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)

```
0x7fffffffde70 \rightarrow 0x7fffffffde80 \rightarrow 0x4011c0 (
                                                                                     r15
                                                         libc csu init) ← push
RSP
      0x7fffffffde60 \( \text{--} 0x6f7470797263 /* 'crypto' */
RIP
     0x40119a (register name+69) ← leave
  0x401185
                   <register name+48>
                                                mov
                                                       rsi, rax
  0x401188
                   <register name+51>
                                                lea
                                                       rdi, [rip + 0xebc]
  0x40118f
                   <register name+58>
                                                mov
                                                       eax, 0
                   <register name+63>
                                                       printf@plt
  0x401194
                                                call
  0x401199
                   <register name+68>
                                                nop
  0x40119a
                   <register name+69>
                                                leave
  0x40119b
                   <register name+70>
                                                ret
  0x4011aa
                   < main + 14 >
                                                mov
                                                       eax, 0
  0x4011af
                   < main + 19 >
                                                       rbp
                                                pop
  0x4011b0
                   <main+20>
                                                ret
  0x7ffff7e02d0a < libc start main+234>
                                                       edi, eax
                                                mov
                                                           ·[ STACK ]
0000:00
         r10 rsp <u>0x7fffffffde60</u> ← 0x6f7470797263 /* 'crypto'
91:0008
                                                                   ebp, ebp
                  0x7ffffffde70 \rightarrow 0x7ffffffde80 \rightarrow 0x4011c0
02:0010
                                                                  ( libc csu init) ← push
03:0018
                  0x7fffffffde78 → 0x4011aa (main+14) ← mov
04:0020
                  0x7fffffffde80 -> 0x4011c0 (
                                                                 push
                                 → 0x7fffff7e02d0a (_libc_start_main+234) ← mov
95:0028
                                     0x7fffffffffffe29a ← '/home/crystal/Des
06:0030
```

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)

```
r15 👤
     0x7fffffffde78 \rightarrow 0x4011aa (main+14) \leftarrow mov
                                                eax, 0 👕
RIP
     0x40119b (register name+70) ← ret
                                               rdi, [rip + 0xebc]
  0x401188
                <register_name+51>
                                         lea
  0x40118f
                                               eax, 0
                                        mov
                <register_name+63>
  0x401194
                                         call
                                               printf@plt
  0x401199
                                         nop
                <register_name+69>
  0x40119a
                                         leave
  0x40119b
                <register_name+70>
                                         ret
  0x4011aa
                <main+14>
                                        mov
                                               eax, 0
  0x4011af
                <main+19>
                                               rbp
                                         pop
                <main+20>
  0x4011b0
                                         ret
  0x7ffff7e02d0a < libc start main+234>
                                        mov
                                               edi, eax
  0x7ffff7e02d0c < libc start main+236>
                                         call
                                               exit
                                                  -[ STACK ]-
00:000
        rsp 0x7ffffffde78 → 0x4011aa (main+14) ← mov
                                                      eax, 0
                         → 0x4011c0 (__libc_csu_init) <- push</pre>
01:0008
                                                              r15
02:0010
           0x7fffffffde88
                         edi, eax
                         03:0018
           0x7fffffffde90
```

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:

Now if we repeat for buffer overflow:

```
eaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaoaaapaaa
     RIP
     0x40119a (register name+69) ← leave
  0x401185 <register name+48>
                                mov
                                       rsi, rax
                                       rdi, [rip + 0xebc]
eax, 0
  0x401188 <register name+51>
                                lea
  0x40118f <register_name+58>
                                mov
                                call
  0x401194 <register name+63>
                                       printf@plt
  0x401199 <register name+68>
                                nop
  0x40119a <register name+69>
                                leave
  0x40119b <register name+70>
                                ret
  0x40119c <main>
                                push
                                       rbp
  0 \times 40119d < main+1>
                                mov
                                       rbp, rsp
  0x4011a0 <main+4>
                                       eax, 0
                                mov
  0x4011a5 <main+9>
                                call
                                       register name
                                                     ·[ STACK ]-
        r10 rsp)0x7ffffffde60 ← 'aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaa
00:000
91:0008
                              'caaadaaaeaaafaaaqaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaa
                <mark>0x7fffffffde70</mark> ← 'eaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
02:0010
                                 'qaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
93:0018
                                 'iaaajaaakaaalaaamaaanaaaoaaapaaa'
94:0020
                                 'kaaalaaamaaanaaaoaaapaaa'
05:0028
06:0030
                                 'maaanaaaoaaapaaa'
97:0038
```

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):

```
0x6161616661616165 ('eaaafaaa')
0x7ffffffde78 ← 'gaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaa'
RIP
     0x40119b (register name+70) ← ret
                                         rdi, [rip + 0xebc]
  0x401188 <register_name+51>
                                  lea
  0x40118f <register name+58>
                                  mov
                                         eax, 0
  0x401194 <register name+63>
                                  call
                                         printf@plt
  0x401199 <register name+68>
                                  nop
  0x40119a <register name+69>
                                  leave
► 0x40119b <register name+70>
                                         <0x6161616861616167>
                                  ret
                                                       -[ STACK ]-
        rsp <u>0x7ffffffde78</u> ← 'gaaahaaaiaaajaaakaaalaaamaaanaaoaaapaaa'
00:0000
             01:0008
02:0010
             0x7ffffffde88 <- 'kaaalaaamaaanaaaoaaapaaa'
             0x7fffffffde90 ← 'maaanaaaoaaapaaa'
03:0018
             0x7fffffffde98 ← 'oaaapaaa'
04:0020
             <u>0x7ffffffdea0</u> → 0x401100 (register_tm_clones+48) ← ret
05:0028
             <u>0x7ffffffdea8</u> → 0x7fffff7e027cf (init cacheinfo+287) ← mov
06:0030
                                                                              rbp, rax
07:0038
             0x7fffffffdeb0 ← 0x0
                                                     · F BACKTRACE 1-
```