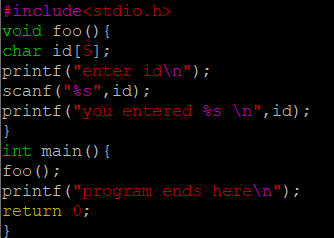
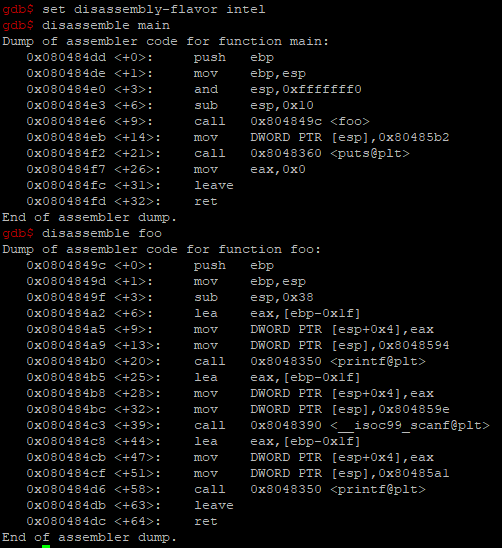
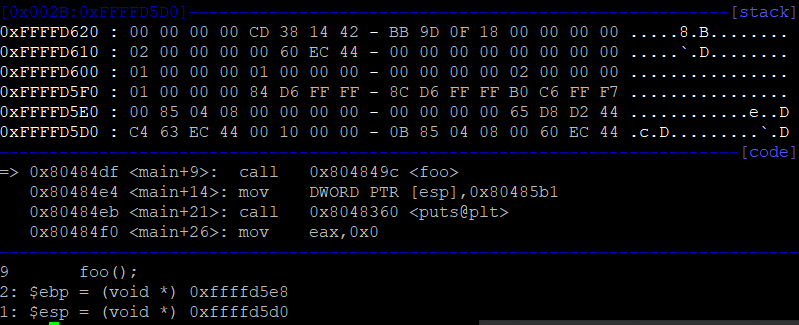
1. Buffer overflow example: f2.c

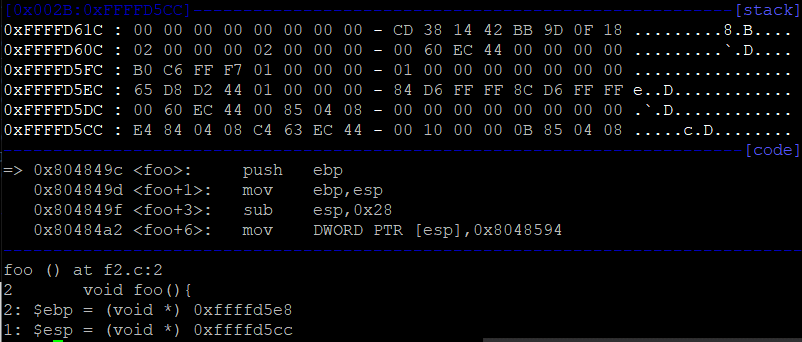






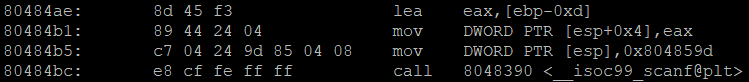
(과제 제출 이후 다시 해보고 추가한거라 조금 다름)

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. Dump the contents of this stack location to confirm that the stack indeed has the return address.

call 0x804849c <foo> 바로 밑에 있는 주소인 0x80484e4가 바로 foo의 리턴주소이다.

si를 사용해서 call foo instruction으로 들어간 다음 분석해보면, call foo를 한 다음에는 esp의 메모리에 리턴주소가 들어가기 때문에 위 그림에서 esp 위치의 스택에는 리턴주소가 들어있다. 실제로 esp인 0xffffd5cc의 스택을 확인해보면, 리턴주소인 80484e4가 그대로 들어있는걸 확인할 수 있다.

2) Find out the starting address of “id[]”. Explain how you found it. How far is id[] from the return address location? 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[].

