Reverse Engineering

Class 10

Exploit Writing III

Return Oriented Programming (ROP)

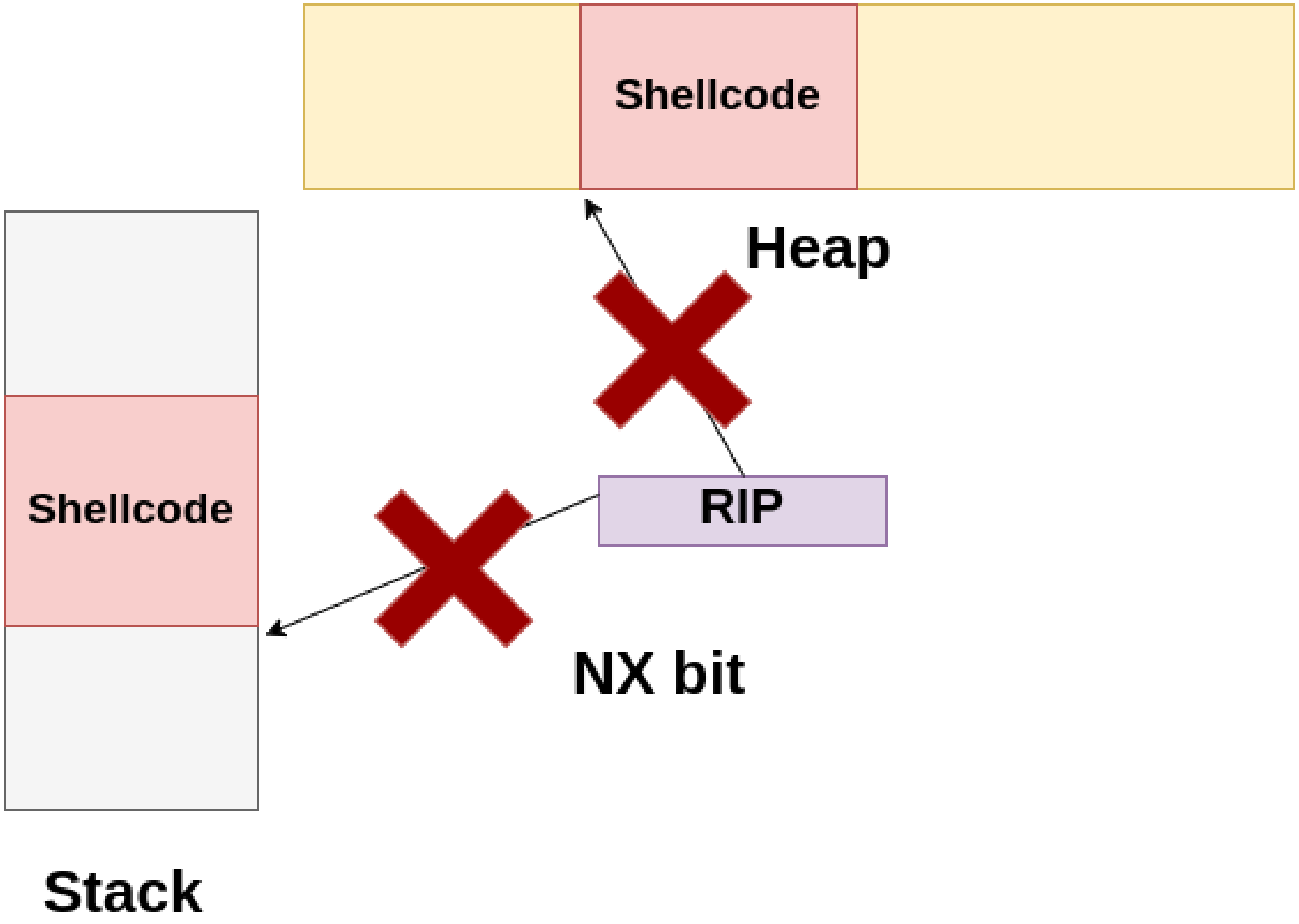
martin.uy

Open by default.

ROP

- ROP: Return Oriented Programming
    - RIP (instruction pointer) is controlled, but:
    - It’s not possible to jump to execute shellcode in the stack, data or heap
        - Data not executable anymore (DEP → Data Execution Prevention)
        - NX bit (x86)
        - This applies to both kernel and user space

ROP



ROP

- NX bit (kernel, x86_64)

```
#define _PAGE_BIT_NX    63 /* No execute: only valid
after cpuid check */

#define _PAGE_NX  (_AT(pteval_t, 1) << _PAGE_BIT_NX)
```

arch/x86/include/asm/pgtable_types.h

```
static inline pte_t pte_mkexec(pte_t pte)
{
   return pte_clear_flags(pte, _PAGE_NX);
}
```

arch/x86/include/asm/pgtable.h

ROP

- NX bit (kernel, x86_64)

```
typedef unsigned long   pteval_t;

typedef struct { pteval_t pte; } pte_t;

arch/x86/include/asm/pgtable_64_types.h
```

ROP

- Stack allocation (kernel, x86_64)

```
stack = __vmalloc_node_range(THREAD_SIZE, THREAD_SIZE,
                VMALLOC_START, VMALLOC_END,
                THREADINFO_GFP | __GFP_HIGHMEM,
                PAGE_KERNEL,
                0, node, __builtin_return_address(0));
```

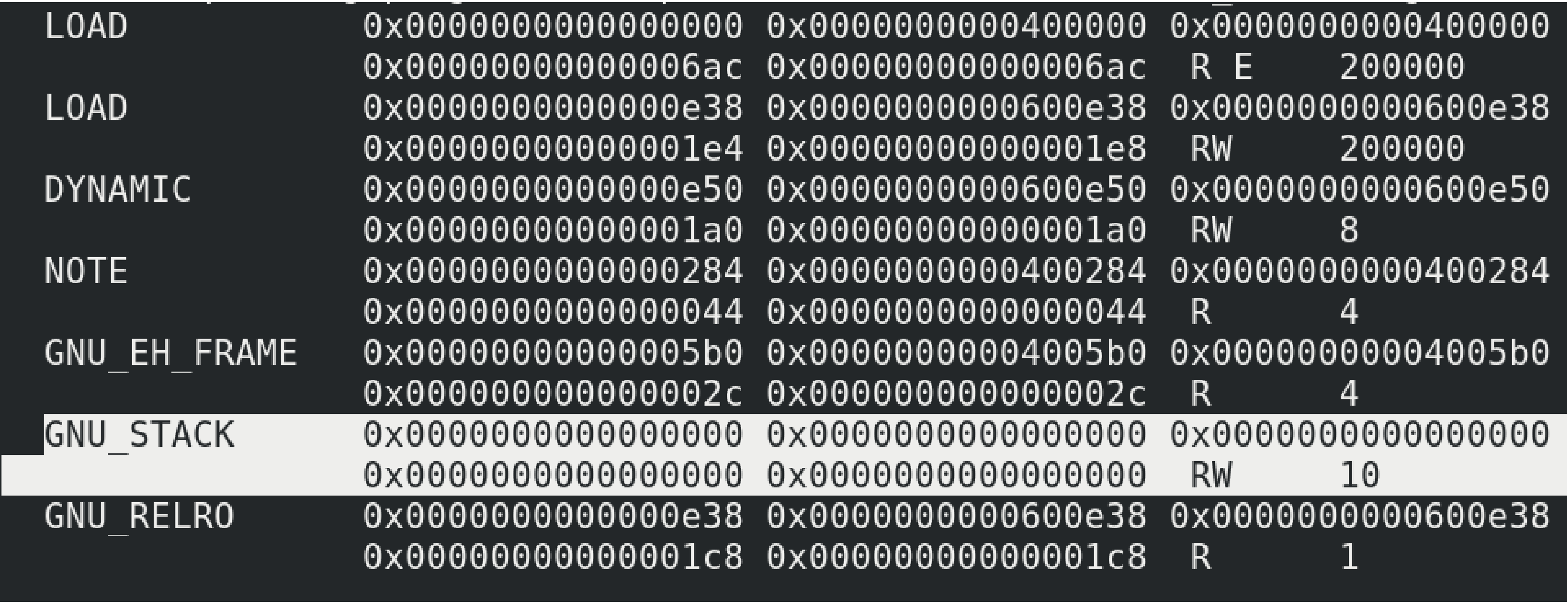
fork.c

```
#define PAGE_KERNEL           __pgprot(__PAGE_KERNEL)

#define __PAGE_KERNEL         (__PAGE_KERNEL_EXEC |
_PAGE_NX)
```

arch/x86/include/asm/pgtable_types.h

ROP

- Stack allocation user main thread (kernel, x86_64)

