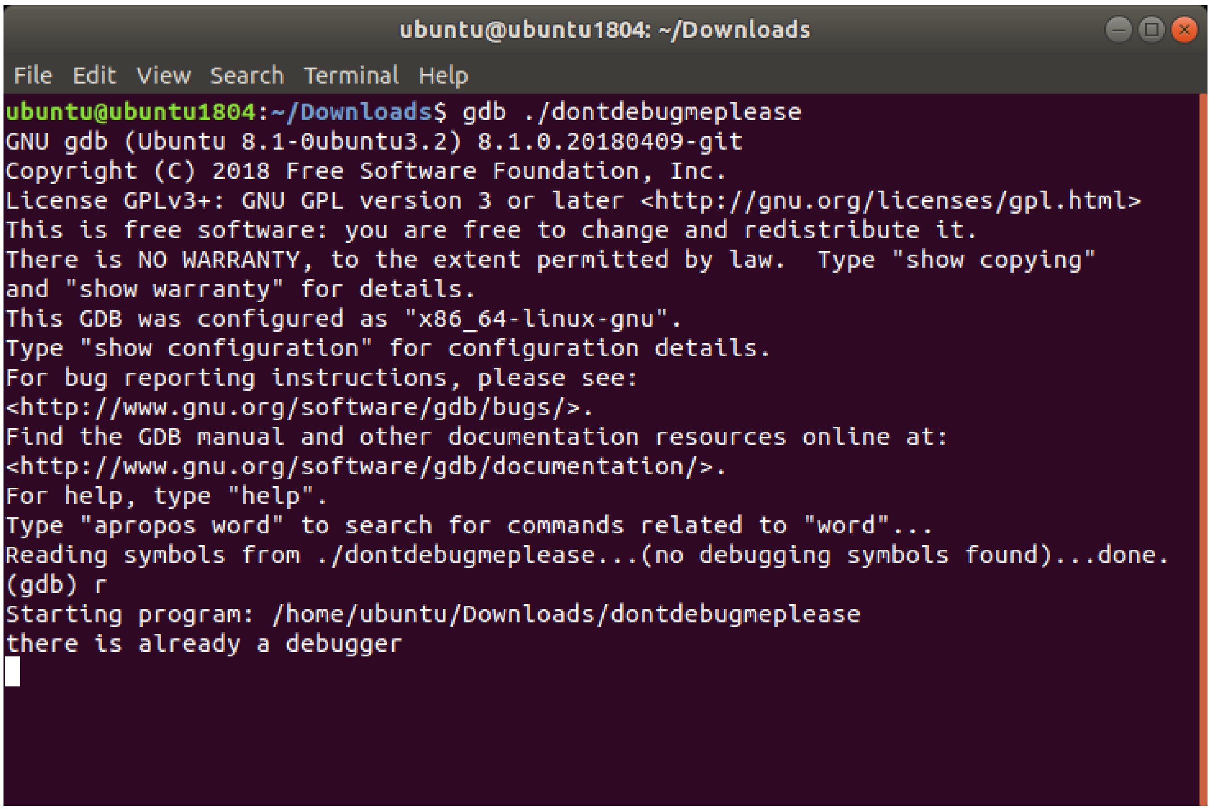
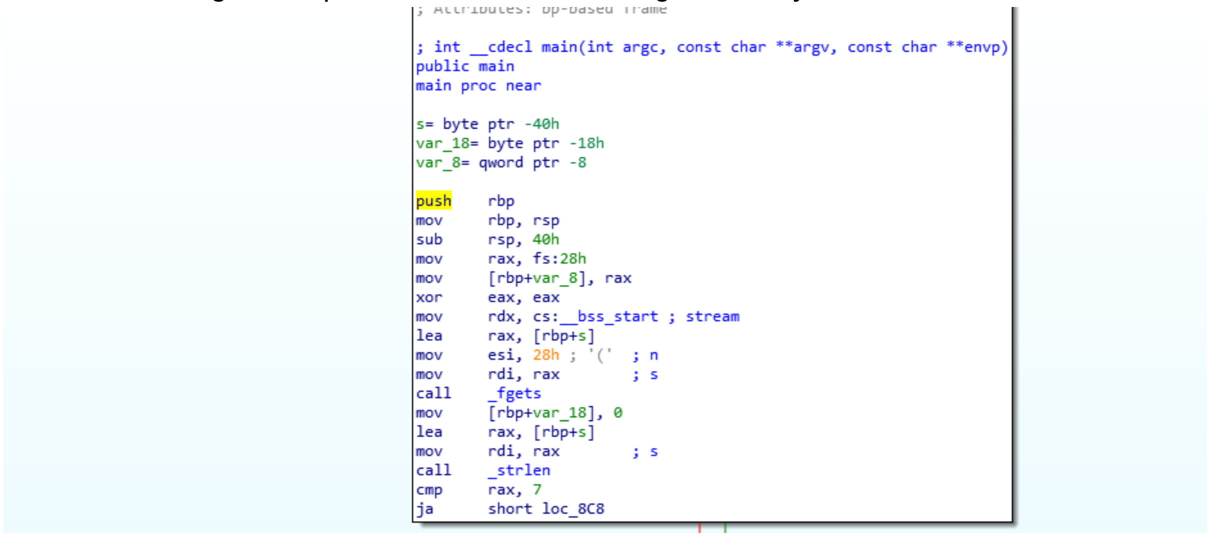
As the title say, the program detects if we are running it in a debugger:


ubuntu@ubuntu1804: ~/Downloads
File Edit View Search Terminal Help

```
ubuntu@ubuntu1804:~/Downloads$ gdb ./dontdebugmeplease
GNU gdb (Ubuntu 8.1-0ubuntu3.2) 8.1.0.20180409-git
Copyright (C) 2018 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./dontdebugmeplease...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/ubuntu/Downloads/dontdebugmeplease
there is already a debugger
```

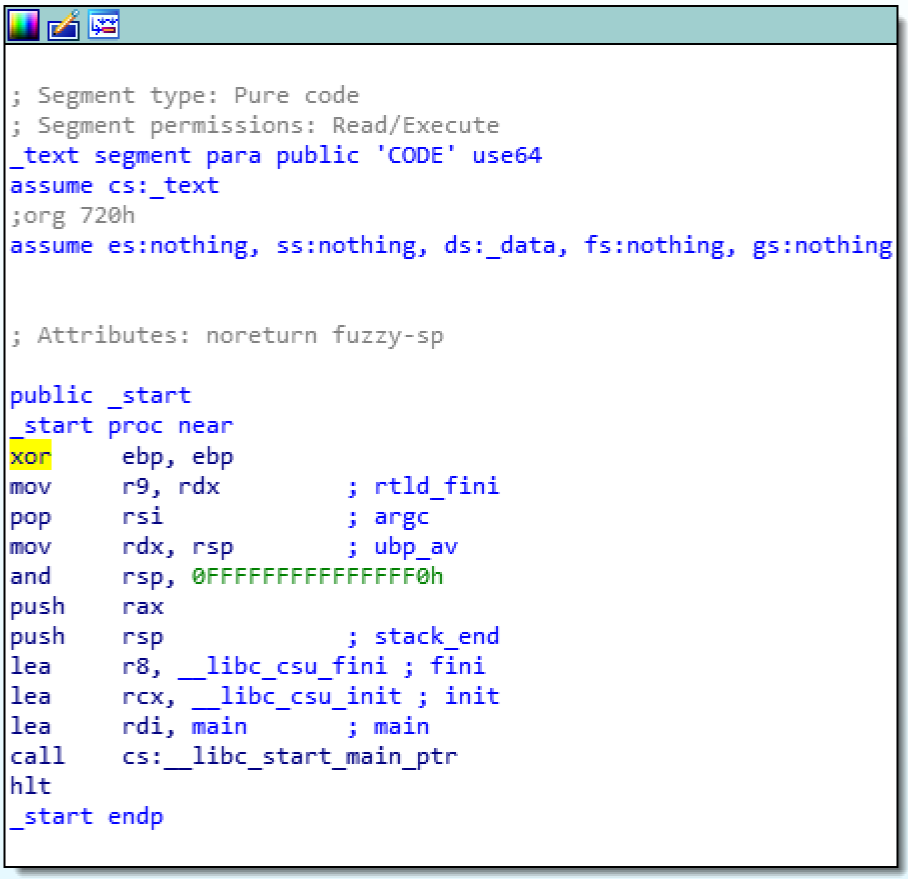
So an anti-debug technique is used. Disassembling the binary we see the main:



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

s= byte ptr -40h
var_18= byte ptr -18h
var_8= qword ptr -8

push    rbp
mov     rbp, rsp
sub     rsp, 40h
mov     rax, fs:28h
mov     [rbp+var_8], rax
xor     eax, eax
mov     rdx, cs:__bss_start ; stream
lea     rax, [rbp+s]
mov     esi, 28h ; '('   ; n
mov     rdi, rax         ; s
call    _fgets
mov     [rbp+var_18], 0
lea     rax, [rbp+s]
mov     rdi, rax         ; s
call    _strlen
cmp     rax, 7
ja      short loc_8C8
```

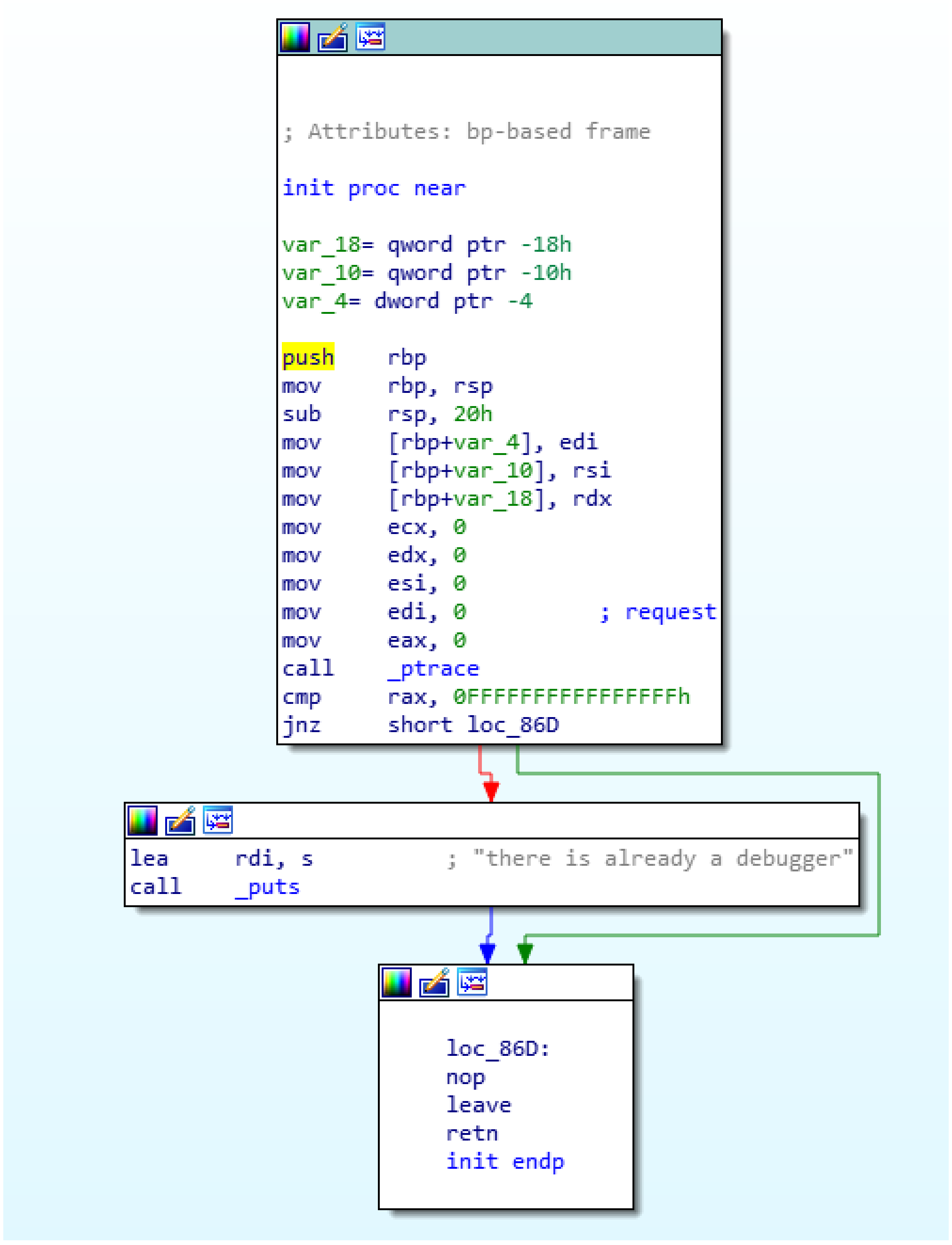
However, there are no calls to check if the debugger is running or the print “there is already a debugger”, so probably the check happens before the main starts.

We can then go to the entrypoint of the program, which is the function called _start.



