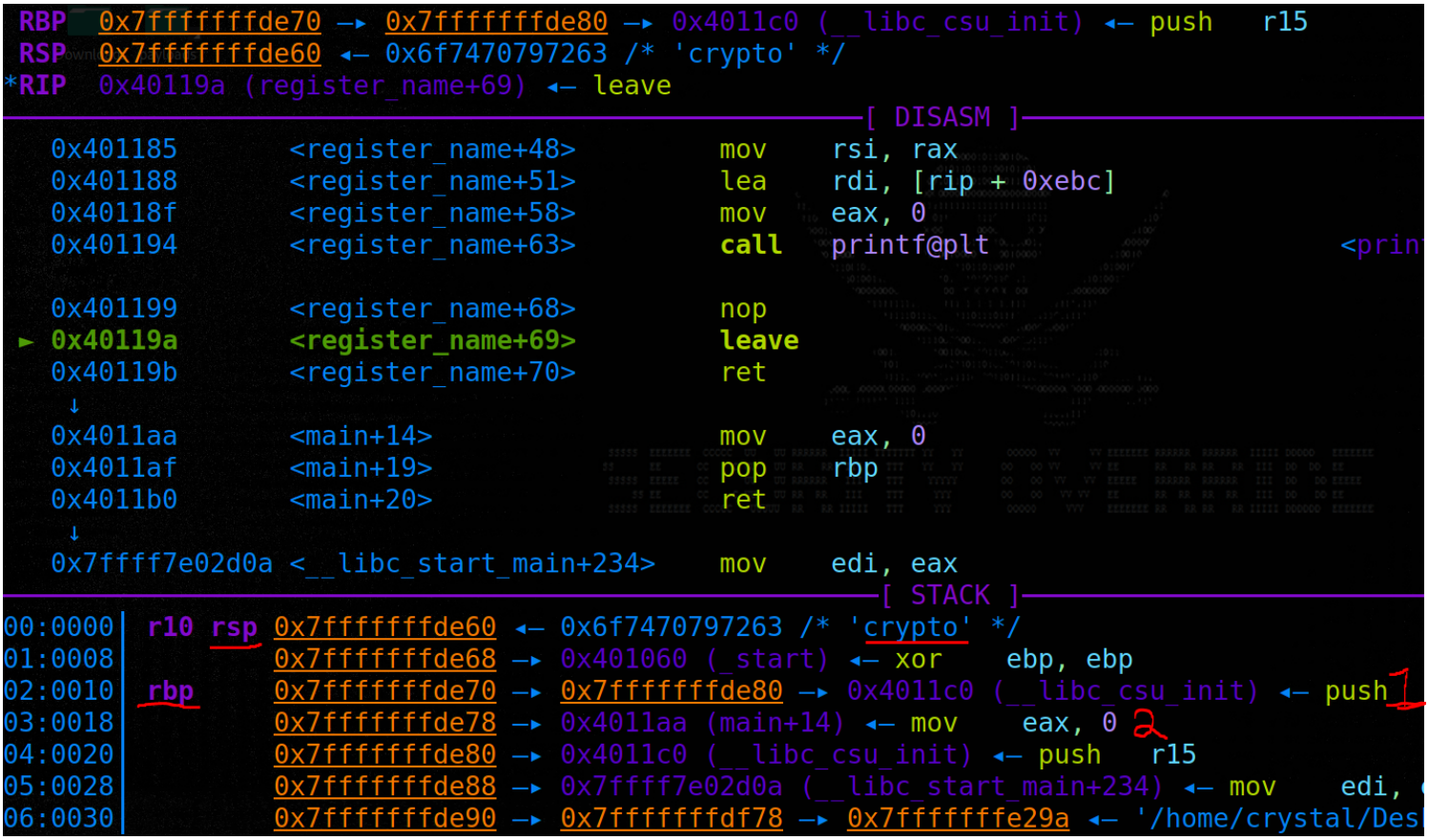
Background: https://zhu45.org/posts/2017/Jul/30/understanding-how-function-call-works/#calling-a-function