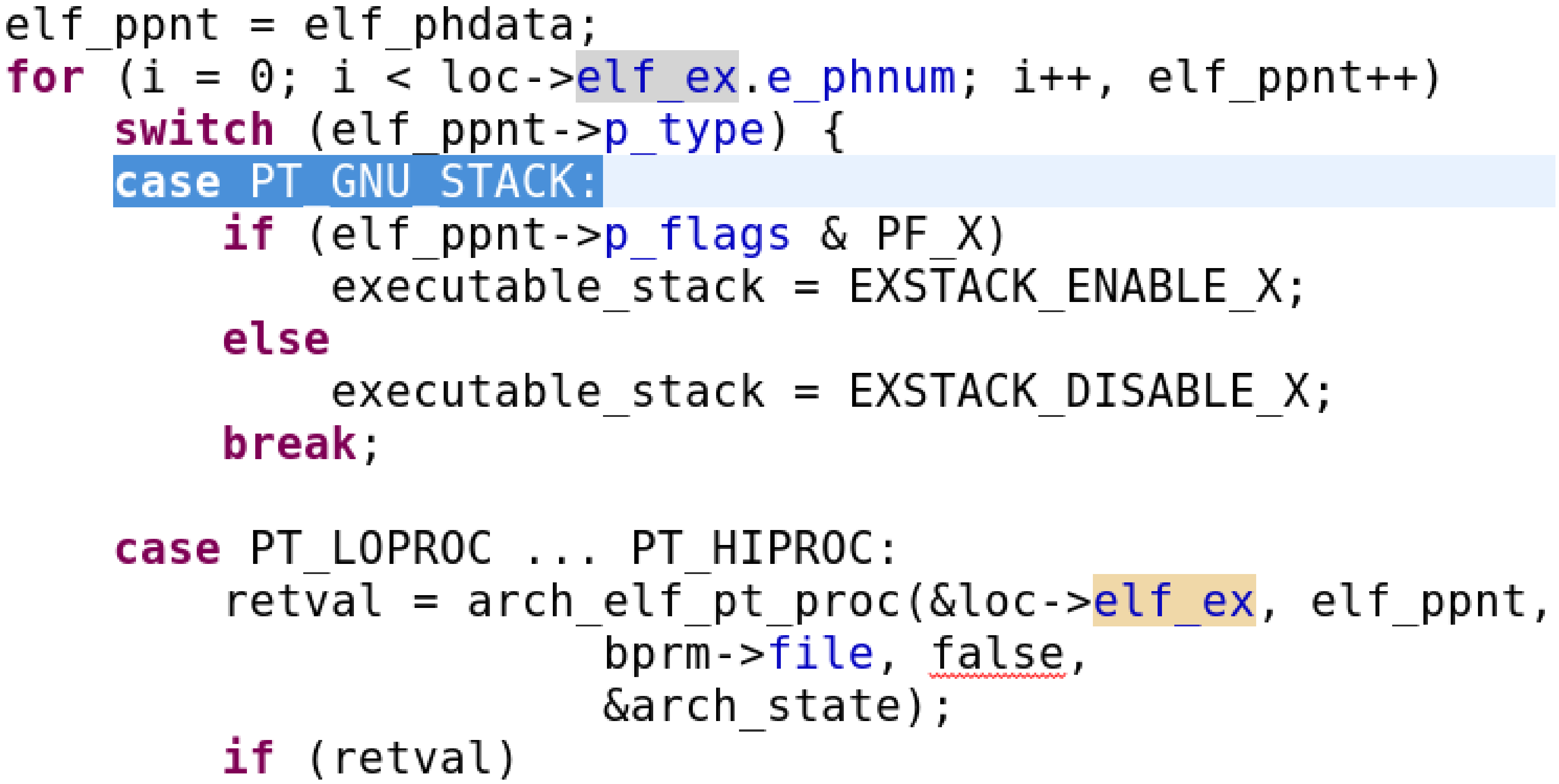


```
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000006ac 0x00000000000006ac  R E    200000
  LOAD           0x0000000000000e38 0x0000000000600e38 0x0000000000600e38
                 0x00000000000001e4 0x00000000000001e8  RW     200000
  DYNAMIC        0x0000000000000e50 0x0000000000600e50 0x0000000000600e50
                 0x00000000000001a0 0x00000000000001a0  RW     8
  NOTE           0x0000000000000284 0x0000000000400284 0x0000000000400284
                 0x0000000000000044 0x0000000000000044  R      4
  GNU_EH_FRAME   0x00000000000005b0 0x00000000004005b0 0x00000000004005b0
                 0x000000000000002c 0x000000000000002c  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000e38 0x0000000000600e38 0x0000000000600e38
                 0x00000000000001c8 0x00000000000001c8  R      1
```

`GNU_STACK section (from “main” binary) has flags RW on`

ROP

- Stack allocation user main thread (kernel, x86_64)

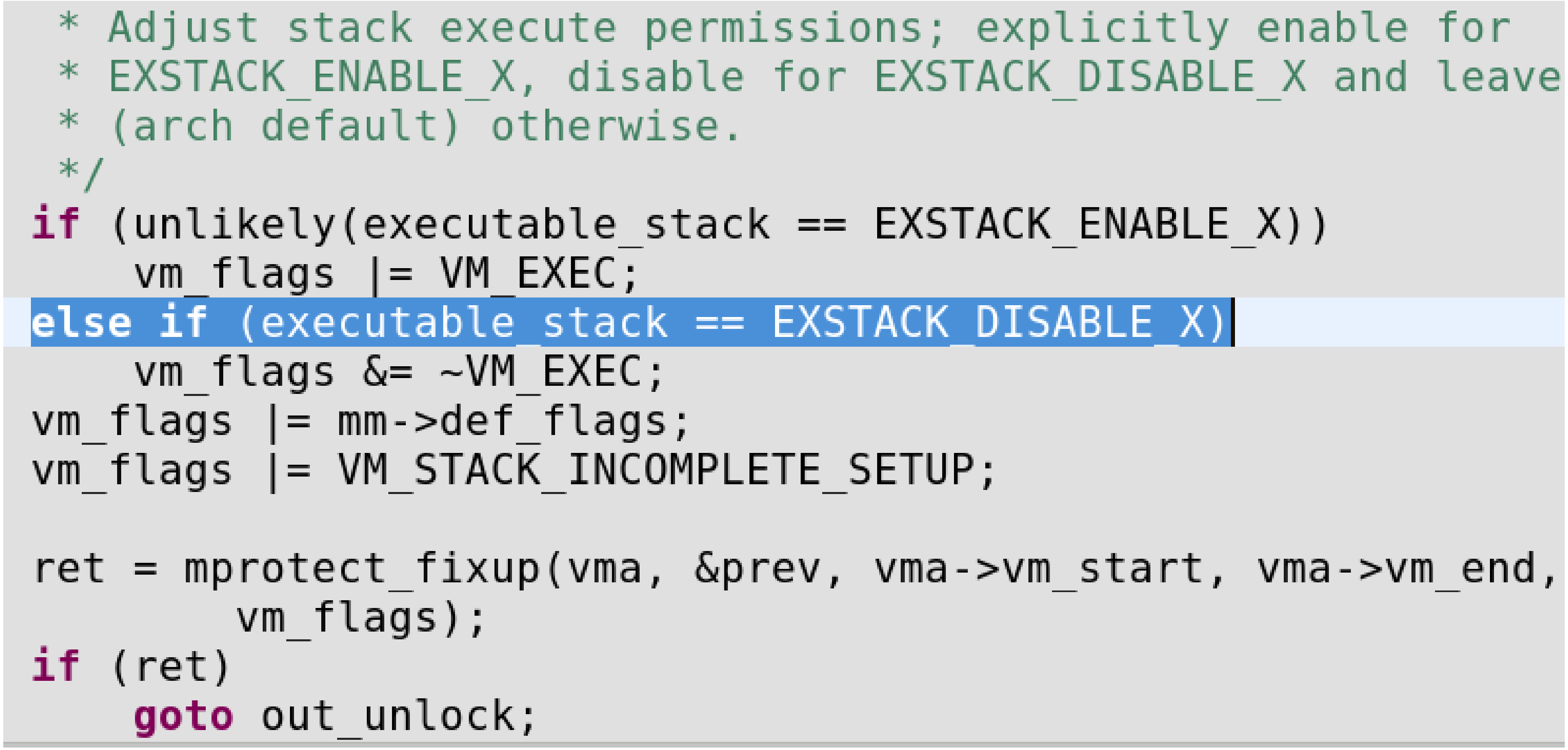


```
elf_ppnt = elf_phdata;
for (i = 0; i < loc->elf_ex.e_phnum; i++, elf_ppnt++)
    switch (elf_ppnt->p_type) {
    case PT_GNU_STACK:
        if (elf_ppnt->p_flags & PF_X)
            executable_stack = EXSTACK_ENABLE_X;
        else
            executable_stack = EXSTACK_DISABLE_X;
        break;

    case PT_LOPROC ... PT_HIPROC:
        retval = arch_elf_pt_proc(&loc->elf_ex, elf_ppnt,
                                  bprm->file, false,
                                  &arch_state);
        if (retval)
```

fs/binfmt_elf.c (Linux kernel)

