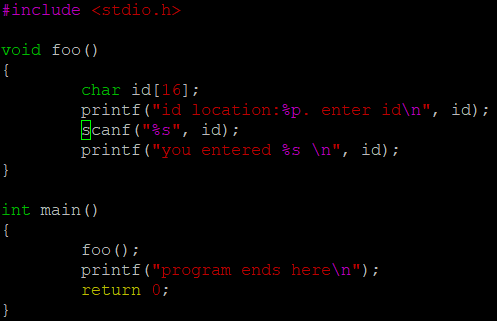
**HW Lect3 buffer overflow attack**

**정보보호론 002분반**

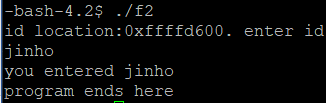
**정보통신공학과 12161719 김진호**

**Homework**

**1) Change f2.c as follows. What is the location of id[] inside gdb? What is the real location of id[] during the execution? How far is id[] from the stack location where the return address for "foo" is stored?**



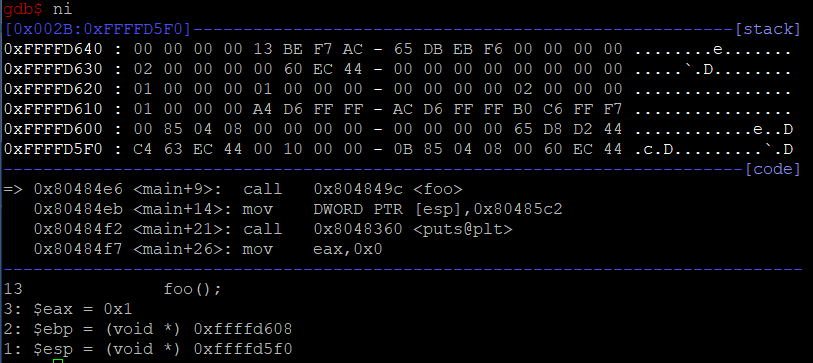
f2.c를 위처럼 수정하였고 gcc -m32 -g -o f2 f2.c를 입력하여 컴파일을 진행하였다.



실행 결과, id의 주소는 0xffffd600으로 확인된다.

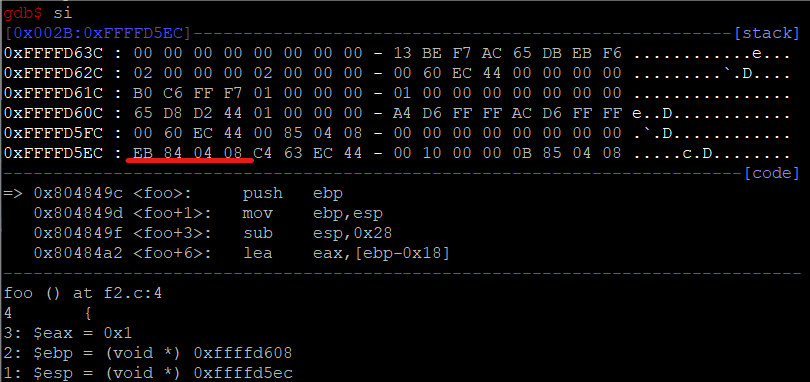
gdb를 통해서 id의 실제 주소(위치)를 확인하기 위해서 foo 함수의 호출 직전 시점부터 메모리 변화를 다음과 같이 추적하였다.

**-sub esp, 0x10-**



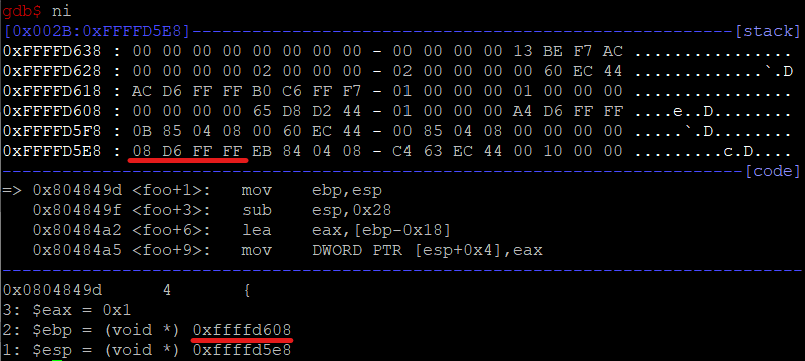
foo 함수의 직전 모습이다.