First without overflow; when we get to the end of the function, the RSP points to our input ("crypto") on top of the stack and RBP points to the base of the stack frame, after that we have two important items on the stack:
1) the frame pointer (previous RBP from the calling function, which will be used to restore the stack layout upon returning)
2) return address (where to resume from the calling function) - below the RBP (in the screenshot, but technically above)



We hit next and the leave instruction (reverse of "enter" that occurs at beginning of function) resets the stack like:

LEAVE
==========
mov %rbp, %rsp # Sets RBP=RSP, both to same value (0x4011c0 libc_csu_init)
pop %rbp # Pops the original RBP , that was saved to the stack at start of function

Note, because this happens in a single instruction we never actually see RBP = RSP. By the time we get to the next instruction, the value from the top of the stack has been popped into the RBP.

Here we can see that (1) from the last screenshot (the original RBP) has been popped back to the RBP and the address to return to in the calling function is in the RSP (2)

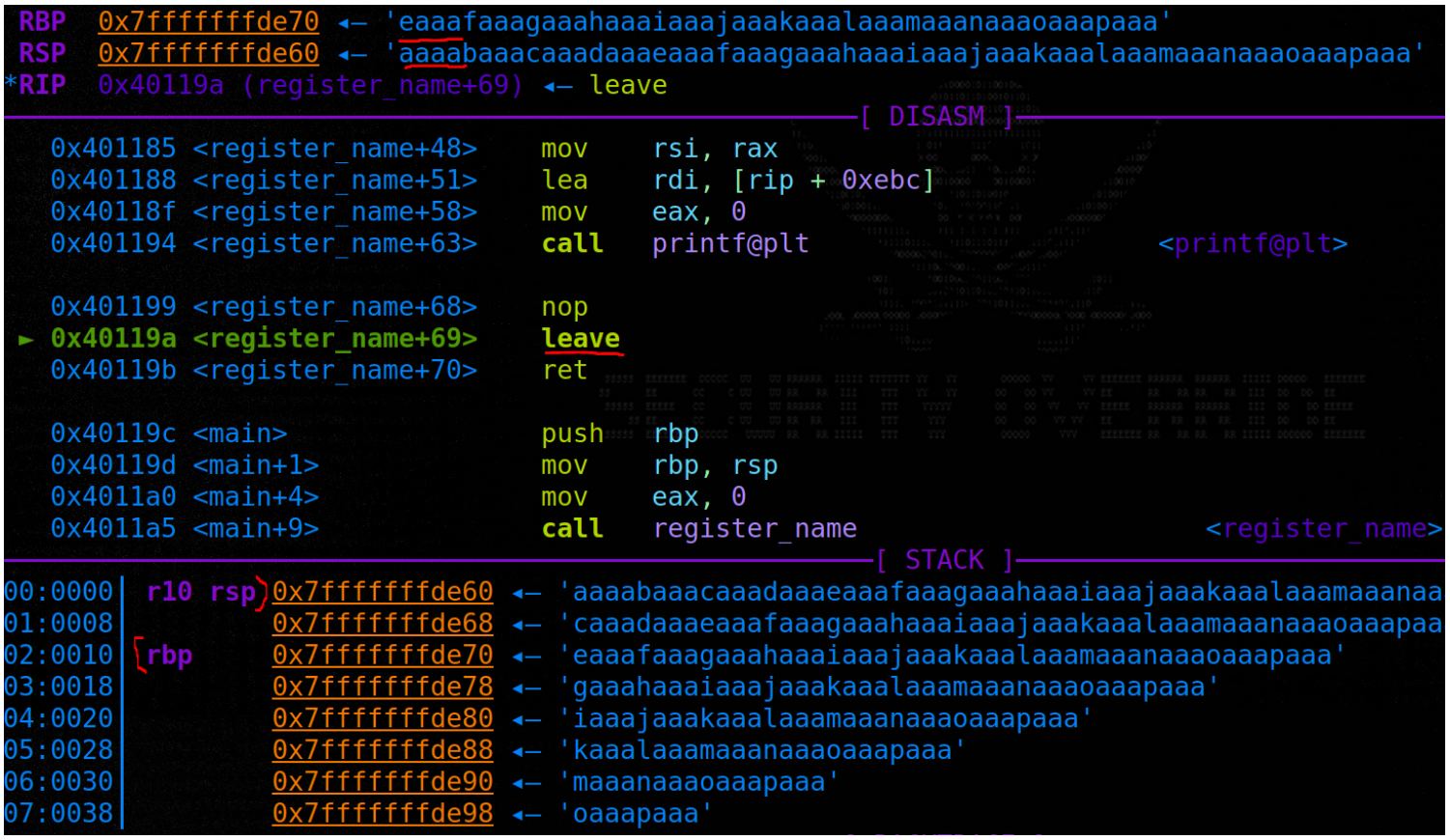
```
