Software Security 4

ROP & JOP
Intro on Heap Exploitation

Carlo Ramponi <carlo.ramponi@unitn.it>

Return Oriented Programming

Return Oriented Programming

Return Oriented Programming (or **ROP**) is based on the idea of chaining together small snippets (or **gadgets**) of assembly with stack control to lead the program to do more complex things.

In this technique, an attacker **gains control of the call stack** to hijack program control flow and then **executes carefully chosen** machine instruction **sequences that are already present in the machine's memory**, called "**gadgets**".
Each gadget typically **ends in a return instruction** and, chained together, these gadgets allow an attacker to perform **arbitrary operations** on a machine employing defenses that thwart simpler attacks. [1]

Traditional code-injection defenses are bypassed (e.g. DEP)

[1] https://en.wikipedia.org/wiki/Return-oriented_programming

ROP Gadgets

In principle, any block of instructions that **ends with a control-flow transfer**, e.g.:

```
pop rax; pop rbx; ret
```

There are different types of gadgets:

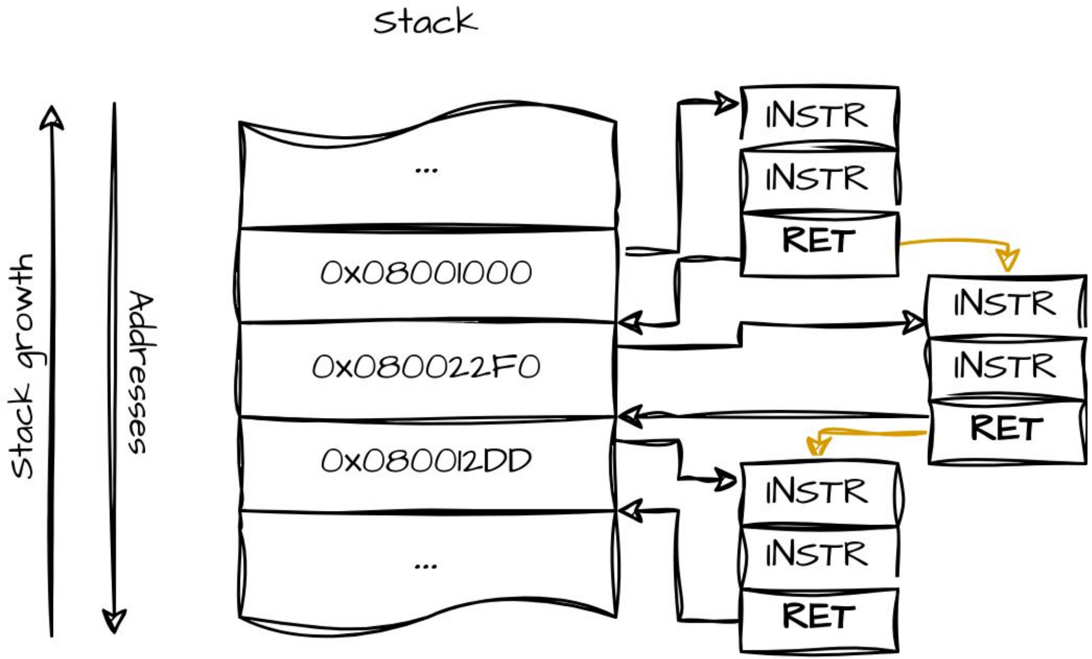
- Gadgets for **loading and storing data**
  - Register to Register
  - Register to Memory
  - Memory to Register
- Gadgets for **arithmetic operations**

With **enough gadgets** (not many), Return Oriented Programming becomes **Turing-Complete**. [1]

[1] Roemer, Ryan, et al. "Return-oriented programming: Systems, languages, and applications." ACM Transactions on Information and System Security (TISSEC) 15.1 (2012): 1-34.

ROP Exploits

- Based on **gadgets ending** with a routine **return** instruction (**`ret`**)
- **`ret`** pops the return location from the stack and **jumps there**
- If the **stack data is corrupted**, a series of *fake* return addresses can be stacked
- Every time a **`ret`** is executed, control is passed to the **next gadget**



