ROP-ing Your Way on Aarch64

Introduction

Sascha Schirra

October, 5th 2022

Who Am I

Sascha Schirra

- Security Consultant @ Recurity Labs
  - Reverse engineering
  - Exploit development
  - Mobile application security
  - Embedded systems
- Twitter: @s4sh_s

What You Will See

- Aarch64 fundamentals
- Return Oriented Programming (ROP) Introduction
- Jump Oriented Programming (JOP) Introduction
- Demo Exploit

Why ARM?

Why ARM?

- Smartphones and Tablets
- Some Computers
- IoT devices
- AWS EC2 Instances

ARM Basics

ARM Introduction

- Reduced Instruction Set Computing
  - Small instruction set
  - Large uniform register file
  - Load / store architecture
  - Simple addressing modes
  - Fixed instruction size
- Conditional instructions (only Aarch32)

ARMv7 - Basics

- 32bit Architecture (Aarch32)
- Different states
  - ARM
  - Thumb
  - and more
- 16 registers
  - `r0-r12` (general purpose)
  - `r13` or `sp`
  - `r14` or `lr`
  - `r15` or `pc`

Privilege Levels

ARM	x86
User(USR)	RING 3
Fast Interrupt Request (FIQ)	
Interrupt Request (IRQ)	
Supervisor (SVC)	RING 0
Monitor (MON)	
Abort (ABT)	
Undefined (UND)	
System (SYS)	

ARMv7 - ARM State

- Default state
- `r0-r12`, `sp`, `lr`, `pc` are accessible

Instruction size	32 bit
Alignment	32 bit

ARMv7 - Thumb State

- Introduced wiht ARMv4T
- Smaller instruction size (16 bit) but less instructions
  - `pc` can only be modified by specific instructions
- better code density - less performance
- Only `r0-r7`, `sp`, `lr`, `pc` are accessible by most instructions
- Thumb-2 state introduced in 2003 with ARMv6T2
  - Extends Thumb state with 32 bit instructions
  - Those instructions can access all registers

Instruction size	16 / 32 bit
Alignment	16 bit

ARMv8 - Basics

- 64bit Architecture (Aarch64)
- Aarch32 state for compatibility reasons
  - A32 and T32
- Exception Levels

ARMv8 - Registers

- `x0-x29` (general purpose)
- `x30` or `lr`
- `sp`
- `pc`
- `xzr`
- `w0-w29` (lower 32bit part)

Exception Levels

Level	Description
EL0	Applications
EL1	OS kernels and associated functions
EL2	Hypervisor
EL3	RING Secure monitor

Function Prologue

ARMv7

```
    push    {fp, lr}
    add     fp, sp, #4
    sub     sp, sp, #136
```

ARMv8

```
    stp x29, x30, [sp, -0xa0]!
    mov x29, sp
```

Function Epilogue

ARMv7

```
    sub sp, fp, #4
    pop {fp, pc}
```

```
    sub sp, fp, #4
    pop {fp, lr}
    bx  lr
```

ARMv8

```
    ldp x29, x30, [sp], 0xa0
    ret
```

Old Way

What is Shellcode?

Shellcode is a sequence of bytes that can be interpreted and executed by the CPU. Historically it is called shellcode, because the first versions spawned a shell.

Mostly, shellcode consists of position indepedent code.

Shellcode must be free of so-called bad bytes. Bad bytes are bytes that interfere with the placement of the shellcode (e.g. a null byte if string operations like `strcpy` are used).

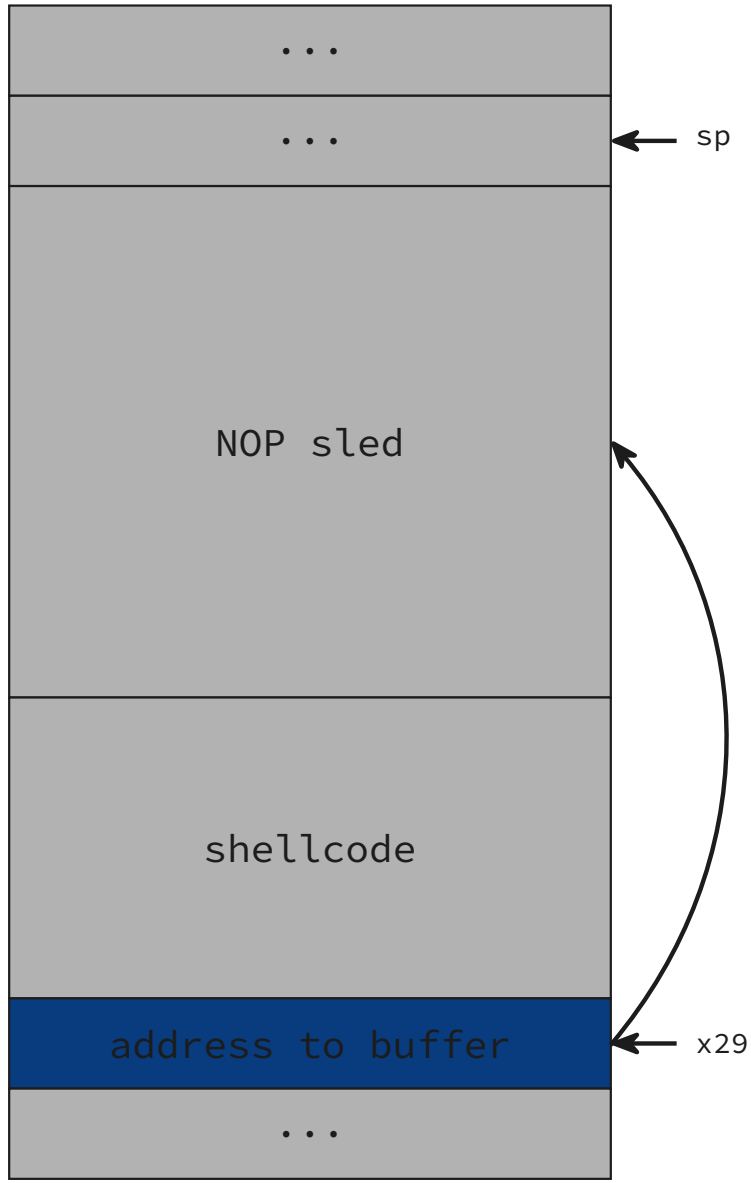
A vulnerability is used to jump to that shellcode