```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment para public 'CODE' use64
assume cs:_text
;org 720h
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


; Attributes: noreturn fuzzy-sp

public _start
_start proc near
xor     ebp, ebp
mov     r9, rdx         ; rtld_fini
pop     rsi             ; argc
mov     rdx, rsp        ; ubp_av
and     rsp, 0FFFFFFFFFFFFFFF0h
push    rax
push    rsp             ; stack_end
lea     r8, __libc_csu_fini ; fini
lea     rcx, __libc_csu_init ; init
lea     rdi, main       ; main
call    cs:__libc_start_main_ptr
hlt
_start endp
```

Looking at the function, we see that fini and init are defined. In particular, the init function is executed before the main, so it could contain the anti-debug check.



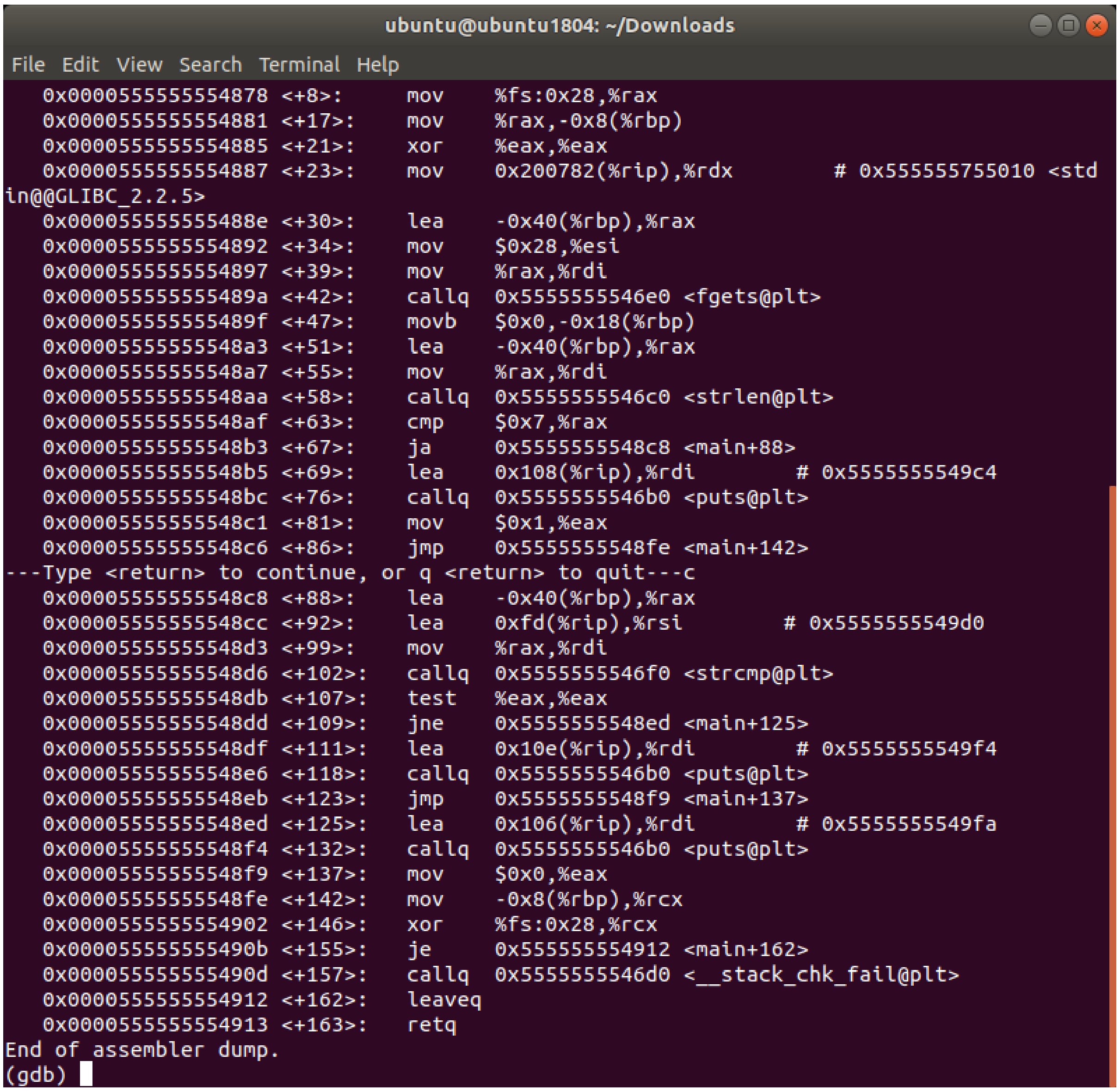
```
; Attributes: bp-based frame

init proc near

var_18= qword ptr -18h
var_10= qword ptr -10h
var_4= dword ptr -4

push    rbp
mov     rbp, rsp
sub     rsp, 20h
mov     [rbp+var_4], edi
mov     [rbp+var_10], rsi
mov     [rbp+var_18], rdx
mov     ecx, 0
mov     edx, 0
mov     esi, 0
mov     edi, 0          ; request
mov     eax, 0
call    _ptrace
cmp     rax, 0FFFFFFFFFFFFFFFFh
jnz     short loc_86D
```