ROP

- Stack allocation user main thread (kernel, x86_64)



```
 * Adjust stack execute permissions; explicitly enable for
 * EXSTACK_ENABLE_X, disable for EXSTACK_DISABLE_X and leave
 * (arch default) otherwise.
 */
if (unlikely(executable_stack == EXSTACK_ENABLE_X))
    vm_flags |= VM_EXEC;
else if (executable_stack == EXSTACK_DISABLE_X)
    vm_flags &= ~VM_EXEC;
vm_flags |= mm->def_flags;
vm_flags |= VM_STACK_INCOMPLETE_SETUP;

ret = mprotect_fixup(vma, &prev, vma->vm_start, vma->vm_end,
        vm_flags);
if (ret)
    goto out_unlock;
```

fs/exec.c (Linux kernel)

ROP

- Stack allocation user (glibc, x86_64)

```
static int
allocate_stack (const struct pthread_attr *attr, struct
pthread **pdp,
      ALLOCATE_STACK_PARMS)
{
...
const int prot = (PROT_READ | PROT_WRITE
         | ((GL(dl_stack_flags) & PF_X) ? PROT_EXEC :
0));
...
mem = mmap (NULL, size, prot,
            MAP_PRIVATE | MAP_ANONYMOUS | MAP_STACK, -1,
0);
```

nptl/allocatestack.c

ROP

- Return to libc
  - Call *system* (Linux) or *WinExec* (Windows)
    - Invoke a command or application (I.e. shell)
  - Call *dlopen* (Linux) or *LoadLibrary* (Windows)
    - Execute code when library is loaded
  - In x86, a memory corruption on the stack may allow control of all parameters for these calls (ABI)

ROP

- Return to libc
  - In x86_64, ABI requires to load registers to send parameters to a function
  - Virtual address space randomization (ASLR): in which virtual addresses are *system*, *dlopen*, *WinExec* and *LoadLibrary* functions located?

ROP

- Return to libc
  - Return to strcpy/memcpy/sprintf/etc
  - Copy shellcode to a writable and executable location
  - W^X: protection against writable and executable segments

Lab

Exercise 10.1: return to Libc

ROP

- Return Oriented Programming (ROP)
  - Control of the stack is required to do ROP
    - Pivot the stack to a controlled area if necessary
  - Concatenate multiple calls to short assembly sequences: gadgets
    - Each “call” is a return to what’s on the top of the stack
    - Gadgets end in a RET instruction (or an equivalent one) that allows to continue controlling the execution flow through the stack
    - Registers and memory state are conveniently modified in each call to a gadget

ROP

- Return Oriented Programming
    - Goals: unprotect memory (*mprotect* syscall in Linux or *VirtualProtect* in Windows) to jump to shellcode or execute a binary (*execve* syscall)
        - Another approach could be allocating new memory with write and execution permissions, and copy the payload to jump there

ROP

- Return Oriented Programming
  - In which address is shellcode located?
  - Example: stack randomization

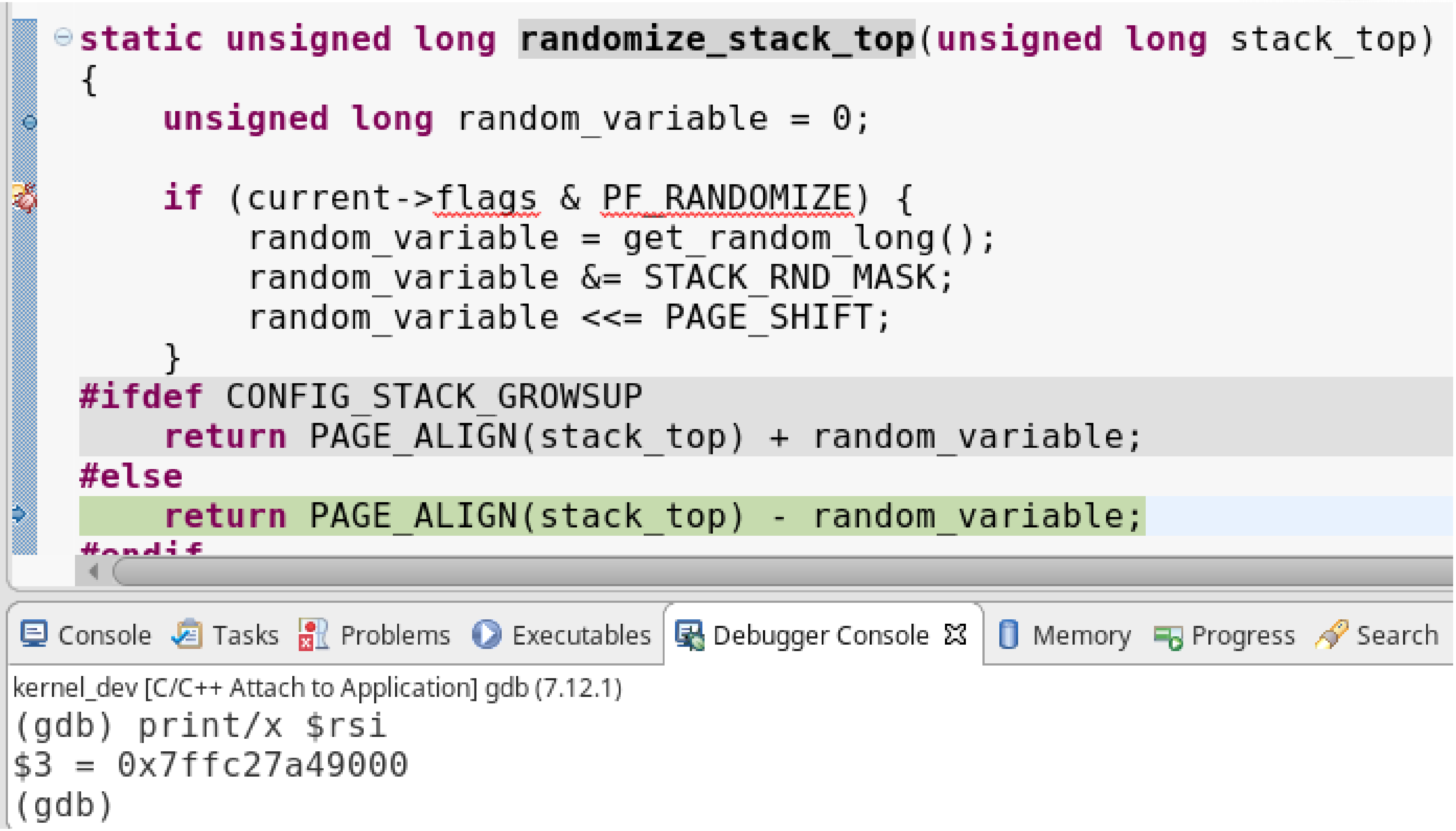
ROP

```
static unsigned long randomize_stack_top(unsigned long
stack_top)
{
   unsigned long random_variable = 0;

   if (current->flags & PF_RANDOMIZE) {
      random_variable = get_random_long();
      random_variable &= STACK_RND_MASK;
      random_variable <<= PAGE_SHIFT;
   }
#ifdef CONFIG_STACK_GROWSUP
   return PAGE_ALIGN(stack_top) + random_variable;
#else
   return PAGE_ALIGN(stack_top) - random_variable;
#endif
}
```

fs/binfmt_elf.c (Linux kernel)

