Return Oriented Programming

When shellcode is not enough!

Marco Bonelli — @mebeim
March 25, 2022

OVERVIEW

- What's Return Oriented Programming.
- Calling function with arbitrary arguments.
- ROP gadgets and how to find/use them.
- Building a ROP chain to call multiple functions.
- 32bit vs 64bit ROP chain.
- Calling library functions using ROP (ret2libc).
- The "magic gadget" exploit.
- More than just calling functions...

What's ROP?

We all know that overwriting a saved return address on the stack can be very interesting… we can make the program jump wherever we want, but that's it more or less.

- ➢ How can we call a function *passing arguments*?
- ➢ And what about calling *multiple* functions one after another?

Here's where Return Oriented Programming comes in handy!

What's ROP?

0xffff00xx	<... locals of func_3>
0xffff0004	<saved $ebp>
0xffff0008	<saved return addr>
0xffff000c	<func_3's arg1>
0xffff0010	<func_3's arg2>
0xffff0014	<func_3's arg3>
0xffff00xx	<... locals of func_2>
0xffff001c	<saved $ebp>
0xffff0020	<saved return addr>
0xffff0024	<func_2's arg1>
0xffff0028	<func_2's arg2>
0xffff00xx	<... locals of func_1>
0xffff0030	<saved $ebp>
0xffff0034	<saved return addr>
0xffff0038	<func_1's arg1>
0xffff003c	<... locals of main>

If we're careful enough about how we place stuff on the stack, we can *simulate fake stack frames* and chain the execution of multiple arbitrary functions with arbitrary arguments.

Calling a function

Let's call a function with arbitrary arguments:

```
void foo(int arg1, int arg2,
         int arg3            ) {
    printf("arg1 is %x\n", arg1);
    printf("arg2 is %x\n", arg1);
    printf("arg3 is %x\n", arg1);
}

int vuln(void) {
    char buf[4];
➡   read(0, buf, 200);
    return 0;
}
```

We want to call: **`foo(1, 2, 3)`**

Stack of `vuln` **before** `read()`:

```
esp  0xffffd000 → 0x00000000 <buf[0..3]>
ebp  0xffffd004 → 0xffffd068 <saved ebp>
     0xffffd008 → 0x080400de <saved return addr>
     0xffffd00c → ...
     0xffffd010 → ...
     0xffffd014 → ...
     0xffffd018 → ...
```

Calling a function

Let's call a function with arbitrary arguments:

```c
void foo(int arg1, int arg2,
         int arg3              ) {
    printf("arg1 is %x\n", arg1);
    printf("arg2 is %x\n", arg1);
    printf("arg3 is %x\n", arg1);
}

int vuln(void) {
    char buf[4];
    read(0, buf, 200);
  ➡ return 0;
}
```

Input to read():

```
AAAABBBB<foo>CCCC\x01\x00\x00\x00\x02\x00\x00\x00
\x03\x00\x00\x00DDDDEEEEFFFFGGGG...
```

Stack of vuln **after** read():

```
esp   0xffffd000 → 0x41414141  <buf[0..3]>
ebp   0xffffd004 → 0x42424242  <saved ebp>
      0xffffd008 → 0x<foo>     <saved return addr>
      0xffffd00c → 0x43434343
      0xffffd010 → 0x1
      0xffffd014 → 0x2
      0xffffd018 → 0x3
```

Calling a function

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
 => 0x56555648: leave (mov esp, ebp; pop ebp)
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret
```

Stack

```
esp 0xffffd000 → 0x41414141 <buf[0..3]>
ebp 0xffffd004 → 0x42424242 <saved ebp>
    0xffffd008 → 0x565555c0 <saved ret addr>
    0xffffd00c → 0x43434343
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x44444444
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers

```
eip = 0x56555648
esp = 0xffffd000
ebp = 0xffffd004
```

Calling a function

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
 => 0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret
```

Stack

```
    0xffffd000 → 0x41414141 <buf[0..3]>
    0xffffd004 → 0x42424242 <saved ebp>
esp 0xffffd008 → 0x565555c0 <saved ret addr>
    0xffffd00c → 0x43434343
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x44444444
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers

```
eip = 0x56555649
esp = 0xffffd008
ebp = 0x42424242
```

Calling a function

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
 => 0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret
```

Stack

```
    0xffffd000 → 0x41414141
    0xffffd004 → 0x42424242
    0xffffd008 → 0x565555c0
esp 0xffffd00c → 0x43434343
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x44444444
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers

```
eip = 0x565555c0
esp = 0xffffd00c
ebp = 0x42424242
```