Example - Buffer Overflow

```
void dosomething(char *msg){
    char buf[128];
    strcpy(buf, msg);
    puts(buf);
}

void main(int argc, char *argv[]){
    dosomething(argv[1]);
}
```

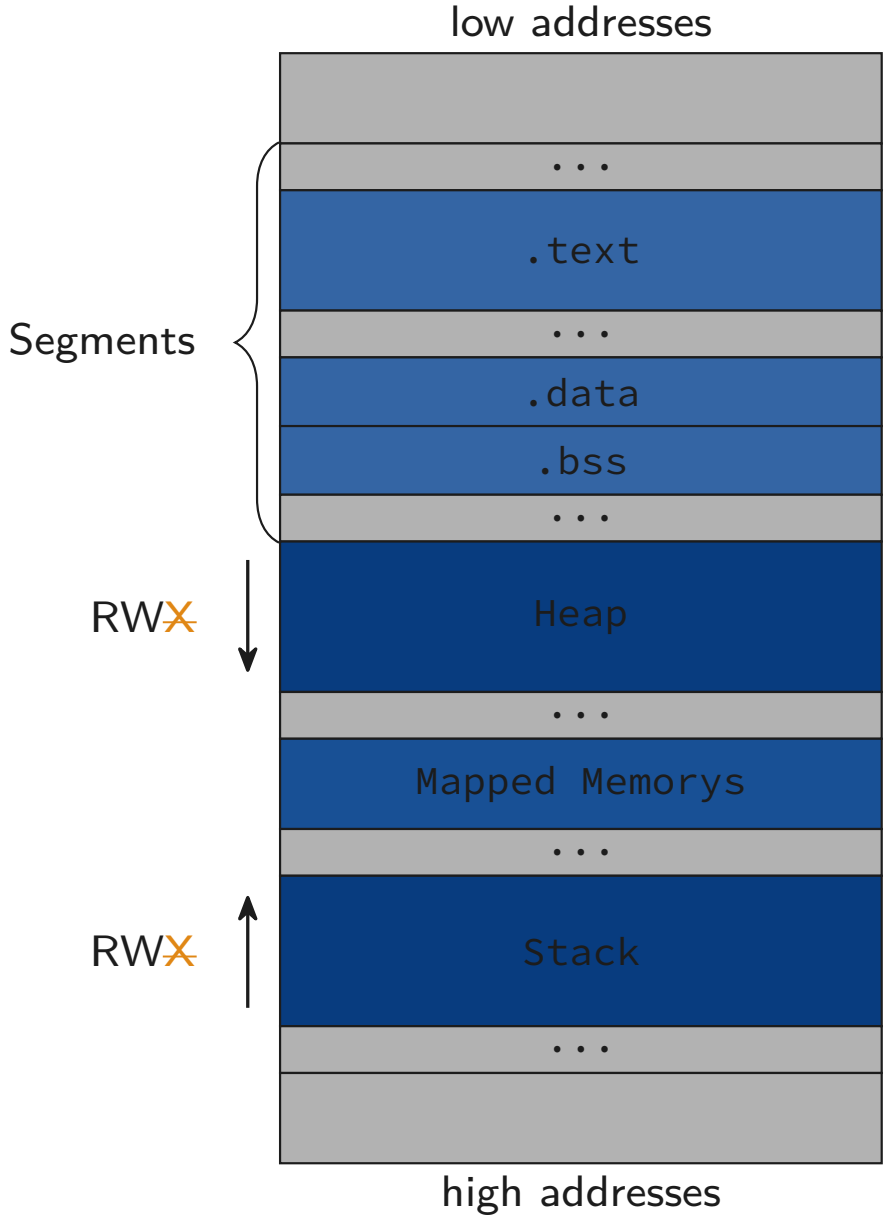
Example - Buffer Overflow



Return Oriented Programming

XN - Introduction

- Introduced by AMD
  - NX - No eXecute
- ARM introduced XN with ARMv6
  - XN - eXecute Never
- Additional bit in page table entry
- Known as
  - DEP
  - XN/NX/XD
  - W xor X



Return Oriented Programming

- Code reuse approach
- Use of small pieces of code called gadgets
- On ARMv7, gadgets end with a branch or pop instruction
  - `bx lr`
  - `pop {reg1, reg2, ..., regN, pc}`
- On ARMv8, gadgets can only end with `ret`
- It is important that `pc` is restored/loaded at the end of a gadget
- ROP chain consists of addresses to gadgets, chain of gadgets
- Each gadget is called by `ret` of the previously gadget

```
ldp x29, x30, [sp], #0x20;
ret
```

```
subs x0, x10, x9;
ret
```

Return Oriented Programming

Where to find gadgets?

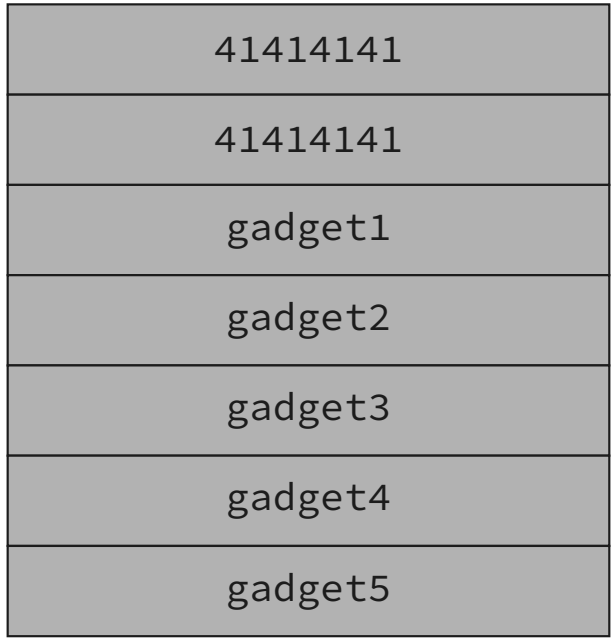
At least at the end of each function.

Different to ARMv7 and x86, gadgets can only be found there.

