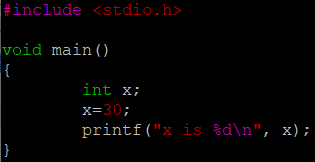
**HW Lect1 Debugging**

**정보보호론 002분반**

**정보통신공학과 12161719 김진호**

**1) Edit, compile, and run ex1.c**

****

vi 편집기를 통해 ex1.c를 위처럼 작성하였다.

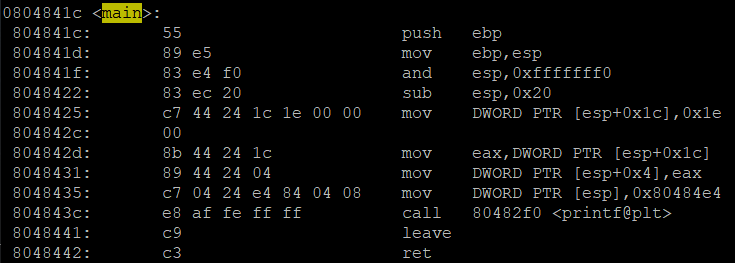
****

gcc 컴파일러를 이용하여 컴파일하고 실행한 결과이다.

**2) Get “ex1.txt” as above and show the asm code for “main”.**

****

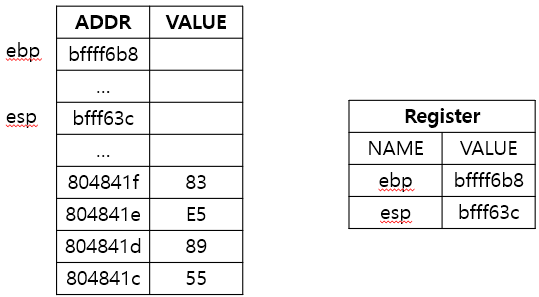
objdump의 -D 옵션을 이용하여 실행파일 ex1의 모든 섹션에 대해 역어셈블을 진행하여 그 결과를 ex1.txt로 리다이렉션 하였다. 그리고 아래 사진은 vi 편집기로 그 txt 파일을 열고 /main을 반복 입력하여 main 함수의 기계어를 찾은 결과이다.

****

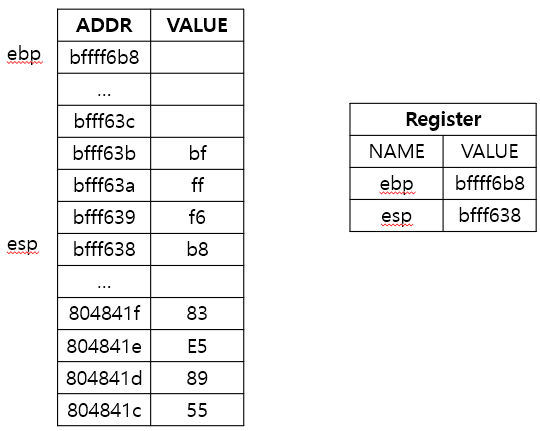
**3) Draw the memory map and show all the changes in registers and memory after each instruction up to the "ret" instruction. Assume esp = 0xbffff63c and ebp = 0xbffff6b8 in the beginning of "main".**

인텔 기반 32bit cpu이므로 리틀 엔디안 방식으로 하위 비트가 스택에 먼저 들어가게 된다. 이때 초기에 주소 값이 큰 ebp를 스택의 바닥으로 인지하는 것이 편리하다.