ROP

Run 1: /usr/bin/ls



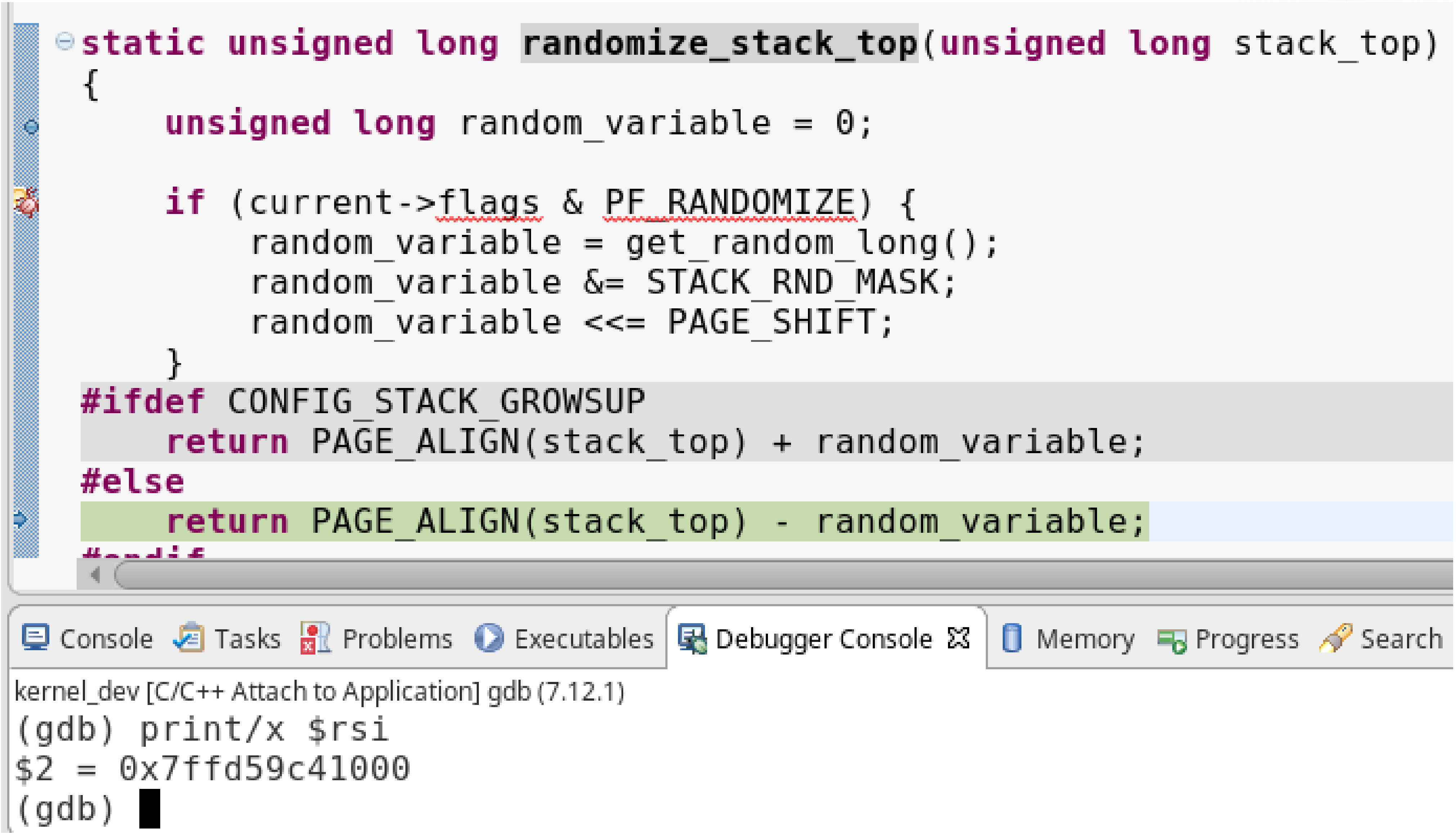
```
static unsigned long randomize_stack_top(unsigned long stack_top)
{
	unsigned long random_variable = 0;

	if (current->flags & PF_RANDOMIZE) {
		random_variable = get_random_long();
		random_variable &= STACK_RND_MASK;
		random_variable <<= PAGE_SHIFT;
	}
#ifdef CONFIG_STACK_GROWSUP
	return PAGE_ALIGN(stack_top) + random_variable;
#else
	return PAGE_ALIGN(stack_top) - random_variable;
#endif
```

kernel_dev [C/C++ Attach to Application] gdb (7.12.1)

```
(gdb) print/x $rsi
$3 = 0x7ffc27a49000
(gdb)
```

ROP

Run 2: /usr/bin/ls



```
static unsigned long randomize_stack_top(unsigned long stack_top)
{
	unsigned long random_variable = 0;

	if (current->flags & PF_RANDOMIZE) {
		random_variable = get_random_long();
		random_variable &= STACK_RND_MASK;
		random_variable <<= PAGE_SHIFT;
	}
#ifdef CONFIG_STACK_GROWSUP
	return PAGE_ALIGN(stack_top) + random_variable;
#else
	return PAGE_ALIGN(stack_top) - random_variable;
#endif
```

kernel_dev [C/C++ Attach to Application] gdb (7.12.1)

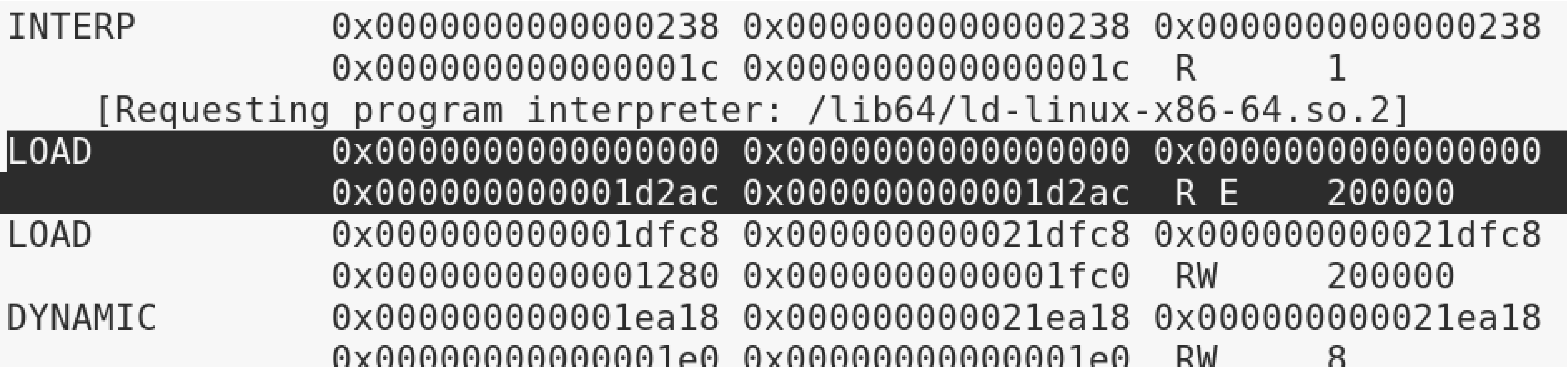
```
(gdb) print/x $rsi
$2 = 0x7ffd59c41000
(gdb) 
```

ROP

- Return Oriented Programming
  - In which address is shellcode located?
    - A ptr leak or a heap spray may be necessary
  - In which addresses are gadgets located?
    - Mapped libraries may be randomized (PIC) but some are not
    - Binary image may be randomized (PIE) or not

ROP

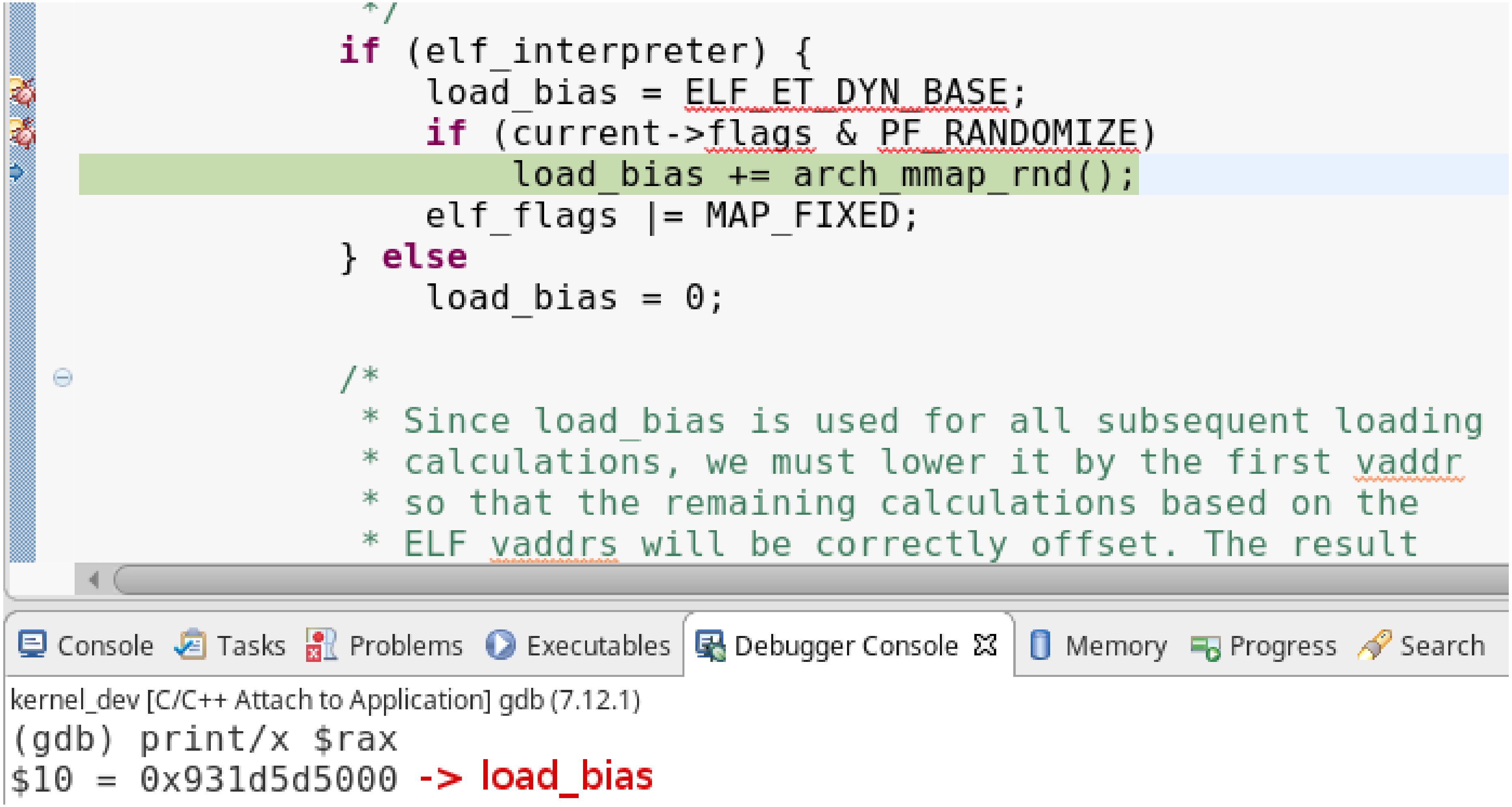
- Return Oriented Programming
  - In which addresses are gadgets located?
    - Example of Position Independent Executable (PIE): /usr/bin/ls (x86_64)



```
INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
               0x000000000000001c 0x000000000000001c  R      1
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x000000000001d2ac 0x000000000001d2ac  R E    200000
LOAD           0x000000000001dfc8 0x000000000021dfc8 0x000000000021dfc8
               0x0000000000001280 0x0000000000001fc0  RW     200000
DYNAMIC        0x000000000001ea18 0x000000000021ea18 0x000000000021ea18
               0x00000000000001e0 0x00000000000001e0  RW     8
```

