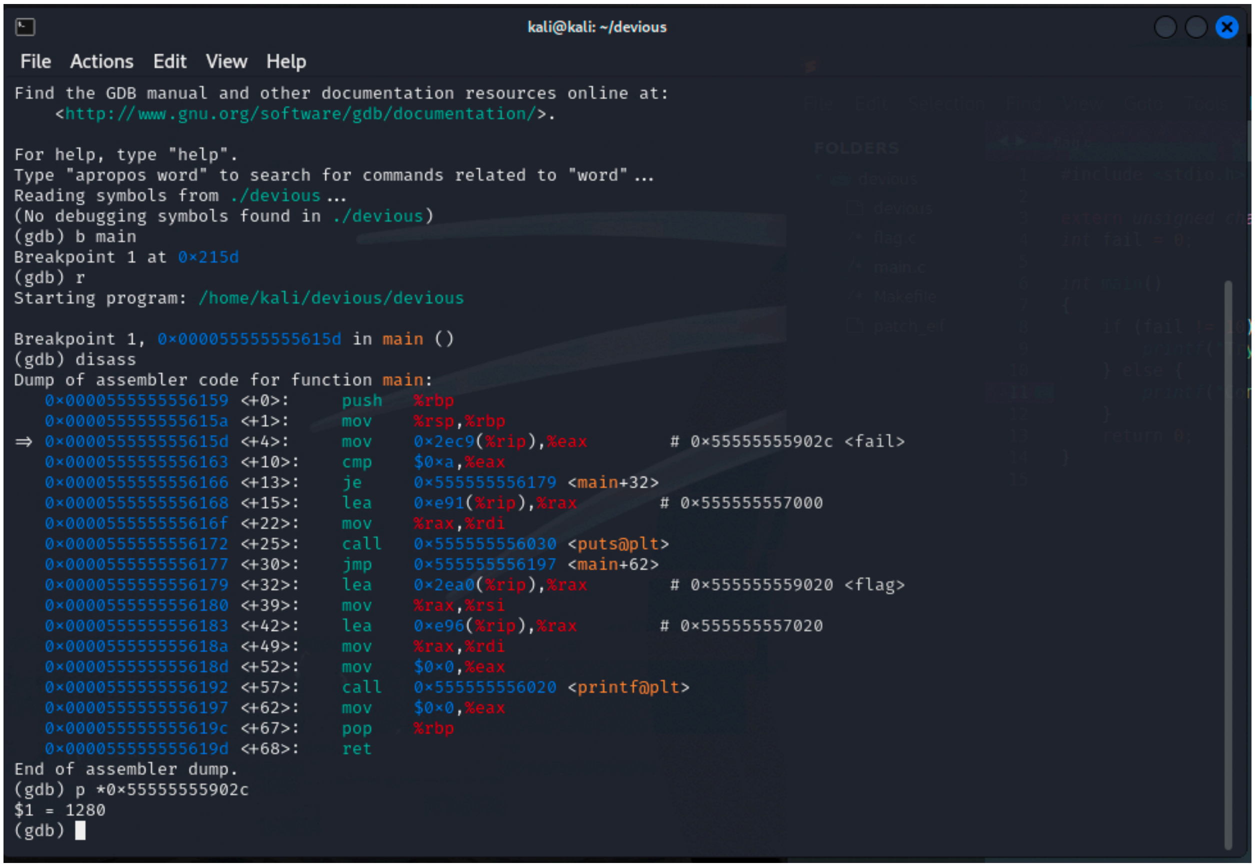
If we study the Makefile, then we can see that flag.c and main.c are compiled into devious_interim. Thereafter, patch_elf does something to it and we get devious.

Let us set the flag variable to “abcdef”.

If we run gdb on devious, set breakpoint at main and examine the fail variable, then we get the following:



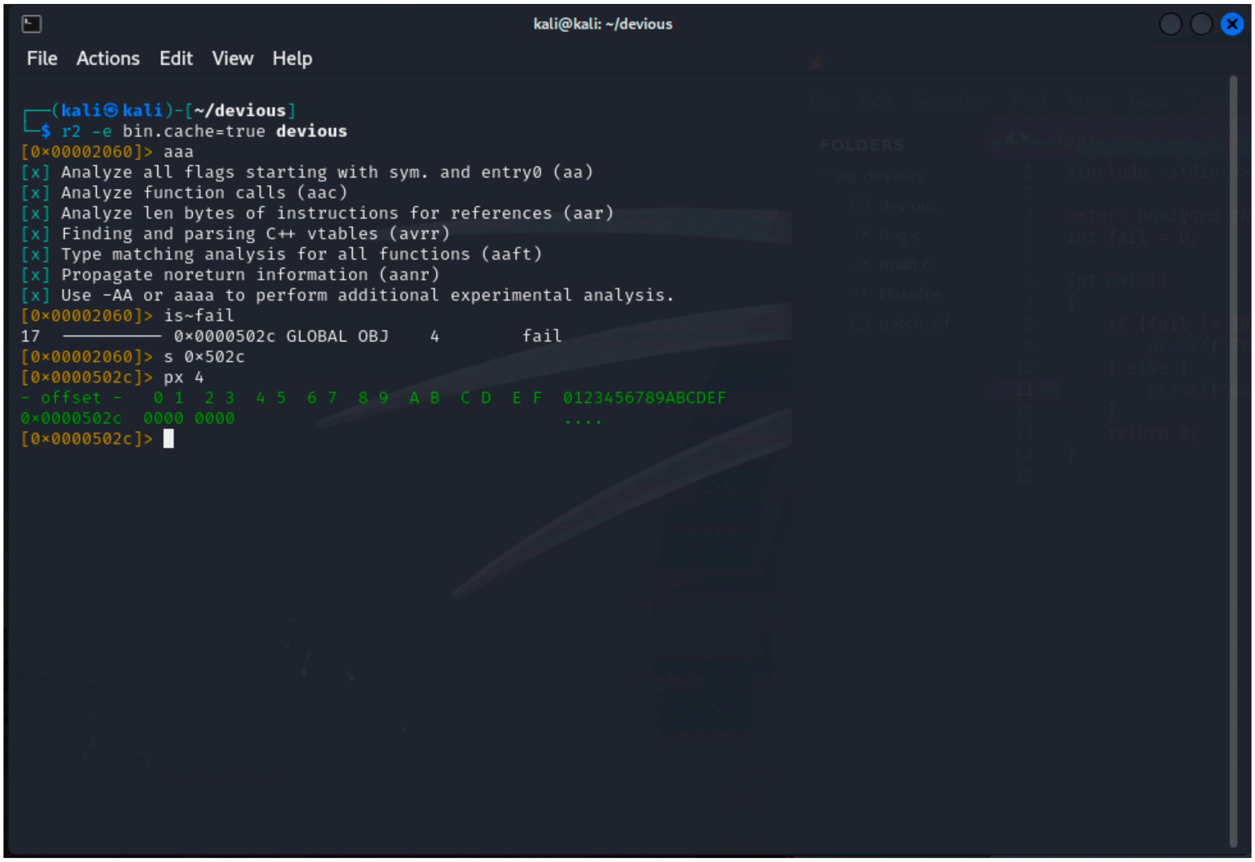
```
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./devious...
(No debugging symbols found in ./devious)
(gdb) b main
Breakpoint 1 at 0x215d
(gdb) r
Starting program: /home/kali/devious/devious

Breakpoint 1, 0x000055555555615d in main ()
(gdb) disass
Dump of assembler code for function main:
   0x0000555555556159 <+0>:     push   %rbp
   0x000055555555615a <+1>:     mov    %rsp,%rbp
=> 0x000055555555615d <+4>:     mov    0x2ec9(%rip),%eax        # 0x55555555902c <fail>
   0x0000555555556163 <+10>:    cmp    $0xa,%eax
   0x0000555555556166 <+13>:    je     0x555555556179 <main+32>
   0x0000555555556168 <+15>:    lea    0xe91(%rip),%rax        # 0x555555557000
   0x000055555555616f <+22>:    mov    %rax,%rdi
   0x0000555555556172 <+25>:    call   0x555555556030 <puts@plt>
   0x0000555555556177 <+30>:    jmp    0x555555556197 <main+62>
   0x0000555555556179 <+32>:    lea    0x2ea0(%rip),%rax        # 0x555555559020 <flag>
   0x0000555555556180 <+39>:    mov    %rax,%rsi
   0x0000555555556183 <+42>:    lea    0xe96(%rip),%rax        # 0x555555557020
   0x000055555555618a <+49>:    mov    %rax,%rdi
   0x000055555555618d <+52>:    mov    $0x0,%eax
   0x0000555555556192 <+57>:    call   0x555555556020 <printf@plt>
   0x0000555555556197 <+62>:    mov    $0x0,%eax
   0x000055555555619c <+67>:    pop    %rbp
   0x000055555555619d <+68>:    ret
End of assembler dump.
(gdb) p *0x55555555902c
$1 = 1280
(gdb) 
```

