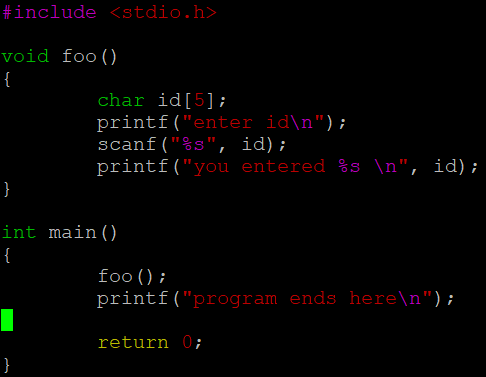
**HW Lect2 flooding-return-address**

**정보보호론 002분반**

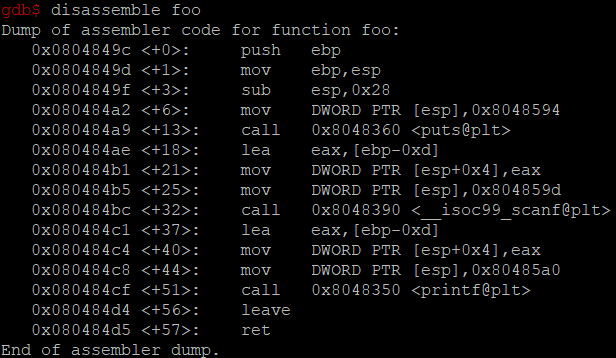
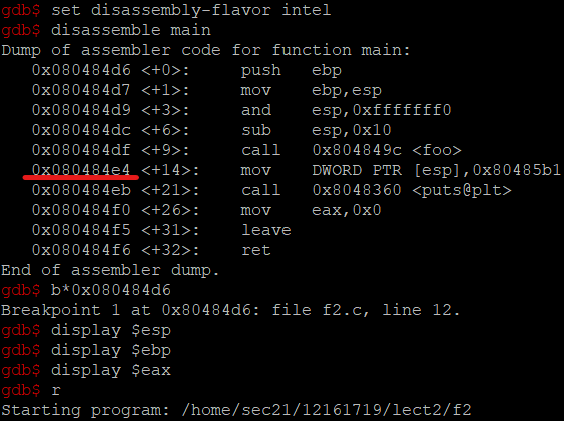
**정보통신공학과 12161719 김진호**

**1) When “foo” is called, the system pushes the return address for “foo” in the stack and jumps to “foo”. Use gdb to find out this return address and the stack location where this address is stored. Examine the contents of this stack location to confirm that the stack indeed has the return address.**

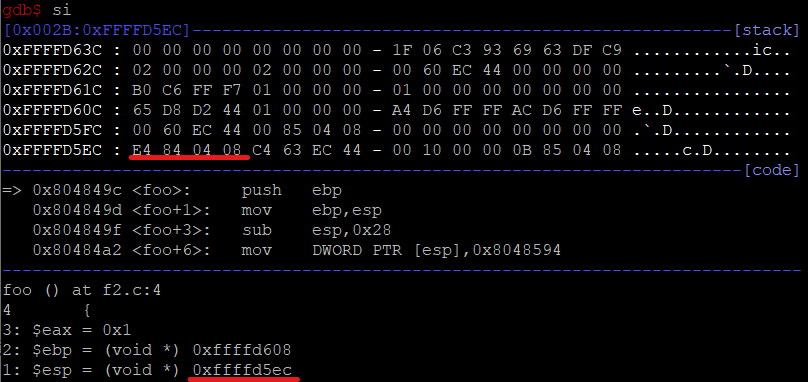
****

먼저, f2.c를 위와 같이 작성하였고 gcc -m32 -g -o f2 f2.c를 입력하여 gdb를 사용할 수 있도록 컴파일하였다.

gdb f2를 입력하여 f2를 디버깅모드로 실행하였다. set disassembly-flavor intel을 입력 후 main 함수를 역어셈블하여 intel format의 어셈블리어를 확인하였다. 그리고 main 함수의 시작 주소인 0x080484d6을 break point로 설정한 후, esp, ebp, eax 레지스터들을 항상 display하도록 하였고 이어서 r을 입력하여 실행하였다.