**<초기 상태>**

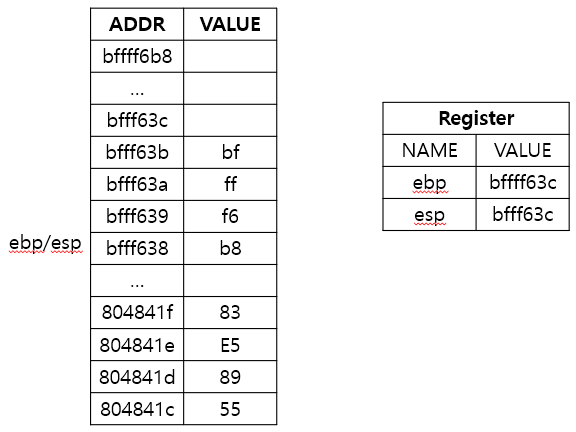


**<push ebp>**



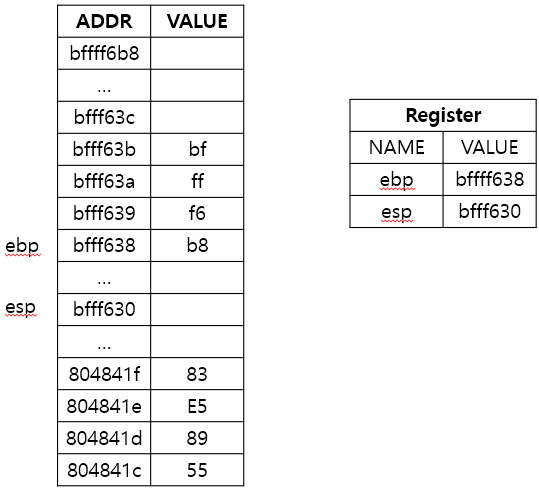
esp -= 4를 해주고 ebp를 스택에 저장된다. 리틀 엔디안 방식으로 데이터가 저장된다.

**<mov ebp, esp>**



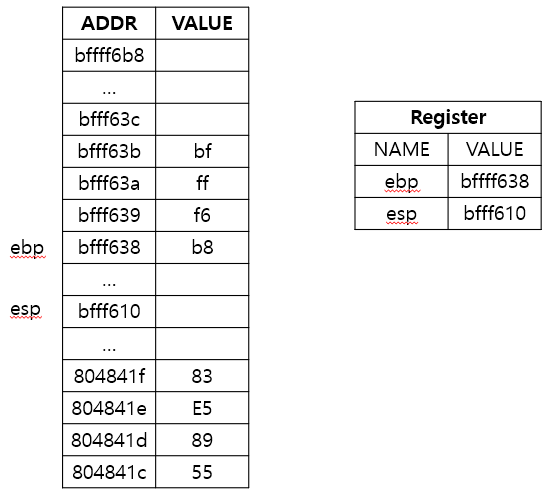
mov a, b는 단순히 b의 값을 a에 덧씌우거나 복사한다는 의미를 갖는다. 따라서 mov ebp, esp의 결과, esp 값이 ebp로 복사된다.

**<and esp, 0xfffffff0>**



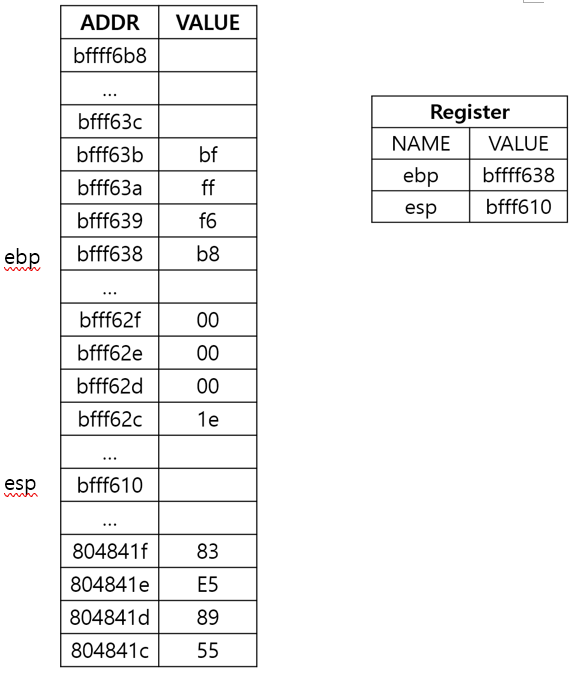
기존의 esp(0xbfff638)과 0xfffffff0의 and 연산 결과는 0x0bfff630이다. 따라서 위처럼 esp의 값이 bfff630으로 변경된다.

**<sub esp, 0x20>**



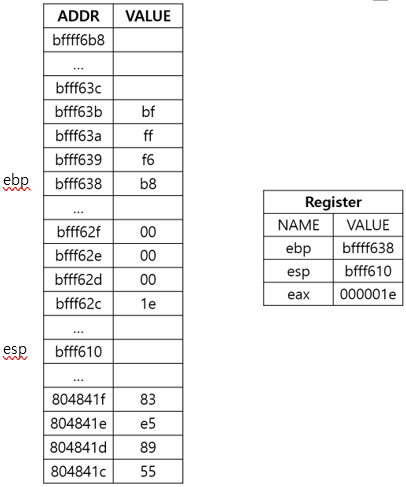
sub는 흔히 알고 있는 마이너스(-) 연산자로, esp-=0x20과 같은 의미이다. 따라서 bfff630이었던 esp는 bfff610으로 변경된다.