```
lea     rdi, s          ; "there is already a debugger"
call    _puts
```

```
loc_86D:
nop
leave
retn
init endp
```

Our intuition was good! Here we can see the program calls _ptrace to check if there is a debugging (rax == -1). We can patch the call _ptrace instruction filling it with nops.

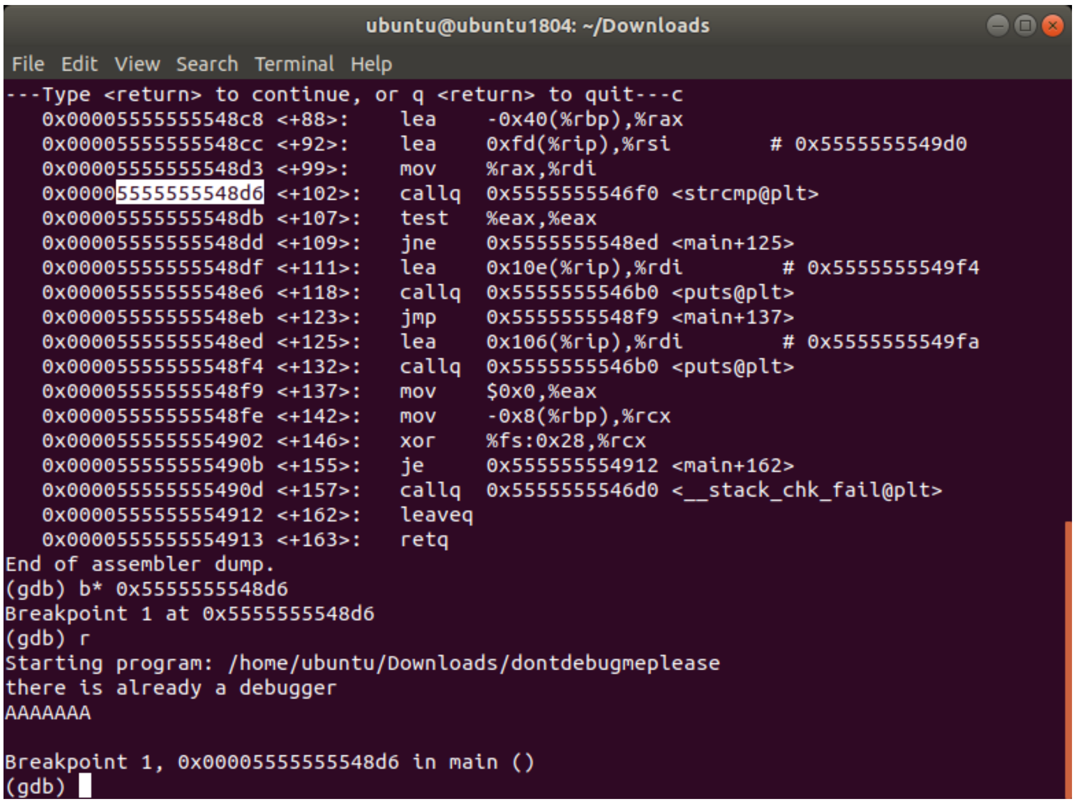
After the patch, we can use gdb to recover our flag.
The program waits for an input, and prints “nope” or “nope dude” if the flag is wrong, depending on the length of the input.
Let’s start disassembling the main


ubuntu@ubuntu1804: ~/Downloads
File Edit View Search Terminal Help