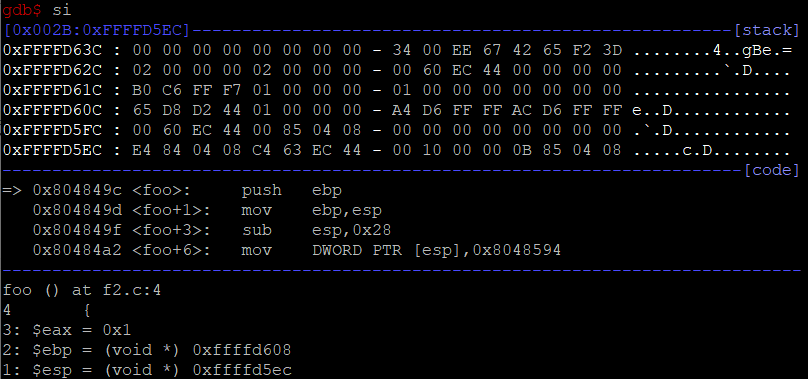


위 사진을 통해서 foo 함수의 call 이후에 돌아와야할 return address는 0x080484e4임을 예상할 수 있는데, 실제로 인스트럭션을 하나씩 실행시키며 확인한 결과는 다음과 같다.



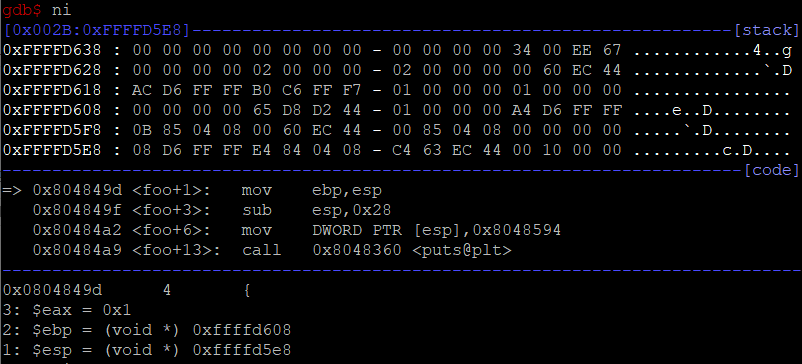
si를 통해서 call 명령어의 세부 과정에 진입하였고, esp 주소에 예상하였던 0x080484e4가 저장된 것을 확인할 수 있었다.

**2) Find out the starting address of “id[]”. Explain how you found it. How far is id[] from the stack location where the return address is stored? Draw a memory map that shows code area and stack area. The code area should show where is main function and where is foo function. The stack area should show where is the return address for "foo" and the location of id[].**



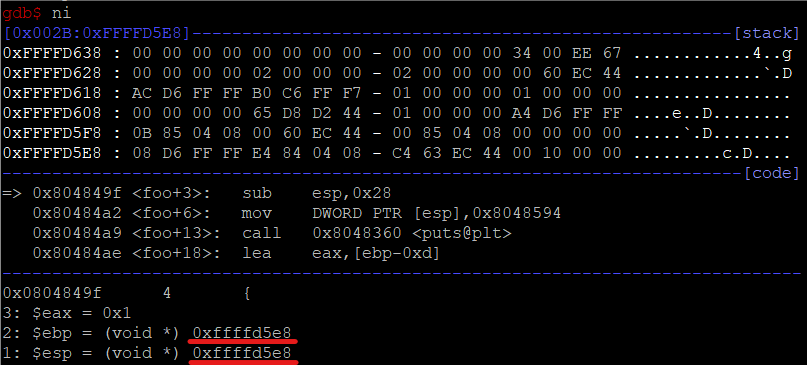
문자열 id의 주소를 찾기 위해서 call 이후 foo 함수로 진입하고 차례로 디버깅한 일련의 과정을 소개한다. 위 사진은 call 함수 직후 foo 함수에 진입한 모습이다. 1번에서 살펴본 것처럼 이 때 esp가 가리키는 주소는 return address를 저장하고 있다.

**-push ebp-**



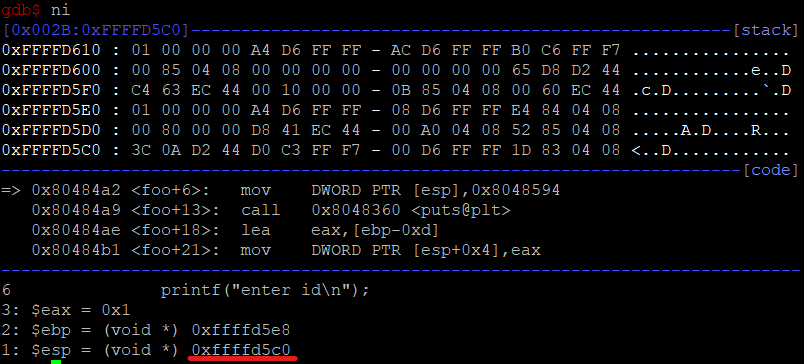
기존의 ebp 값(old ebp)를 스택에 저장하는 과정이다. push 명령어로 인해 이전에 비해서 esp가 4만큼 감소한 것을 확인할 수 있다.