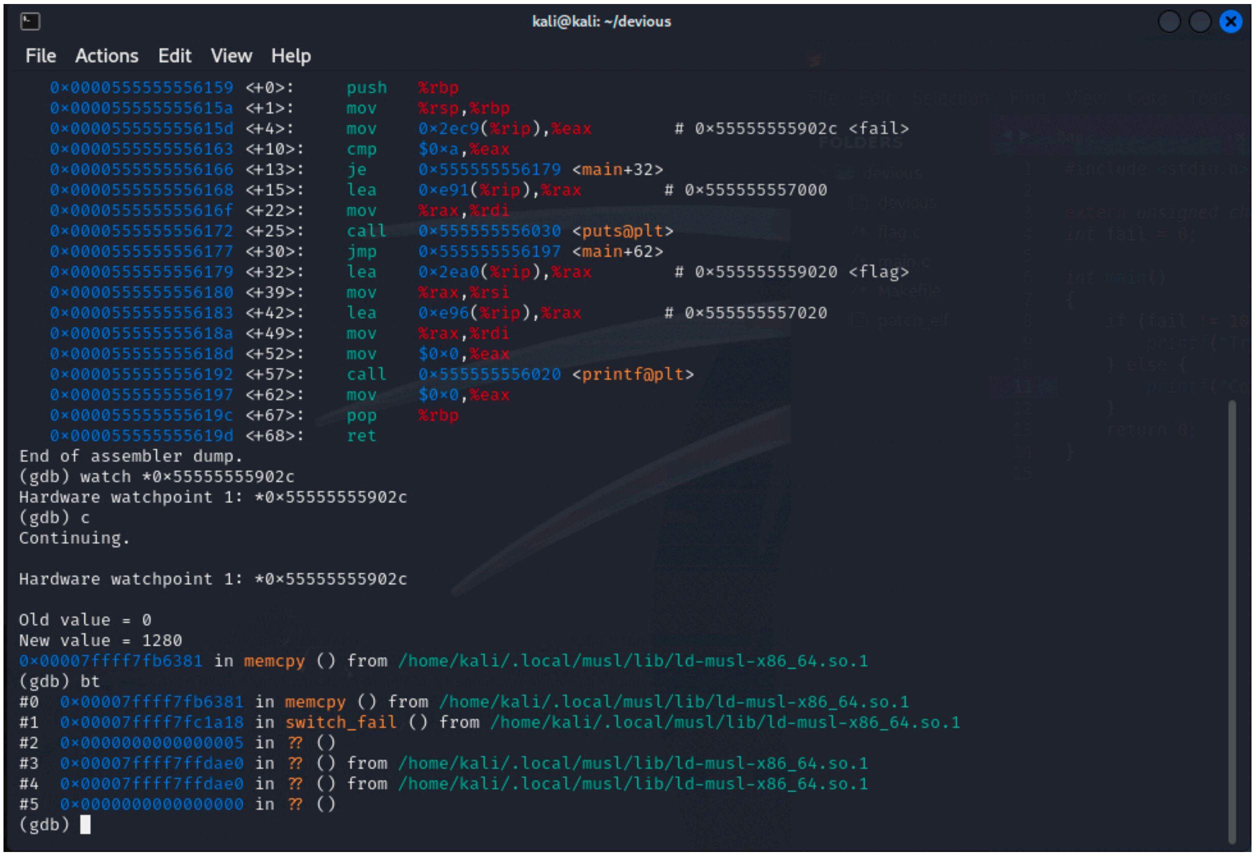
fail has been set to 1280. This is different from the value set to main.c, which is 0. Therefore, either patch_elf is doing this or something is running before main and doing it.

To verify if patch_elf is doing it or not, we can simply open devious in r2 and check what value fail contains.



```
┌──(kali㉿kali)-[~/devious]
└─$ r2 -e bin.cache=true devious
[0x00002060]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Finding and parsing C++ vtables (avrr)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information (aanr)
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x00002060]> is~fail
17   ────────── 0x0000502c GLOBAL OBJ    4        fail
[0x00002060]> s 0x502c
[0x0000502c]> px 4
- offset -   0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x0000502c  0000 0000                                ....
[0x0000502c]> 
```

fail is still 0 over here, so patch_elf is not setting fail. Something must be running before fail. To find this out, we need to set a watchpoint on the memory location of fail.
However, we have ASLR enabled and so the base offset of the program will vary from run to run. To workaround this, we can use the starti command in GDB to halt program execution at the first execution. Since the ELF file must already be loaded into memory at this point, we can simply disassemble main and get the address of fail.