ROP

Run 1: /usr/bin/ls



```
            */
            if (elf_interpreter) {
                load_bias = ELF_ET_DYN_BASE;
                if (current->flags & PF_RANDOMIZE)
                    load_bias += arch_mmap_rnd();
                elf_flags |= MAP_FIXED;
            } else
                load_bias = 0;

            /*
             * Since load_bias is used for all subsequent loading
             * calculations, we must lower it by the first vaddr
             * so that the remaining calculations based on the
             * ELF vaddrs will be correctly offset. The result
```

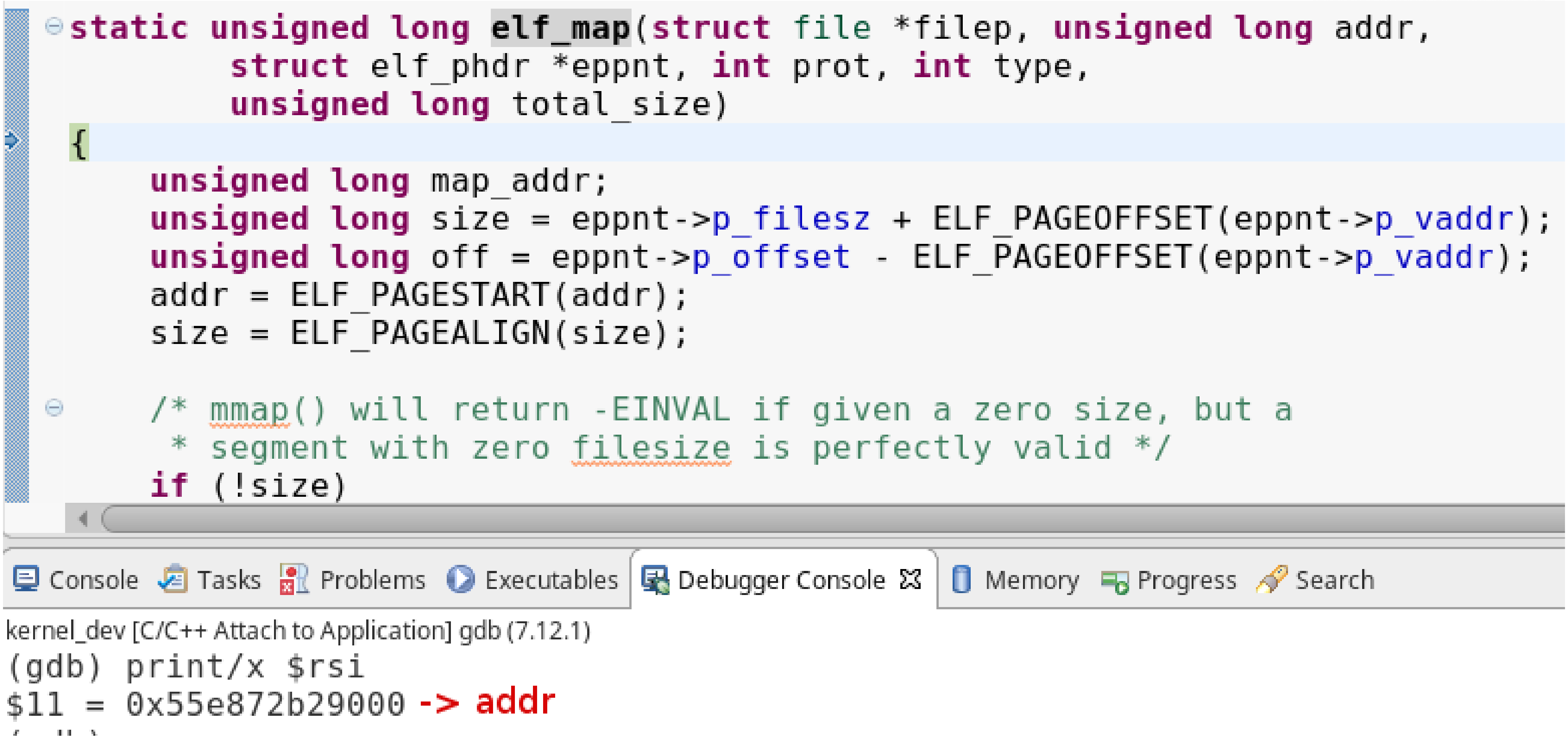
kernel_dev [C/C++ Attach to Application] gdb (7.12.1)

```
(gdb) print/x $rax
$10 = 0x931d5d5000 -> load_bias
```

fs/binfmt_elf.c (Linux kernel)

ROP

Run 1: /usr/bin/ls



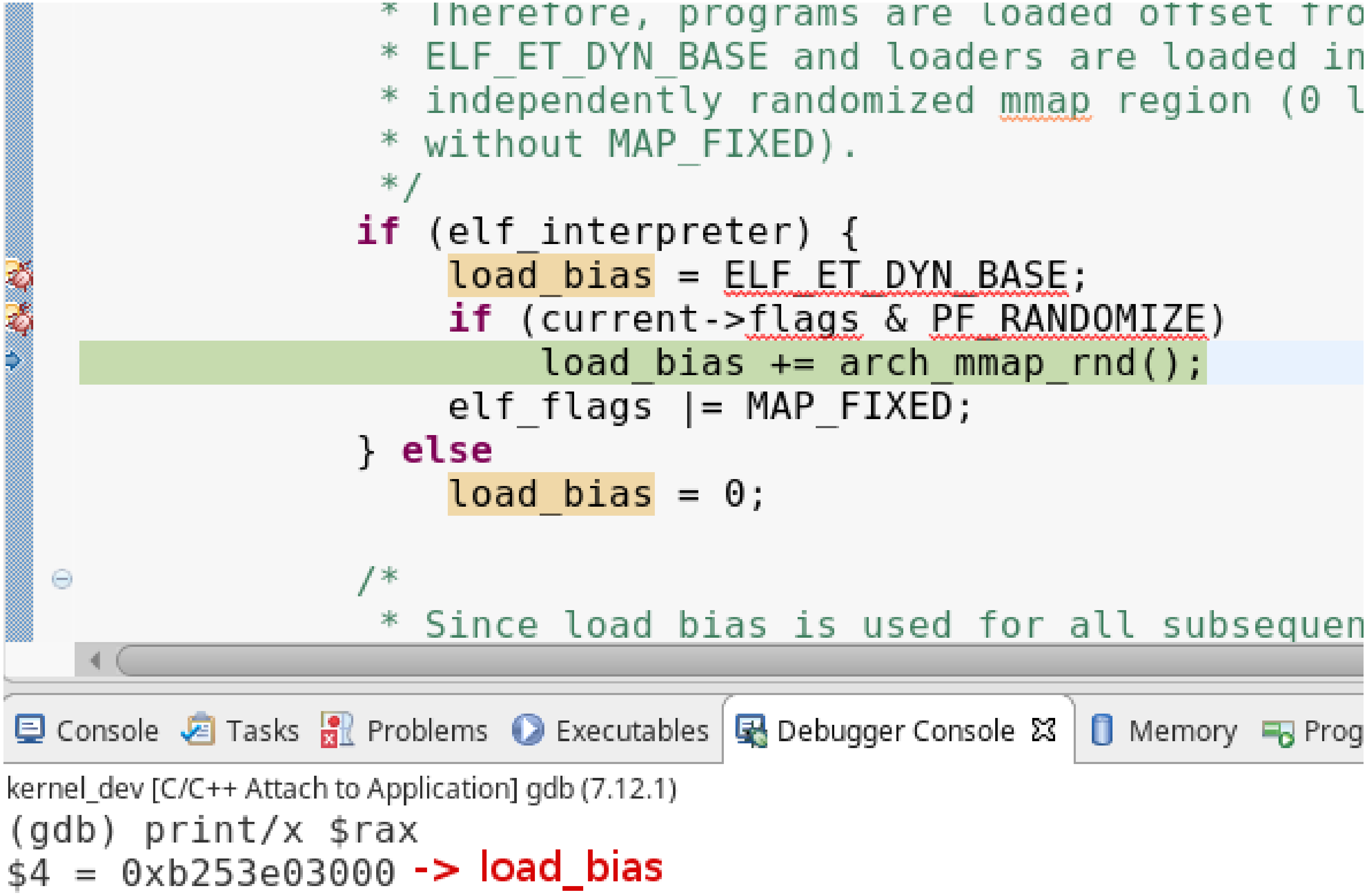
```
static unsigned long elf_map(struct file *filep, unsigned long addr,
        struct elf_phdr *eppnt, int prot, int type,
        unsigned long total_size)
{
    unsigned long map_addr;
    unsigned long size = eppnt->p_filesz + ELF_PAGEOFFSET(eppnt->p_vaddr);
    unsigned long off = eppnt->p_offset - ELF_PAGEOFFSET(eppnt->p_vaddr);
    addr = ELF_PAGESTART(addr);
    size = ELF_PAGEALIGN(size);

    /* mmap() will return -EINVAL if given a zero size, but a
     * segment with zero filesize is perfectly valid */
    if (!size)
```

```
kernel_dev [C/C++ Attach to Application] gdb (7.12.1)
(gdb) print/x $rsi
$11 = 0x55e872b29000 -> addr
```