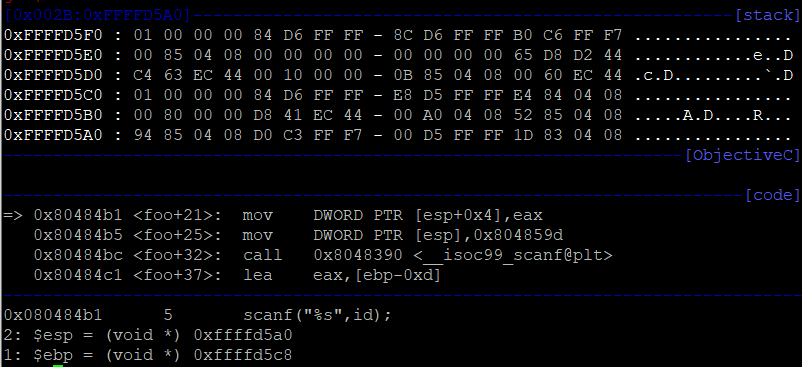


scanf 함수에서는 인자가 두개가 들어가는데, 그럴 땐 인자가 뒤에있는 것부터 차례대로 메모리에 저장된다. mov 명령어를 보면 esp+0x4의 위치에는 eax가 들어가고, esp에는 주소 0x804859d가 들어간다. 즉, 이것들이 scanf의 인자라면 주소 0x804859d에 있는 것이 첫번째 인자가 되고 eax에 있는 것이 두번째 인자가 된다.



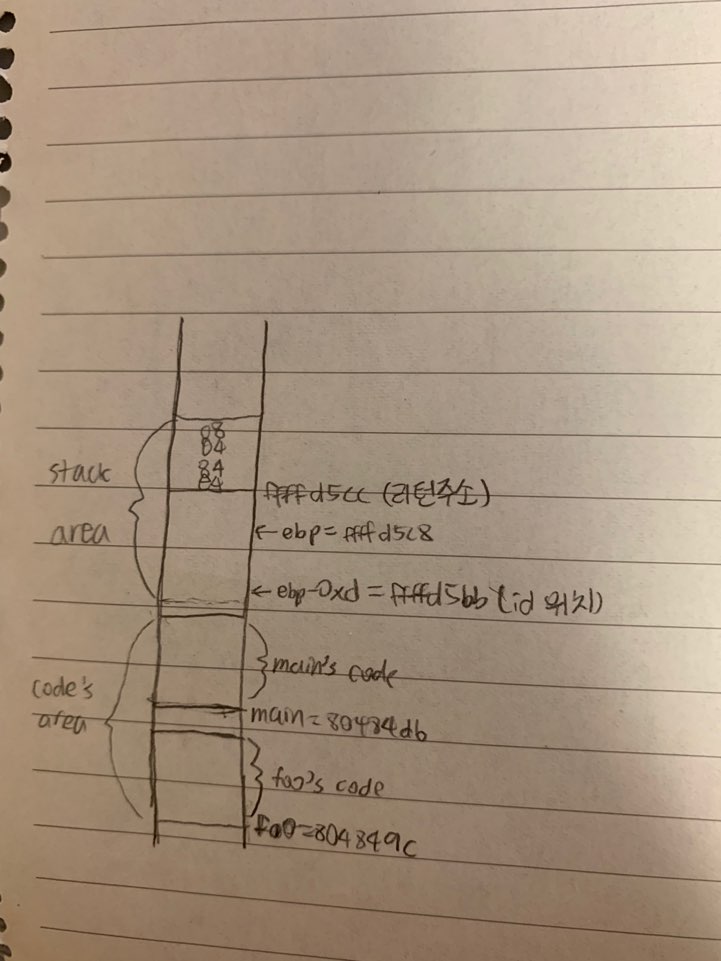
위 그림에서 네번째 숫자인 25가 804859d의 위치에 들어있는 것인데, 25 73를 아스키코드로 확인해보면 그것이 %s, 즉 scanf의 첫번째 인자라는 것을 알 수 있다. 위 그림의 두번째줄 명령에서 eax(두번째 인자)가 id의 주소를 포함한다고 할 수 있고, 첫번째줄 명령을 봤을 때 eax에 ebp-0xd가 들어가기 때문에 id의 주소는 ebp-0xd가 된다는걸 알 수 있다.

(한마디로 걍 0x804859d = %s, eax = id란 뜻임. 그 eax=ebp-0xd이고.)