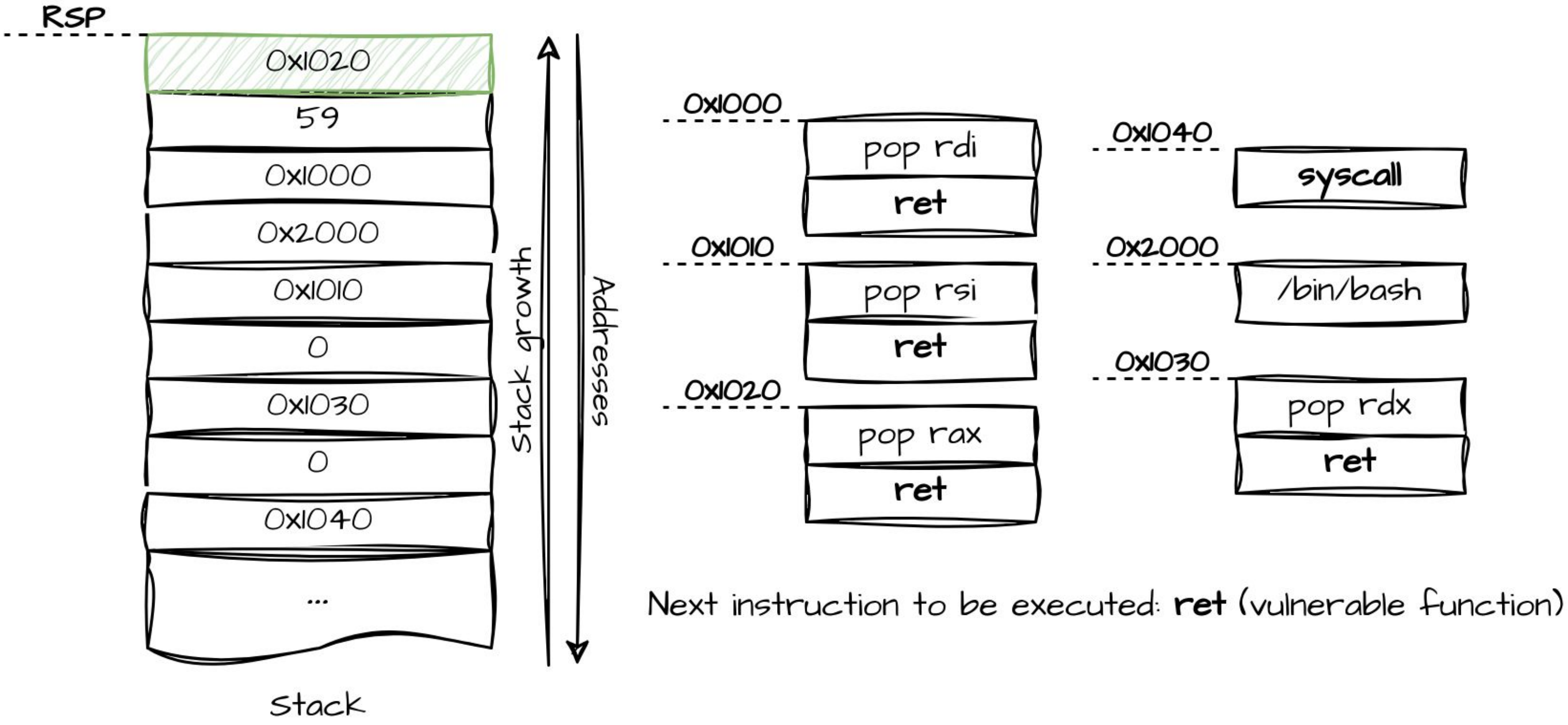
ROP Exploits

To perform an **attack based on ROP** one has to:

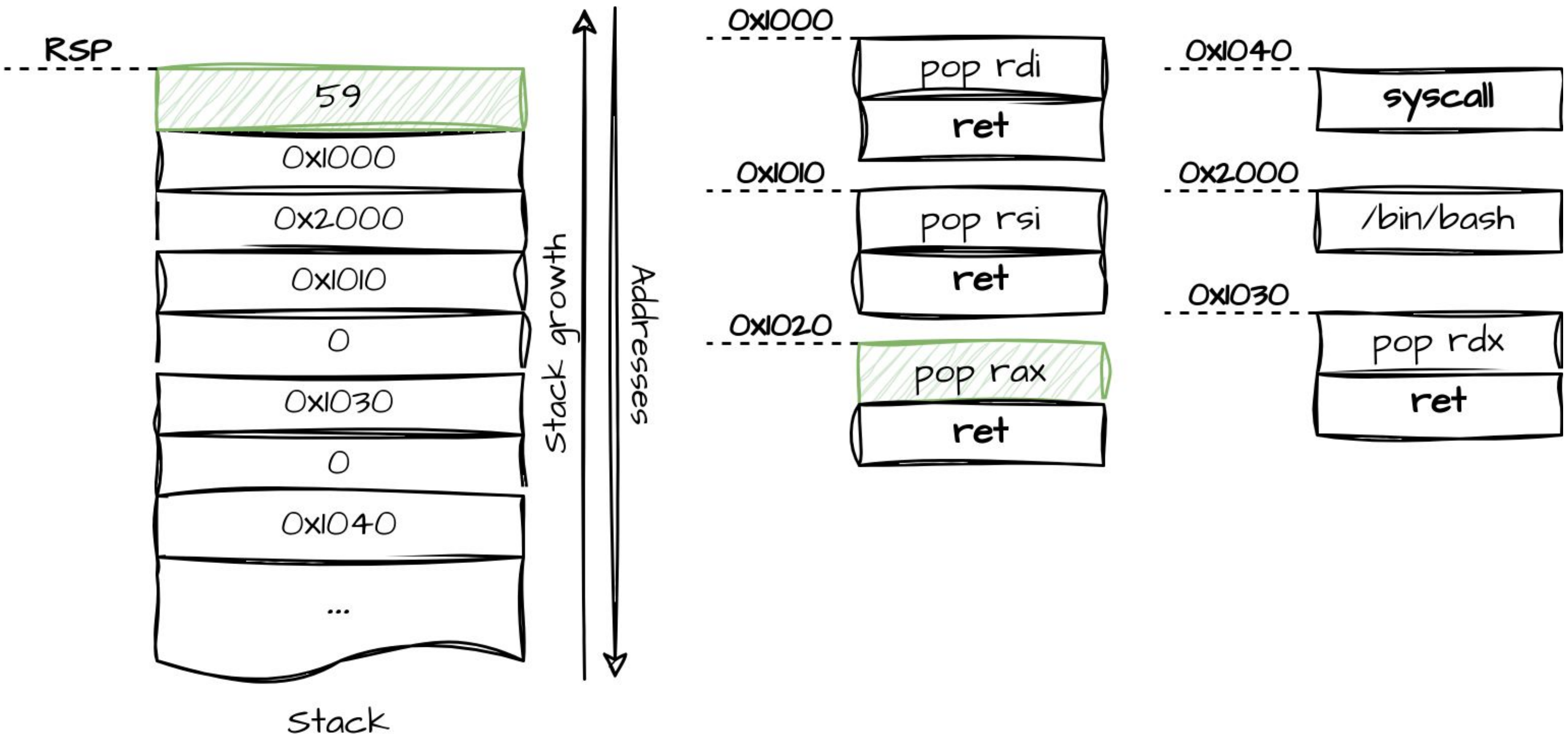
1. **Find the chain of gadgets** that induces the **expected behavior**
   - Store the addresses of the gadgets on the stack
   - Also store any value that the gadgets will pop from the stack

2. The value of **RIP** register must be **overwritten** with the address of the **first gadget** (any control-flow hijacking attack will do, e.g. Stack BoF)