Calling a function

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
 => 0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret
```

Stack

```
    0xffffd000 → 0x41414141
    0xffffd004 → 0x42424242
esp 0xffffd008 → 0x42424242 <saved ebp>
    0xffffd00c → 0x43434343
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x44444444
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers

```
eip = 0x565555c1
esp = 0xffffd008
ebp = 0x42424242
```

Calling a function

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
 =>   ...
    0x56555615: leave
    0x56555616: ret
```

Stack

```
    0xffffd000 → 0x41414141
    0xffffd004 → 0x42424242
esp 0xffffd008 → 0x42424242 <saved ebp>
    0xffffd00c → 0x43434343
    0xffffd010 → 0x1        <arg 1>
    0xffffd014 → 0x2        <arg 2>
    0xffffd018 → 0x3        <arg 3>
    0xffffd01c → 0x44444444
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers / vars

```
eip = 0x56555xxx      arg1 [ebp+0x08]: 0x1
esp = 0xffffd008      arg2 [ebp+0x0c]: 0x2
ebp = 0xffffd008      arg3 [ebp+0x10]: 0x3
```

Calling a function

CYBER CHALLENGE
CyberChallenge.IT

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push    ebp
    0x565555c1: mov     ebp, esp
      ...
 => 0x56555615: leave  (mov esp, ebp; pop ebp)
    0x56555616: ret
```

Stack

```
esp 0xffffd000 → ...
    0xffffd004 → ...
ebp 0xffffd008 → 0x42424242 <saved ebp>
    0xffffd00c → 0x43434343
    0xffffd010 → 0x1        <arg 1>
    0xffffd014 → 0x2        <arg 2>
    0xffffd018 → 0x3        <arg 3>
    0xffffd020 → 0x44444444
    0xffffd024 → 0x45454545
    0xffffd028 → 0x46464646
```

Registers / vars

```
eip = 0x56555615      arg1 [ebp+0x08]: 0x1
esp = 0xffffd000      arg2 [ebp+0x0c]: 0x2
ebp = 0xffffd008      arg3 [ebp+0x10]: 0x3
```

Calling a function

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
 => 0x56555616: ret
                  ↓
    0x43434343: ???  SIGSEGV
```

Stack

```
    0xffffd000 → ...
    0xffffd004 → ...
    0xffffd008 → 0x42424242 <saved ebp>
esp 0xffffd00c → 0x43434343
    0xffffd010 → 0x1        <arg 1>
    0xffffd014 → 0x2        <arg 2>
    0xffffd018 → 0x3        <arg 3>
    0xffffd01c → 0x44444444
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers

```
eip = 0x56555616
esp = 0xffffd00c
ebp = 0x42424242
```

CYBER CHALLENGE
CyberChallenge.IT

Calling more than one function?

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
 => 0x56555616: ret

<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

Stack

```
    0xffffd000 → ...
    0xffffd004 → ...
    0xffffd008 → 0x42424242 <saved ebp>
esp 0xffffd00c → ???   What can go here
    0xffffd010 → 0x1   to call  bar(4, 5, 6)?
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x44444444   And here?
    0xffffd020 → 0x45454545
    0xffffd024 → 0x46464646
```

Registers

```
eip = 0x56555616
esp = 0xffffd00c
ebp = 0x42424242
```

Calling more than one function?

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
 => 0x56555616: ret

<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

Stack

```
    0xffffd000 → ...
    0xffffd004 → ...
    0xffffd008 → 0x42424242 <saved ebp>
esp 0xffffd00c → 0x565557b2
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x4
    0xffffd020 → 0x5
    0xffffd024 → 0x6
```

Does this work?

Registers

```
eip = 0x56555616
esp = 0xffffd00c
ebp = 0x42424242
```

Calling more than one function?

CYBER CHALLENGE
CyberChallenge.IT

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<bar>:
 => 0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

Stack

```
    0xffffd000 → ...
    0xffffd004 → ...
    0xffffd008 → 0x42424242
    0xffffd00c → 0x565557b2
esp 0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x4
    0xffffd020 → 0x5
    0xffffd024 → 0x6
```

Registers

```
eip = 0x565557b2
esp = 0xffffd010
ebp = 0x42424242
```

Calling more than one function?

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push    ebp
    0x565555c1: mov     ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<bar>:
    0x565557b2: push    ebp
 => 0x565557b3: mov     ebp, esp
```

Stack

```
    0xffffd000 → ...
    0xffffd004 → ...
    0xffffd008 → 0x42424242
esp 0xffffd00c → 0x42424242 <saved ebp>
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x4
    0xffffd020 → 0x5
    0xffffd024 → 0x6
```

Registers

```
eip = 0x565557b3
esp = 0xffffd00c
ebp = 0x42424242
```

Calling more than one function?