**<mov DWORD PTR[esp+0x1c], 0x1e>**



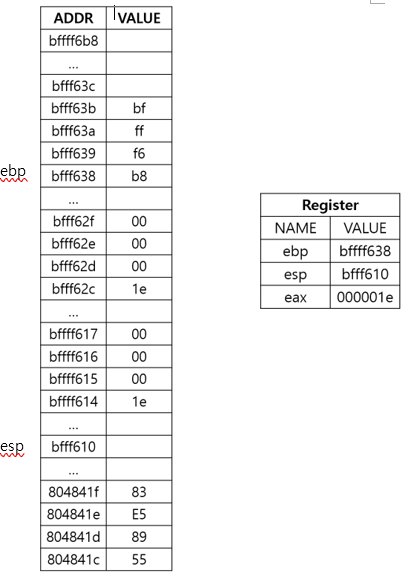
“mov DWORD PTR[esp+0x1c], 0x1e”는 esp+0x1c 주소에 0x1e라는 데이터를 저장하라는 의미이다. 따라서 bfff62f부터 bfff62c까지 차례로 00 00 00 1e의 데이터를 갖게 된다. 이 때 0x0000001e는 10진수로 31이며 ex1.c 코드에서 x=30; 에 해당되는 내용이다.

**<mov eax, DWORD PTR[esp+0x1c]>**



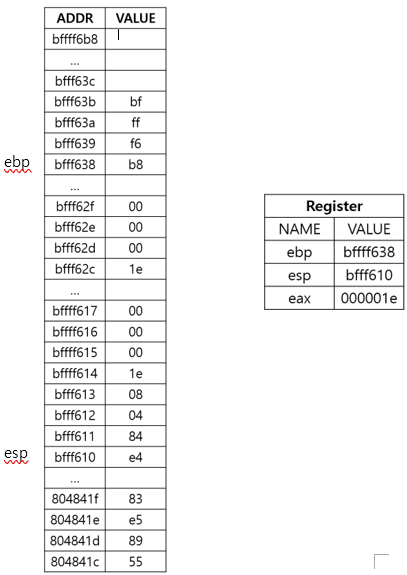
“mov eax, DWORD PTR[esp+0x1c]”은 esp+0x1c 주소에 저장된 데이터를 eax에 저장하라는 의미로, esp+0x1c 주소의 데이터는 0x0000001e이기에 eax 또한 0x0000001e를 갖는다.

**< Mov DWORD PTR [esp+0x4], eax>**

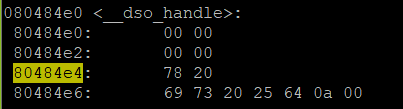


eax를 esp+0x4 주소의 데이터로 저장하라는 의미이다. esp+0x4는 0x0bffff614이므로 0x0bfff617부터 0x0bfff614까지 00 00 00 1e가 차례로 저장된다.

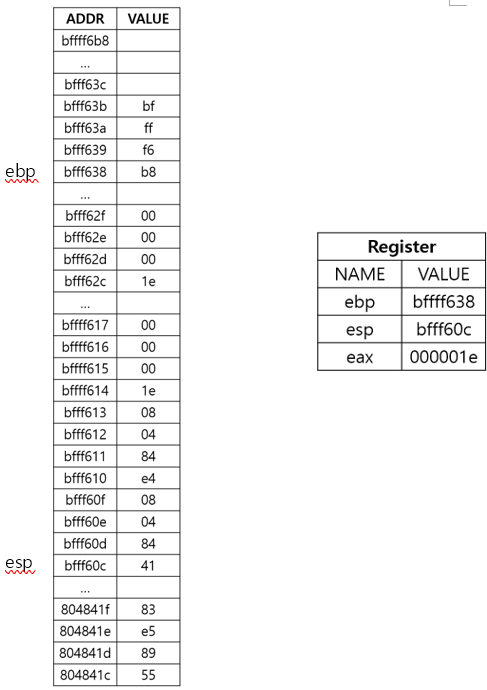
**<mov DWORD PTR [esp], 0x80484e4>**



esp 주소에 0x080484e4 라는 데이터를 저장한다. 이 때, 아래 사진의 메모리 정보를 보면 0x080484e4 주소에는 x is %d\n이라는 문자열이 저장되어 있는 것을 알 수 있다.