**-call 0x804849c <foo>-**



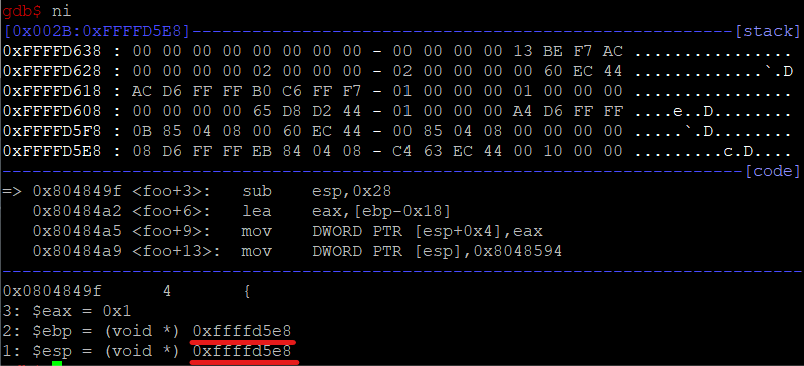
call 명령어를 통해서 return address인 0x080484eb가 스택에 저장되고 esp는 4만큼 감소한 것을 확인할 수 있다. 이 때 단순 ni가 아닌 si를 통해서 foo 함수로 진입하였다.

**-push ebp-**



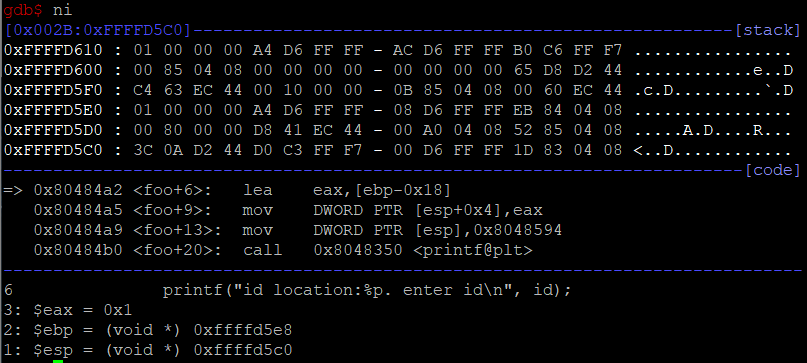
old ebp인 0xffffd608이 스택에 저장되었고 esp는 4 감소하였다.

**-mov ebp, esp-**



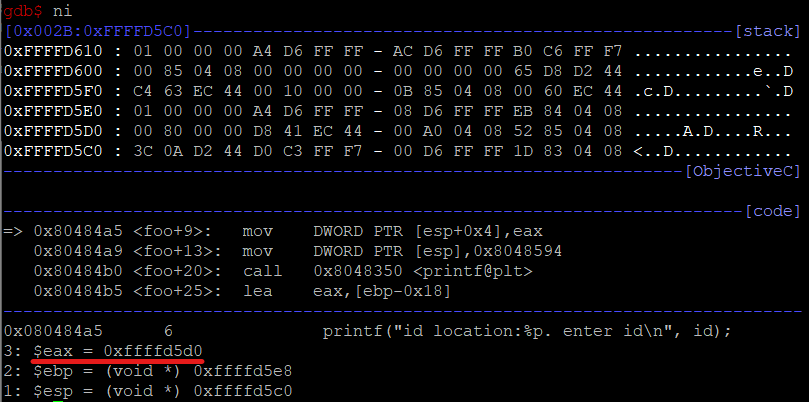
esp 값이 ebp 값으로 복사되어 두 레지스터가 같은 주소를 가리키고 있는 것을 확인할 수 있다.