**-move ebp, esp-**



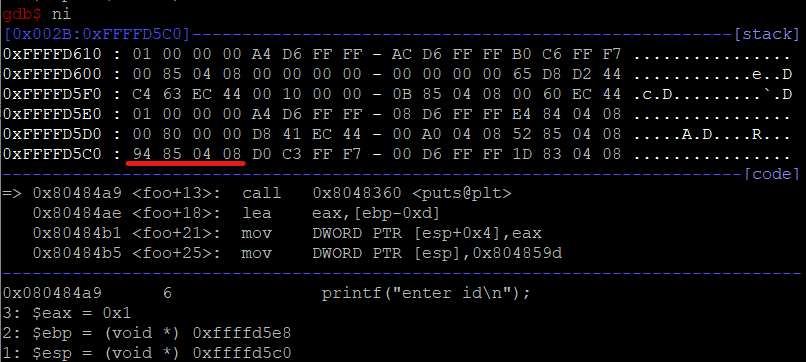
현재 esp의 값을 esp로 복사한다. 즉, esp가 가리키는 주소를 ebp 또한 가리키고 있다.

**-sub esp, 0x28-**

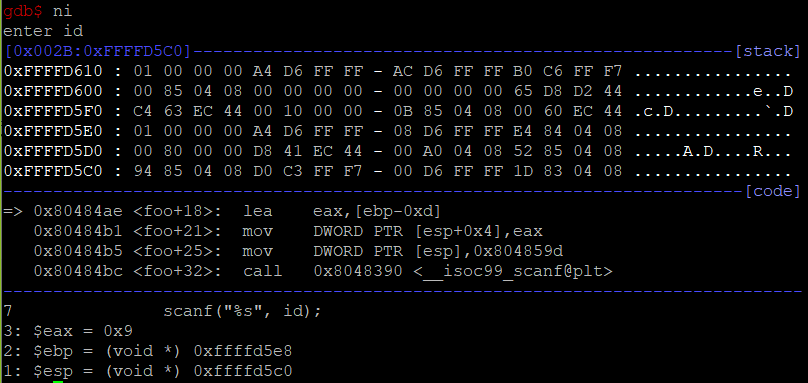


esp 값에서 0x28만큼 빼주었기 때문에 esp가 감소한 것을 확인할 수 있다. 이제서야 비로소 foo 함수의 스택 프레임이 완성됐다고 할 수 있다.

**-mov DWORD PTR [esp], 0x8048594- -call 0x8048360<puts@plt>-**

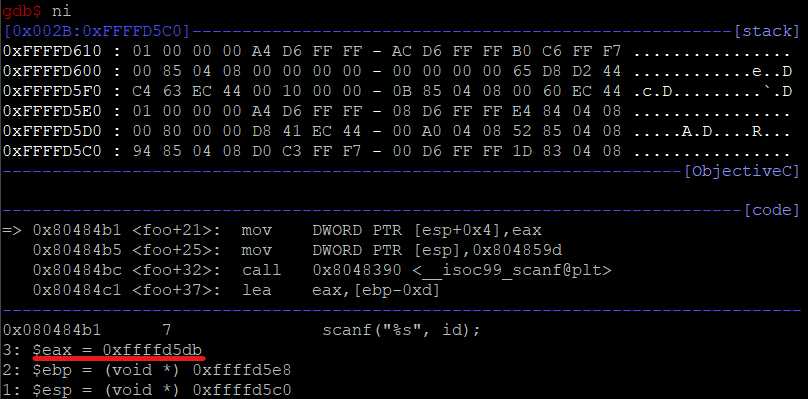
****

esp가 가리키는 주소에 0x08048594라는 값이 저장된 것을 확인할 수 있는데 0x08048594는 printf(“enter id\n”); 코드 중 enter id \n이라는 문자열이 저장된 주소이다. 그리고 이어지는 명령어 call 0x8048360<puts@plt>는 printf 함수의 호출을 의미하는데 시스템 함수인 printf 문으로 진입하지는 않겠다.