*RBP  0x7fffffffde80 —▸ 0x4011c0 (__libc_csu_init) ◂— push   r15 1
*RSP  0x7fffffffde78 —▸ 0x4011aa (main+14) ◂— mov    eax, 0 2
*RIP  0x40119b (register_name+70) ◂— ret
[ DISASM ]
   0x401188       <register_name+51>         lea    rdi, [rip + 0xebc]
   0x40118f       <register_name+58>         mov    eax, 0
   0x401194       <register_name+63>         call   printf@plt                  <prin

   0x401199       <register_name+68>         nop
   0x40119a       <register_name+69>         leave
 ► 0x40119b       <register_name+70>         ret                                <0x4011
    ↓
   0x4011aa       <main+14>                  mov    eax, 0
   0x4011af       <main+19>                  pop    rbp
   0x4011b0       <main+20>                  ret
    ↓
   0x7ffff7e02d0a <__libc_start_main+234>    mov    edi, eax
   0x7ffff7e02d0c <__libc_start_main+236>    call   exit                <exit>
[ STACK ]
00:0000| rsp 0x7fffffffde78 —▸ 0x4011aa (main+14) ◂— mov    eax, 0
01:0008| rbp 0x7fffffffde80 —▸ 0x4011c0 (__libc_csu_init) ◂— push   r15
02:0010|     0x7fffffffde88 —▸ 0x7ffff7e02d0a (__libc_start_main+234) ◂— mov    edi, eax
03:0018|     0x7fffffffde90 —▸ 0x7fffffffdf78 —▸ 0x7fffffffe29a ◂— '/home/crystal/Desktop
```

