```
@ubuntu:~/Downloads/akco$ cat code.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

        char buf[0x80];
        printf("buf = (%p)\n", buf);
        scanf("%80s", buf);

        return 0;
}
```

위와 같이 c언어로 코드를 짰다.

0x80 크기의 buf를 선언하고, buf의 주소를 출력해준다. 이후, %80s까지 사용자의 입력을 받을 수 있고 0을 리턴하는 프로그램이다.

```
@ubuntu:~/Downloads/akco$ gcc -o code code.c -no-pie
@ubuntu:~/Downloads/akco$ ls
code  code.c
```

gcc로 컴파일해보면 code라는 파일이 생성된 것을 알 수 있다.

본격적으로 gdb를 사용하여 어셈블리 코드를 분석해보자.

```
pwndbg> disass main
Dump of assembler code for function main:
   0x0000000000401176 <+0>:     endbr64
   0x000000000040117a <+4>:     push   rbp
   0x000000000040117b <+5>:     mov    rbp,rsp
   0x000000000040117e <+8>:     sub    rsp,0xa0
   0x0000000000401185 <+15>:    mov    DWORD PTR [rbp-0x94],edi
   0x000000000040118b <+21>:    mov    QWORD PTR [rbp-0xa0],rsi
   0x0000000000401192 <+28>:    mov    rax,QWORD PTR fs:0x28
   0x000000000040119b <+37>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000040119f <+41>:    xor    eax,eax
   0x00000000004011a1 <+43>:    lea    rax,[rbp-0x90]
   0x00000000004011a8 <+50>:    mov    rsi,rax
   0x00000000004011ab <+53>:    lea    rdi,[rip+0xe52]        # 0x402004
   0x00000000004011b2 <+60>:    mov    eax,0x0
   0x00000000004011b7 <+65>:    call   0x401070 <printf@plt>
   0x00000000004011bc <+70>:    lea    rax,[rbp-0x90]
   0x00000000004011c3 <+77>:    mov    rsi,rax
   0x00000000004011c6 <+80>:    lea    rdi,[rip+0xe43]        # 0x402010
   0x00000000004011cd <+87>:    mov    eax,0x0
   0x00000000004011d2 <+92>:    call   0x401080 <__isoc99_scanf@plt>
   0x00000000004011d7 <+97>:    mov    eax,0x0
   0x00000000004011dc <+102>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000004011e0 <+106>:   xor    rdx,QWORD PTR fs:0x28
   0x00000000004011e9 <+115>:   je     0x4011f0 <main+122>
   0x00000000004011eb <+117>:   call   0x401060 <__stack_chk_fail@plt>
   0x00000000004011f0 <+122>:   leave
   0x00000000004011f1 <+123>:   ret
End of assembler dump.
```

```
pwndbg> b *main+43
Breakpoint 1 at 0x4011a1
pwndbg> b *main+70
Breakpoint 2 at 0x4011bc
```

*main+43을 보면 변수 buf의 주소는 [rbp-0x90]에 저장되어있는 걸 알 수 있다.

따라서 *main+43과 printf 직후에 break point를 걸어주었다.

<어셈블리 코드 분석>

main+0 endbr64 : CET가 활성화됐음을 알리는 프롤로그

main+4 : 함수의 프롤로그. main함수를 호출한 이전 함수의 rbp 값을 push한다.

main+5 : 함수의 프롤로그. 스택이 여기에서부터 쌓일 것을 알려주는 베이스 포인터(rbp)이다.

main+8 : rsp(스택 포인터 레지스터)의 주소를 0xa0(160)만큼 빼두어 함수를 이용할 준비를 한다.

main+15 : edi의 값을 rbp-0x94의 주소가 가리키는 곳에 넣는다. DWORD이므로 해당 주소에서 32비트를 읽는다.

main+21 : rsi의 값을 rbp-0xa0의 주소가 가리키는 곳에 넣는다. QWORD이므로 해당 주소에서 64비트를 읽는다.

main+28 : fs:0x28(랜덤값)을 rax에 넣는다. 이 부분은 스택 카나리를 설정하는 부분이다.

main+37 : rax에 있던 fs:0x28의 값을 rbp-0x8이 가리키는 곳에 넣는다. 0x8만큼의 공간을 카나리를 위해 비워두는 작업이다.

main+41 : eax끼리 XOR 연산을 한다. 같은 값을 xor함으로써 eax레지스터를 0으로 초기화한다. 레지스터 값을 0으로 변경할 때 자주 쓰이는 연산이다. (mov보다 메모리를 덜 차지한다.)