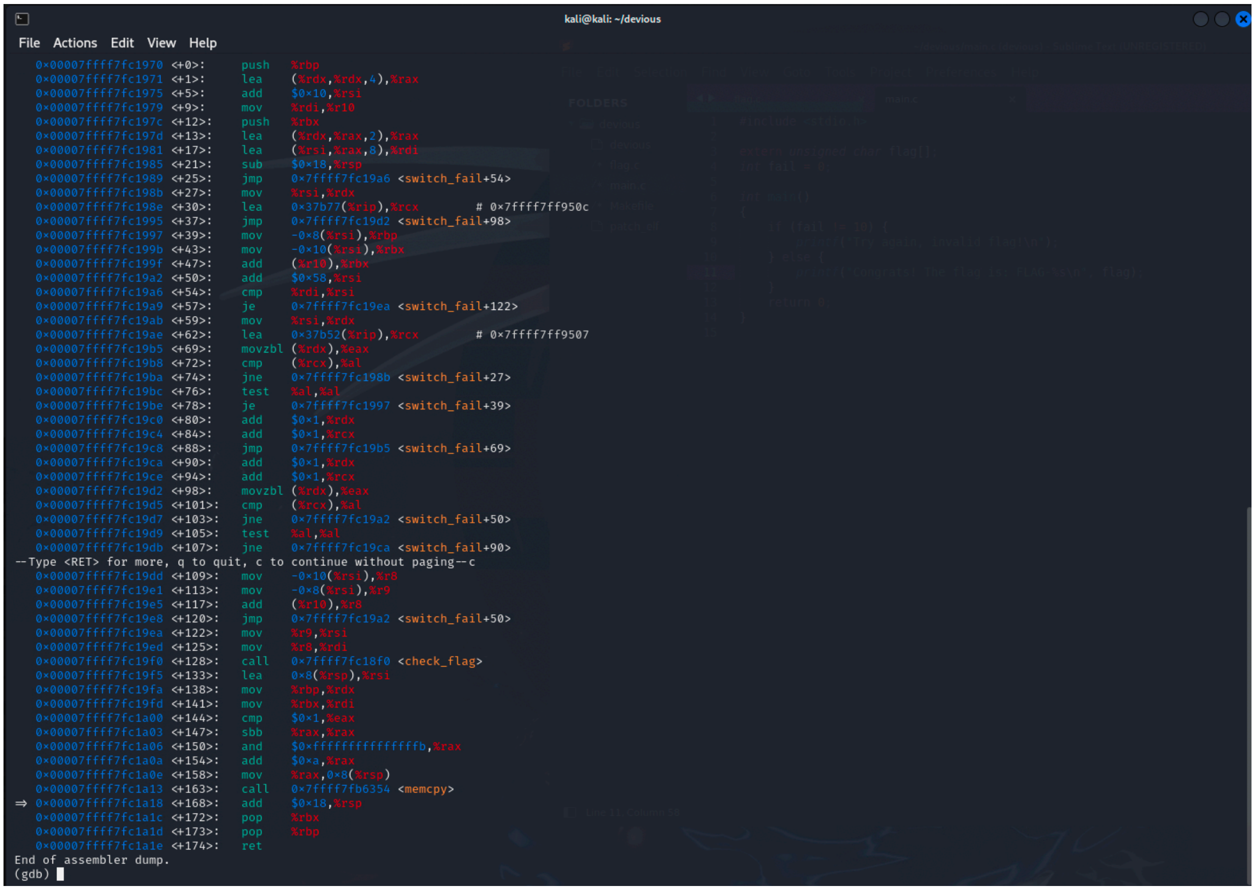


```
   0x0000555555556159 <+0>:     push   %rbp
   0x000055555555615a <+1>:     mov    %rsp,%rbp
   0x000055555555615d <+4>:     mov    0x2ec9(%rip),%eax        # 0x55555555902c <fail>
   0x0000555555556163 <+10>:    cmp    $0xa,%eax
   0x0000555555556166 <+13>:    je     0x555555556179 <main+32>
   0x0000555555556168 <+15>:    lea    0xe91(%rip),%rax        # 0x555555557000
   0x000055555555616f <+22>:    mov    %rax,%rdi
   0x0000555555556172 <+25>:    call   0x555555556030 <puts@plt>
   0x0000555555556177 <+30>:    jmp    0x555555556197 <main+62>
   0x0000555555556179 <+32>:    lea    0x2ea0(%rip),%rax        # 0x555555559020 <flag>
   0x0000555555556180 <+39>:    mov    %rax,%rsi
   0x0000555555556183 <+42>:    lea    0xe96(%rip),%rax        # 0x555555557020
   0x000055555555618a <+49>:    mov    %rax,%rdi
   0x000055555555618d <+52>:    mov    $0x0,%eax
   0x0000555555556192 <+57>:    call   0x555555556020 <printf@plt>
   0x0000555555556197 <+62>:    mov    $0x0,%eax
   0x000055555555619c <+67>:    pop    %rbp
   0x000055555555619d <+68>:    ret
End of assembler dump.
(gdb) watch *0x55555555902c
Hardware watchpoint 1: *0x55555555902c
(gdb) c
Continuing.

Hardware watchpoint 1: *0x55555555902c

Old value = 0
New value = 1280
0x00007ffff7fb6381 in memcpy () from /home/kali/.local/musl/lib/ld-musl-x86_64.so.1
(gdb) bt
#0  0x00007ffff7fb6381 in memcpy () from /home/kali/.local/musl/lib/ld-musl-x86_64.so.1
#1  0x00007ffff7fc1a18 in switch_fail () from /home/kali/.local/musl/lib/ld-musl-x86_64.so.1
#2  0x0000000000000005 in ?? ()
#3  0x00007ffff7ffdae0 in ?? () from /home/kali/.local/musl/lib/ld-musl-x86_64.so.1
#4  0x00007ffff7ffdae0 in ?? () from /home/kali/.local/musl/lib/ld-musl-x86_64.so.1
#5  0x0000000000000000 in ?? ()
(gdb) 
```