Disassembly

```
<vuln>:
    0x56555638: call   0x56555410 <read@plt>
      ...
    0x56555648: leave
    0x56555649: ret

<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
 =>   ...
```

Stack

```
    0xffffd000 → ...
    0xffffd004 → ...
    0xffffd008 → 0x42424242
esp 0xffffd00c → 0x42424242 <saved ebp>
    0xffffd010 → 0x1        <return addr>
    0xffffd014 → 0x2        <arg 1>
    0xffffd018 → 0x3        <arg 2>
    0xffffd01c → 0x4        <arg 3>
    0xffffd020 → 0x5
    0xffffd024 → 0x6
```

NOPE!

Registers / vars

```
eip = 0x565557xx      arg1 [ebp+0x08]: 0x2
esp = 0xffffd00c      arg2 [ebp+0x0c]: 0x3
ebp = 0xffffd00c      arg3 [ebp+0x10]: 0x4
```

ROP gadgets

A gadget is a sequence of useful instructions followed by an instruction that gives control back to us (usually a **`ret`**).

If we want to call more than one function, we need something to clean the stack to continue the chain, the easiest way is a gadget like: `pop regX; pop regY; ret`

Gadgets can be found manually analyzing a binary or with automated programs.

ROP gadgets: useful tools

CYBER CHALLENGE
CyberChallenge.IT

Useful tools to find ROP gadgets are:

- Ropper: github.com/sashs/Ropper
- ROPgadget: github.com/JonathanSalwan/ROPgadget
- rp++: github.com/0vercl0k/rp
- ropshell (cool online gadget library): ropshell.com
- one_gadget: github.com/david942j/one_gadget
- xrop: github.com/acama/xrop

ROP gadgets: useful tools

Example using ropper:

```
$ ropper -f myprogram
0x080487bd: adc al, 0x41; ret;
0x0804842e: adc al, 0x50; call edx;
0x08048466: adc byte ptr [eax - 0x3603a275], dl; ret;
0x080484f7: adc byte ptr [eax], bh; mov ebx, dword ptr [ebp - 4]; leave; ret;
0x08048531: add al, 0x24; ret;
0x08048529: add al, 0x59; pop ebp; lea esp, dword ptr [ecx - 4]; ret;
...
```

Save to a text file:

```
$ ropper --nocolor -f myprogram > gadgets.txt
```

Calling more than one function

Disassembly

```
<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
 => 0x56555616: ret
                  ↓
<gadget>:
    0x56555aaa: pop    eax
    0x56555aab: pop    ebx
    0x56555aac: pop    ecx
    0x56555aad: ret
                  ↓
<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

With a gadget we can advance the stack pointer to skip the previous function arguments and then return into a new function.

Stack

```
esp 0xffffd00c → 0x56555aaa <gadget>
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x565557b2 <bar>
    0xffffd020 → 0x45454545 <bar's ret addr>
    0xffffd024 → 0x4        <bar's arg 1>
    0xffffd028 → 0x5        <bar's arg 2>
    0xffffd02c → 0x6        <bar's arg 3>
```

Calling more than one function

Disassembly

```
<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<gadget>:
 => 0x56555aaa: pop    eax
    0x56555aab: pop    ebx
    0x56555aac: pop    ecx
    0x56555aad: ret
                  ↓
<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

With a gadget we can advance the stack pointer to skip the previous function arguments and then return into a new function.

Stack

```
    0xffffd00c → 0x56555aaa <gadget>
esp 0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x565557b2 <bar>
    0xffffd020 → 0x45454545 <bar's ret addr>
    0xffffd024 → 0x4        <bar's arg 1>
    0xffffd028 → 0x5        <bar's arg 2>
    0xffffd02c → 0x6        <bar's arg 3>
```

Calling more than one function

Disassembly

```
<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<gadget>:
    0x56555aaa: pop    eax
 => 0x56555aab: pop    ebx
    0x56555aac: pop    ecx
    0x56555aad: ret
                 ↓
<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

With a gadget we can advance the stack pointer to skip the previous function arguments and then return into a new function.

Stack