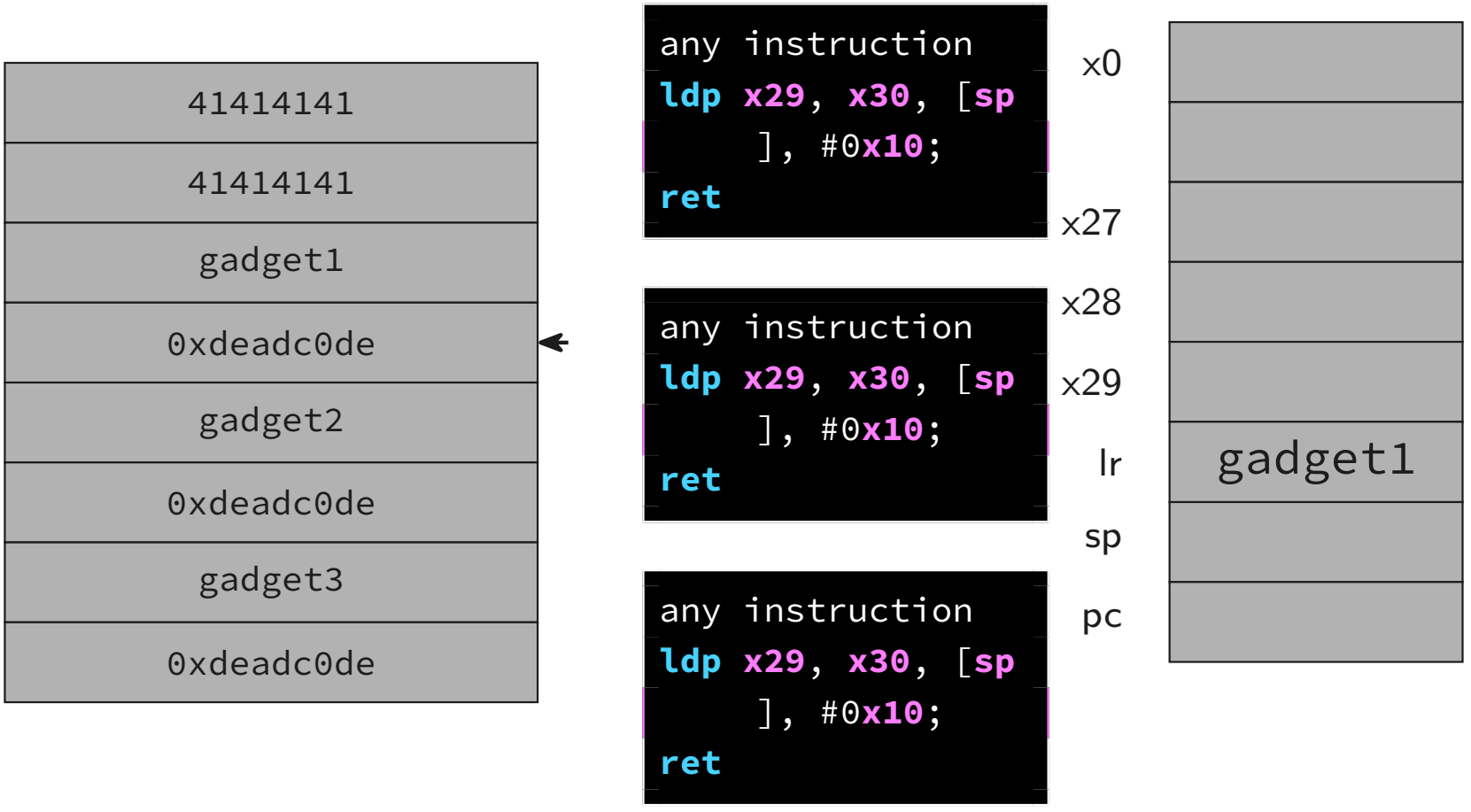
Return Oriented Programming

```
adds x3, x3, #1
ldp x29, x30, [sp], #0x10;
ret
```