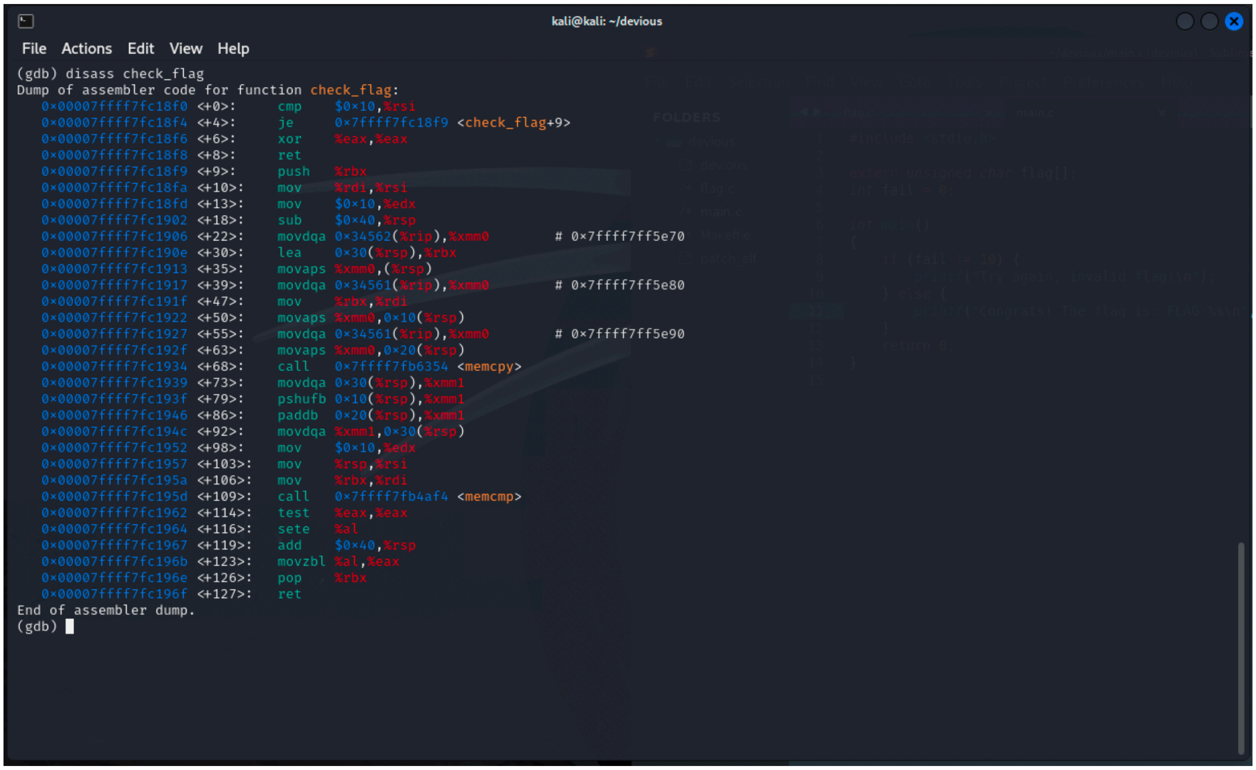
So the value of fail is changing from somewhere in ld-musl-x86_64.so.1, which is the dynamic linker. The memcpy happens in the switch_fail function. So we must check the disassembly of that function:



```
   0x00007ffff7fc1970 <+0>:     push   %rbp
   0x00007ffff7fc1971 <+1>:     lea    (%rdx,%rdx,4),%rax
   0x00007ffff7fc1975 <+5>:     add    $0x10,%rsi
   0x00007ffff7fc1979 <+9>:     mov    %rdi,%r10
   0x00007ffff7fc197c <+12>:    push   %rbx
   0x00007ffff7fc197d <+13>:    lea    (%rdx,%rax,2),%rax
   0x00007ffff7fc1981 <+17>:    lea    (%rsi,%rax,8),%rdi
   0x00007ffff7fc1985 <+21>:    sub    $0x18,%rsp
   0x00007ffff7fc1989 <+25>:    jmp    0x7ffff7fc19a6 <switch_fail+54>
   0x00007ffff7fc198b <+27>:    mov    %rsi,%rdx
   0x00007ffff7fc198e <+30>:    lea    0x37b77(%rip),%rcx        # 0x7ffff7ff950c
   0x00007ffff7fc1995 <+37>:    jmp    0x7ffff7fc19d2 <switch_fail+98>
   0x00007ffff7fc1997 <+39>:    mov    -0x8(%rsi),%rbp
   0x00007ffff7fc199b <+43>:    mov    -0x10(%rsi),%rbx
   0x00007ffff7fc199f <+47>:    add    (%r10),%rbx
   0x00007ffff7fc19a2 <+50>:    add    $0x58,%rsi
   0x00007ffff7fc19a6 <+54>:    cmp    %rdi,%rsi
   0x00007ffff7fc19a9 <+57>:    je     0x7ffff7fc19ea <switch_fail+122>
   0x00007ffff7fc19ab <+59>:    mov    %rsi,%rdx
   0x00007ffff7fc19ae <+62>:    lea    0x37b52(%rip),%rcx        # 0x7ffff7ff9507
   0x00007ffff7fc19b5 <+69>:    movzbl (%rdx),%eax
   0x00007ffff7fc19b8 <+72>:    cmp    (%rcx),%al
   0x00007ffff7fc19ba <+74>:    jne    0x7ffff7fc198b <switch_fail+27>
   0x00007ffff7fc19bc <+76>:    test   %al,%al
   0x00007ffff7fc19be <+78>:    je     0x7ffff7fc1997 <switch_fail+39>
   0x00007ffff7fc19c0 <+80>:    add    $0x1,%rdx
   0x00007ffff7fc19c4 <+84>:    add    $0x1,%rcx
   0x00007ffff7fc19c8 <+88>:    jmp    0x7ffff7fc19b5 <switch_fail+69>
   0x00007ffff7fc19ca <+90>:    add    $0x1,%rdx
   0x00007ffff7fc19ce <+94>:    add    $0x1,%rcx
   0x00007ffff7fc19d2 <+98>:    movzbl (%rdx),%eax
   0x00007ffff7fc19d5 <+101>:   cmp    (%rcx),%al
   0x00007ffff7fc19d7 <+103>:   jne    0x7ffff7fc19a2 <switch_fail+50>
   0x00007ffff7fc19d9 <+105>:   test   %al,%al
   0x00007ffff7fc19db <+107>:   jne    0x7ffff7fc19ca <switch_fail+90>
--Type <RET> for more, q to quit, c to continue without paging--c
   0x00007ffff7fc19dd <+109>:   mov    -0x10(%rsi),%r8
   0x00007ffff7fc19e1 <+113>:   mov    -0x8(%rsi),%r9
   0x00007ffff7fc19e5 <+117>:   add    (%r10),%r8
   0x00007ffff7fc19e8 <+120>:   jmp    0x7ffff7fc19a2 <switch_fail+50>
   0x00007ffff7fc19ea <+122>:   mov    %r9,%rsi
   0x00007ffff7fc19ed <+125>:   mov    %r8,%rdi
   0x00007ffff7fc19f0 <+128>:   call   0x7ffff7fc18f0 <check_flag>
   0x00007ffff7fc19f5 <+133>:   lea    0x8(%rsp),%rsi
   0x00007ffff7fc19fa <+138>:   mov    %rbp,%rdx
   0x00007ffff7fc19fd <+141>:   mov    %rbx,%rdi
   0x00007ffff7fc1a00 <+144>:   cmp    $0x1,%eax
   0x00007ffff7fc1a03 <+147>:   sbb    %rax,%rax
   0x00007ffff7fc1a06 <+150>:   and    $0xfffffffffffffffb,%rax
   0x00007ffff7fc1a0a <+154>:   add    $0xa,%rax
   0x00007ffff7fc1a0e <+158>:   mov    %rax,0x8(%rsp)
   0x00007ffff7fc1a13 <+163>:   call   0x7ffff7fb6354 <memcpy>
⇒  0x00007ffff7fc1a18 <+168>:   add    $0x18,%rsp
   0x00007ffff7fc1a1c <+172>:   pop    %rbx
   0x00007ffff7fc1a1d <+173>:   pop    %rbp
   0x00007ffff7fc1a1e <+174>:   ret
End of assembler dump.
(gdb) 
```