**-sub esp, 0x28-**



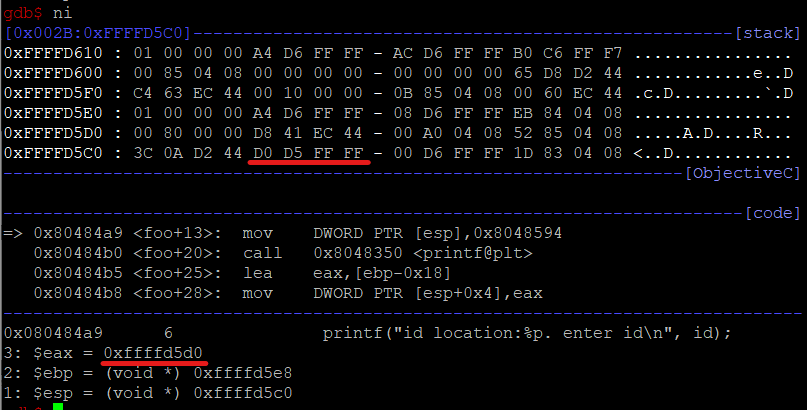
esp가 0x28만큼 감소하였다.

**-lea eax, [ebp-0x18]-**



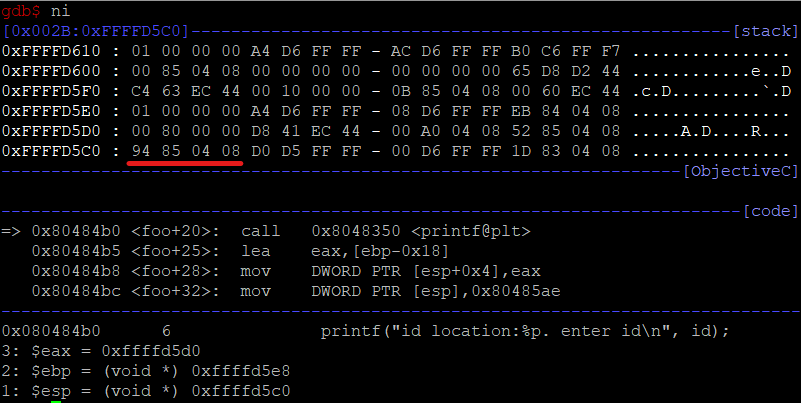
**mov eax, ebp-0x18**과 동일한 의미로 eax 값이 0xffffd5d0이 되었다.

**-mov DWORD PTR [esp+0x4], eax-**

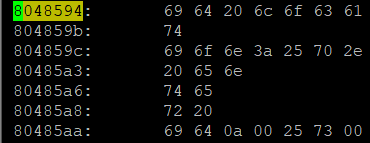


eax 값이 esp+0x4 주소에 저장되었다. 이 떄의 eax 값이 id의 주소값이다. 즉 id의 실제 위치는 0xffffd5d0이라고 할 수 있다.

**-mov DWORD PTR [esp], 0x8048594-**

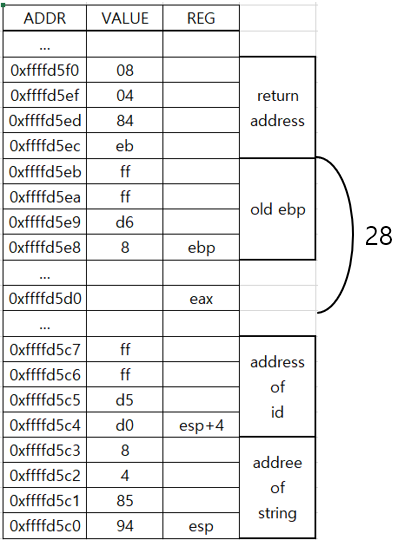


esp가 가리키고 있는 주소에 0x08048594를 저장한다. objdump -D -M intel f2 > f2.txt를 통해 실행파일 f2를 역어셈블한 후, 0x080484954에는 무엇이 있는지를 살펴보았다.



