분석한 메모리 맵을 보면, 입력하는 경우에서 `$rsp`로부터 `0x28` offset 떨어진 곳에 return addess가 담겨 있습니다. stack buffer overflow 취약점을 이용해 이 return address를 임의의 주소를 입력하면 그 주소로 control flow를 변경할 수 있습니다.

3. custom canary의 취약점

디컴파일 된 코드를 보면, 다음과 같은 로직으로 custom canary를 만듭니다.

```
v0 = time(0LL);
srand(v0);
v1 = rand();
canary = v1;
```

C의 random 모듈에는 여러 취약점이 존재합니다. 위 코드에서는 `time(0)`, 즉 현재 시간을 사용해 srand()의 seed를 설정합니다. 이 현재 시간은 바이너리가 빌드된 시간이 아니라, 파일을 실행할 때, 결정되므로, 이 seed를 파일을 실행할 때의 시간으로 얻어낼 수 있습니다.
srand()의 seed를 설정하면, rand()를 실행할 때, 항상 동일한 output을 얻을 수 있으므로, 충분히 canary 값을 복구할 수 있습니다. 풀이에서는 python의 `ctype` 모듈을 이용해 c와 같은 방식으로 srand의 seed 설정과 rand()를 사용할 수 있는 방법을 사용했습니다.
C에서의 rand()를 통해 만들어지는 것은 4bytes의 정수이고, 이것이 x86-64에서 저장되므로, 8bytes 공간에 little endian으로 넣습니다.

4. 특정 바이트 금지 로직

```
for ( i = 0; i <= 63; ++i )
  {
    if ( v3[i] > 32 && v3[i] <= 84 )
    {
      ...
```

```
        exit(-1);
      }
    }
```

C언어에서의 random 모듈 LCG (선형 합동 생성기)를 이용합니다.

$$X_{n+1} = (1664525X_n + 1013904223) \mod 2^{32}$$

-> 선형 합동 생성기의 예시.
이 자체로 취약한 난수 생성을 하지만, 모든 바이트가 균일하게 나온다고 가정하면, 0 ~ 32, 85 ~ 255인 바이트만 검사 로직을 통과할 수 있으므로, `204/256`의 확률로 통과할 수 있습니다.
canary는 4bytes로, 나머지는 `\x00`으로 little endian에 맞춰 패딩되어 있습니다. canary는 C의 rand()로 완전하지는 않지만 랜덤하게 생성되므로, 모든 바이트가 생성될 확률이 같다고 가정하면, 4개의 bytes가 `204/256` 확률을 모두 통과하면 됩니다.
`print_flag()` 함수의 주소를 탐색하면, MSB의 byte는 55 또는, 56 이고, LSB의 bytes는 ea로 고정되어 있습니다. 즉 6개의 bytes만 검사 로직을 통과하면 됩니다.
즉, `204/256`의 확률을 10번 통과하면 되므로, 우리가 만든 exploit이 통과할 확률은 다음과 같습니다.

```
(204/256) ^ 10
```

위를 계산하면, 약 10%의 확률이 나옵니다. 그러므로, exploit을 여러번 반복해 합리적으로 통과할 수 있습니다.

5. ASLR로 print_flag()를 찾는 로직

프로그램을 실행하면 `main`의 주소를 얻을 수 있습니다. gdb로 target을 실행하고, `info func` 을 입력하면, `print_flag()`의 주소를 구하기 위해 offset을 구해야합니다. main()의 offset은 `0x0000000000001457`, print_flag()의 offset은 `0x00000000000012ea` 이므로, 다음과 같은 계산으로 flag의 주소를 구할 수 있습니다.

```
address_flag = address_main - (0x0000000000001457 - 0x00000000000012ea
```

`vulnerable()`의 return 주소를 여기서 구한 `print_flag()`의 주소로 덮어쓰면, flag를 얻을 수 있습니다.

```
csed415-lab02@csed415:/tmp/jjong22/lab02$ python3 exploit.py
[+] Starting local process '/home/csed415-lab02/target': pid 111
canary: 0x78d27804
0x5568da02e2ea
[*] Switching to interactive mode
Enter input: [*] Process '/home/csed415-lab02/target' stopped with exit code 0 (pid 111)
This is your flag:
```

2번째 hex는 `print_flag()`의 주소이다.