****

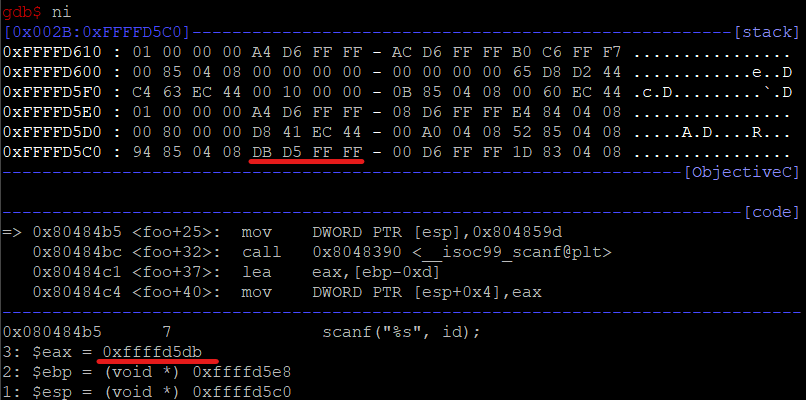
call 명령어를 ni로 실행한 결과 enter id가 출력된 것을 확인할 수 있다.

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

****

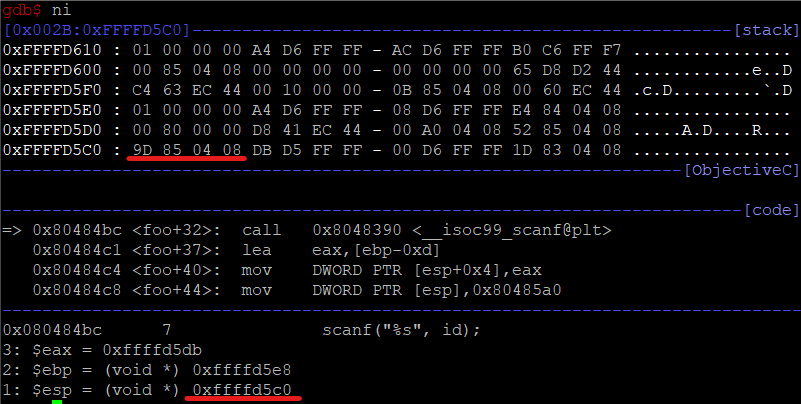
scanf(“%s”, id)가 실행되기 시작하는 가장 첫 번째 부분이라고 할 수 있다. ebp-0xd 값이 eax에 저장된다. 따라서 eax는 0xffffd5c0-0xd=0xffffd5db가 된다. 현재 ebp에서 0xd 즉, 13바이트만큼 떨어진 곳을 eax가 가리키고 있게 된다.

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

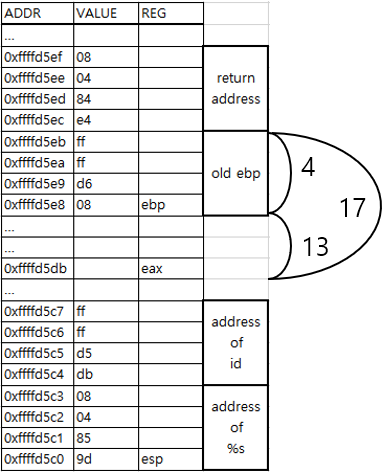


eax에 담긴 값이 esp+0x4라는 주소의 값으로 저장된다. 이 때의 eax 값은 배열 id의 주소값이다.

**-mov DWORD PTR [esp], 0x804859d-**

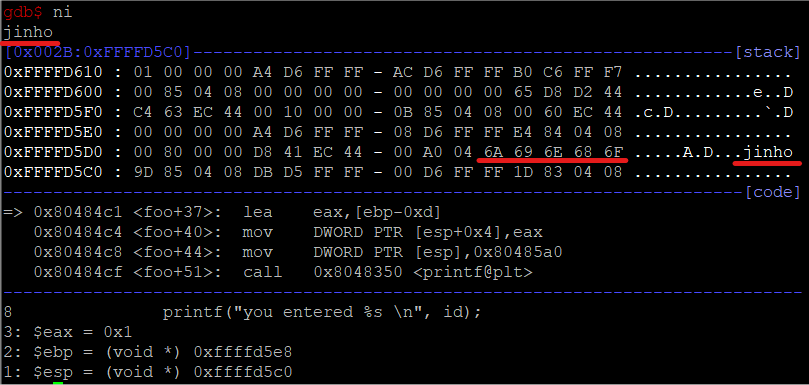
****

현재 esp가 가리키고 있는 주소에 0x804859d라는 값을 저장한 것을 확인할 수 있다. 0x804859d는 %s라는 문자열이 저장된 주소로 예상된다.