main+43 : rbp-0x90의 주소를 rax로 옮긴다. [rbp-0x90]은 버퍼를 위해 0x90만큼 비워둔 곳의 주소이다. 스택은 높은 주소부터 낮은 주소로 쌓이기 때문에 뺄셈 연산을 통해 공간을 확보한다.

main+50 : rax에 있던 [rbp-0x90]을 rsi로 옮긴다.

main+53 : [rip+0xe52]의 주소를 rdi로 옮긴다. rip는 현재 명령어의 위치를 가리킨다. rip로부터 0xe52만큼 전에 위치한 명령어를 가져온다.

main+60 : eax의 값을 0으로 만든다. (이때 왜 xor eax, eax를 안했는지 모르겠다.)

main+65 : printf 함수를 호출한다.

main+70 : [rbp-0x90]의 주소를 rax에 담는다.

main+77 : rax에 담긴 값([rbp-0x90])을 rsi에 옮긴다.

main+80 : [rip+0xe43]의 주소를 rdi로 옮긴다.

main+87 : eax의 값을 0으로 만든다.

main+92 : scanf 함수를 호출한다.

main+97 : eax의 값을 0으로 만든다.

main+102 : [rbp-0x8]에 저장된 값을 64비트 읽어와 rdx에 옮긴다.

main+106 : rdx에 담긴 값과 fs:0x28에 있는 랜덤값을 xor연산한다. 카나리가 변조되었는지 비교하는 부분이다.

main+115 : 위 xor 연산에서 두 값이 같으면 main+122로 분기하고, 다르면 다음 명령어를 실행한다.

main+117 : __stack_chk_fail함수를 호출해서 실행에 오류가 있음을 알린다.

main+122 : 함수의 에필로그. 여러 레지스터와 메모리 공간을 정리한다.

main+123 : 함수의 에필로그. 리턴 어드레스를 eip에 넣고 jmp를 통해 분기하여 함수를 종료한다.

<브레이크 포인트 분석>


첫 번째 bp에서 레지스터의 값은 위와 같다.

ni를 통해 printf가 있는 부분까지 왔다.

```
pwndbg> ni
buf = (0x7fffffffdfa0)

Breakpoint 2, 0x00000000004011bc in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
─────────────────────────────────[ REGISTERS ]──────────────────────────────────
*RAX  0x17
 RBX  0x401200 (__libc_csu_init) ◂— endbr64
*RCX  0x0
*RDX  0x0
*RDI  0x7ffff7fb27e0 (_IO_stdfile_1_lock) ◂— 0x0
*RSI  0x4052a0 ◂— 'buf = (0x7fffffffdfa0)\n'
 R8   0x0
*R9   0x17
*R10  0x40200d ◂— 0x73303825000a29 /* ')\n' */
*R11  0x246
 R12  0x401090 (_start) ◂— endbr64
 R13  0x7fffffffe120 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x7fffffffe030 ◂— 0x0
 RSP  0x7fffffffdf90 —▸ 0x7fffffffe128 —▸ 0x7fffffffe43d ◂— '/home/      Download
s/akco/code'
*RIP  0x4011bc (main+70) ◂— lea    rax, [rbp - 0x90]
```

**buf의 주소였던 0x7fffffffdfa0**를 출력해준 것을 확인할 수 있다. 이후 rax에 다시 buf의 주소를 옮겨둔다.

```
pwndbg> ni
aaaa
0x00000000004011d7 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
```

ni를 하면 scanf함수 때문에 사용자 입력을 기다린다. aaaa를 입력해주었다.

```
─────────────────────────────────[ STACK ]──────────────────────────────────
00:0000│ rsp 0x7fffffffdf90 —▸ 0x7fffffffe128 —▸ 0x7fffffffe43d ◂— '/home/      /D
ownloads/akco/code'
01:0008│     0x7fffffffdf98 ◂— 0x100000000
02:0010│     0x7fffffffdfa0 ◂— 0x61616161 /* 'aaaa' */
03:0018│     0x7fffffffdfa8 ◂— 0x0
... ↓       4 skipped
```

rsp+0x10의 위치에 'aaaa'가 들어가있다. a는 아스키코드로 0x61이기 때문에 0x7fffffffdfa0에는 0x61616161가 들어있다.

```
pwndbg> x/10wx 0x7fffffffdfa0
0x7fffffffdfa0: 0x61616161      0x00000000      0x00000000      0x00000000
0x7fffffffdfb0: 0x00000000      0x00000000      0x00000000      0x00000000
0x7fffffffdfc0: 0x00000000      0x00000000
```

buf의 주소를 메모리덤프해보면 aaaa가 들어가있는 것을 알 수 있다.

```
pwndbg> ni
[Inferior 1 (process 2951) exited normally]
```

프로그램이 종료되었다.