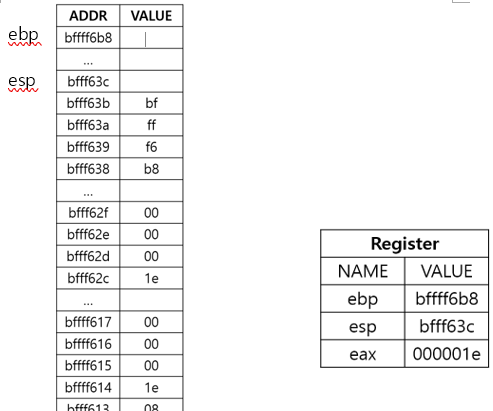


**<call 80482f0 <printf@plt>**

****

call x는 두단계로 이루어진다. 먼저 call 다음에 실행되어야 할 명령어(인스트럭션)의 주소를 esp에 push하고, x로 jump한다. 따라서 esp-=4를 한 후, 다음 인스트럭션의 주소인 0x08048441을 DWORD PTR [esp]에 저장하게 된다.

**<leave>**

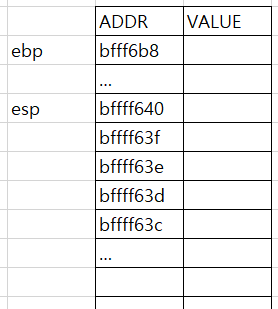


leave 명령어는 스택 프레임을 정리해주는 또한 두 단계로 구성된다.

mov esp, ebp와 pop ebp이다.

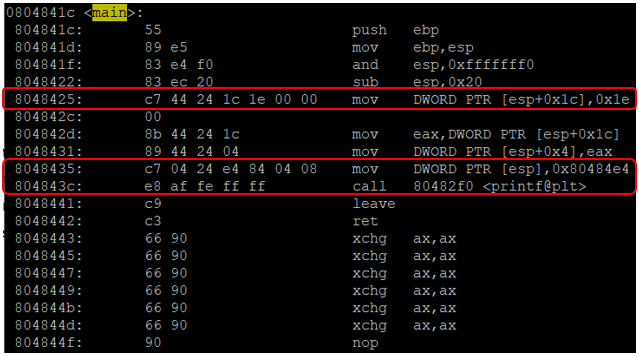
mov esp, ebp에서 현재 ebp와 esp는 0x0bfff638을 값으로 갖는데, pop ebp를 통해 ebp에는 DWORD PTR [esp]가 저장되고 esp+= 4가 된다. 즉 초기의 ebp 값을 돌아가게 된다. ebp와 esp가 초기 상태로 돌아갔기 때문에 아래의 스택들은 의미가 없다고 생각하여 고의적으로 아래 부분은 자른 상태로 캡처하여 사진을 위와 같이 업로드하였다.

**<ret>**



ret은 pop eip, jmp eip로 이루어진다. pop으로 인해 esp+=4 가 될 것으로 예상 가능하다.

**4)** **Find corresponding instructions for "x=30;" and "printf("x is %d\n",x);" in the ASM code.**

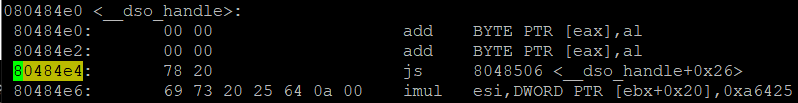
****

3번에서 설명한 내용과 동일하다. 위에 표시한 빨간 박스가 x=30; 해당되고 그 다음 박스가 printf(“x is %d\n, x);에 해당되는 내용이다.

**5) What is the memory location of the variable x?**

3번의 그림을 보면 변수 x는 0x0bfff62c에 위치하는 것을 확인할 수 있다.

**6) Find the memory address where 속 string “x is %d\n” is stored. Confirm the ascii codes for “x is %d\n” at that address.**

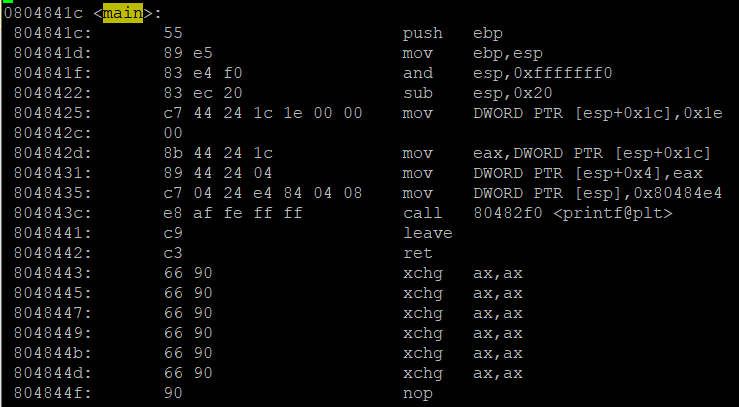
****

3, 4번에서 mov DWORD PTR [esp], 0x8048e4 인스트럭션이 printf(“x is %d\n”, x);와 관련이 있고 0x8048e4에 문자열 “x is %d”가 저장되어 있다고 하였다. ex1.txt에서 메모리 0x08048e4의 내용을 실제로 확인할 결과는 위와 같다. 이 데이터를 아스키 코드로 변경하면 다음과 같다.