```
   0x0000555555554878 <+8>:     mov    %fs:0x28,%rax
   0x0000555555554881 <+17>:    mov    %rax,-0x8(%rbp)
   0x0000555555554885 <+21>:    xor    %eax,%eax
   0x0000555555554887 <+23>:    mov    0x200782(%rip),%rdx        # 0x555555755010 <std
in@@GLIBC_2.2.5>
   0x000055555555488e <+30>:    lea    -0x40(%rbp),%rax
   0x0000555555554892 <+34>:    mov    $0x28,%esi
   0x0000555555554897 <+39>:    mov    %rax,%rdi
   0x000055555555489a <+42>:    callq  0x5555555546e0 <fgets@plt>
   0x000055555555489f <+47>:    movb   $0x0,-0x18(%rbp)
   0x00005555555548a3 <+51>:    lea    -0x40(%rbp),%rax
   0x00005555555548a7 <+55>:    mov    %rax,%rdi
   0x00005555555548aa <+58>:    callq  0x5555555546c0 <strlen@plt>
   0x00005555555548af <+63>:    cmp    $0x7,%rax
   0x00005555555548b3 <+67>:    ja     0x5555555548c8 <main+88>
   0x00005555555548b5 <+69>:    lea    0x108(%rip),%rdi        # 0x5555555549c4
   0x00005555555548bc <+76>:    callq  0x5555555546b0 <puts@plt>
   0x00005555555548c1 <+81>:    mov    $0x1,%eax
   0x00005555555548c6 <+86>:    jmp    0x5555555548fe <main+142>
---Type <return> to continue, or q <return> to quit---c
   0x00005555555548c8 <+88>:    lea    -0x40(%rbp),%rax
   0x00005555555548cc <+92>:    lea    0xfd(%rip),%rsi        # 0x5555555549d0
   0x00005555555548d3 <+99>:    mov    %rax,%rdi
   0x00005555555548d6 <+102>:   callq  0x5555555546f0 <strcmp@plt>
   0x00005555555548db <+107>:   test   %eax,%eax
   0x00005555555548dd <+109>:   jne    0x5555555548ed <main+125>
   0x00005555555548df <+111>:   lea    0x10e(%rip),%rdi        # 0x5555555549f4
   0x00005555555548e6 <+118>:   callq  0x5555555546b0 <puts@plt>
   0x00005555555548eb <+123>:   jmp    0x5555555548f9 <main+137>
   0x00005555555548ed <+125>:   lea    0x106(%rip),%rdi        # 0x5555555549fa
   0x00005555555548f4 <+132>:   callq  0x5555555546b0 <puts@plt>
   0x00005555555548f9 <+137>:   mov    $0x0,%eax
   0x00005555555548fe <+142>:   mov    -0x8(%rbp),%rcx
   0x0000555555554902 <+146>:   xor    %fs:0x28,%rcx
   0x000055555555490b <+155>:   je     0x555555554912 <main+162>
   0x000055555555490d <+157>:   callq  0x5555555546d0 <__stack_chk_fail@plt>