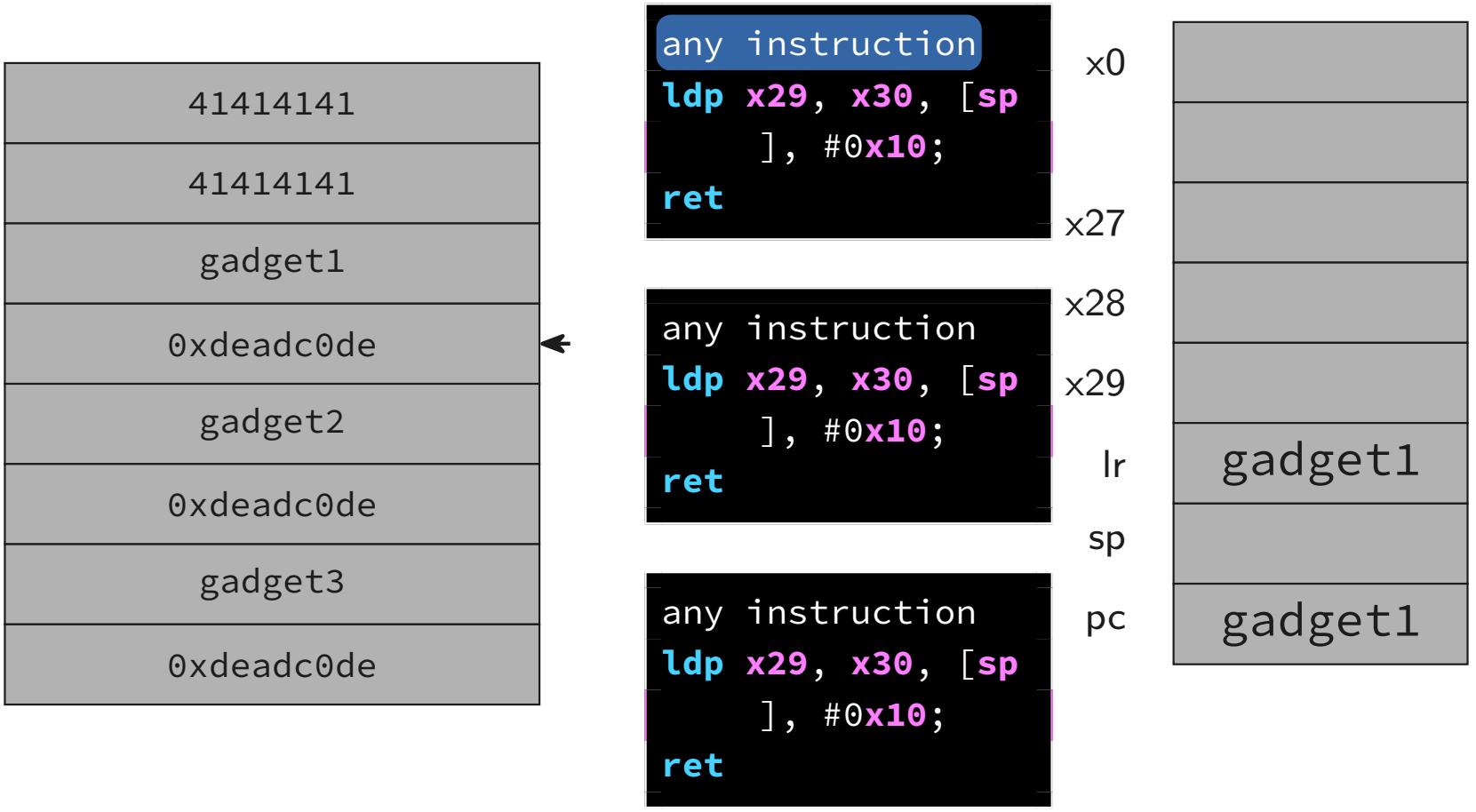
ROP Chain Example



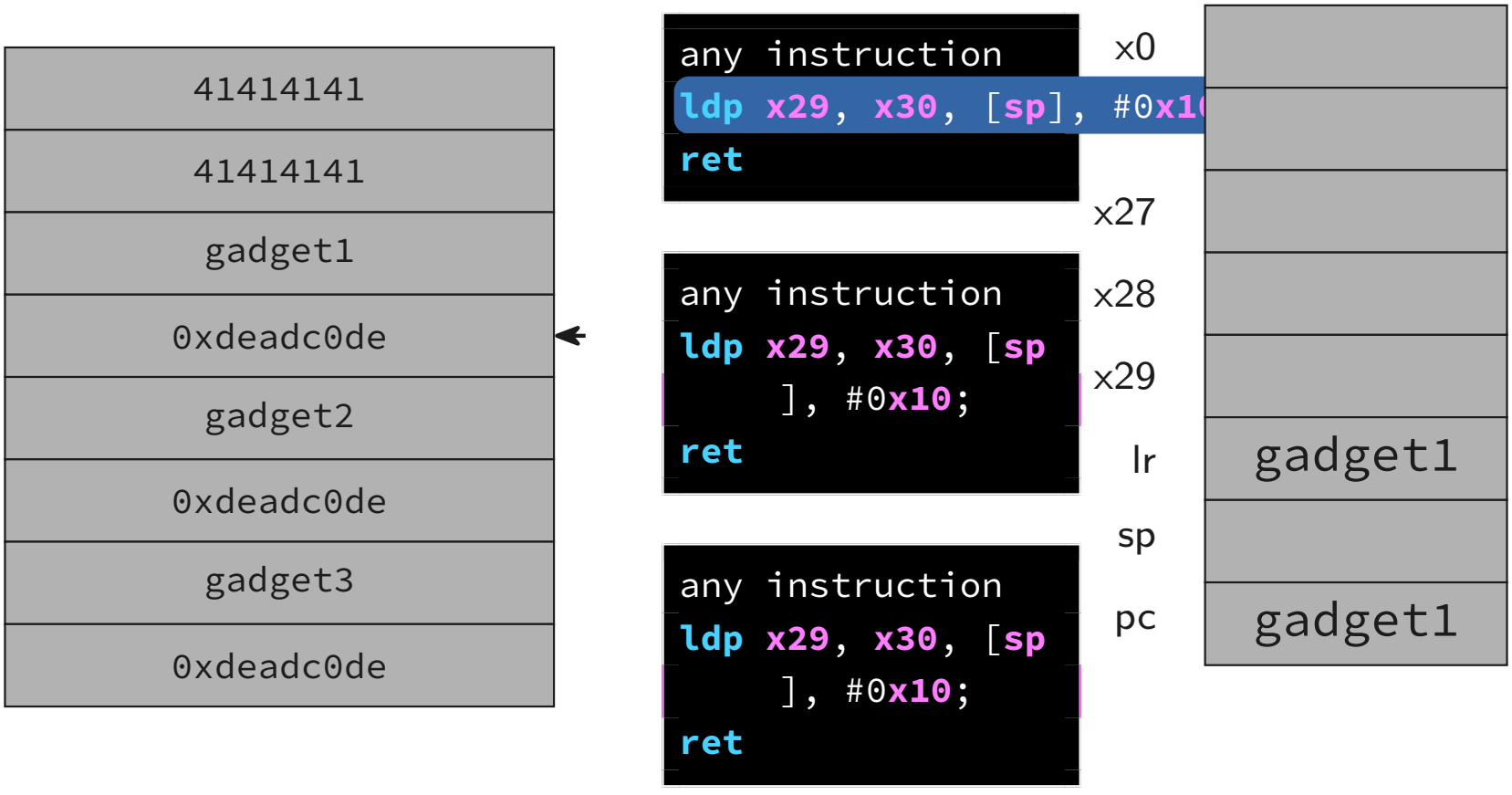
ROP Chain Example



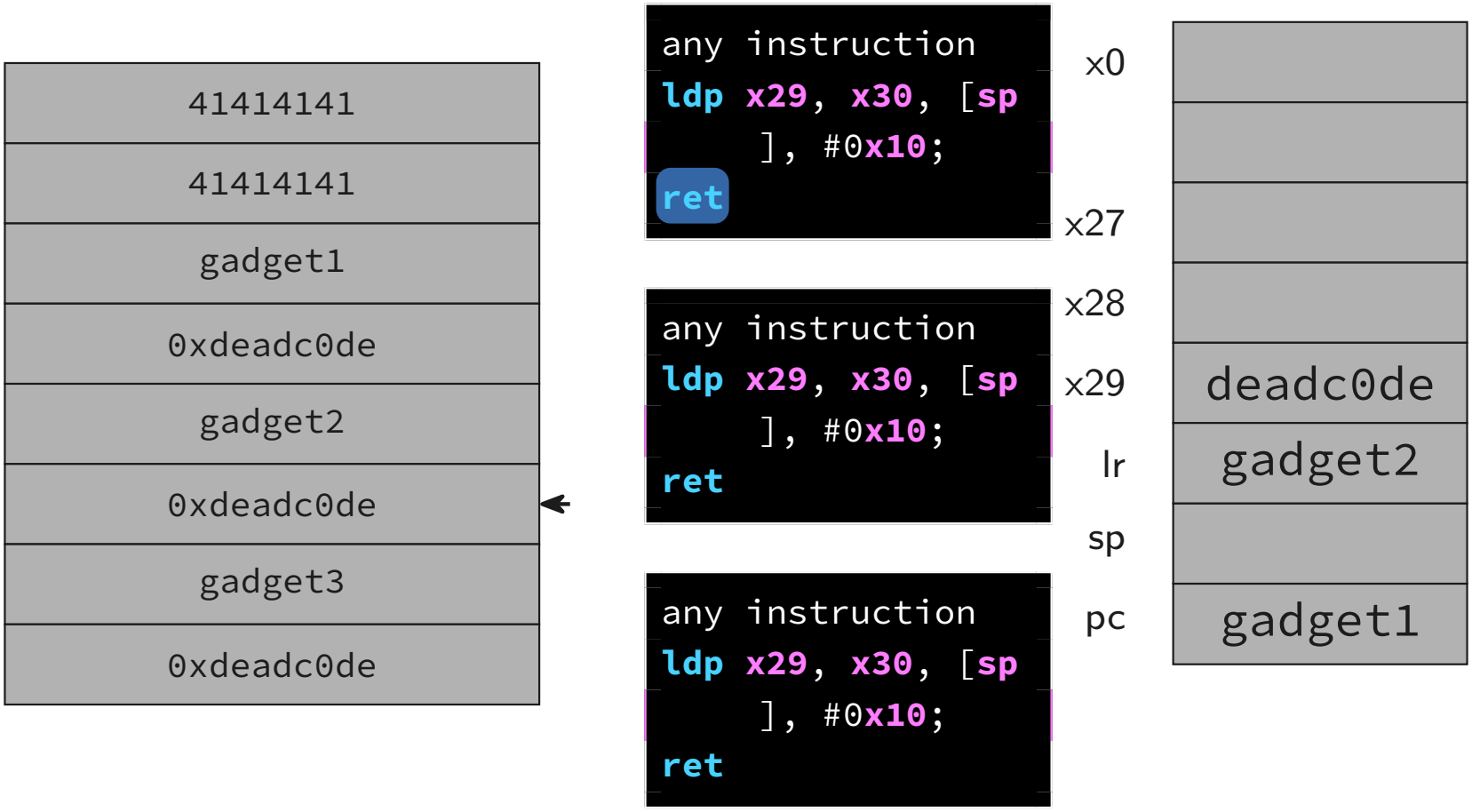
ROP Chain Example



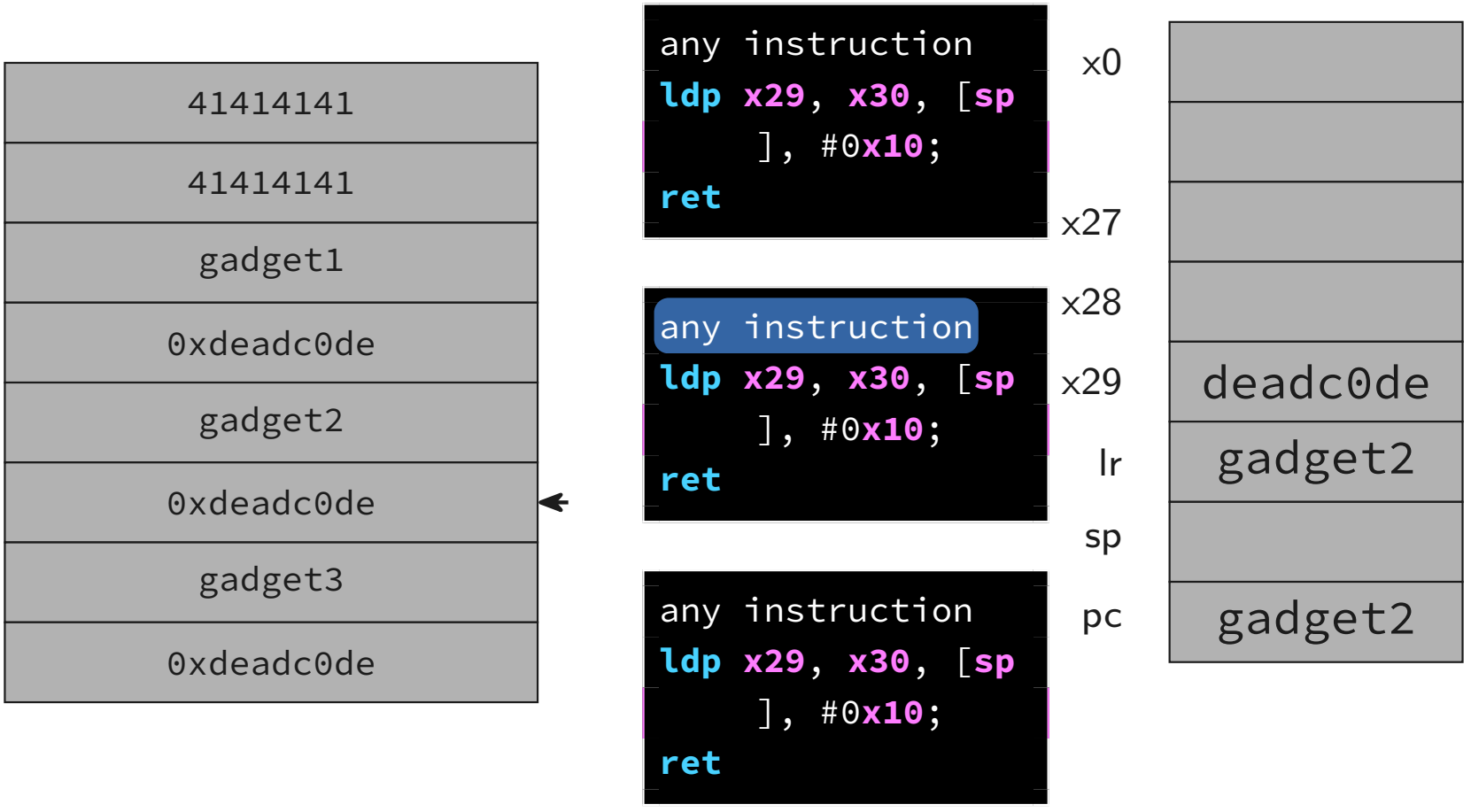
ROP Chain Example



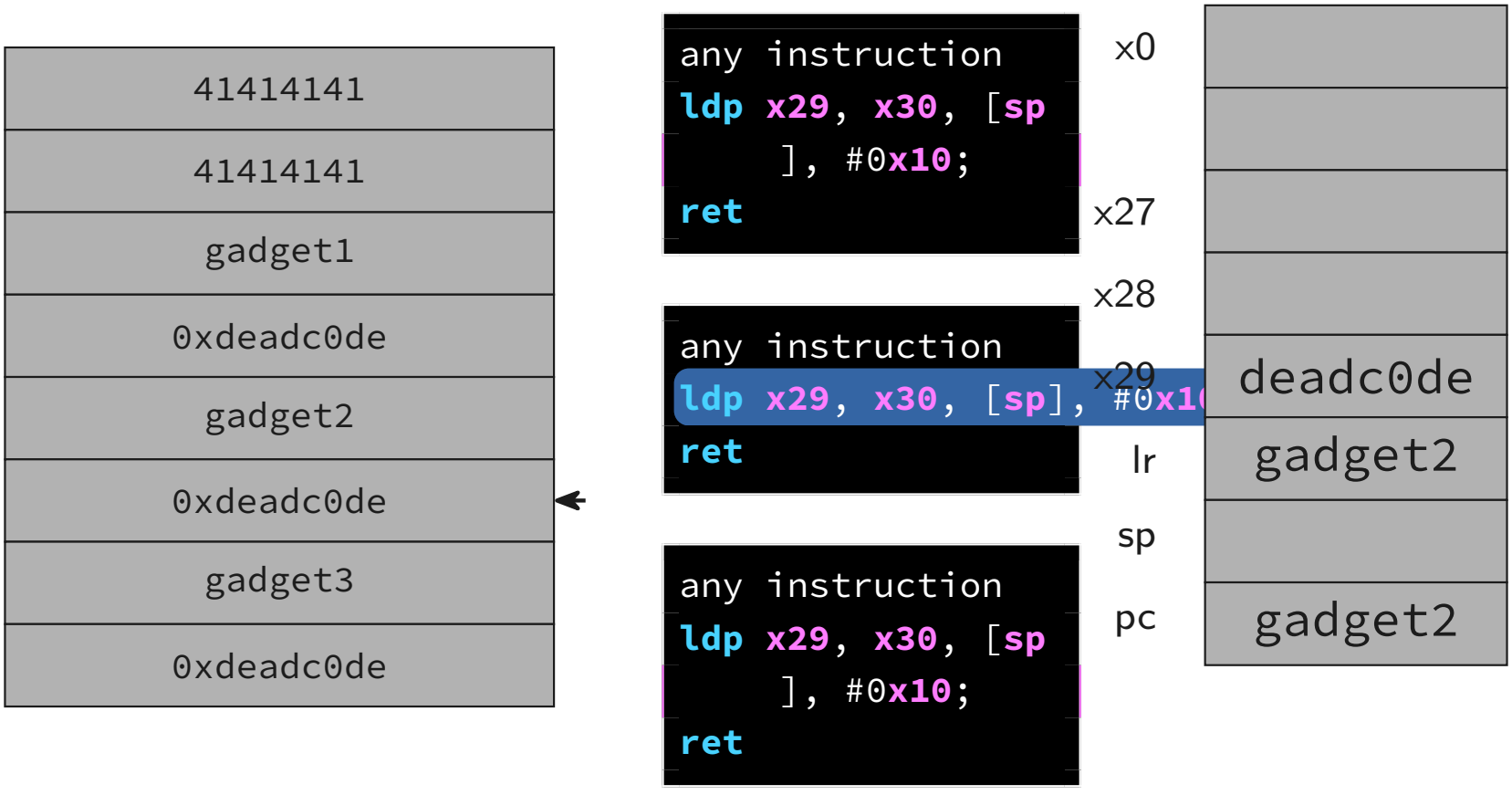
ROP Chain Example



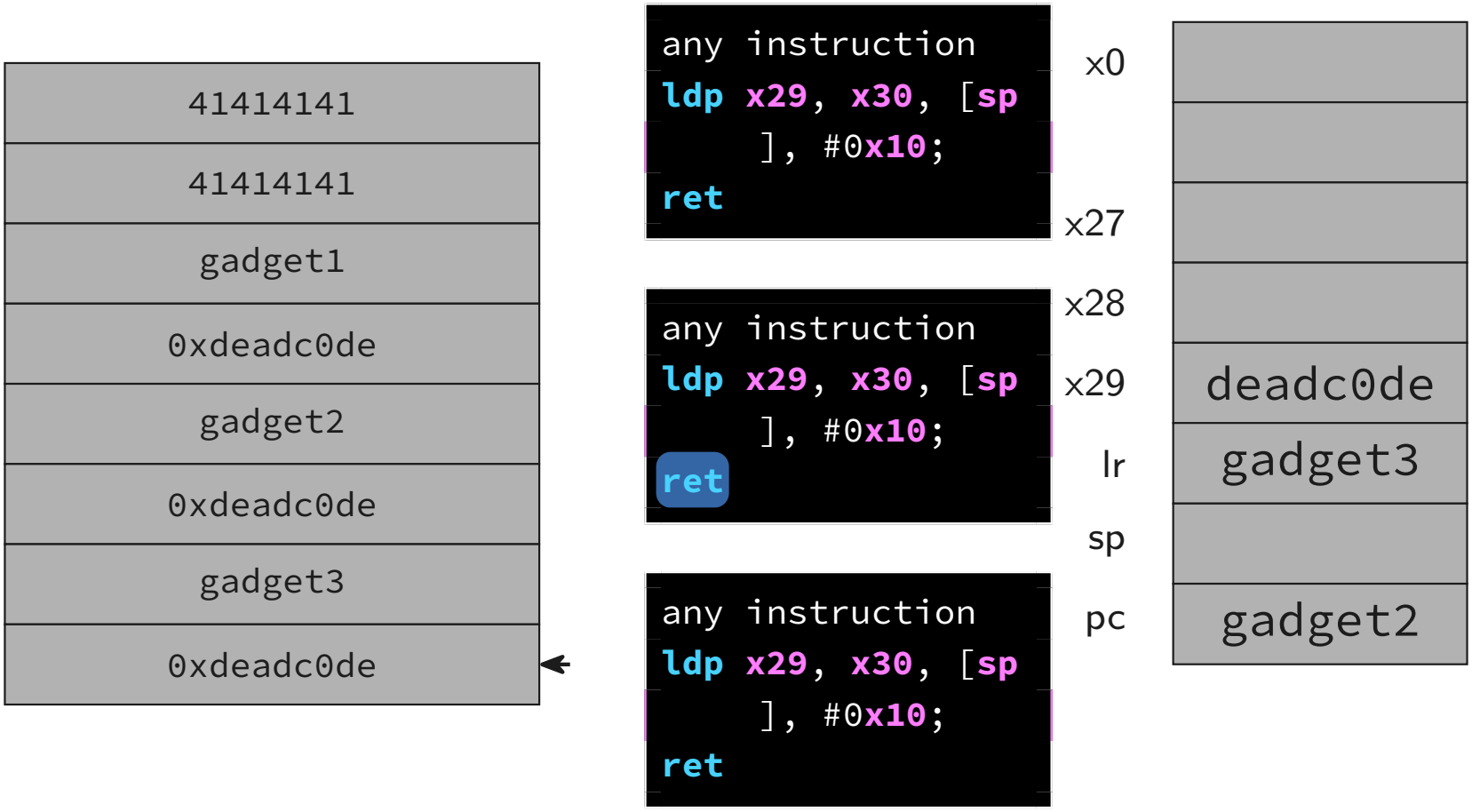
ROP Chain Example



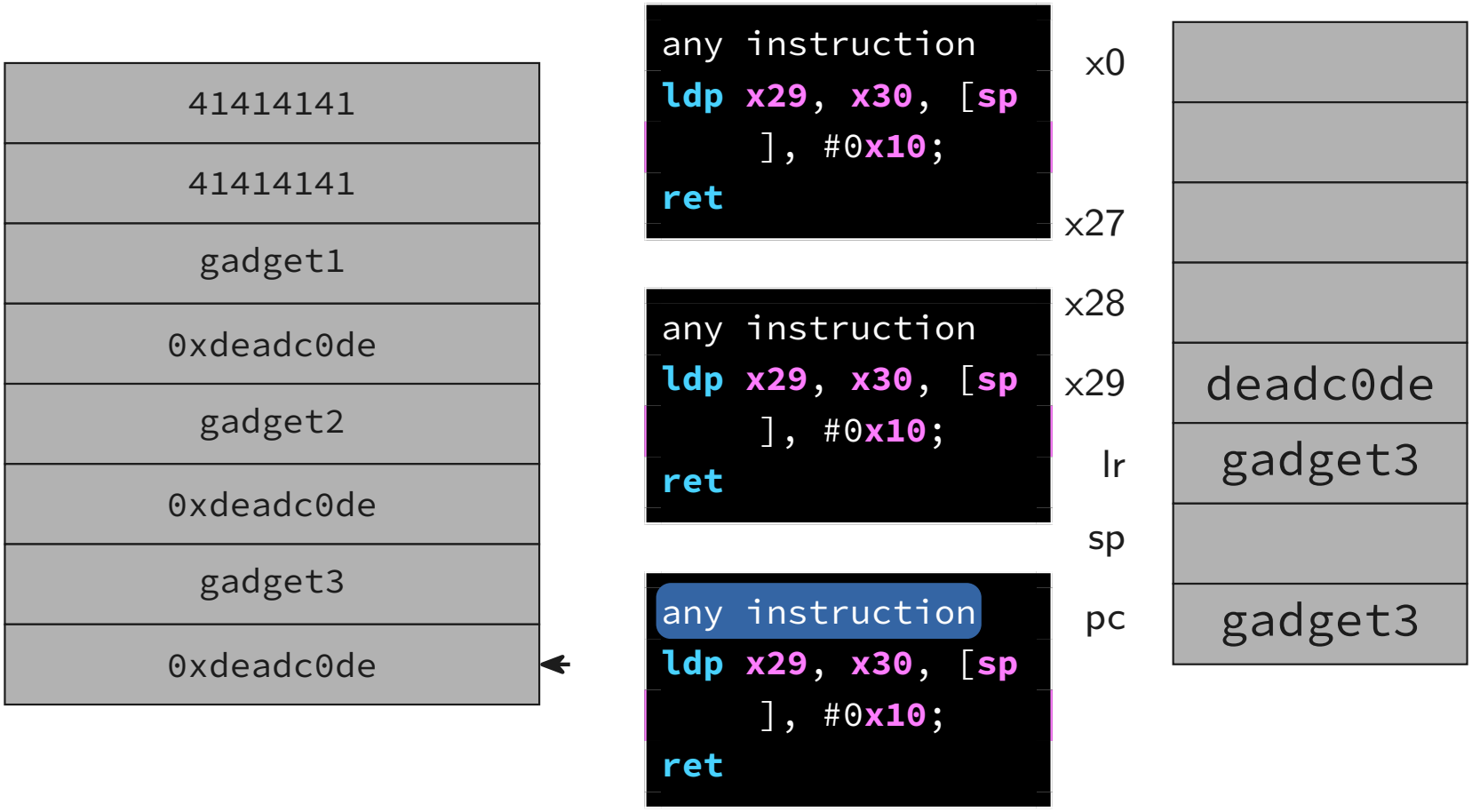
ROP Chain Example



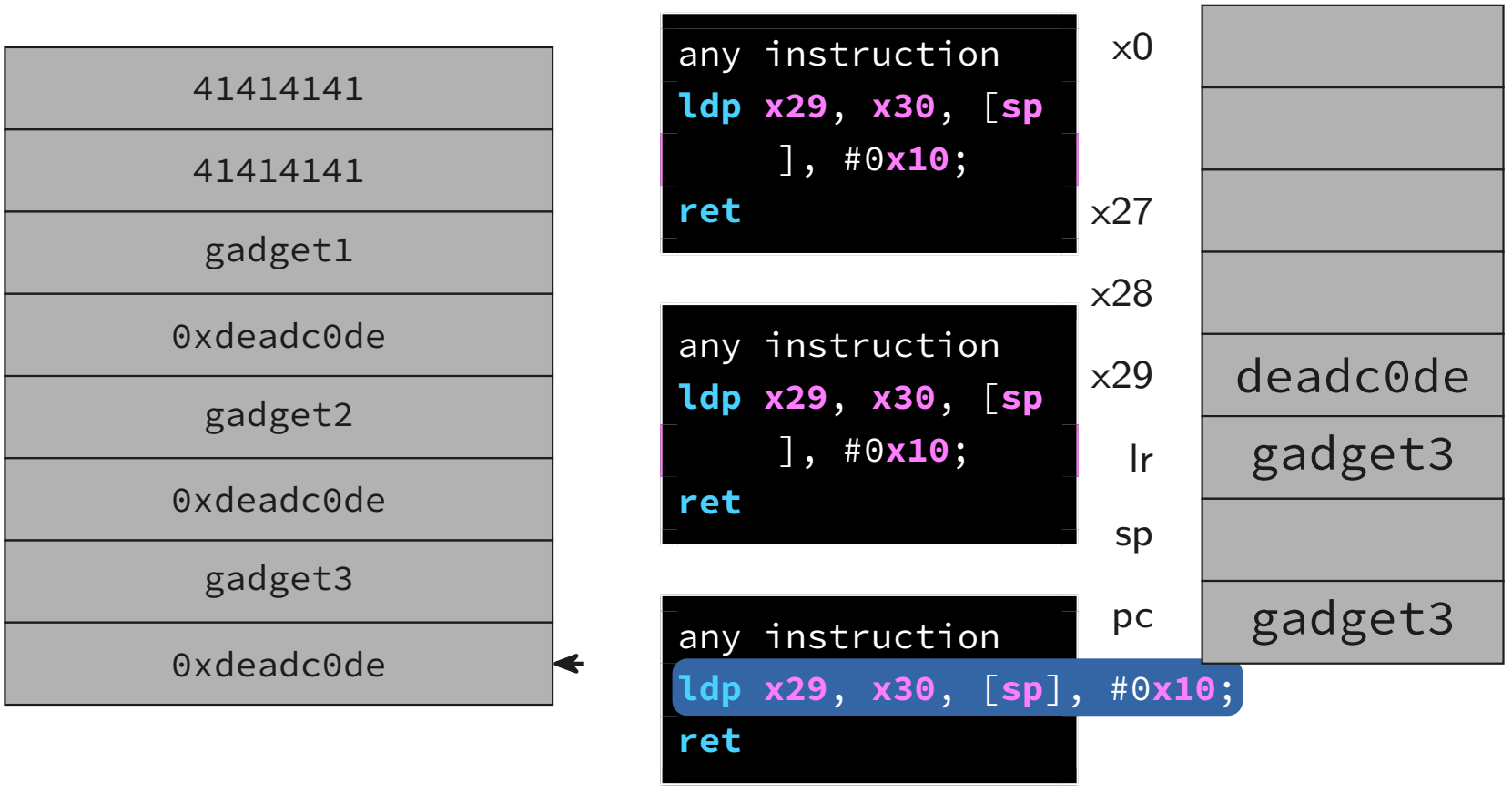
ROP Chain Example



ROP Chain Example



ROP Chain Example



Jump Oriented Programming

Jump Oriented Programming

- Similar to ROP
- Gadgets ends with `br <reg>` or `blr <reg>`
- `<reg>` has to prepared with a gadget before