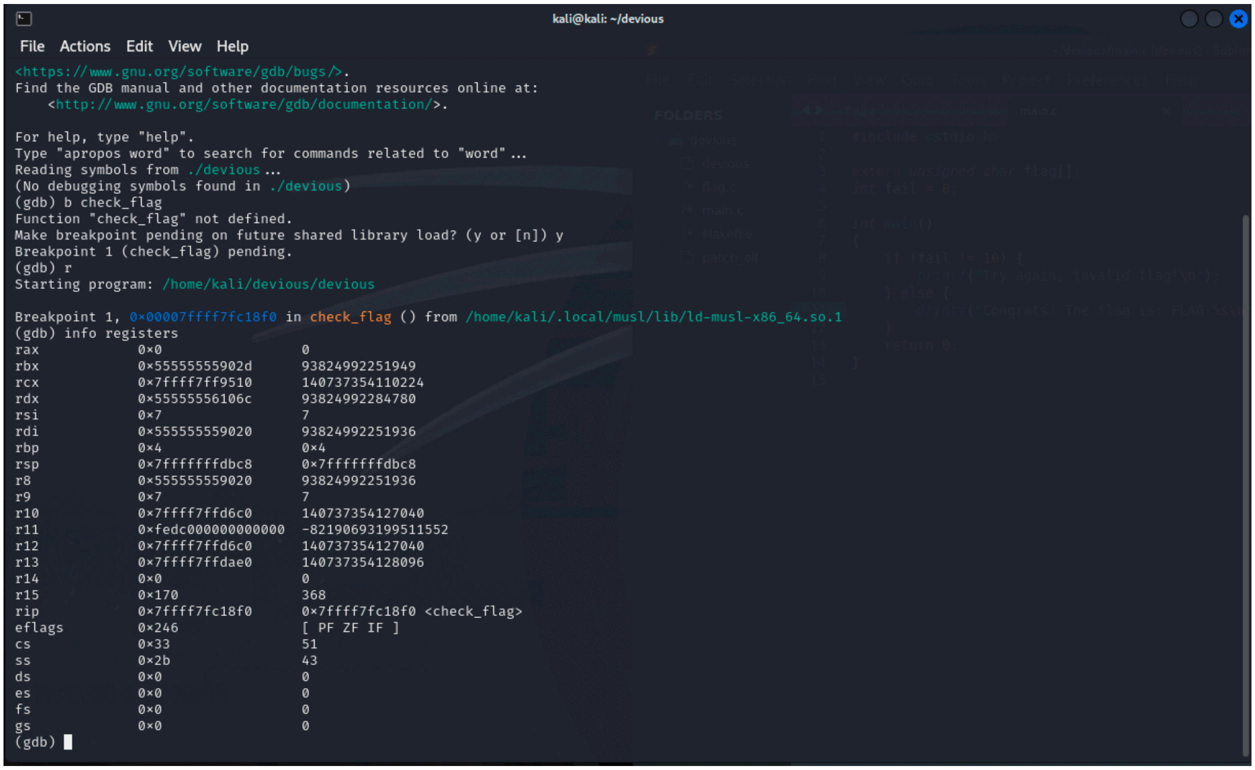
A few instructions before the memcpy call, there is a call to a function called check_flag. Presumably, this has something to do with the flag checking logic. Between the call to check_flag and call to memcpy, there is a “cmp $0x1, %eax”, which means probably check_flag returns 1 in some case (so perhaps boolean return value, which checks out with the name)