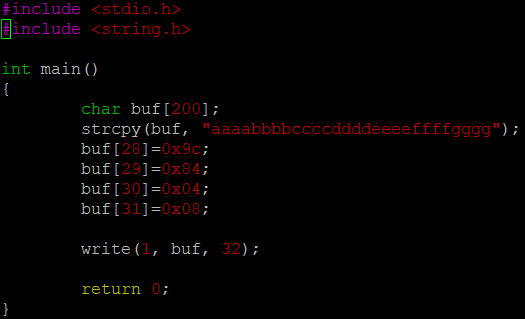
위 일련의 숫자들은 16진수 아스키 코드로, id location:%p. enter id\n라는 문자열을 의미한다. 즉 id location:%p. enter id\n라는 문자열이 존재하는 주소를 esp가 가리키는 주소에 저장한 것이다.

foo 함수 이후 복귀하게 될 return address 0x080484eb는 0xffffd5ec에 존재하고 id는 실제로 0xffffd5d0에 존재한다. 따라서 return address가 저장된 주소와, id가 저장된 주소는 0x1c 즉, 10진수로 28바이트만큼 떨어져 있는 것을 확인할 수 있다.

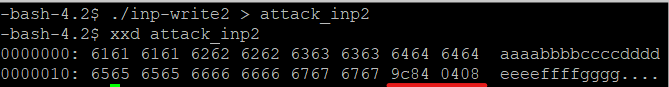


**2) Make an attack input file, attack-inp2, so that this program repeats "enter id" twice.**

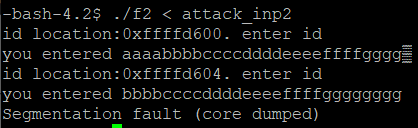
id가 실제로 위치하는 주소와 return address가 존재하는 주소가 28바이트만큼 떨어져 있기 때문에 id로, 29바이트부터 32바이트까지 foo 함수의 address를 갖는 문자열을 입력하면 foo 함수가 한 번 처리된 이후 한 번 더 foo로 돌아가기 때문에 enter id가 두 번 등장하게 된다.



위는 inp-write2.c 코드이다. 문자열 buf에 먼저 aaaa….gggg의 28바이트를 복사해준 상태에서 foo 함수의 주소 0x0804849c를 추가해주었다. 그리고 터미널 화면에 출력되도록 하는데 이 결과를 >를 이용해서 attack-inp2 라는 파일의 형태로 저장한다.



attack\_inp2 파일은 위와 같다. xxd로 확인한 결과, 마지막에 foo 함수의 주소가 제대로 들어가 있는 것을 확인할 수 있다.

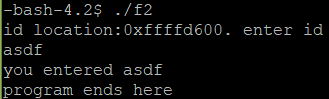


attack\_inp2의 내용을 f2의 입력 값(id)로 제공한 결과, 한 번 실행 이후 변경된 return address로 인해서 foo 함수가 한 번 더 호출된 것을 위처럼 확인할 수 있다.

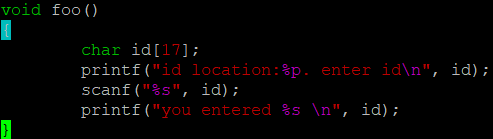
**3) Make an attack input file, attack-inp3, that contains attack code using inp-write3.c below. Attack f2 with this input file. f2 should execute the injected code and prints a. The return address in inp-write3.c should be changed appropriately. You also need "-z execstack" option when compiling f2.c to allow executable stack. The scanf() in f2.c stops reading bytes when it sees a whitespace (0x9, 0xa, 0xb, 0xc, 0xd, or 0x20) or end-of-string(0x00), so make sure your attack-inp does not contain any of these. If it contains any of them, you should change the size of id[] to avoid them.**

2번에서는 f2에서 id로 입력될 내용을 코드로 만들었다면, 3번에서는 foo 함수 내부의 scanf를 통해 f2의 스택 프레임으로 침투하여 그 내부에서 실행될 실행 파일을 작성하는 코드, inp-write3.c를 작성하였다.

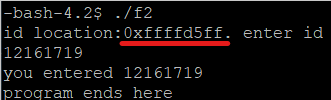
inp-wrtie3.c를 작성하기에 앞서, 몇 가지를 다음과 같이 변경하였다. 먼저, 현재 id의 위치를 다시 한 번 출력하면 다음과 같다.



id의 위치는 0xffffd600인데, 이 위치는 inp-write3.c에서 ret 주소로 들어가야할 위치이다. 이 때 문제점이 발생하는데, f2에서 호출되는 scanf 함수는 0x9, 0xa, 0xb, 0xc, 0xd, 0x20, 0x00를 만나면 스캔을 중단하게 된다. 따라서 id의 주소를, 위의 16진수를 포함하지 않도록 변경할 필요가 있다. 현재 f2.c에서 id가 16바이트의 길이를 갖도록 초기화되고 있는데 이 길이를 증가하면, id의 위치가 내려갈 것(주소 숫자가 커질 것)으로 예상할 수 있다.