ROP Exploits - Example

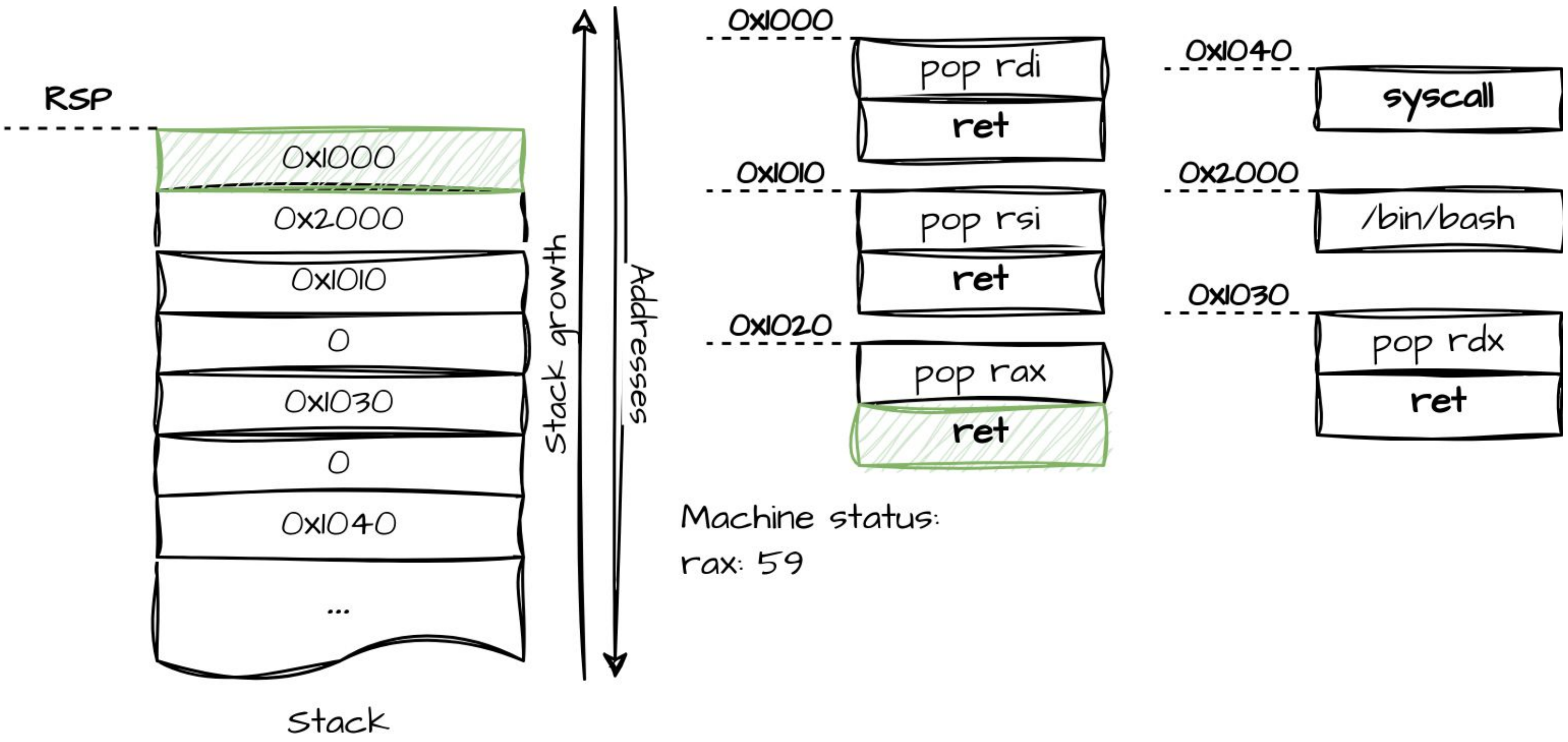


ROP Exploits - Example



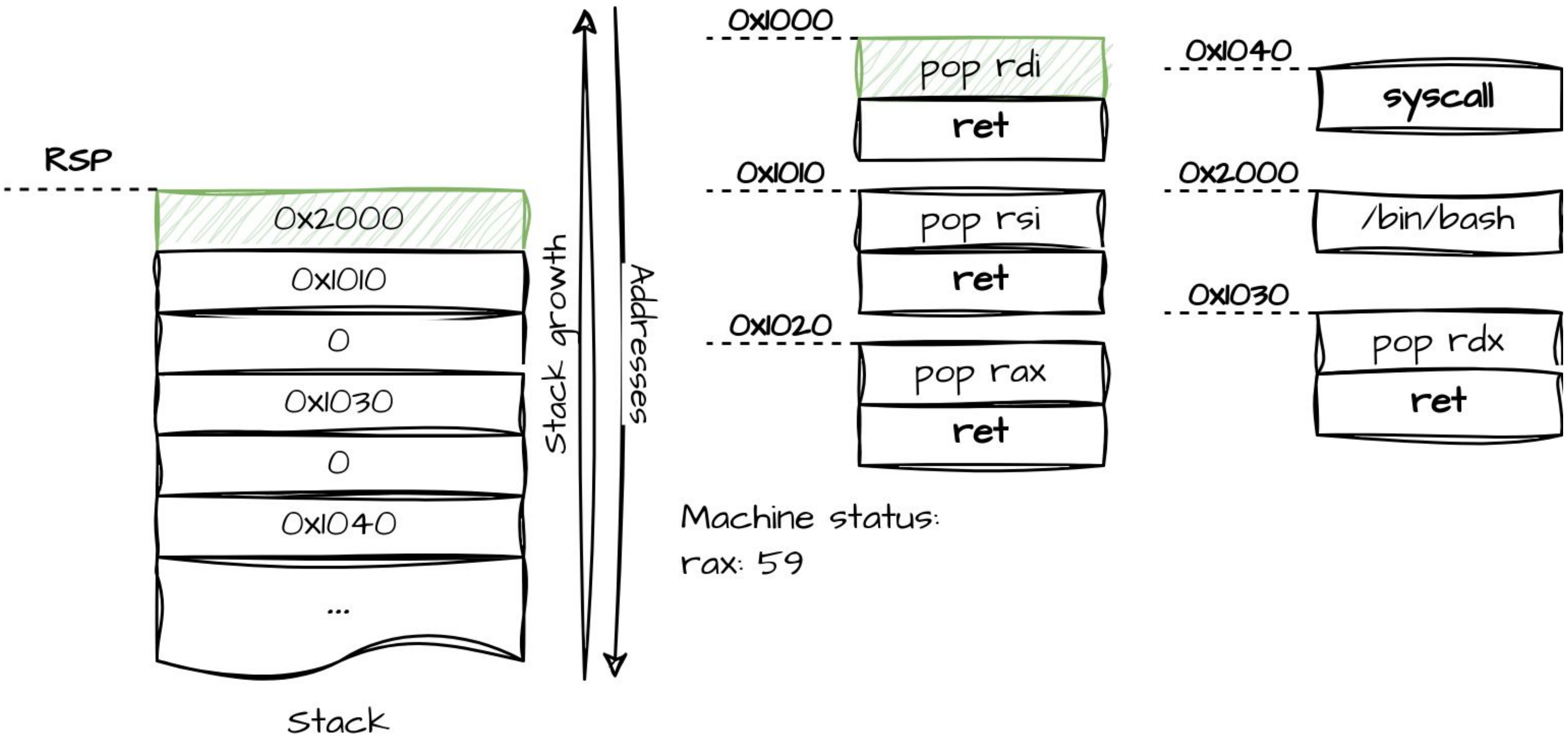
Carlo Ramponi

ROP Exploits - Example

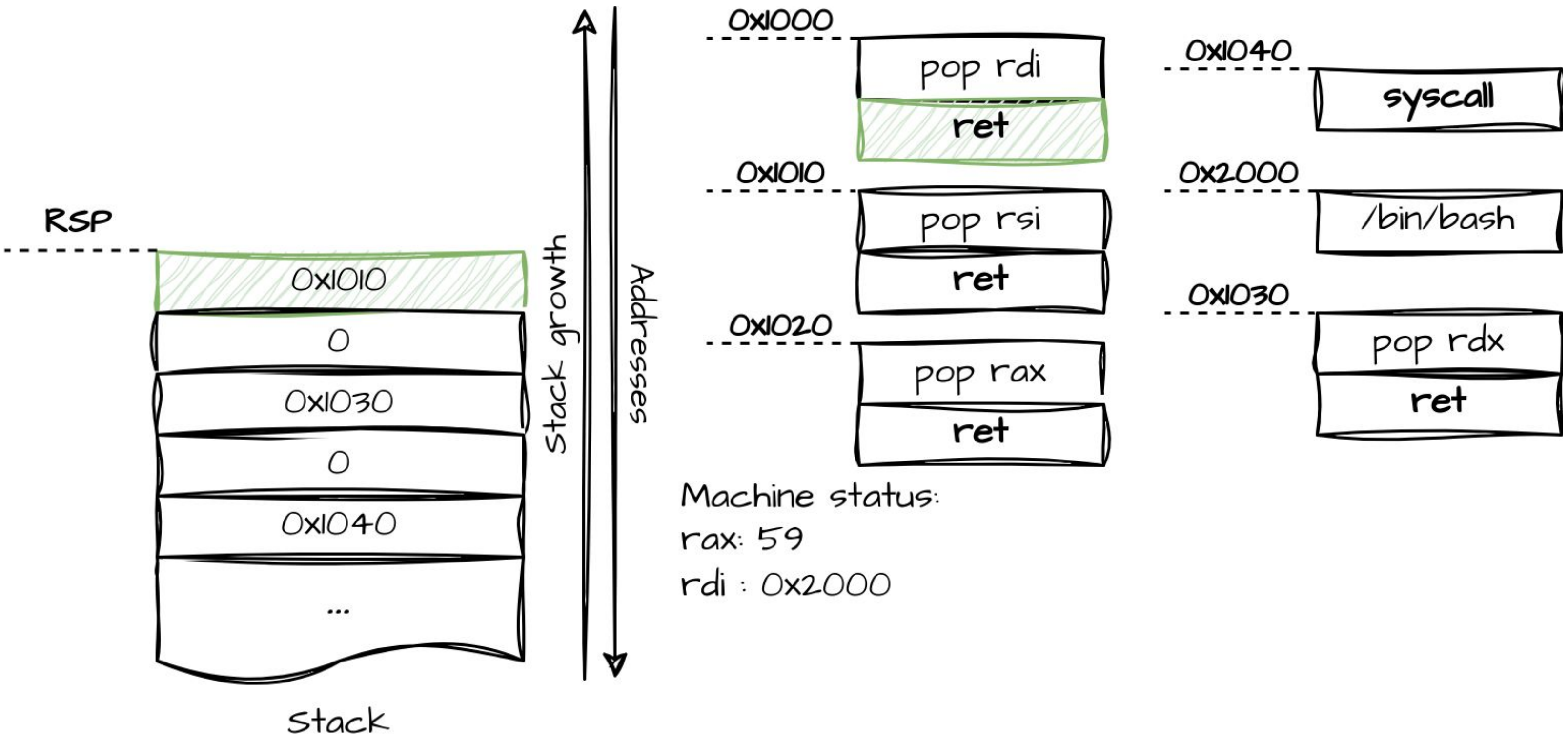


Carlo Ramponi

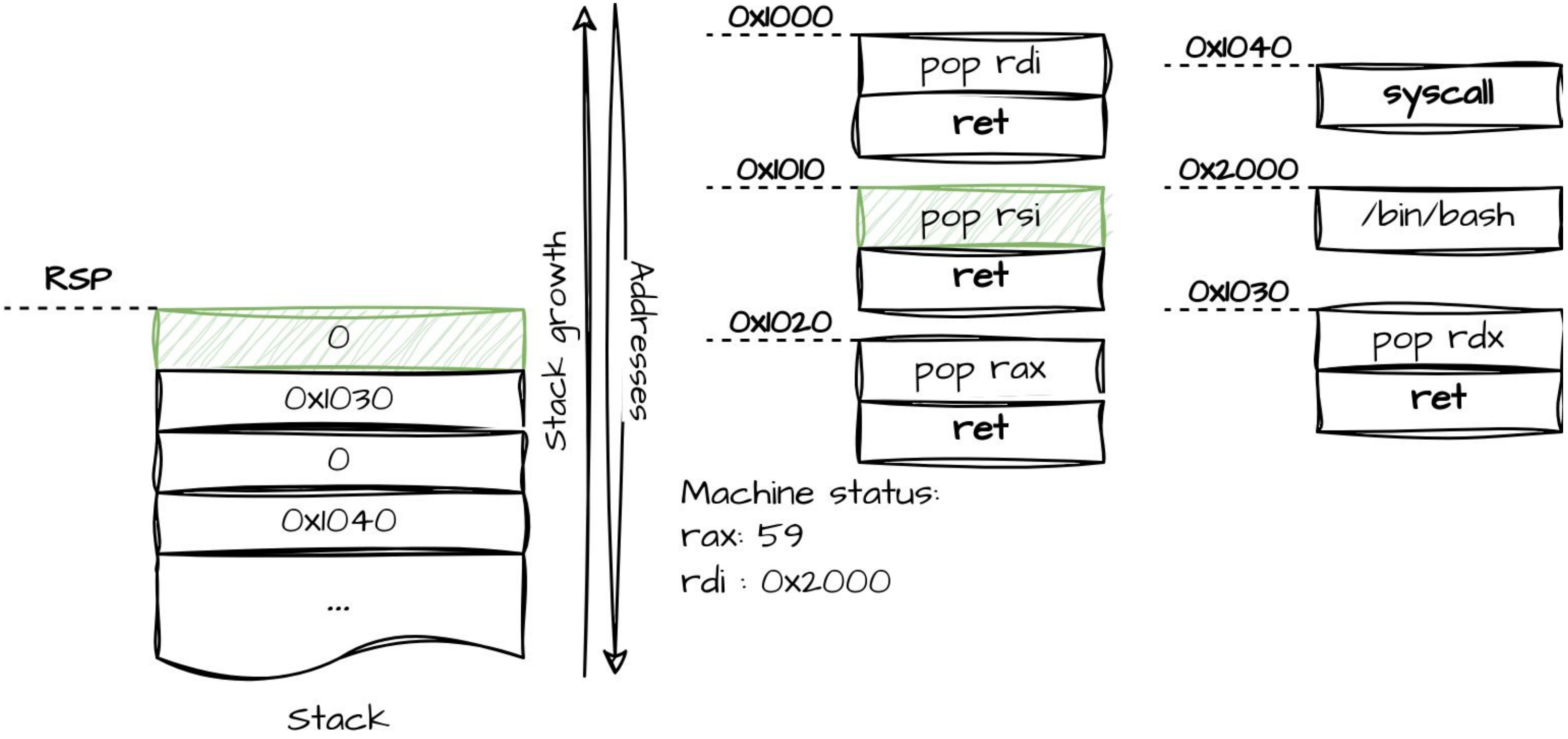
ROP Exploits - Example



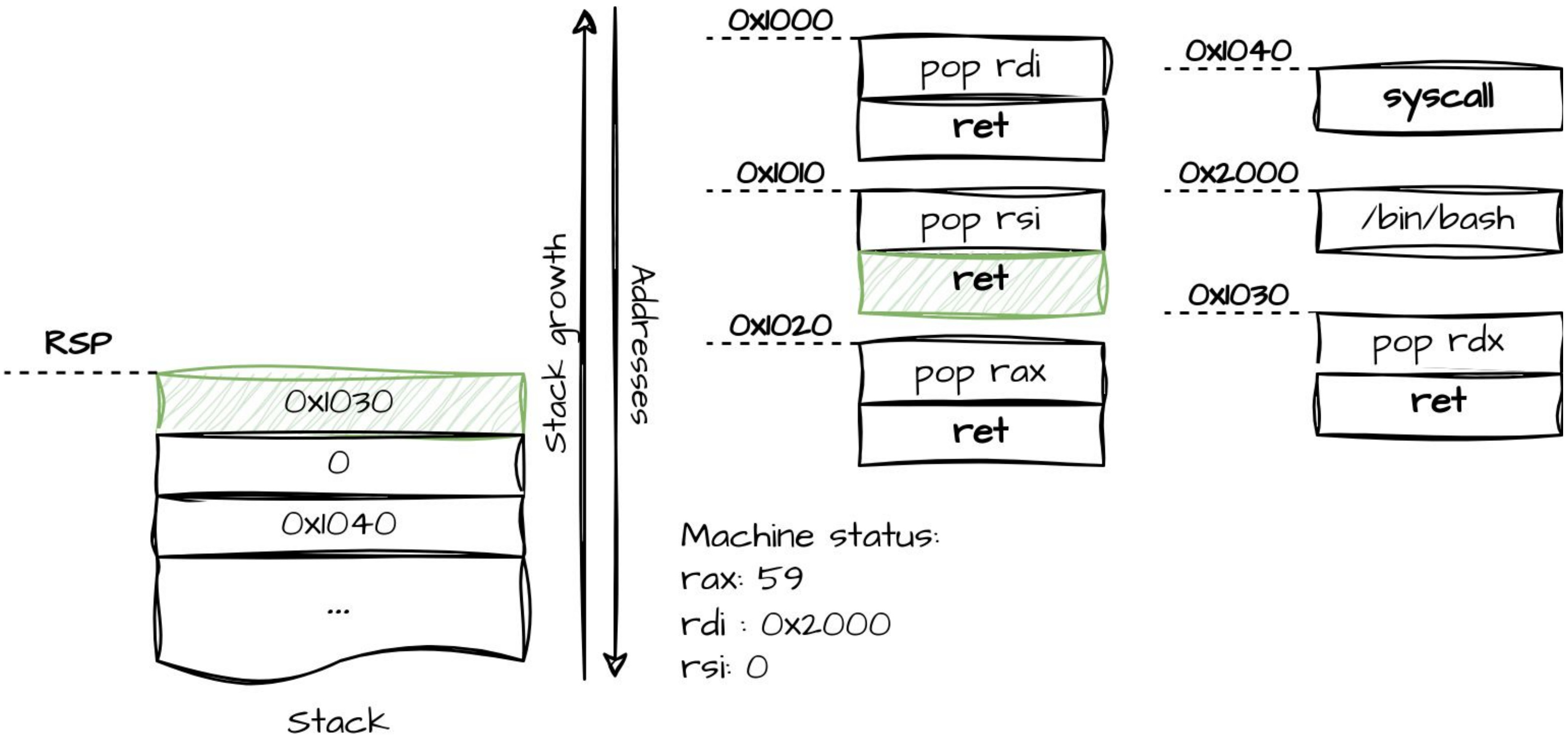
ROP Exploits - Example



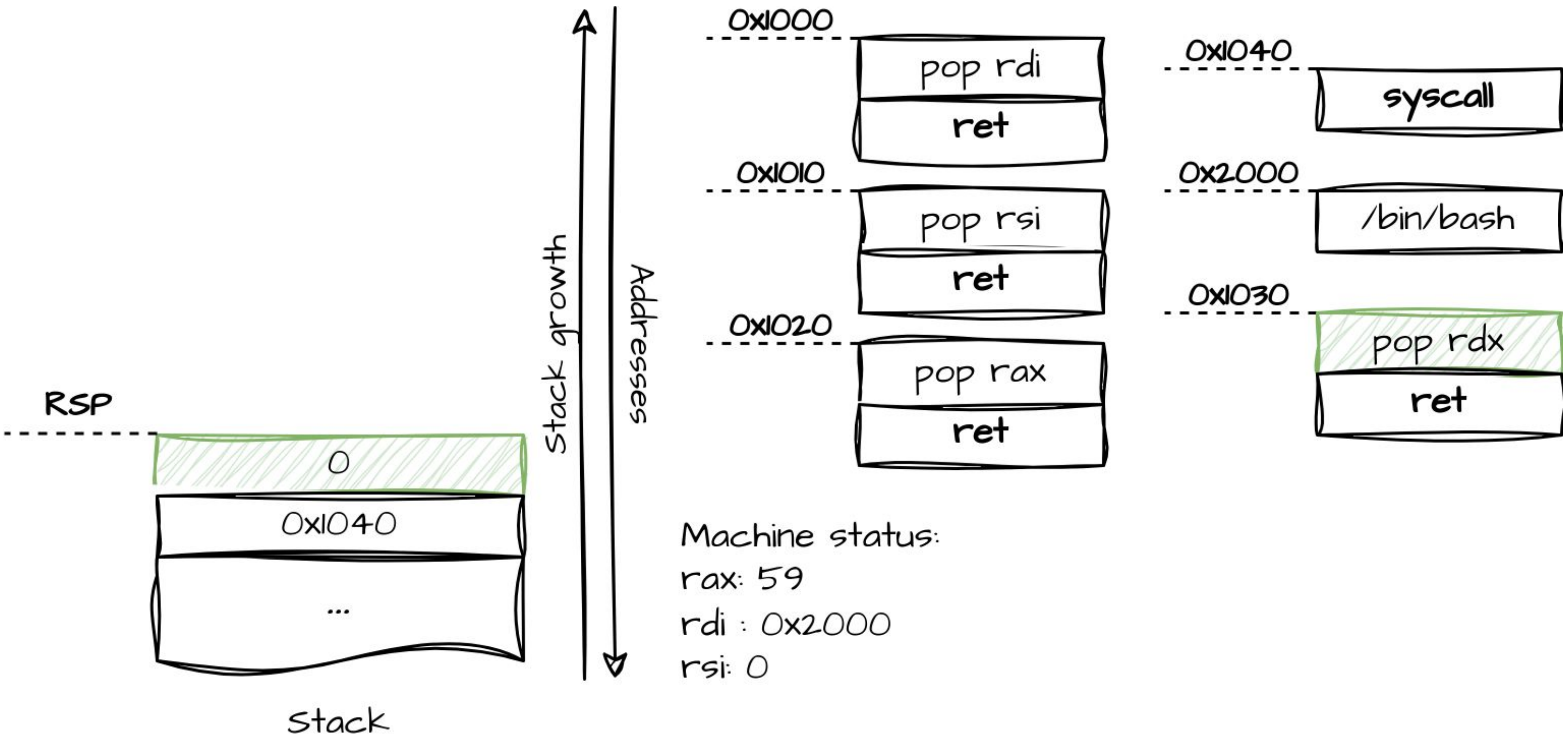
ROP Exploits - Example



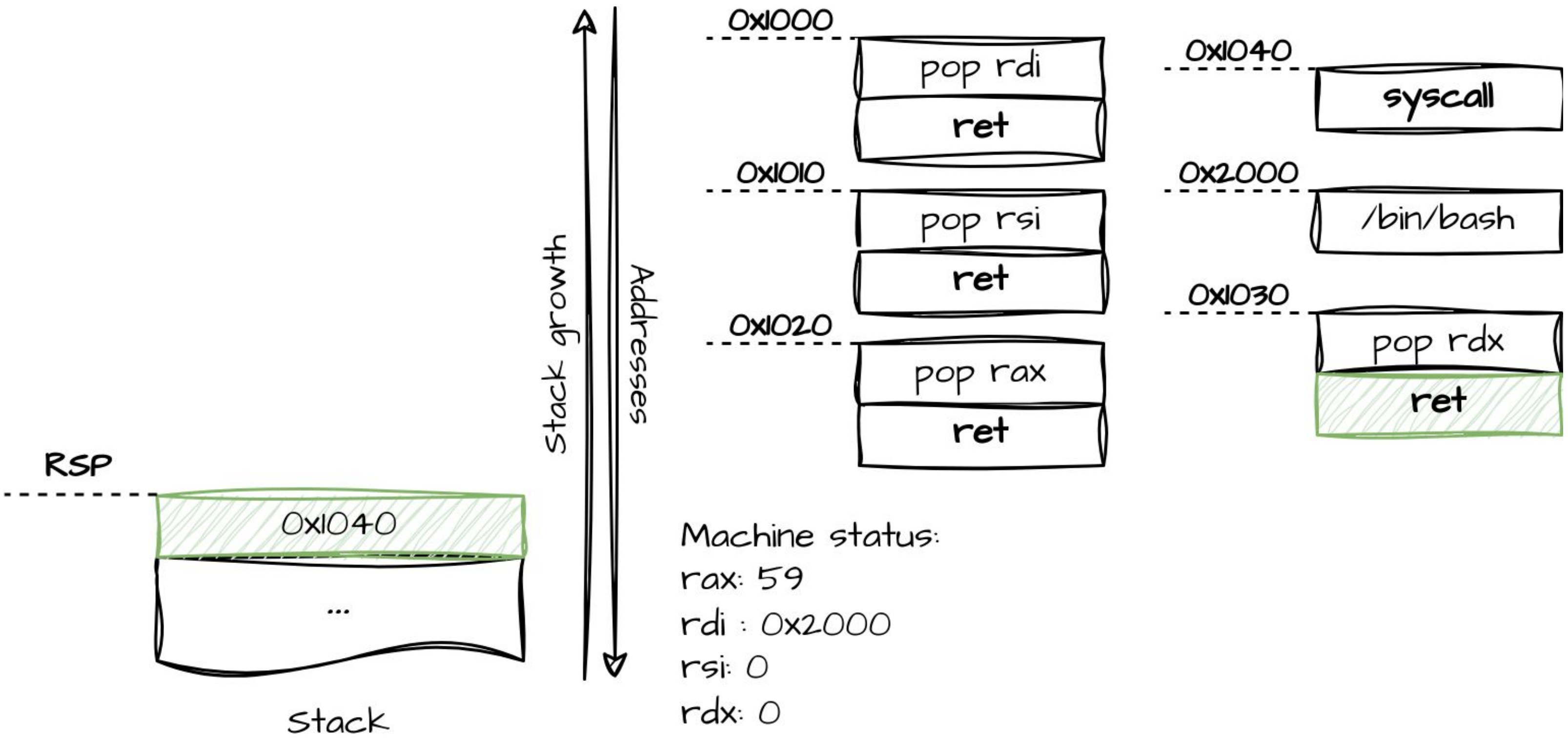
ROP Exploits - Example



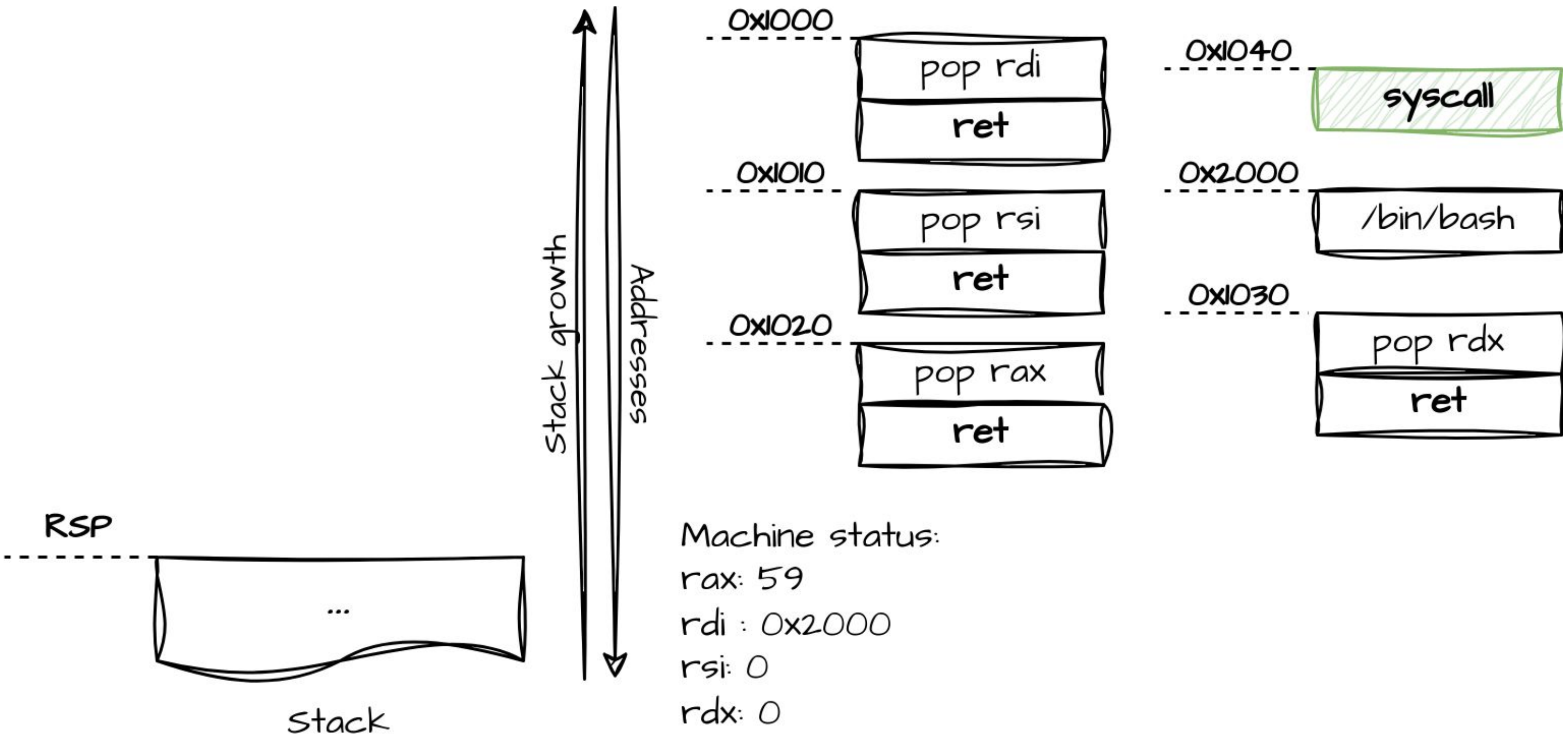
ROP Exploits - Example