78 20 69 73 20 25 64 0a 00

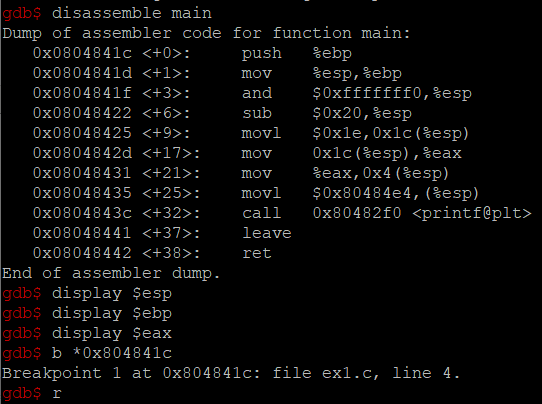
x SP i s SP % d LF \0

**7) Show the memory address where main) begins.**

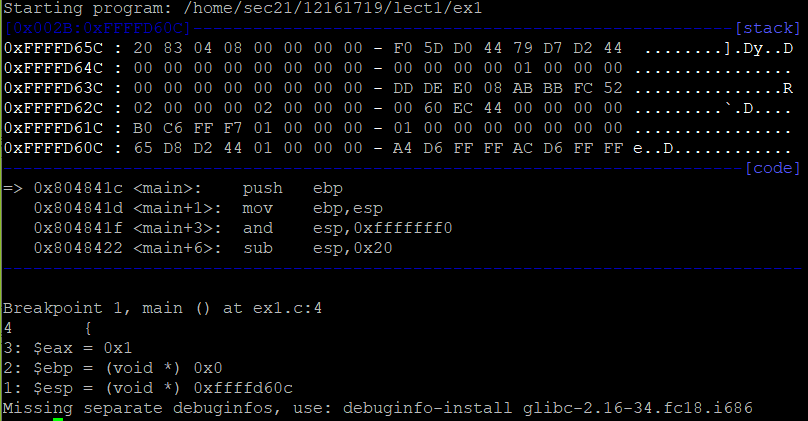
****

위 사진은 ex1.txt를 vi 편집기를 이용하여 확인한 결과이다. 위 사진을 통해서 main 함수는 메모리 0x0804841c에서 시작함을 알 수 있다.

**8) Follow above steps to show the content of the registers or memory that have been changed after each instruction in main(). You should indicate the changed part in your picture (the captured output screen from gdb) for all instructions one by one. For "call" instruction use "si" command to enter the function and show the changes in the stack and register.**

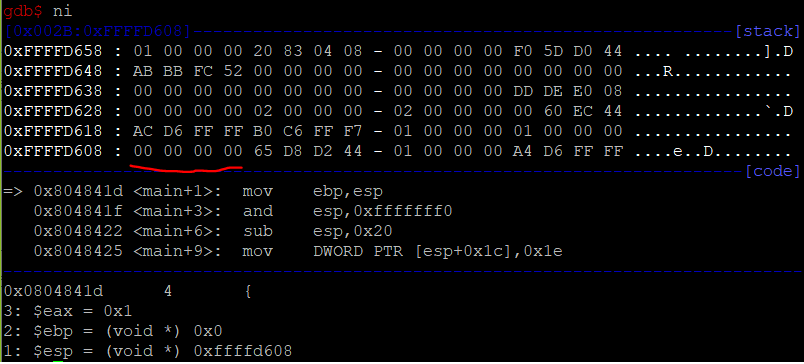
****

gdb 디버거를 실행하기에 앞서 .gdbinit 파일을 ../../linuxer1에서 12161719 디렉토리로 복사해왔다. 그리고 위처럼 ex1을 gdb 디버깅 모드로 실행하여 main 함수를 disassemble하고 하나의 명령어가 실행될 때마다 esp ebp eax 레지스터를 출력하도록 display 명령어를 입력하였다. 이어서 main 함수의 시작점의 breakpoint를 설정하고 r을 입력하여 실행하였다.