fs/binfmt_elf.c (Linux kernel)

ROP

Run 2: /usr/bin/ls



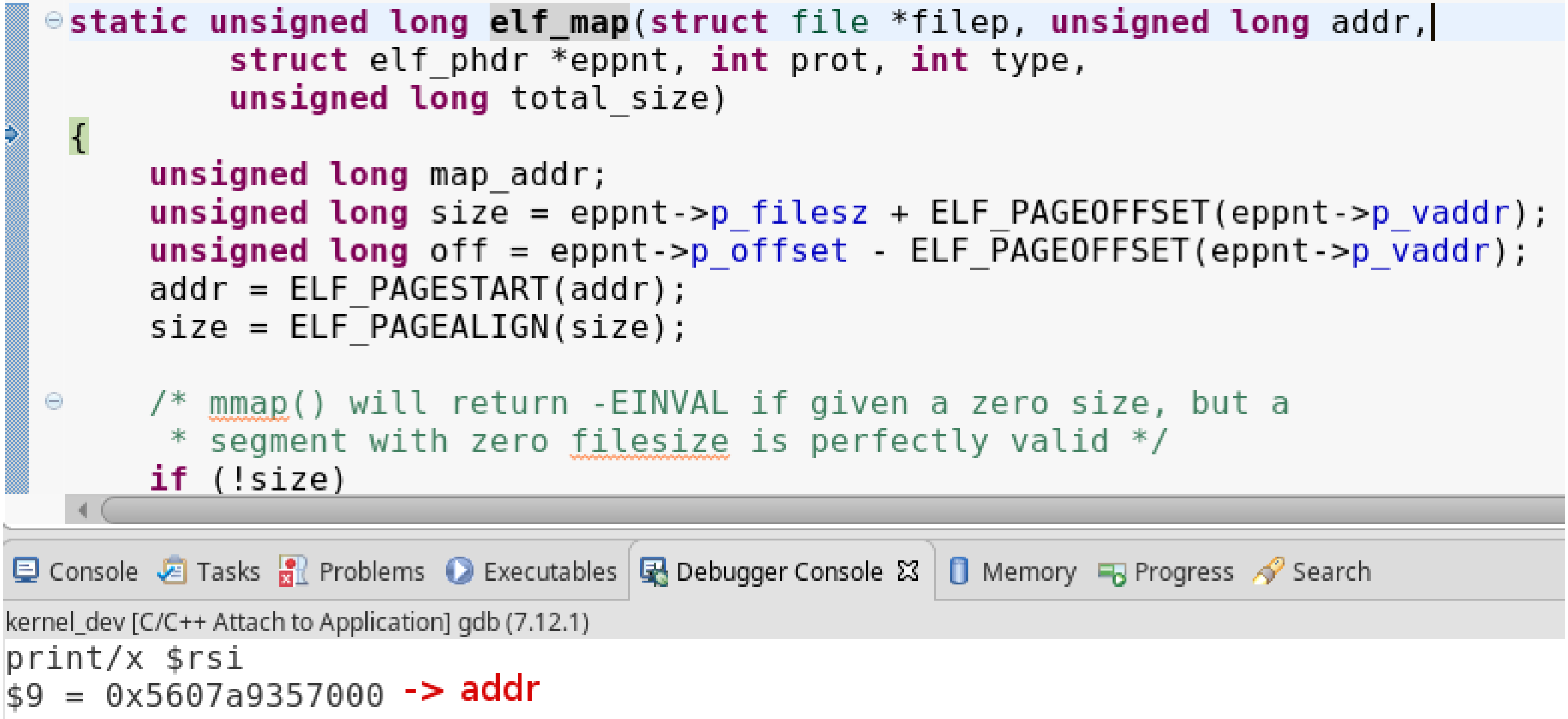
```
         * Therefore, programs are loaded offset fro
         * ELF_ET_DYN_BASE and loaders are loaded in
         * independently randomized mmap region (0 l
         * without MAP_FIXED).
         */
        if (elf_interpreter) {
            load_bias = ELF_ET_DYN_BASE;
            if (current->flags & PF_RANDOMIZE)
                load_bias += arch_mmap_rnd();
            elf_flags |= MAP_FIXED;
        } else
            load_bias = 0;

        /*
         * Since load bias is used for all subsequen
```

```
kernel_dev [C/C++ Attach to Application] gdb (7.12.1)
(gdb) print/x $rax
$4 = 0xb253e03000 -> load_bias
```

fs/binfmt_elf.c (Linux kernel)

ROP

Run 2: /usr/bin/ls



```
static unsigned long elf_map(struct file *filep, unsigned long addr,
        struct elf_phdr *eppnt, int prot, int type,
        unsigned long total_size)
{
    unsigned long map_addr;
    unsigned long size = eppnt->p_filesz + ELF_PAGEOFFSET(eppnt->p_vaddr);
    unsigned long off = eppnt->p_offset - ELF_PAGEOFFSET(eppnt->p_vaddr);
    addr = ELF_PAGESTART(addr);
    size = ELF_PAGEALIGN(size);

    /* mmap() will return -EINVAL if given a zero size, but a
     * segment with zero filesize is perfectly valid */
    if (!size)
```

kernel_dev [C/C++ Attach to Application] gdb (7.12.1)

```
print/x $rsi
$9 = 0x5607a9357000
```

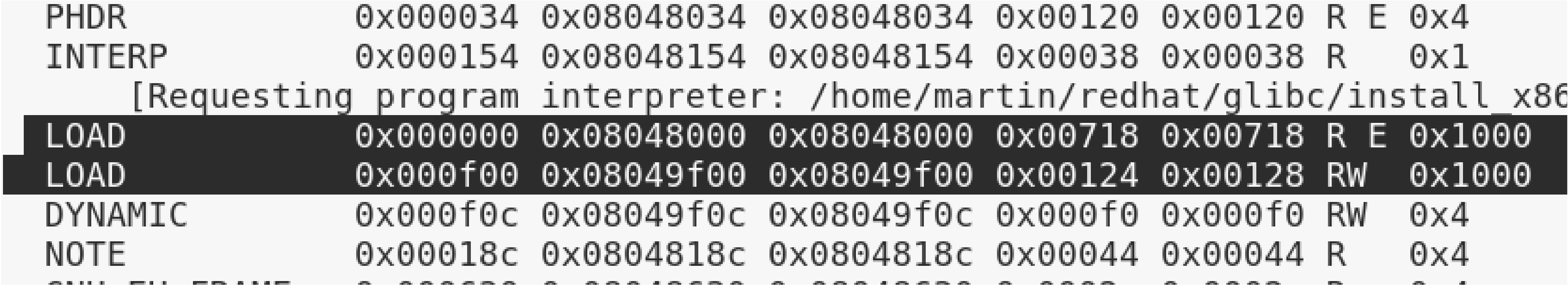
-> addr

fs/binfmt_elf.c (Linux kernel)

ROP

- Return Oriented Programming
  - X86 ELF binaries used not to be PIE, and the virtual address to be mapped was specified in the program header