```
mov x0, x26
blr x3
```

```
add x2, x2, x3
blr x5
```

How to find Gadgets

Several tools available:

- ropper
  - https://github.com/sashs/ropper
- ropgadget
  - https://github.com/JonathanSalwan/ROPgadget # ROP on Aarch64

ROP on Aarch64

`x0`, `x1`, `x3`, ... need to be prepared for function calls or callinc syscalls

Values mostly need to be read from the stack.

Mostly no gadgets that set those registers with values from stack:

```
ldp x0, x1, [sp, #0x10]
ldp x29, x30, [sp], #0x20
ret
```

ROP on Aarch64

`x0`, `x1`, `x3`, ... need to be prepared for function calls or callinc syscalls

Values mostly need to be read from the stack.

It is possible to find that kind of gadgets

```
ldp x20, x21, [sp, #0x10]
ldp x29, x30, [sp], #0x20
ret
```

ROP on Aarch64

`x0`, `x1`, `x3`, ... need to be prepared for function calls or callinc syscalls

Values mostly need to be read from the stack.

It is possible to find that kind of gadgets

```
ldp x20, x21, [sp, #0x10]
ldp x29, x30, [sp], #0x20
ret
```

```
mov x0, x20
blr x3
```

ROP on Aarch64

- ROP and JOP is necessary
- Prepare higher registers
- Move values to lower registers
- Register of `br` or `blr` instruction has to prepared as well

```
ldp x20, x21, [sp, #0x10]
ldp x29, x30, [sp], #0x20
ret
```

```
mov x0, x20
blr x3
```

Vulnerabilities

- ASLR, StackCanaries, CFI and other mitigations
  - Requires another vulnerability