ROP Exploits - Example



ROP Exploits - Example



ROP Exploits - Example

The **ROP chain** executes a call to the system call number **59**, which is **execve** [1]
The exploit also prepares the arguments of the system call which has the following signature:

```
int execve(const char *pathname, char *const _Nullable argv[],
                char *const _Nullable envp[]);
```

rdi	rsi	rdx
const char *pathname	char *const argv[]	char *const envp[]
/bin/bash	0	0

This translates to: `execve(“/bin/bash”, NULL, NULL);`

[1] https://filippo.io/linux-syscall-table/

ROP Exploits

Question: How do we find such gadgets?

- **By hand**, e.g. by inspecting the disassembly (old school approach)
- **By using** one of the available **tools**:
    - Ropper - https://github.com/sashs/Ropper
    - ROPGadget - https://github.com/JonathanSalwan/ROPgadget
    - Pwntools - https://github.com/Gallopsled/pwntools

Some tools can also assist in building common gadget chains (e.g. **`execve("/bin/sh")`**), but they require a large amount of gadgets to succeed.

ROP Exploits

```
 carlo@carlo-pc  ~  ropper --file /bin/ls --search "pop rbx"
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
[INFO] Searching for gadgets: pop rbx

[INFO] File: /bin/ls
0x000000000000d051: pop rbx; idiv edi; jmp qword ptr [rsi + 0x66];
0x0000000000008423: pop rbx; pop rbp; pop r12; pop r13; jmp rax;
0x00000000000051c4: pop rbx; pop rbp; pop r12; pop r13; pop r14; ret;
0x000000000000637e: pop rbx; pop rbp; pop r12; pop r13; ret;
0x0000000000006b0e: pop rbx; pop rbp; pop r12; ret;
0x0000000000006b3f: pop rbx; pop rbp; cdqe; pop r12; add qword ptr [rip + 0x197c4], rax; ret;
0x00000000000065e4: pop rbx; pop rbp; ret;
0x0000000000006f9c: pop rbx; ret;
```

ROP Countermeasures

Traditional mitigations such as **Stack Canaries** and **ASLR** can help in **preventing** the stack to be corrupted to perform a **Control-Flow Hijacking attack**.

Some mitigations **specific to ROP** attacks have been developed:

Detect ROP attacks:

- Observe the **execution of small set of instructions** ending with a **`ret`**
- Observe **`pop`**s of return addresses all pointing at the **same memory space**

Prevent ROP attacks:

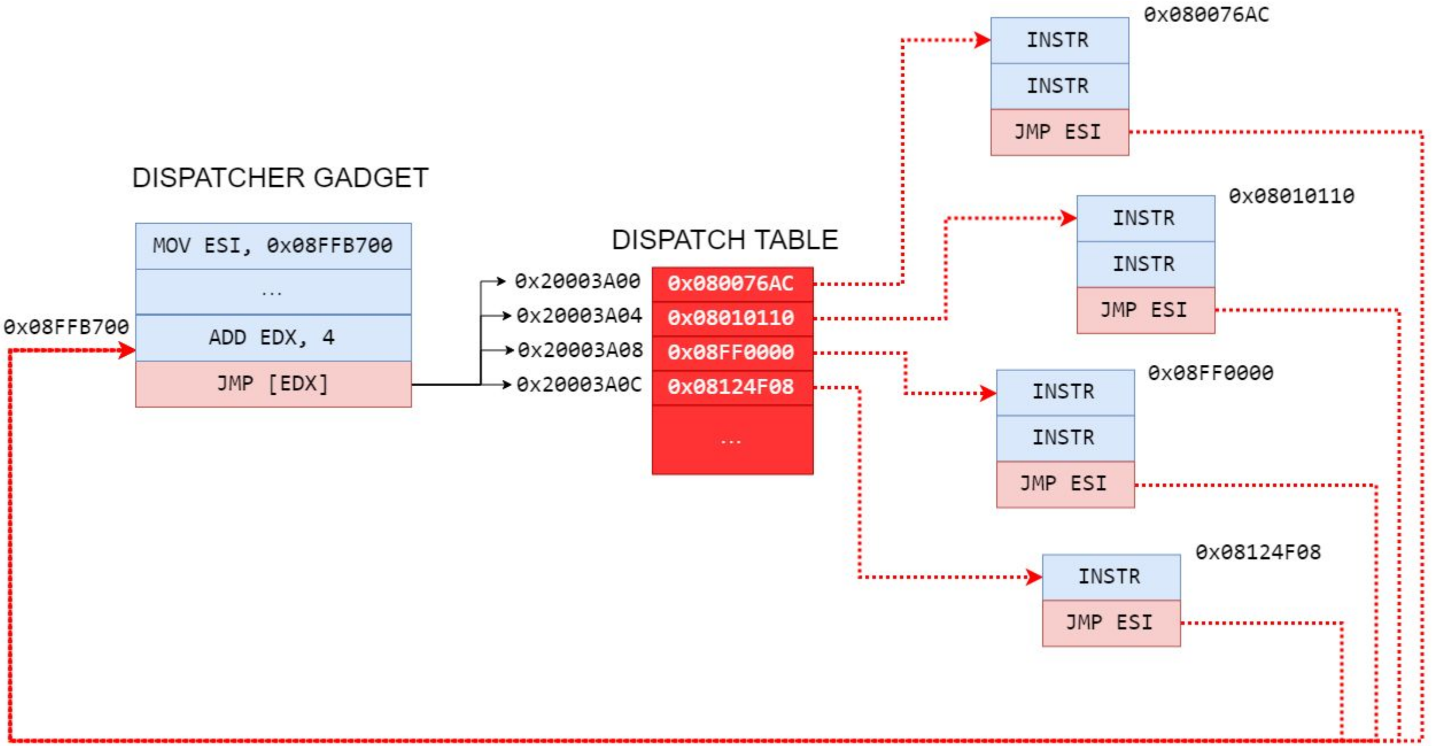
- *Return-less* approach: tell the compiler **not to use** the **`ret`** instruction

Jump Oriented Programming - JOP

- **Jump-Oriented Programming** (JOP) is a technique that triggers the execution of arbitrary code via a sequence of indirect jump instructions

- Similarly to ROP, JOP is based on a **sequence of small gadgets**
  - in ROP, each gadget **ends with a return instruction** (**`ret`**)
  - in JOP each gadget **ends with an unconditional jump** instruction (**`jmp`**)

Jump Oriented Programming - JOP



Jump-oriented programming: a new class of code-reuse attack
Tyler Bletsch, Xuxian Jiang, Vincent W. Freeh, Zhenkai Liang ACM ASIACCS'11

Intro on

Heap Exploitation

Carlo Ramponi

Heap Exploitation

The **Heap** section of a process is where **dynamically allocated memory** sections resides.

Memory can be managed using the **`malloc`** and **`free`** library functions, which internally will **keep a state** of the current and past allocations, trying to **optimize memory usage** by using different **allocation strategies** [1]

Each allocated block will be **preceded by its metadata**, containing for instance the size of the allocated block.

Corrupting the metadata of an allocation or **exploiting errors** in the management of heap memory by **leveraging allocation strategies** enable a **wide range of attacks**.

[1] https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

Heap Exploitation

Many exploitation techniques on the heap require **a deep understanding of the allocation strategies** and the **inner workings of the in-use allocator**, unfortunately we don't have the time to dig this deep into it.

If you are interested, you can follow this nice tutorial:
https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

We'll only look at the (probably) **easiest class of vulnerabilities** on the heap, namely **Use After Free (UAF)**

Heap Exploitation

Many exploitation techniques on the heap require **a deep understanding of the allocation strategies** and the **inner workings of the in-use allocator**, unfortunately we don’t have the time to dig this deep into it.

If you are interested, you can follow this nice tutorial:
https://azeria-labs.com/heap-exploitation-part-1-understanding-the-glibc-heap-implementation/

We’ll only look at the (probably) **easiest class of vulnerabilities** on the heap, namely **Use After Free (UAF)**

Other interesting vulnerabilities are **Heap Overflow** and **Double free**

Heap Exploitation - Use After Free

When a programmer is **finished with an allocation** from **`malloc`**, the programmer releases it back to the heap manager by passing it to **`free`**.

Internally, the heap manager **needs to keep track of freed chunks** so that **`malloc`** can reuse them during allocation requests. This ensures that a **new allocation request**, if a **previously freed chunk** that can hold the new one is available, will be served using that chunk.

The algorithm is **not trivial**, but by knowing the **state of the heap manager**, one can **deterministically know** which chunk will be chosen.

Easy assumption: if a **chunk is freed** and the next heap operation is a **malloc of the same size**, that **chunk will be used**.

Heap Exploitation - Use After Free

```
carlo@carlo-pc  ~/Desktop/EH24/Code  bat uaf.c
File: uaf.c
#include <stdio.h>
#include <stdlib.h>

int main() {
    char *p = malloc(256);
    free(p);
    char *pp = malloc(256);
    printf("p = %p, pp = %p, equal? %s\n", p, pp, p == pp ? "Yes": "No");
    return 0;
}
carlo@carlo-pc  ~/Desktop/EH24/Code  ./uaf
p = 0x64f28da072a0, pp = 0x64f28da072a0, equal? Yes
```

Heap Exploitation - Use After Free

```
carlo@carlo-pc  ~/Desktop/EH24/Code  bat uaf2.c
```

```
File: uaf2.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main() {
    char *p = malloc(256);
    free(p);
    char *pp = malloc(256);
    strncpy(pp, "Legit content of pp", 256);
    strncpy(p, "Use after free??", 256);

    printf("%s\n", pp);
    return 0;
}
```

Question: what will this program print?

Heap Exploitation - Use After Free

```
carlo@carlo-pc  ~/Desktop/EH24/Code  bat uaf2.c
       File: uaf2.c
       #include <stdio.h>
       #include <stdlib.h>
       #include <string.h>

       int main() {
           char *p = malloc(256);
           free(p);
           char *pp = malloc(256);
           strncpy(pp, "Legit content of pp", 256);
           strncpy(p, "Use after free??", 256);

           printf("%s\n", pp);
           return 0;
       }
carlo@carlo-pc  ~/Desktop/EH24/Code  ./uaf2
Use after free??
```

We modified the content of the newly allocated memory **using the old pointer**!


That's All
Folks