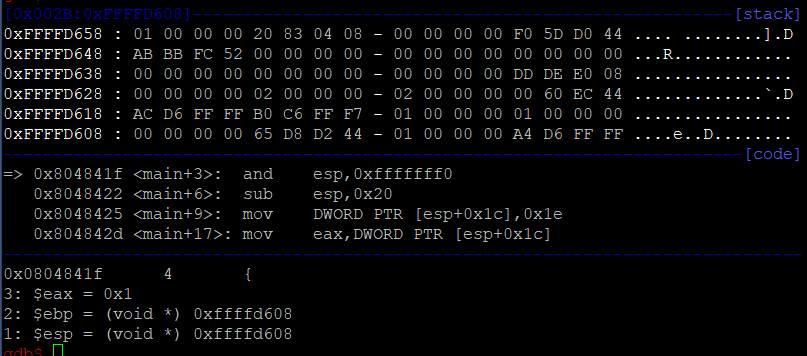


가장 초기의 상태이다.

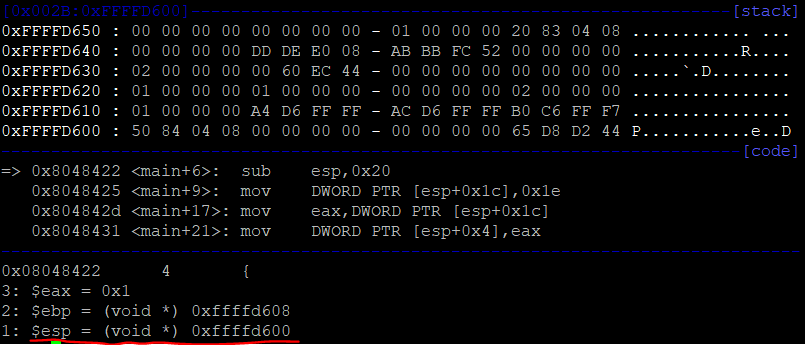
ebp: 0x0 esp: 0xffffd60c 이다.



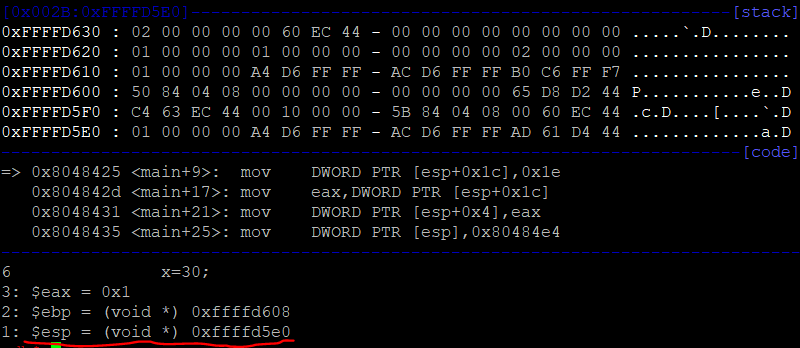
ni를 입력하여 push ebp가 실행된 후의 스택 모습인데, esp가 push로 인해 4만큼 줄어든 것을 확인할 수 있고, 빨간 밑줄 친 부분을 보면 esp에 ebp 값(0x0)이 삽입된 것을 볼 수 있다.



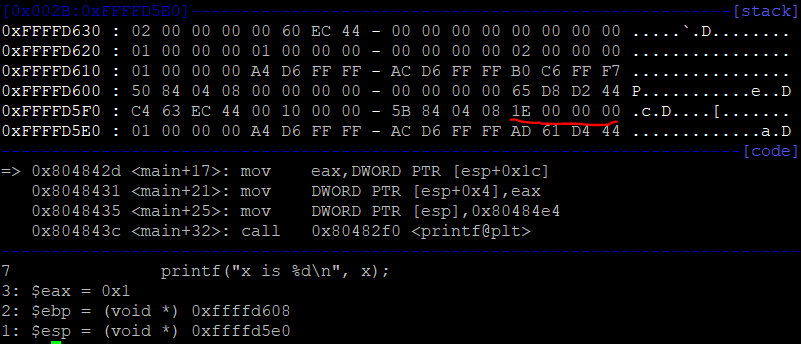
그 다음 인스트럭션인 push ebp, esp가 실행된 결과이다. ebp에 esp의 값이 복사되어 두 레지스터 모두 0xffffd608의 값을 가지고 있는 것을 확인할 수 있다.