```
    0xffffd00c → 0x56555aaa <gadget>
    0xffffd010 → 0x1
esp 0xffffd014 → 0x2
    0xffffd018 → 0x3
    0xffffd01c → 0x565557b2 <bar>
    0xffffd020 → 0x45454545 <bar's ret addr>
    0xffffd024 → 0x4        <bar's arg 1>
    0xffffd028 → 0x5        <bar's arg 2>
    0xffffd02c → 0x6        <bar's arg 3>
```

Calling more than one function

Disassembly

```
<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<gadget>:
    0x56555aaa: pop    eax
    0x56555aab: pop    ebx
 => 0x56555aac: pop    ecx
    0x56555aad: ret

<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

With a gadget we can advance the stack pointer to skip the previous function arguments and then return into a new function.

Stack

```
    0xffffd00c → 0x56555aaa <gadget>
    0xffffd010 → 0x1
    0xffffd014 → 0x2
esp 0xffffd018 → 0x3
    0xffffd01c → 0x565557b2 <bar>
    0xffffd020 → 0x45454545 <bar's ret addr>
    0xffffd024 → 0x4        <bar's arg 1>
    0xffffd028 → 0x5        <bar's arg 2>
    0xffffd02c → 0x6        <bar's arg 3>
```

Calling more than one function

Disassembly

```
<foo>:
    0x565555c0: push   ebp
    0x565555c1: mov    ebp, esp
      ...
    0x56555615: leave
    0x56555616: ret

<gadget>:
    0x56555aaa: pop    eax
    0x56555aab: pop    ebx
    0x56555aac: pop    ecx
 => 0x56555aad: ret
                 ↓
<bar>:
    0x565557b2: push   ebp
    0x565557b3: mov    ebp, esp
```

Works like a charm!

Stack

```
    0xffffd00c → 0x56555aaa <gadget>
    0xffffd010 → 0x1
    0xffffd014 → 0x2
    0xffffd018 → 0x3
esp 0xffffd01c → 0x565557b2 <bar>
    0xffffd020 → 0x45454545 <bar's ret addr>
    0xffffd024 → 0x4        <bar's arg 1>
    0xffffd028 → 0x5        <bar's arg 2>
    0xffffd02c → 0x6        <bar's arg 3>
```

A complete ROP chain

To build a chain of multiple calls we can use different gadgets to clean the stack after calling each function:

ROP CHAIN

```
<...padding...>
<func_1>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_2>
<gadget_2>
<func_2's arg1>
<func_3>
<gadget_3>
<func_3's arg1>
<...>
```

```
pop <some_reg>
pop <some_reg>
ret
```

```
add esp, 0x4
ret
```

```
pop <some_reg>
ret
```

What about library functions?

If we want to call an external function we have two options:

1. If the program already uses the function we want to call, then there must be a PLT entry for it. If we know the location of the PLT we can just jump to it to call the function.
2. If the program does not use the function we want to call, then we must obtain its address at runtime and jump there directly.

What about library functions?

Since libraries are dynamically loaded in memory at random positions we cannot know where their functions are upfront.

So, for the second option, we need to:

1. Know which library is being used (e.g. which version).
2. *Leak* the address of some *known* symbol at runtime.
3. Use that to compute the *base address* of the library.
4. Do some math to get the position of any other symbol.

Calling library functions: ret2libc

If we somehow manage to leak the address of a symbol in the libc at runtime and we also know the libc version (e.g. we have a local copy), we are all set!

Suppose we leak `printf = 0x7f058cf98190`; then we check its offset in libc:

```
$ objdump -T libc-2.24.so | grep printf
000000000004f190 g    DF .text   00000000000000a1  GLIBC_2.2.5 printf
```

We can use it to calculate the base address of libc in memory:

```
libc_base = 0x7f058cf98190 - 0x4f190 = 0x7f058cf49000
```

And now we know the address of any other symbol:

```
other_symbol = libc_base + symbol_offset
```

Calling library functions: ret2libc

So now we can also call any libc library function with arbitrary arguments… pretty cool, right?