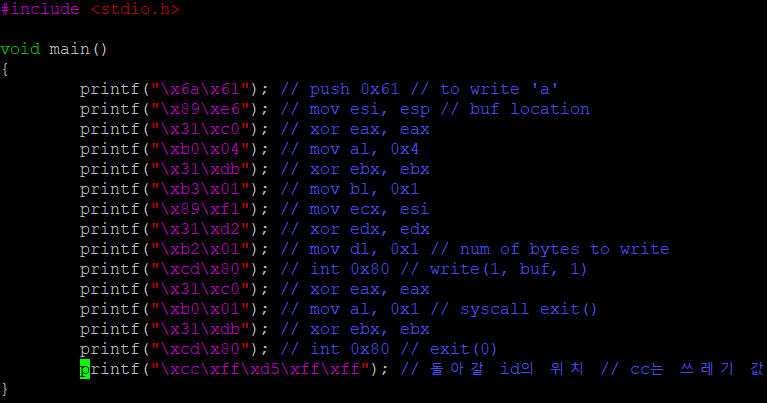


따라서 foo 함수에서 id를 위처럼 변경하였다. (id[16] -> id[17])



이후 재 컴파일하고 실행한 결과, 예상대로 id의 위치가 1바이트 감소한 것을 확인할 수 있다.

id의 위치를 적절히 변경하였으니 inp-write3.c 코드를 다음과 같이 작성하였다.

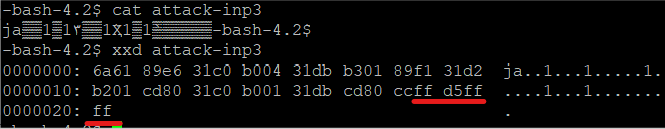


전체적인 코드는 강의 노트에서 제공된 내용이며, 마지막 ret address 4바이트를 id의 위치로 변경해주었다. 이 때 강의노트에서는 마지막 ret address 부분에 4바이트만을 넣고 있지만, 현재 id 배열의 길이를 1바이트 증가시켰으므로 한 개의 무의미한 쓰레기 값이 추가되어야 id 위치가 return address로 잘 들어가게 된다. 따라서 xcc라는 쓰레기 값을 추가하였다. 이는 id 길이가 16일 때, ret address와 id 위치 간의 간격 28바이트였지만, 현재 id 길이가 17이 되었으므로 그 간격이 29바이트로 증가하는데, 공격 코드의 길이가 28 바이트이므로 1 바이트를 쓰레기 값으로 채워 넣고 나머지 4바이트를 ret address에 우겨 넣기 위함이다.

이 코드를 컴파일하고 attack-inp3라는 파일로 리다이렉션(>)하고, attack-inp3를 f2로 리다이렉션하여 f2를 실행할 것이다. 위 코드의 주석을 보면 printf 내부의 문자열들이 기계어의 형태를 가지고 있다.

가장 첫 줄 \x6a\x61을 살펴보겠다. \x6a는 x6a를 한 바이트로 인식하여 그대로 출력하기 위한 형식이다. x6a x61 중 x6a는 push를 의미하는 opcode이므로 이는 x61 즉, 알파벳 a를 push하라는 의미를 갖는다.

이처럼 기계어 형태의 문자열들을 출력하는 것이 inp-write3.c인데 이것의 실행 결과를 리다이렉션한 attack-inp3는 일종의 실행 파일이 될 것이다.

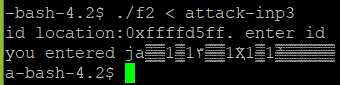


attack-inp3를 확인한 결과, cat의 경우 아스키 코드 값 범위를 벗어나는 바이트들을 온전히 표현할 수 없고, xxd로 확인하였더니 inp-write3.c에서 출력한 바이트 시퀀스가 정상적으로 담겨 있는 것을 확인할 수 있었다.

attack-inp3를 f2의 스택 내부에서 실행할 수 있도록 f2를 다음과 같이 컴파일하였다.