At any rate, if we check what is inside check_flag:



```
(gdb) disass check_flag
Dump of assembler code for function check_flag:
   0x00007ffff7fc18f0 <+0>:     cmp    $0x10,%rsi
   0x00007ffff7fc18f4 <+4>:     je     0x7ffff7fc18f9 <check_flag+9>
   0x00007ffff7fc18f6 <+6>:     xor    %eax,%eax
   0x00007ffff7fc18f8 <+8>:     ret
   0x00007ffff7fc18f9 <+9>:     push   %rbx
   0x00007ffff7fc18fa <+10>:    mov    %rdi,%rsi
   0x00007ffff7fc18fd <+13>:    mov    $0x10,%edx
   0x00007ffff7fc1902 <+18>:    sub    $0x40,%rsp
   0x00007ffff7fc1906 <+22>:    movdqa 0x34562(%rip),%xmm0        # 0x7ffff7ff5e70
   0x00007ffff7fc190e <+30>:    lea    0x30(%rsp),%rbx
   0x00007ffff7fc1913 <+35>:    movaps %xmm0,(%rsp)
   0x00007ffff7fc1917 <+39>:    movdqa 0x34561(%rip),%xmm0        # 0x7ffff7ff5e80
   0x00007ffff7fc191f <+47>:    mov    %rbx,%rdi
   0x00007ffff7fc1922 <+50>:    movaps %xmm0,0x10(%rsp)
   0x00007ffff7fc1927 <+55>:    movdqa 0x34561(%rip),%xmm0        # 0x7ffff7ff5e90
   0x00007ffff7fc192f <+63>:    movaps %xmm0,0x20(%rsp)
   0x00007ffff7fc1934 <+68>:    call   0x7ffff7fb6354 <memcpy>
   0x00007ffff7fc1939 <+73>:    movdqa 0x30(%rsp),%xmm1
   0x00007ffff7fc193f <+79>:    pshufb 0x10(%rsp),%xmm1
   0x00007ffff7fc1946 <+86>:    paddb  0x20(%rsp),%xmm1
   0x00007ffff7fc194c <+92>:    movdqa %xmm1,0x30(%rsp)
   0x00007ffff7fc1952 <+98>:    mov    $0x10,%edx
   0x00007ffff7fc1957 <+103>:   mov    %rsp,%rsi
   0x00007ffff7fc195a <+106>:   mov    %rbx,%rdi
   0x00007ffff7fc195d <+109>:   call   0x7ffff7fb4af4 <memcmp>
   0x00007ffff7fc1962 <+114>:   test   %eax,%eax
   0x00007ffff7fc1964 <+116>:   sete   %al
   0x00007ffff7fc1967 <+119>:   add    $0x40,%rsp
   0x00007ffff7fc196b <+123>:   movzbl %al,%eax
   0x00007ffff7fc196e <+126>:   pop    %rbx
   0x00007ffff7fc196f <+127>:   ret
End of assembler dump.
(gdb)
```