Virtual Address

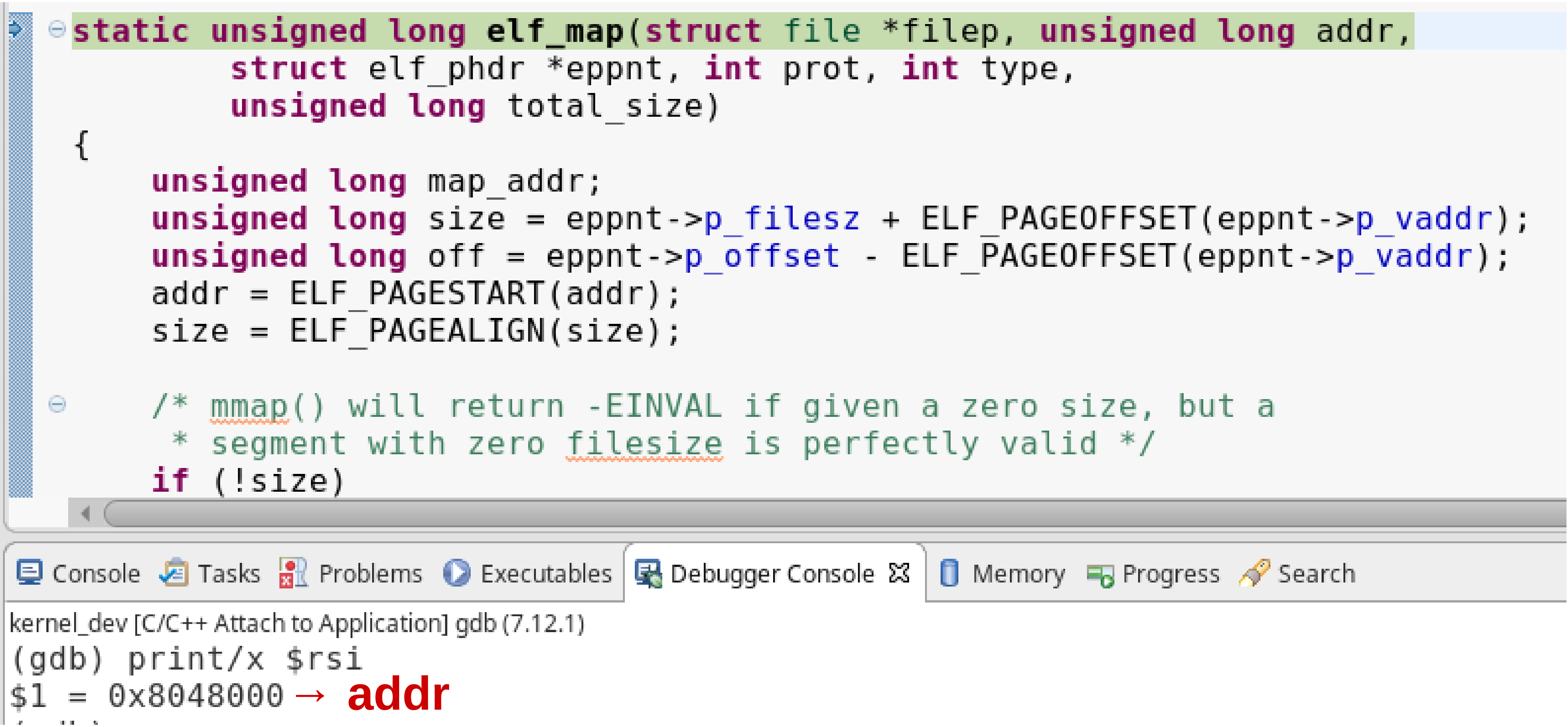


```
  PHDR           0x000034 0x08048034 0x08048034 0x00120 0x00120 R E 0x4
  INTERP         0x000154 0x08048154 0x08048154 0x00038 0x00038 R   0x1
      [Requesting program interpreter: /home/martin/redhat/glibc/install_x86
  LOAD           0x000000 0x08048000 0x08048000 0x00718 0x00718 R E 0x1000
  LOAD           0x000f00 0x08049f00 0x08049f00 0x00124 0x00128 RW  0x1000
  DYNAMIC        0x000f0c 0x08049f0c 0x08049f0c 0x000f0 0x000f0 RW  0x4
  NOTE           0x00018c 0x0804818c 0x0804818c 0x00044 0x00044 R   0x4
```

main-static (ELF 32)

ROP

Run: main-static (ELF 32)



```
static unsigned long elf_map(struct file *filep, unsigned long addr,
        struct elf_phdr *eppnt, int prot, int type,
        unsigned long total_size)
{
    unsigned long map_addr;
    unsigned long size = eppnt->p_filesz + ELF_PAGEOFFSET(eppnt->p_vaddr);
    unsigned long off = eppnt->p_offset - ELF_PAGEOFFSET(eppnt->p_vaddr);
    addr = ELF_PAGESTART(addr);
    size = ELF_PAGEALIGN(size);

    /* mmap() will return -EINVAL if given a zero size, but a
     * segment with zero filesize is perfectly valid */
    if (!size)
```

```
kernel_dev [C/C++ Attach to Application] gdb (7.12.1)
(gdb) print/x $rsi
$1 = 0x8048000
```

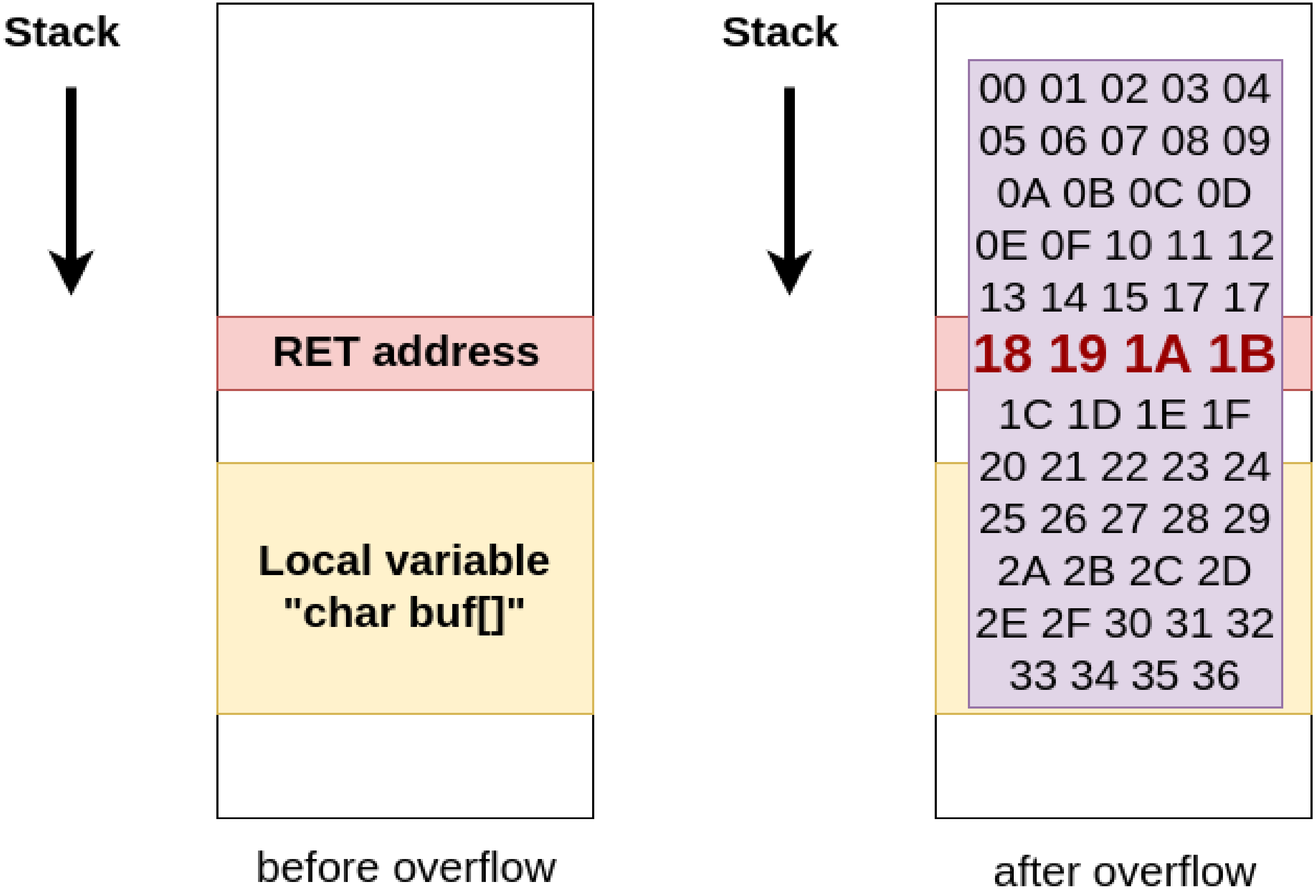
→ **addr**

fs/binfmt_elf.c (Linux kernel)