-z execstack 옵션을 추가하였다.



./f2 < attack-inp3를 통해서 f2의 내부에서 attack-inp3가 실행되도록 하였다. foo 함수 내부의 scanf가 attack-inp3를 id의 값으로 한 바이트 한 바이트 입력 받게 된다. 그리고 마지막에 return address를 id의 위치로 한다. 그리고 foo 함수의 ret 명령어 이후, esp는 ret address가 담겼던 지점 바로 위(주소 값을 나타내는 16진수가 큰 쪽을 위라고 표현하였다.)에 존재하게 되고 eip는 id에 저장되어 있는 명령어들(혹은 attack-inp3)를 실행하게 된다.

이번 문제의 해결을 통해, 준비된 파일을 이용하여 프로그램이 원래의 의도와 다르게 동작하도록 하는 buffer overflow attack을 할 수 있음을 알게 되었다.

**※** 5번 하단의 inp-write3.c에 대한 설명을 뒤늦게 확인하였다. inp-write3.c는 다음과 같은 코드의 어셈블리언어를 출력하도록 하는 프로그램이라고 한다.

void main()

{

char buf;

buf=’a’;

write(1, &buf, 1);

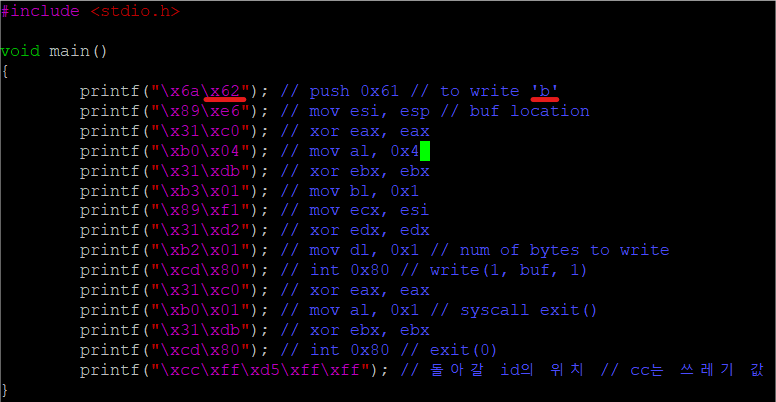
exit(0);

}

즉 inp-write3.c로 생성한 attack-inp3.c는 위 코드와 동일한 역할을 한다. 그래서 비정상적으로 a만을 출력한 채 프로그램이 종료되게 된다.

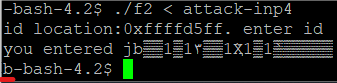
**4) Modify your code such that it displays 'b' instead of 'a'.**

inp-write3.c의 가장 첫 줄, printf(“\x6a\x61”)이 a를 스택 프레임으로 push하라는 내용을 담고 있는 만큼, x61을 x62로 변경하고 3번과 동일한 과정을 거치면 a 대신 b가 출력될 것으로 예상할 수 있다.



inp-write4.c를 위처럼 작성하였다.

그리고 이를 이용하여 만든 attack-inp4를 f2 내부에서 실행되도록 한 결과는 다음과 같다.

****

a 대신 b가 출력되는 것을 확인할 수 있다.

**5) Modify your code such that it displays 'ab'.**

기존의 a나 b, 하나의 문자 출력이 아니고 ab를 출력해야 한다. 따라서 스택에 두개의 문자가 push 되어야 하며 그 순서는 b가 먼저 push되고 a가 push되어야 한다. 따라서 코드는 다음과 같이 시작한다.

printf(“\x6a\x62”);

printf(“\x6a\x61”);

현재 putty로 사용 중인 서버는 32bit cpu 기반이므로 push가 4바이트 단위로 이루어 진다. 따라서 스택에서는 a null null null b null null null (오른쪽으로 갈수록 주소값이 크다.)의 형태 즉, 8 바이트를 차지하고 있다.

그리고 printf(“\x89\xe6”) 부터 printf(“\xcd\x80”)까지는 3번과 4번에서 사용된 공격 프로그램과 동일하다.