Now everything is in place to safely return to where we left off in main(). The "ret" instruction will pop the next address (0x4011aa) off the stack into RIP and execute:

```
   0x40119a       <register_name+69>         leave
 ► 0x40119b       <register_name+70>         ret
    ↓
   0x4011aa       <main+14>                  mov    eax, 0
   0x4011af       <main+19>                  pop    rbp
```

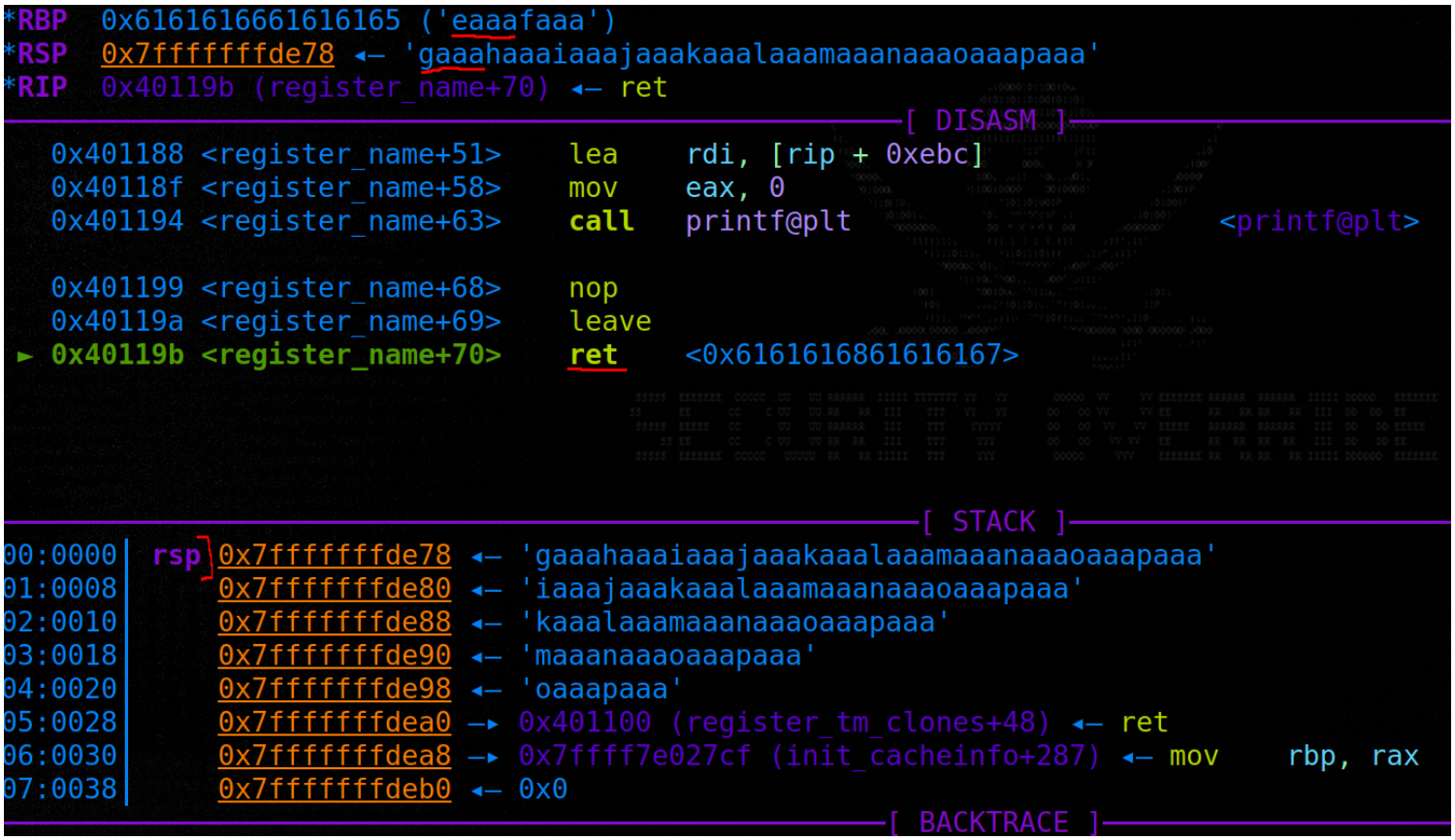
Now if we repeat for buffer overflow:



We've overwritten everything, on the stack including our saved RBP and return pointer from the calling function. So when we LEAVE this time:

LEAVE
==========
mov %rbp, %rsp # Sets RBP=RSP, both to same value (in our case "eaaafaaa...")
pop %rbp # Pops what *should* be original RBP value (but instead is "eaaafaaa...") to the RBP

Again, we don't see this properly because it happens in one instruction. However, we can see "eaaafaaa" in RBP and the RSP contains the value right after it "gaaahaaa", which would be popped to the RIP and executed (if it were a valid address):


*RBP 0x6161616661616165 ('eaaafaaa')
*RSP 0x7fffffffde78 ◂— 'gaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
*RIP 0x40119b (register_name+70) ◂— ret
[DISASM]
0x401188 <register_name+51> lea rdi, [rip + 0xebc]
0x40118f <register_name+58> mov eax, 0
0x401194 <register_name+63> call printf@plt <printf@plt>
0x401199 <register_name+68> nop
0x40119a <register_name+69> leave
► 0x40119b <register_name+70> ret <0x6161616861616167>
[STACK]
00:0000 rsp 0x7fffffffde78 ◂— 'gaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
01:0008 0x7fffffffde80 ◂— 'iaaajaaakaaalaaamaaanaaaoaaapaaa'
02:0010 0x7fffffffde88 ◂— 'kaaalaaamaaanaaaoaaapaaa'
03:0018 0x7fffffffde90 ◂— 'maaanaaaoaaapaaa'
04:0020 0x7fffffffde98 ◂— 'oaaapaaa'
05:0028 0x7fffffffdea0 —▸ 0x401100 (register_tm_clones+48) ◂— ret
06:0030 0x7fffffffdea8 —▸ 0x7ffff7e027cf (init_cacheinfo+287) ◂— mov rbp, rax
07:0038 0x7fffffffdeb0 ◂— 0x0
[BACKTRACE]