현재 foo 함수 호출과 scanf 함수 call 직전까지의 레지스터 모습을 위와 같이 나타냈다. scanf의 결과로 입력되는 id 값은 eax로 위로(높은 주소로) 저장될 것인데 return address와의 거리가 17바이트이므로 이것을 넘는 바이트가 입력될 때 return address 자체가 손상되는 치명적인 문제가 발생하게 될 것을 예상할 수 있다.

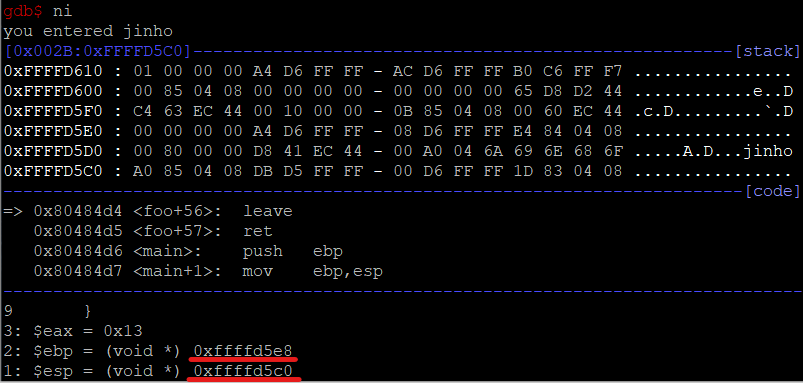
**3) After you enter an ID at “enter id” prompt, examine the memory to confirm that the entered ID is stored correctly. At which address are they stored? Is that address same as the location of id[] you found in Problem 2)?**

****

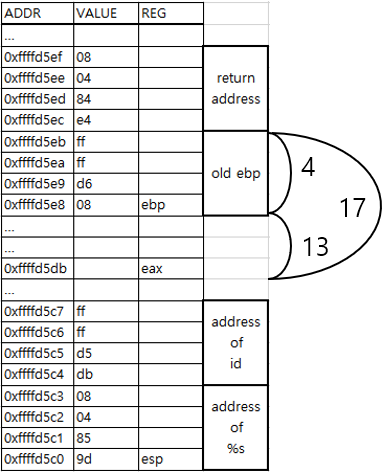
enter id라는 문자열이 출력된 이후 실제로 id를 입력하려면 foo 함수를 si 명령어를 이용해서 진입해야 한다. 그리고 ni를 여러 번 입력하면 scanf 이후 id를 입력해야 하는 시점이 생기는데 위와 같다.

문제 2번에서 id의 주소는 0xffffd5db에 저장될 것이라고 예상한 바가 있고, 실제로 위 사진을 통해서 입력한 jinho라는 id가 16진수 아스키 코드 값 6a 69 6e 68 6f로 0xffffd5db에 잘 저장되어 있는 것을 확인할 수 있었다.

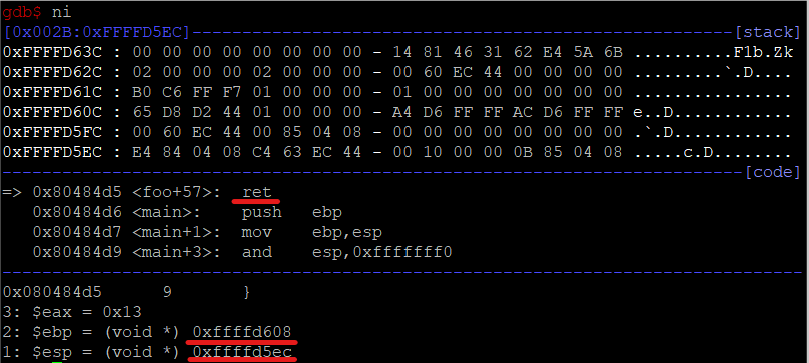
**4) Step through the program until you are at “ret” instruction (right before running "ret"). What is the value of esp at this point? Instruction "ret" will make the cpu return to the location written in the stack where esp is pointing to. When the system executes "ret", where the cpu should return? Do “ni” and confirm that the program correctly returns to the return address you have predicted.**