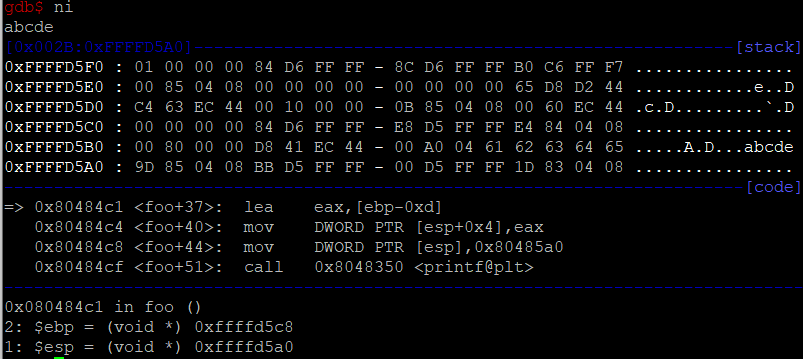
lea eax,[ebp-0xd]를 실행한 후, ebp는 0xffffd5c8이기 때문에 id의 위치는 ebp-0xd인 0xffffd5bb가 된다.

(원래 lea eax,[ebp-0xd]까지 ni를 했어야함..)



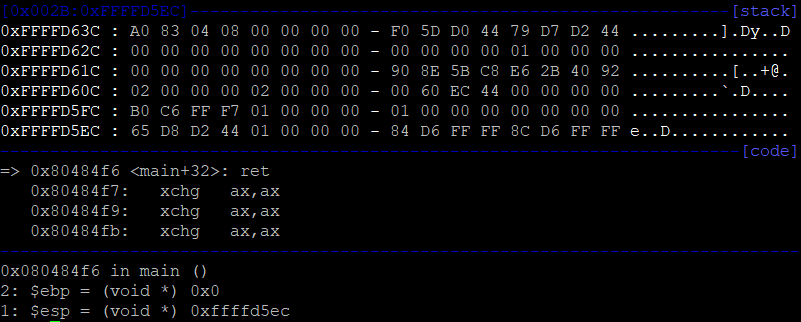
리턴주소의 위치와 id의 위치는 11만큼 차이난다. (decimal로는 17)

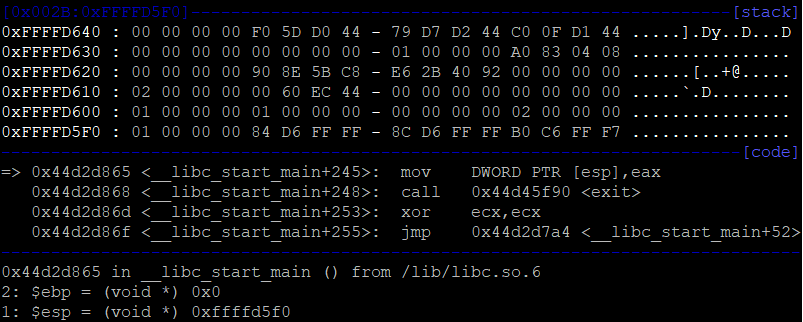
3) After you enter an ID at “enter id” prompt, dump the memory to confirm that the entered ID is stored correctly.



아이디로 abcde를 넣었을 때, ffffd5bb에서의 스택을 확인해보면 abcde의 아스키코드인 61 62 63 64 65가 들어가있는걸 확인할 수 있다.

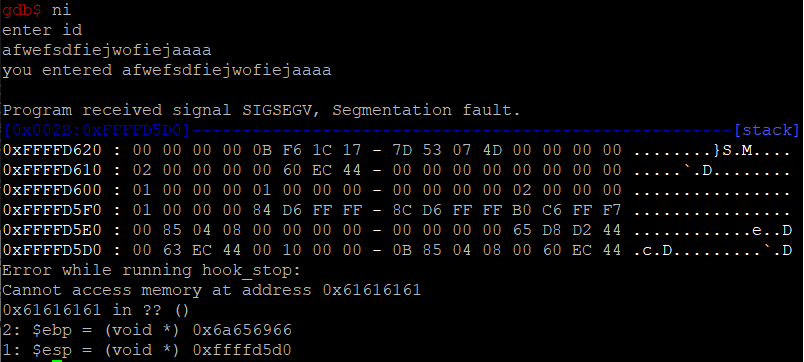
4) Step through the program until you are at “ret” instruction. Do “ni” and confirm that the program returns to the return address in main() correctly.