주의해야할 부분은 아이디의 길이인데 공격 코드의 사이즈, 스택 구조가 변경되었으므로 돌아갈 id의 주소를 신중히 정해야 한다. 문자 두개를 스택에 넣으면서 중간에 코드의 변화가 일어나는 부분이 있는데, 강의 노트에 수록된 inp-write3.c의 설명을 참조하여, 한 줄 한 줄 하기에 서술하겠다.

printf(“\x6a\x62”); // 문자열 b를 스택에 push 한다.

printf(“\x6a\x61”); // 문자열 a를 스택에 push 한다.

printf(“\x89\e6”); // mov esi, esp. buf location. // 현재 esp는 스택에서 a null null null b null null null이 위치한 주소 값을 저장하고 있다. 따라서 이제는 esi가 문자열 a\0\0\0b\0\0\0을 가리키고 있다고 볼 수 있다.

printf(“\31\c0”); // xor eax, eax // eax는 자기 자신과의 xor 연산 결과, 0으로 초기화된다.

printf(“\xb0\x04”); // mov al, 0x4 // write 함수의 호출에 앞서, write는 syscall 4번에 해당하므로 al 레지스터에 4를 저장한다. al 레지스터는 eax 레지스터의 하위 8 bit를 의미한다.

printf(“\x31\xdb”); // xor ebx, ebx // ebx는 자기 자신과의 xor 연산 결과, 0으로 초기화된다.

printf(“\b3\x01”); // mov 0x1, bl // write를 어디에 할 것인지 즉, 터미널의 파일 디스크립터 1을 bl이라는 레지스터에 저장한다. bl 레지스터는 주로 주소를 지정하는 ebx 레지스터의 하위 8 bit를 의미한다.

printf(“\x89\xf1”); // mov ecx, esi // 현재 esi에는 a\0\0\0b\0\0\0이 위치한 주소가 저장되어 있는데 이를 ecx에도 저장한다.

printf(“\x31\xd2”); // xor edx, edx // edx는 자기 자신과의 xor 연산으로 인해 0으로 초기화된다. edx도 eax처럼 주로 연산에 사용되는 레지스터이다.

**printf(“\b2\x08”);** **// mov dl, 0x8 // dl 레지스터는 edx의 하위 8 bit에 해당되는 레지스터이며 여기에 0x08을 저장한다. 이 명령어는 esi가 가리키는 주소에 존재하는 문자열을 몇 바이트나 읽어올 것인지를 의미한다. a\0\0\0b\0\0\0이므로 8바이트를 읽어온다고 지정해주었다.**

printf(“\xcd\x80”); // INT 0x80 // 터미널에 write(System function)할 수 있도록 system interrupt가 필요하다.

printf(“\x31\xc0”); // xor eax, eax // eax 초기화

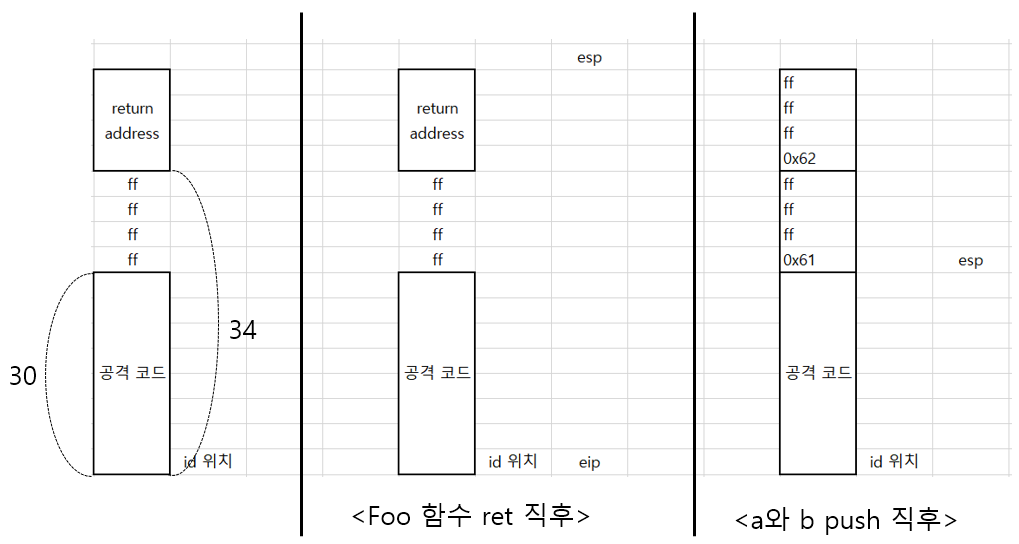
printf(“\xb0\x01”); // mov al, 0x01 // exit 함수의 syscall number 1을 al 레지스터에 저장한다.