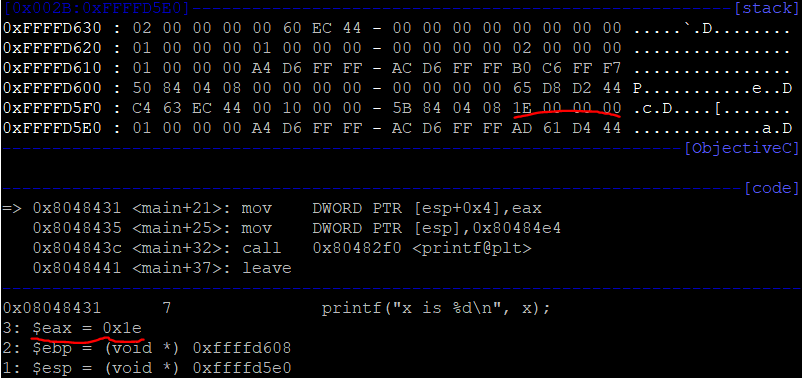
and esp, 0xfffffff0 이 실행된 결과이다. esp와 0xfffffff0의 and 연산 결과인 0xffffd600이 esp이 저장되어 있는 것을 확인할 수 있다.



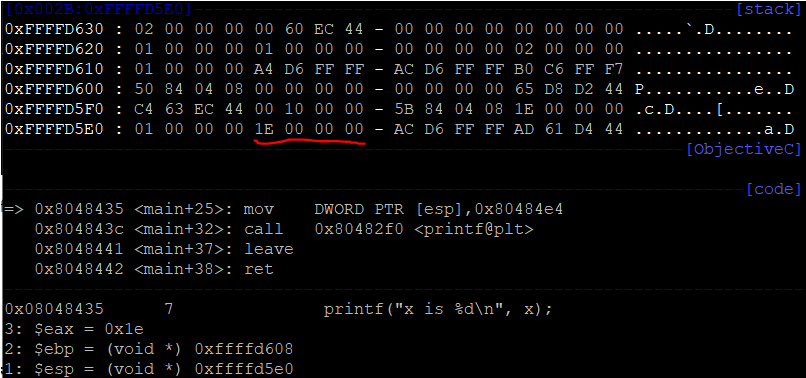
sub esp, 0x20이 실행된 결과이다. esp가 0xffffd5e0으로 변경되었다.



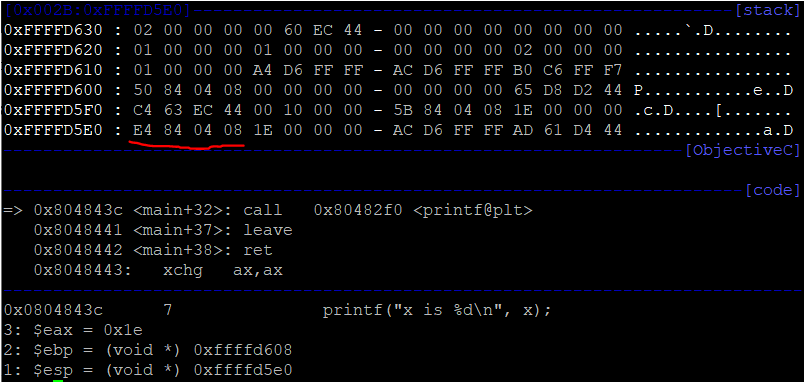
mov DWORD PTR [esp+0x1c], 0x1e의 실행 결과이다. esp+0x1c 즉, 0xffffd5fc 주소에 0x1e를 저장하라는 의미로, 빨간 줄 쳐진 곳을 보면 실제로 해당 주소에 0x1e가 저장된 것을 확인할 수 있다.



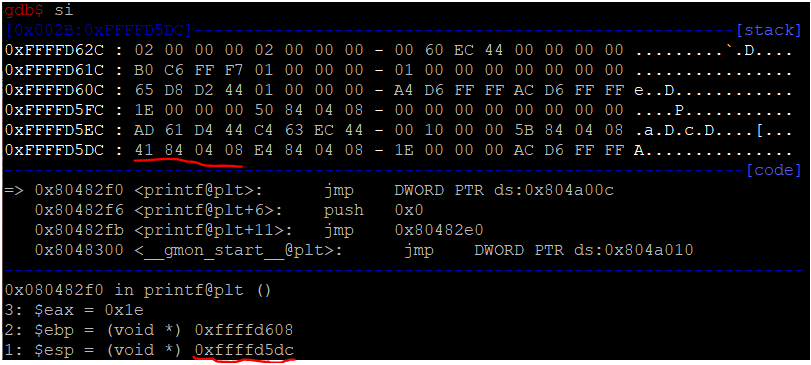
mov eax, DWORD PTR [esp+0x1c]의 결과이다. 0xffffd5fc 주소에 있는 값, 0x1e가 eax 레지스터에 복사되었다.