There are some SIMD instructions (movaps, movdqa, pshufb, paddb). If we want to find out about the arguments of check_flag, we must first set a breakpoint to it and then check the registers %rdi, %rsi, %rdx etc.

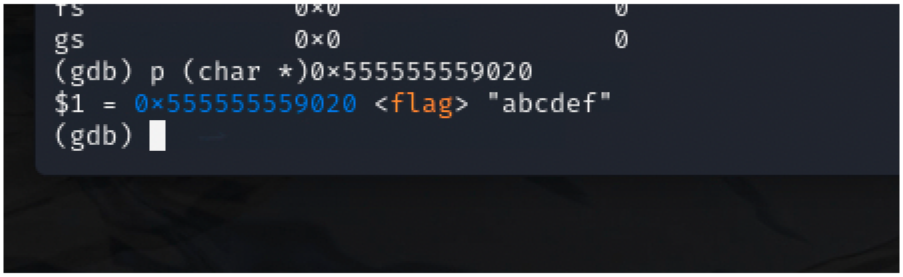


```
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./devious...
(No debugging symbols found in ./devious)
(gdb) b check_flag
Function "check_flag" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (check_flag) pending.
(gdb) r
Starting program: /home/kali/devious/devious

Breakpoint 1, 0x00007ffff7fc18f0 in check_flag () from /home/kali/.local/musl/lib/ld-musl-x86_64.so.1
(gdb) info registers
rax            0x0                 0
rbx            0x55555555902d      93824992251949
rcx            0x7ffff7ff9510      140737354110224
rdx            0x55555556106c      93824992284780
rsi            0x7                 7
rdi            0x555555559020      93824992251936
rbp            0x4                 0x4
rsp            0x7fffffffdbc8      0x7fffffffdbc8
r8             0x555555559020      93824992251936
r9             0x7                 7
r10            0x7ffff7ffd6c0      140737354127040
r11            0xfedc000000000000  -82190693199511552
r12            0x7ffff7ffd6c0      140737354127040
r13            0x7ffff7ffdae0      140737354128096
r14            0x0                 0
r15            0x170               368
rip            0x7ffff7fc18f0      0x7ffff7fc18f0 <check_flag>
eflags         0x246               [ PF ZF IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) 
```

%rsi is set to 7, which is 6 (length of flag) + 1 (null terminator) probably. %rdi is set to a memory address. If we try to print the value present there:



```
fs             0x0                 0
gs             0x0                 0
(gdb) p (char *)0x555555559020
$1 = 0x555555559020 <flag> "abcdef"
(gdb) 
```

That's our flag. So evidently the flag is being checked here.

This function is fairly straight-forward to reverse. It involves a permutation of the characters plus a shift for each position. The bytes obtained by this transformation are checked against expected bytes. If they match, check_flag returns 1 otherwise returns 0. Also note that the function straight away returns 0 if %rsi is not 16, meaning that we have a 15 letter string as flag.

The expected value of flag is “W@2d0I$Proud0fU”, which is the flag to submit.