Stack:

```
0x41414141
0x41414141
0x41414141 <old $ebp>
0x7ffa4f20 <ret addr>
0x41414141 <fake ret addr>
0x7ffb0090 <cmd>
...
...
```

`libc-2.24.so:`

```
101110<system>0010111101010110<fopen>010
11001100101101001100110101011<strlen>111
0101<puts>1010111<printf>101100101010010
101"GLIBC_2.2.5"01010<malloc>00111010010
0101011011001001010"/bin/sh"001101011111
```

Result:

```
system("/bin/sh")
```

Calling library functions: ret2libc

The `pwntools` have a handful of very useful features to automatically get offsets, symbols, etc. from an ELF:

```
from pwn import *

libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')  # Load an ELF.
libc_system = libc.symbols['system']           # Find the address of a symbol.
libc_binsh = next(libc.search('/bin/sh\x00'))  # Search for a sequence of bytes.

myelf = ELF('./myprogram')
got_puts = myelf.got['puts']                   # Find the address of a GOT entry.
plt_puts = myelf.plt['puts']                   # Find the address of a PLT entry.

rop = ROP('/lib/x86_64-linux-gnu/libc.so.6')   # Load ROP gadgets from an ELF.
gadget = rop.find_gadget(['pop rsi'])          # Find a gadget containing a specific instr.
```

Magic gadgets

Every single libc binary must contain some code to execute `/bin/sh` somehow, since libc provides the `system(cmd)` function, which basically does `execl("/bin/sh", ...)`.

A "magic gadget", also called "one gadget", is a gadget that **can spawn a shell alone if the program jumps to it!**

There usually are several magic gadgets laying around in the libc binary, each requiring different constraints to work.

Magic gadgets

The `one_gadget` tool is a very cool Ruby program which can automatically find magic gadgets and their constraints:

```
$ one_gadget /lib/x86_64-linux-gnu/libc.so.6

0x3f306 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x3f35a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xd6b9f execve("/bin/sh", rsp+0x60, environ)
constraints:
  [rsp+0x60] == NULL
```

/lib/x86_64-linux-gnu/libc.so.6:

```
3f35a:  mov   rax,QWORD PTR [rip+0x359b57]
3f361:  lea   rdi,[rip+0x1228b1]
3f368:  lea   rsi,[rsp+0x30]
3f36d:  mov   DWORD PTR [rip+0x35c109],0x0
3f377:  mov   DWORD PTR [rip+0x35c103],0x0
3f381:  mov   rdx,QWORD PTR [rax]
3f384:  call  b8640 <execve@@GLIBC_2.2.5>
```

ROP chain: 32bit vs 64bit

In x86 32bit arguments are almost always passed on the stack (as per the **cdecl** calling convention), but in x86 64bit arguments are usually passed in registers (as per the **System V ABI** calling convention).

If we want to build a 64bit ROP chain we need to use gadgets to *pop the arguments from our chain to the needed registers*. Even if we're only calling one function!

ROP chain: 32bit vs 64bit

32bit **cdecl** convention: arguments on the stack

64bit **System V** convention: arguments in `RDI, RSI, RDX, RCX, R8, R9, XMM0…7`

32 bit

```
<...padding...>
<func_1>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_2>
<gadget_2>
<func_2's arg1>
<func_3>
<gadget_3>
<func_3's arg1>
<...>
```

```
pop <some_reg>
pop <some_reg>
ret
```

```
add esp, 0x4
ret
```

```
pop <some_reg>
ret
```

64 bit

```
<...padding...>
<gadget_1>
<func_1's arg1>
<func_1's arg2>
<func_1>
<gadget_2>
<func_2's arg1>
<func_2>
<gadget_3>
<func_3's arg1>
<func_3>
<...>
```

```
pop rdi
pop rsi
ret
```

```
pop rdi
sub eax, 0x8
ret
```

```
pop rdi
ret
```

ROP chain: not only calling functions

Sometimes you cannot call functions, but who needs to call library functions when you've got the right gadgets?

```
pop rbp
ret
```

```
mov DWORD PTR [rsi], ebx
sub rsp, 0x20
ret
```

```
xchg rsi, rdi
ret
```

```
add al, 0x48
add edx, 1
syscall
```

```
pop r15
pop r10
pop r13
ret
```

```
pop rbp
mov edi, 0x61e600
jmp rax
```

```
int 0x80
```

```
mov dword ptr [rdi + 0x10], ecx
xor ch, ch
mov byte ptr [rdi + 0x12], ch
ret
```

More than ROP...

If you're interested, you might want to also take a look at SROP: Sigreturn Oriented Programming.

This technique takes advantage of the `sigreturn` syscall to take control of the registers (and thus the execution) by using gadgets which are usually always in memory at runtime.

SROP is generally "simpler" than ROP and often only needs one gadget (to execute the `sigreturn` syscall).

CYBER CHALLENGE
CyberChallenge.IT

Got any questions?