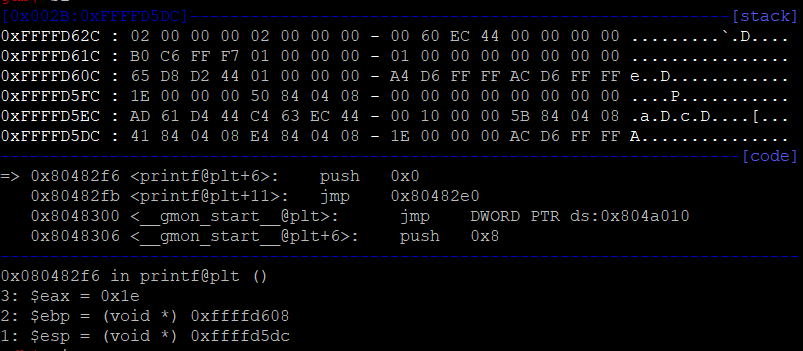
mov DWORD PTR[esp+0x4], eax의 결과이며 0xffffd5e4에 0x1e가 담긴 것을 확인할 수 있다.



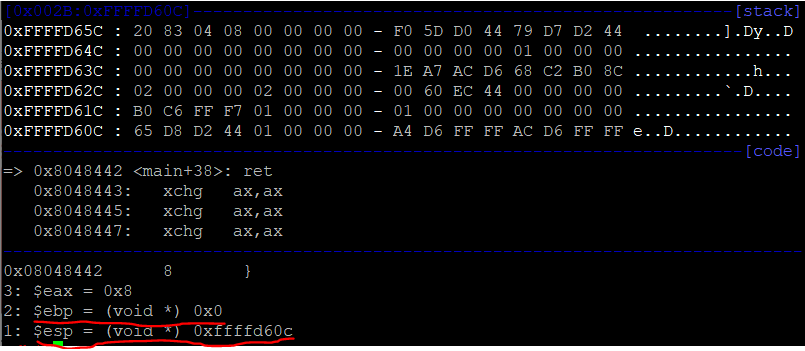
mov DWORD PTR [esp], 0x80484e4의 실행결과이다. esp에 저장된 주소, 0xffffd5e0에 0x80484e4가 저장되어 있는 것을 확인할 수 있다.



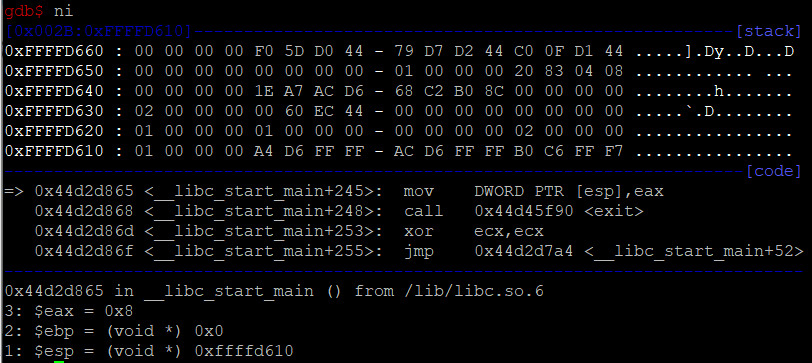
다음으로 si를 입력하여 call 0x80482f0 <printf@plt>이 실행된 결과이다. 먼저, call 이후 실행될 인스트럭션의 주소를 esp에 push하기 위해 esp가 4byte감소하고 0x08048441이 저장된 것을 확인할 수 있다.



call 이후 printf 함수의 기계어 인스트럭션이 진행되는 모습이다. 이후에도 printf의 인스트럭션이 연이어서 진행되지만 바로 leave 인스트럭션이 실행된 결과를 아래에 첨부하였다.



leave 이후, 3번에서 유추하였던 메모리 맵대로, 가장 초기 상태의 스택으로 esp, ebp가 초기화된 것을 확인할 수 있다.



ret 이후의 결과이다. ret은 pop eip의 과정을 포함하기 때문에 esp가 4 증가한 것을 확인할 수 있다.

eax

eax

eax