   0x0000555555554912 <+162>:   leaveq
   0x0000555555554913 <+163>:   retq
End of assembler dump.
(gdb) 
```

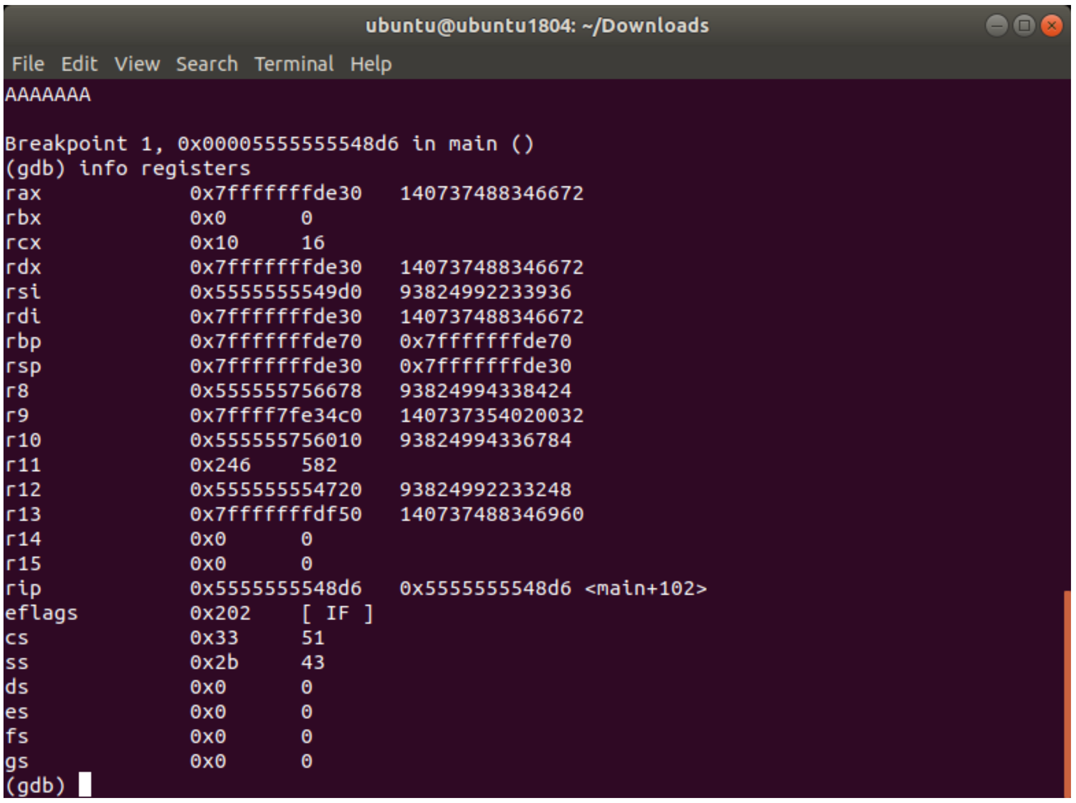
We can see that it calls fgets to get the user input, and then uses strlen to compare the input length with 7. If above, it continues the checks, otherwise it ends.
So we know that the input must be longer than 7.
Then it uses a strcmp to check two strings, the user input again and another string, hopefully our flag. If we check the registers rdi and rsi before the strcmp call, we might be able to retrieve the flag.

So let's put a breakpoint at the call address (0x5555555548d6), and run the program giving an input with 7 letters


ubuntu@ubuntu1804: ~/Downloads
File Edit View Search Terminal Help

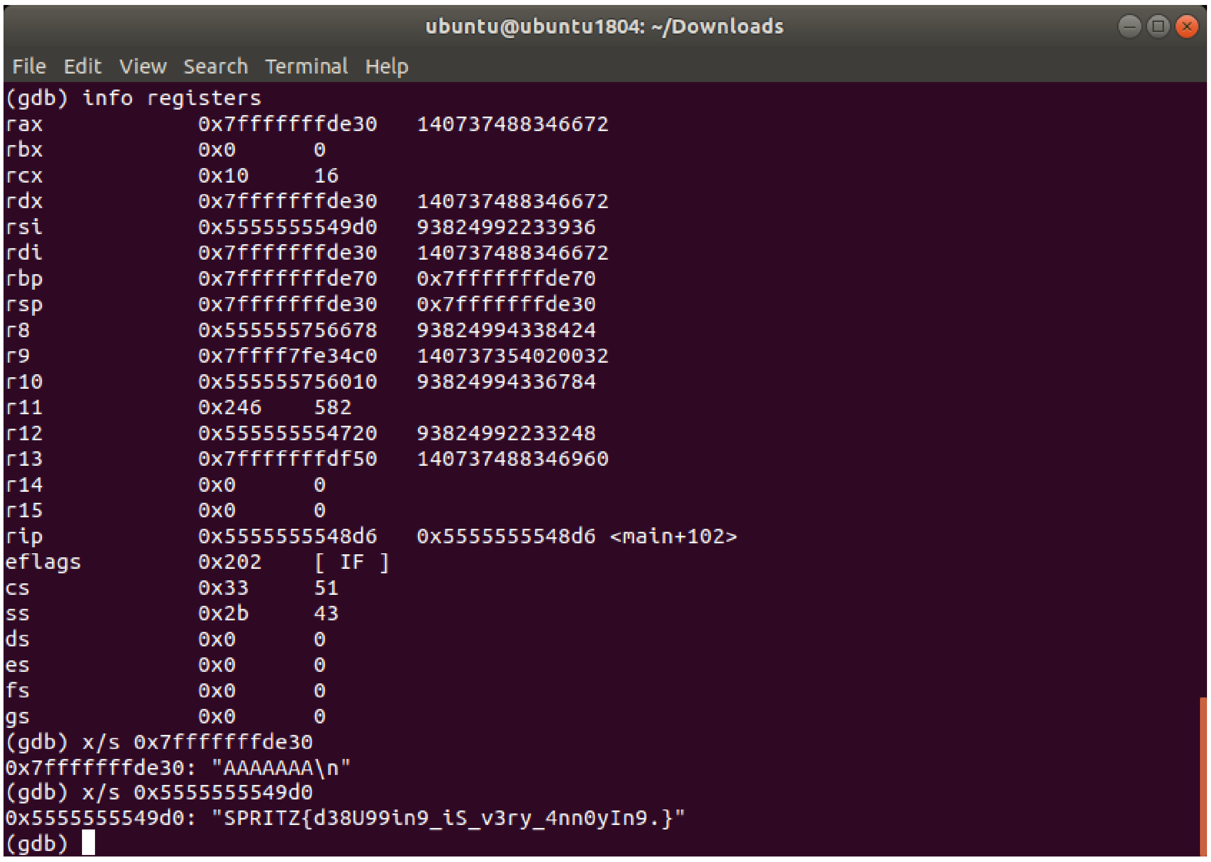
```
---Type <return> to continue, or q <return> to quit---c
   0x00005555555548c8 <+88>:    lea    -0x40(%rbp),%rax
   0x00005555555548cc <+92>:    lea    0xfd(%rip),%rsi        # 0x5555555549d0
   0x00005555555548d3 <+99>:    mov    %rax,%rdi
   0x00005555555548d6 <+102>:   callq  0x5555555546f0 <strcmp@plt>
   0x00005555555548db <+107>:   test   %eax,%eax
   0x00005555555548dd <+109>:   jne    0x5555555548ed <main+125>
   0x00005555555548df <+111>:   lea    0x10e(%rip),%rdi        # 0x5555555549f4
   0x00005555555548e6 <+118>:   callq  0x5555555546b0 <puts@plt>
   0x00005555555548eb <+123>:   jmp    0x5555555548f9 <main+137>
   0x00005555555548ed <+125>:   lea    0x106(%rip),%rdi        # 0x5555555549fa
   0x00005555555548f4 <+132>:   callq  0x5555555546b0 <puts@plt>
   0x00005555555548f9 <+137>:   mov    $0x0,%eax
   0x00005555555548fe <+142>:   mov    -0x8(%rbp),%rcx
   0x0000555555554902 <+146>:   xor    %fs:0x28,%rcx
   0x000055555555490b <+155>:   je     0x555555554912 <main+162>
   0x000055555555490d <+157>:   callq  0x5555555546d0 <__stack_chk_fail@plt>
   0x0000555555554912 <+162>:   leaveq
   0x0000555555554913 <+163>:   retq
End of assembler dump.
(gdb) b* 0x5555555548d6
Breakpoint 1 at 0x5555555548d6
(gdb) r
Starting program: /home/ubuntu/Downloads/dontdebugmeplease
there is already a debugger
AAAAAAA