- Mostly in the heap
  - Double Free
  - Use After Free
  - Buffer Overflows
- ROP relies on the sp
- Only one gadget can be executed

Stack Pivot

- Necessary to move sp to the memory region which can be controlled
  - mostly heap
- Stack pivot is necessary
- Gadget that moves sp to that memory region

```
mov sp, x20
ldp x29, x30, [sp], #0x10
ret
```

Example

Example

- Android app
  - Using JNI (native code)
- Connects to REST service
- Get JSON data
- Parse JSON data and call native function

Example

```
{
    length: 123
    message: <base64 encoded>
}
```

Example

```
JNIEXPORT jstring JNICALL
Java_de_scoding_ropyourway_MainActivity_parseMessage(
        JNIEnv* env,
        jobject /* this */,
        jbyteArray message, jint length){

    jboolean isCopy;
    char buffer[32];
    jstring string = env->NewString(reinterpret_cast<const jchar *>("")
         , 5);

    char * nativeString = (char*)env->GetByteArrayElements( message, &
         isCopy);
    memcpy(buffer, nativeString, length);
return string;
}
```

Example

- Classic buffer overflow on the stack
- Return address will be overwritten

Example - Exploit

- Call `system` with a ROP chain to execute a command
- `x0` has to be prepared with the argument
  - Address to the command
  - Gadget that moves `sp` to `x0`
- Place the command on the stack

```
mov x0, sp
blr x??
```

Example - Exploit