printf(“\x31\xdb”); // xor ebx, ebx // ebx 초기화

printf(“\xcd\x80”); // INT 80 // system interrupt를 통한 exit 함수 처리

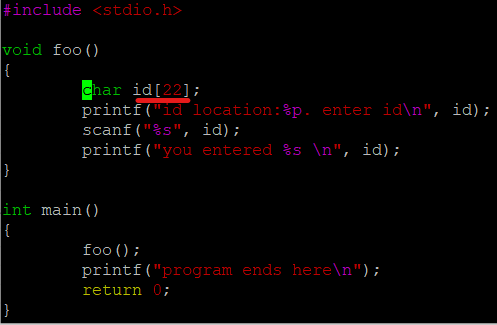
**printf(“\xff\xff\xff\xff”); // 이 라인과 다음 라인의 이유는 아래의 그림과 함께 설명하겠다.**

**printf(“\xff\xff\xd5\xfa”); //**

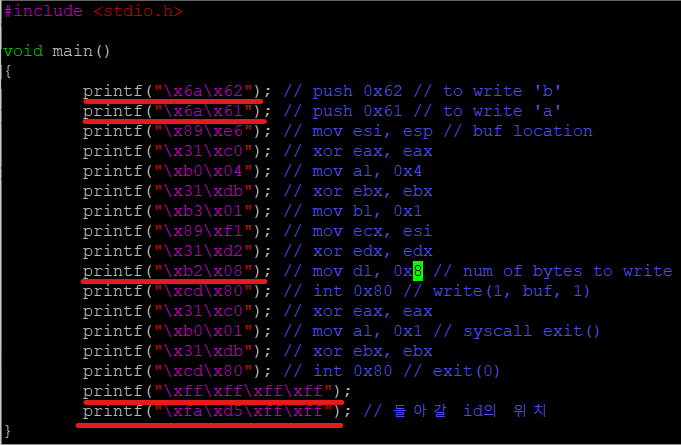


먼저 push a와 push b가 이루어져야 하므로 return address의 변경을 제외한 ‘공격 코드’는 30 바이트가 된다. (3번, 4번에서 공격 코드는 28 바이트였다.) foo 함수의 ret 이후에 공격 코드가 실행되기 시작하는 것인데 foo 함수의 ret 직후의 모습은 위 그림과 같다. 기존에 ret address가 들어있는 주소의 바로 위에 esp가 존재하게 되고 push a와 push b가 차례로 일어나는 것을 고려하면 return address와 공격 코드는 4 바이트의 여유 공간이 필요함을 알 수 있다. 따라서 공격 코드 이후 쓰레기 값 ff를 4 바이트 추가하고 id의 위치 0xffff5dfa를 추가한 것이 attack-inp5.c이다.

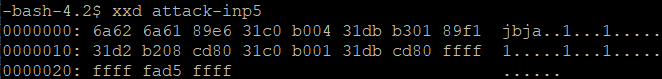
이 때 id의 위치는 return address와 34 바이트 이상 떨어져 있어야 하는데 위의 그림 대로 하기 위해서 34 바이트 떨어진 0xffff5dfa로 설정하였다. 자연스레 id 배열의 크기도 23 바이트가 된다.



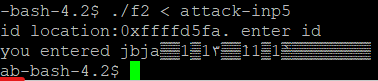
따라서 f2.c는 위처럼 변경된다.



inp-write5.c는 위와 같다. 유의미하거나 inp-write3.c inp-write4.c와 비교해서 변경이 있는 부분은 빨간 줄로 표시하였다.



attack-inp5의 xxd 출력 결과이다.



attack-inp5를 이용해서 f2에 공격을 시도하자, 의도한대로 ab가 출력되는 것을 확인할 수 있다.