****

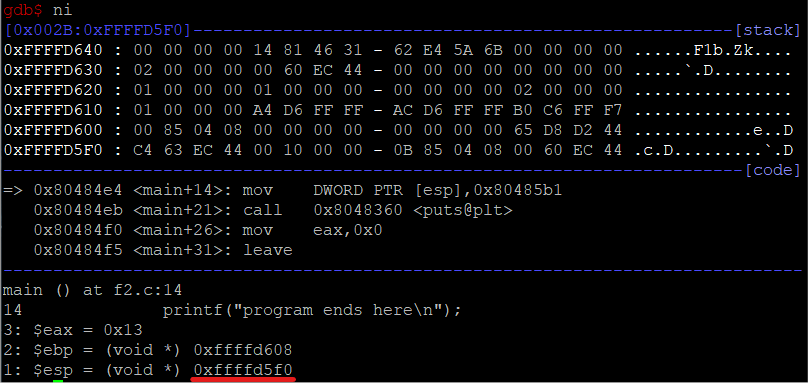
개인적으로, 이해를 위해서 leave 명령어 실행 직전의 순간부터 살펴보도록 하겠다.



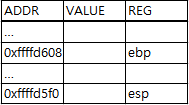
문제 2에 첨부하였던 메모리 그림을 다시 가져왔다. 현재 esp는 0xffffd5c0, ebp는 0xffffd5e8에 위치하고 있다. 위 그림과 잘 매칭되고 있는 상태이다.

****

그리고 다음은 leave 명령어가 실행된 후, 그리고 ret 명령어가 실행되기 직전의 모습이다. 현재 esp가 0xffffd5ec에 있는 것을 확인할 수 있는데 위 메모리 그림에 따르면, return address가 저장되어 있는 주소를 esp가 가리키고 있는 것이다. ebp의 경우 메모리 그림의 old ebp로 잘 돌아간 것을 확인할 수 있다. 즉, 현재 ebp는 foo 함수 호출 전으로 돌아갔고 return address를 스택에서 추출하기만 하면 된다.

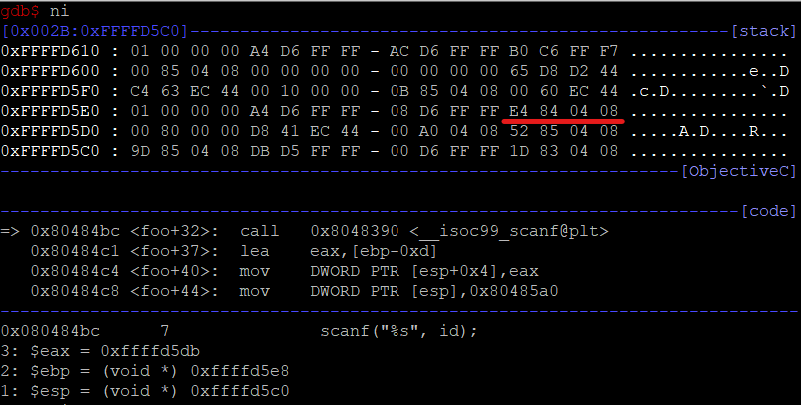


ret을 실행한 결과, 스택에서 return address가 pop되며 esp의 주소가 4만큼 증가한 것을 확인할 수 있다.

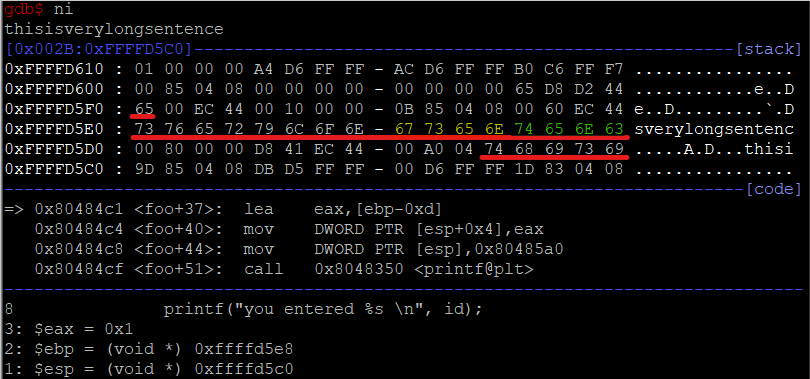


위와 같은 상태로 스택프레임이 정리된 것을 확인할 수 있다.