Breakpoint 1, 0x00005555555548d6 in main ()
(gdb) 
```

When we reach the breakpoints, we can inspect the registers using info registers


ubuntu@ubuntu1804: ~/Downloads
File Edit View Search Terminal Help

```
AAAAAAA

Breakpoint 1, 0x00005555555548d6 in main ()
(gdb) info registers
rax            0x7fffffffde30   140737488346672
rbx            0x0      0
rcx            0x10     16
rdx            0x7fffffffde30   140737488346672
rsi            0x5555555549d0   93824992233936
rdi            0x7fffffffde30   140737488346672
rbp            0x7fffffffde70   0x7fffffffde70
rsp            0x7fffffffde30   0x7fffffffde30
r8             0x555555756678   93824994338424
r9             0x7ffff7fe34c0   140737354020032
r10            0x555555756010   93824994336784
r11            0x246    582
r12            0x555555554720   93824992233248
r13            0x7fffffffdf50   140737488346960
r14            0x0      0
r15            0x0      0
rip            0x5555555548d6   0x5555555548d6 <main+102>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) 
```

They contain memory addresses, so let's inspect the memory, interpreting it as a string, using x/s 0xaddress


ubuntu@ubuntu1804: ~/Downloads
File Edit View Search Terminal Help

```
(gdb) info registers
rax            0x7fffffffde30   140737488346672
rbx            0x0      0
rcx            0x10     16
rdx            0x7fffffffde30   140737488346672
rsi            0x5555555549d0   93824992233936
rdi            0x7fffffffde30   140737488346672
rbp            0x7fffffffde70   0x7fffffffde70
rsp            0x7fffffffde30   0x7fffffffde30
r8             0x555555756678   93824994338424
r9             0x7ffff7fe34c0   140737354020032
r10            0x555555756010   93824994336784
r11            0x246    582
r12            0x555555554720   93824992233248
r13            0x7fffffffdf50   140737488346960
r14            0x0      0
r15            0x0      0
rip            0x5555555548d6   0x5555555548d6 <main+102>
eflags         0x202    [ IF ]
cs             0x33     51
ss             0x2b     43
ds             0x0      0
es             0x0      0
fs             0x0      0
gs             0x0      0
(gdb) x/s 0x7fffffffde30
0x7fffffffde30: "AAAAAAA\n"
(gdb) x/s 0x5555555549d0
0x5555555549d0: "SPRITZ{d38U99in9_iS_v3ry_4nn0yIn9.}"
(gdb) 
```

As we thought, the program compares the flag with our input.

Flag: SPRITZ{d38U99in9_iS_v3ry_4nn0yIn9.}