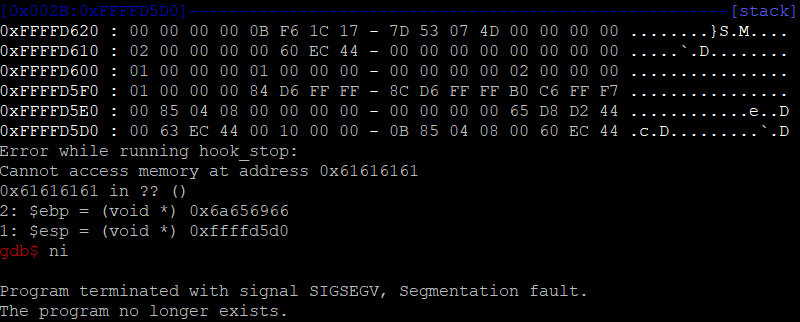


gdb를 실행해서 ni를 계속 누르고 ret에 도달했을 때 ni를 다시 한번 눌러봤는데, 프로그램 동작에 아무런 문제가 없었기 때문에 프로그램이 main의 리턴 주소로 잘 돌아간걸 확인할 수 있었다.

5) Rerun the debugger, 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?



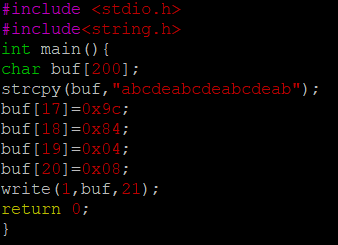
id 크기의 최대인 13비트(0xd)와 old ebp값의 공간인 4비트를 더해서 17비트는 랜덤으로 아무거나 입력하고 원래 리턴 주소가 있어야할 다음 4비트에는 aaaa를 집어넣은 결과, 리턴주소는 aaaa인 0x61616161을 가리키게 됐다.



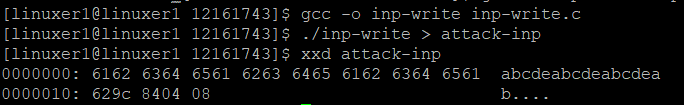
0x61616161은 허용되지 않은 segment이기 때문에 여기에서 ni를 더 쓰면 segmentation fault가 일어난다.

6) Can you give an input such that the program asks "enter id" more than once? Remember you are not changing the program.

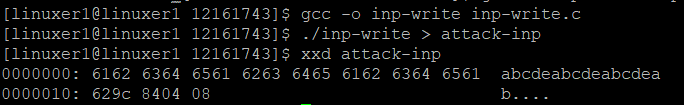
enter id를 한번 이상 출력하게 하려면 리턴주소 바로 전까지 아무거나 입력한 다음 리턴주소가 입력되는 부분에는 foo함수의 시작주소를 넣어야하지만, foo함수의 시작주소인 804849c는 아스키코드상 존재하지 않기 때문에 직접 키보드로 입력하는 방식으로는 enter id를 한번 이상 출력하게 할 수 없다. 따라서 이 문제를 풀려면 파일을 이용해서 원하는 input을 넣어준 다음 f2가 그걸 읽게 해야한다.



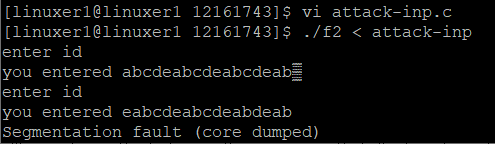
리턴 주소까지 닿을려면 기존 id배열의 크기인 13바이트와 old ebp 4바이트를 합쳐서 총 17바이트를 입력해야한다. 그렇기 때문에 strcpy로 buf에 먼저 17바이트를 랜덤으로 넣어주고 원래 리턴주소가 들어가야할 마지막 4바이트에 foo 함수의 시작주소를 넣어주면 foo가 재실행되기 때문에 enter id를 한번 더 출력할 수 있게 된다.



./inp-write > attack-inp를 하게되면 inp-write의 출력값을 그대로 attack-inp 파일에 넘기기 때문에 buf에 있는 21바이트가 그대로 attack-inp에 들어가게 된다.



xxd를 사용해서 attack-inp의 hexdump를 봐보면 strcpy로 buf에 넣었던 17바이트와 끝에 넣은 foo의 시작주소 4바이트까지 그대로 있는걸 볼 수 있다.



f2를 실행시킬 때, <를 사용해서 attack-inp 파일에 있는 input을 f2가 읽게 했더니 enter id를 두번 출력하게 돼었다.