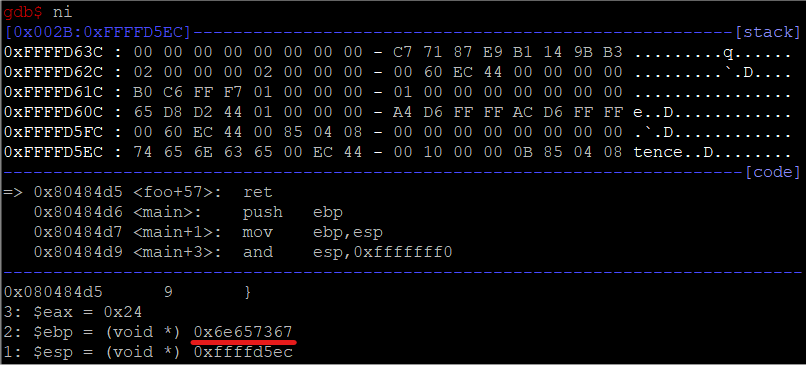
**5) Rerun the debugger with "r" command, and this time enter a long ID such that it changes the return address. Dump the memory starting from “id[]” up to the return address location to see the changed return address. When the gdb executes “ret” instruction, where does it return?**

****

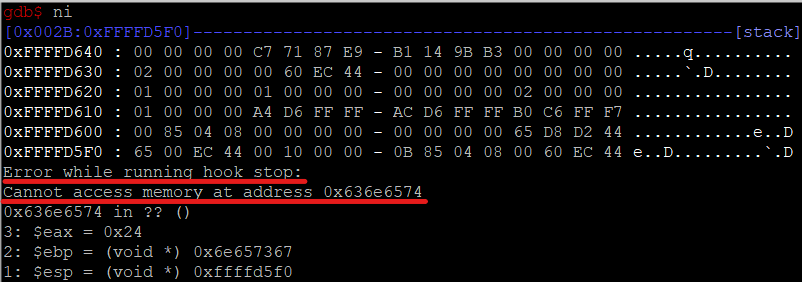
위 캡처는 매우 긴 id를 입력하기 전의 순간으로, 0xffffd5ec에 return address 0x080484e4가 정상적으로 저장되어 있는 것을 확인할 수 있다.



그리고 id로, thisisverylongsentence라는 17바이트가 넘는 문자열을 입력해준 이후의 결과이다. old ebp는 0x6e657367, return address는 0x636e6574으로 변경되어 버렸다.

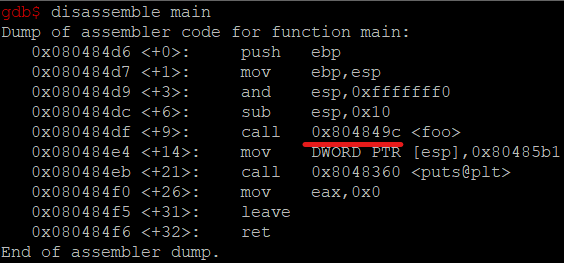


그리고 ni를 연이어 입력하여 leave 명령어 직후, ret 명령어 직전의 모습이다. old ebp값으로 돌아갔어야할 ebp 값이, 위에서 변경되어 버린 old ebp 값, 0x6e657367이 되어 버린 것을 확인할 수 있다.

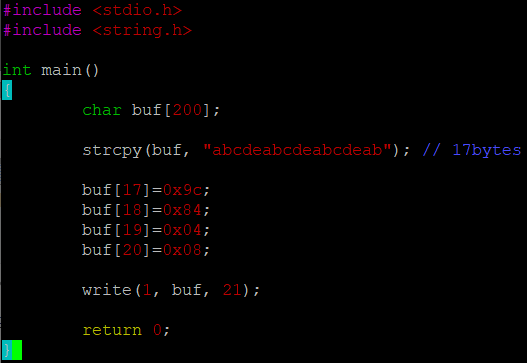
그리고 ret 명령어를 실행하니 위처럼 error가 발생하였다. 0x636e6574라는 메모리가 오류의 원인으로 해석되는데, 이 메모리 주소는 17바이트 이상의 id 입력으로 인해 변경된 return address가 존재하는 주소이다. 즉, return address의 변경은 치명적인 오류를 야기하는 것으로 보인다.

**6) Can you give an input such that the program asks "enter id" more than once? Remember you are not changing the program. You only give some strange input that would confuse the program. You will need to redirect the standard input in order to provide numeric input as follows.**

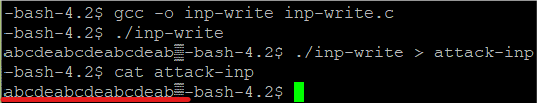
enter id라는 문자열이 터미널에 두 번 이상 반복적으로 출력되게 하려면, foo 함수 호출 이후 return address를, 다시 foo 함수의 주소로 변경하면 된다. 이는 foo 함수의 주소를 id의 18byte부터 4byte만큼 입력하면 된다.

****

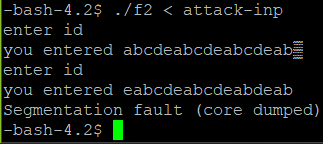
그러나 위 main 함수의 disassemble 결과를 보면, foo 함수의 주소는 0x0804849c인데, 이름 18~21byte에 입력해줄 수가 없다. 우리의 키보드 입력의 아스키 값은 0x00~0x7f가 범위를 가지고 있기 때문이다. 따라서 inp-write.c라는 코드와 리다이렉션을 이용해서 f2 실행파일 시점에 0x0804849c를 입력으로 줄 수 있도록 할 것이다.

****

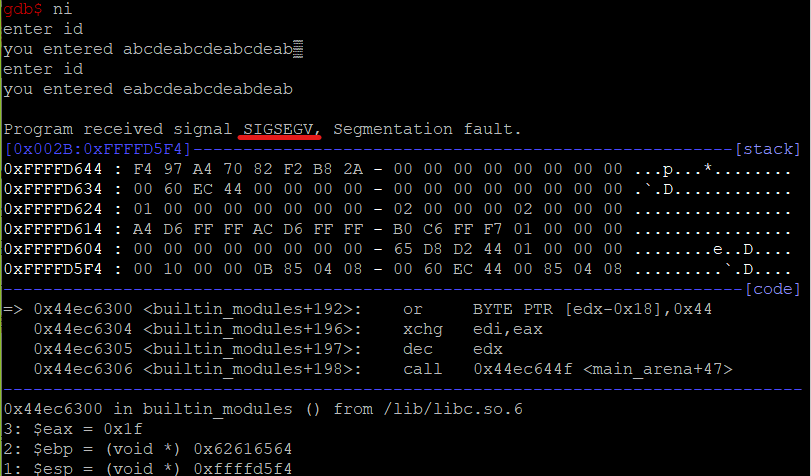
먼저 inp-write.c를 위와 같이 작성하였다. buf라는 문자열에 abcdeabcdeab라는 17바이트 문자열을 복사해주고, 18byte 부터는 foo 함수의 주소(0x0804849c)를 넣어주었다. 그리고 마지막에 write(1, buf, 21)이라는 코드를 통해 file descriptor 1번 즉, 터미널 화면에 buf가 21bytes 만큼 출력되도록 하였다. 이 출력 결과를 attack-inp라는 별도의 파일에 저장하고, < 를 통해 f2의 입력 인자(id)가 될 것이다.



실제로 attack-inp를 만들고 확인한 결과이다. 마지막에 이상한 문자가 뜨는 것은 내가 buf 문자열에 넣어준 foo 함수의 address가 아스키 코드 값 범위 밖의 숫자라서 터미널에 출력되지 못한 것이다.



실제 실행결과이다. attack-inp의 내용을 읽어서 f2의 id 인자로 사용한 결과이다. 처음에 의도한 id가 잘 입력되어 foo 함수가 한번 더 호출된 거을 확인할 수 있다. 두번째 id 입력은 약간의 변형이 일어났고, 이로 인해서 return address가 foo 함수의 address로 제대로 변경되지는 못하였지만, return address의 비 정상적인 변경으로 프로그램이 종료되는 것이 보인다.



이를 gdb에서 run < inp-attack이라는 명령어를 이용해서 확인하자, 프로그램 종료 직전에 SIGSEGV라는 signal을 확인할 수 있었다.

SIGSEGV는 비정상 종료 혹은 메모리 엑세스 오류 등으로 인해 OS의 프로그램 강제 종료 시 발생하는 signal이라고 한다.