ROP

- I.e: suppose that this binary main-static (ELF 32, not PIE) has a stack overflow and EIP can be controlled
  - Stack canary → no
  - DEP → yes (not executable stack)
  - ASLR → yes for libs, not for the executable image

ROP



Stack

RET address

Local variable
"char buf[]"

before overflow

Stack

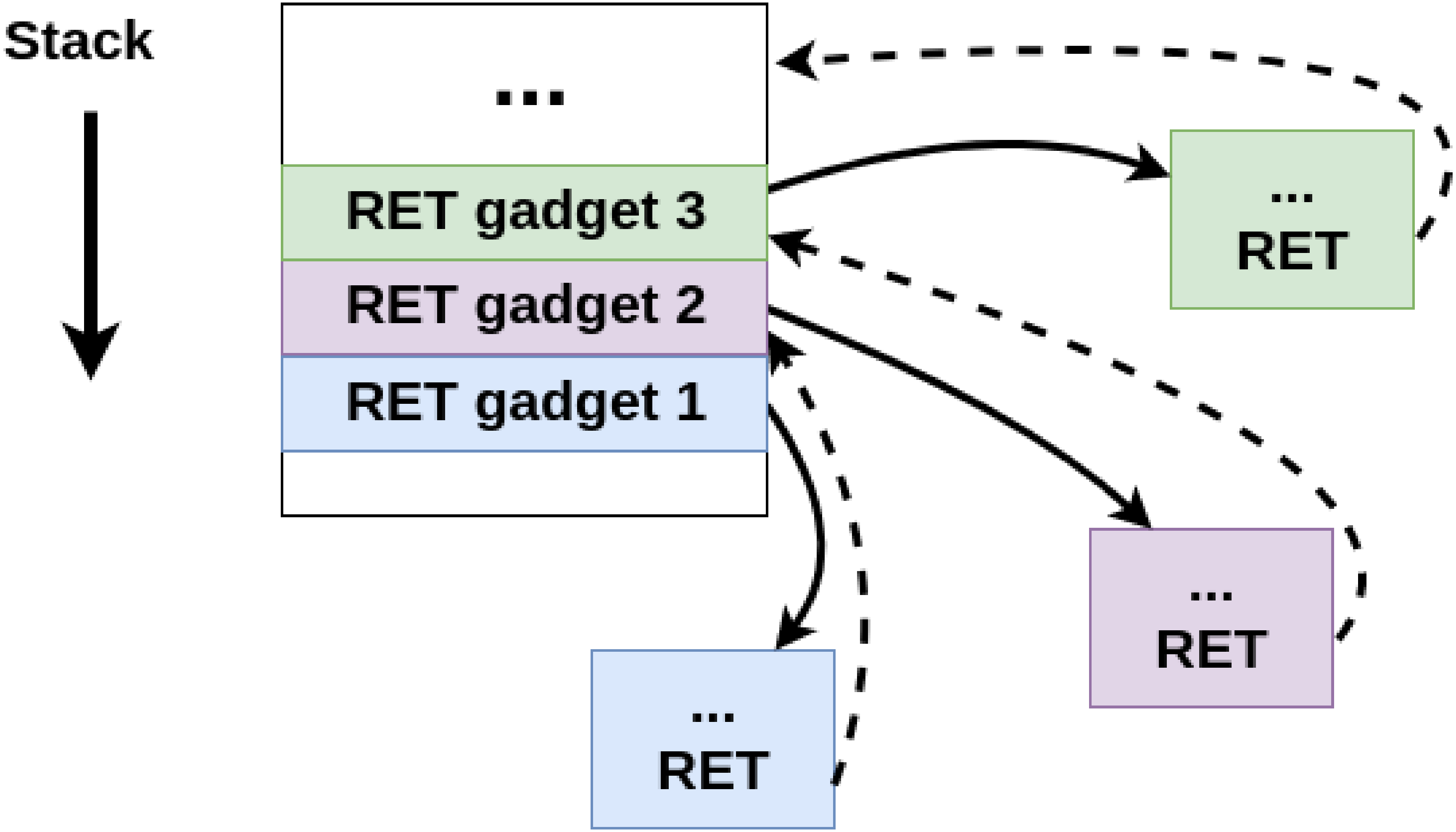
00 01 02 03 04
05 06 07 08 09
0A 0B 0C 0D
0E 0F 10 11 12
13 14 15 17 17
18 19 1A 1B
1C 1D 1E 1F
20 21 22 23 24
25 26 27 28 29
2A 2B 2C 2D
2E 2F 30 31 32
33 34 35 36

after overflow

ROP

- I.e: if we want to call *sys_execve* in order to execute */bin/bash* in Linux x86, what has to be done according to syscalls ABI?
  - eax = 0xb (syscall number)
  - ebx = pointer to “/bin/bash” (parameter 1)
  - ecx = null pointer (parameter 2 - argv)
  - edx = null pointer (parameter 3 - envp)
  - eip = pointer to “int 80” instruction

ROP



Stack

...

RET gadget 3

RET gadget 2

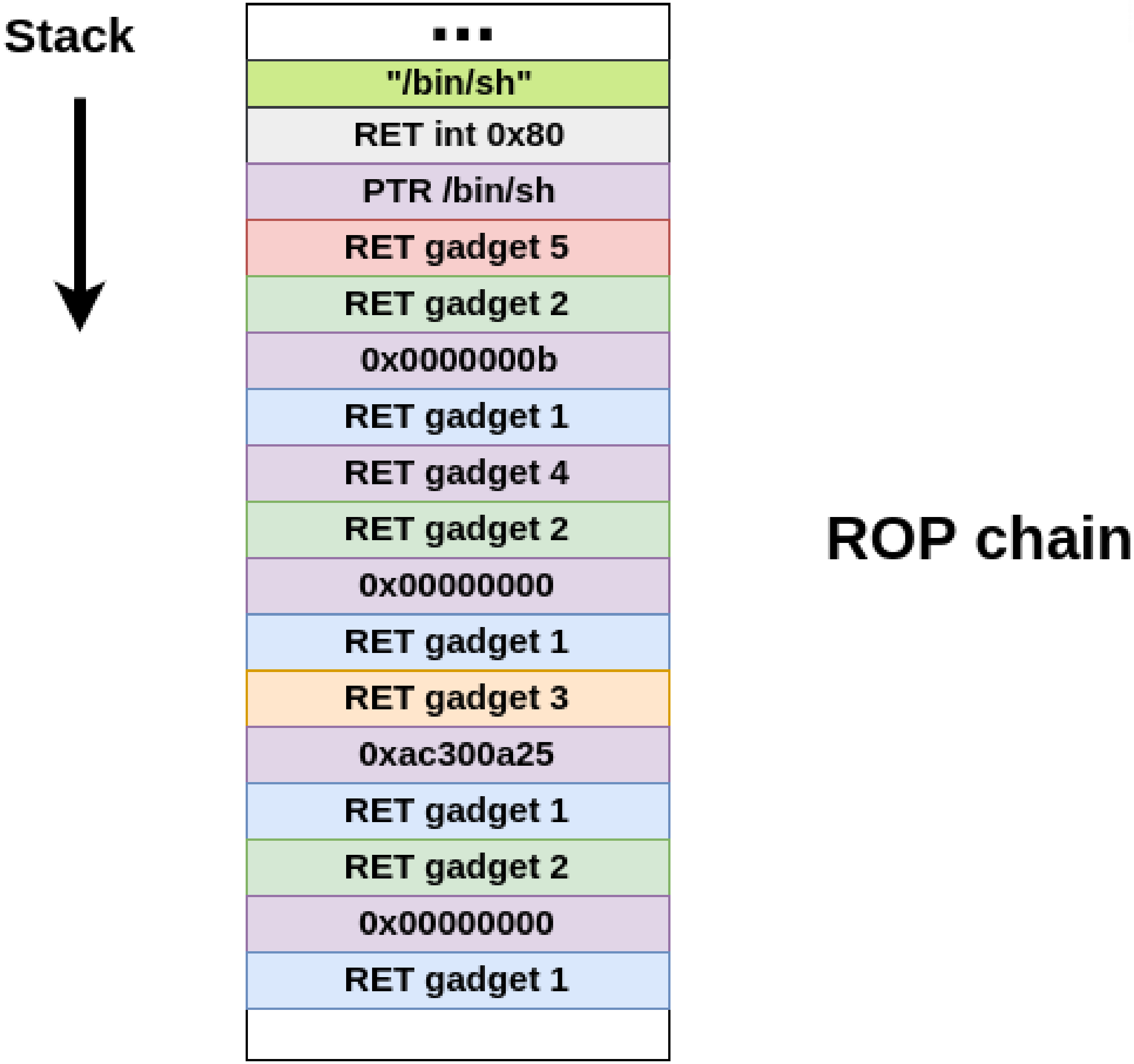
RET gadget 1

...
RET

...
RET

...
RET

ROP



Stack

...

"/bin/sh"

RET int 0x80

PTR /bin/sh

RET gadget 5

RET gadget 2

0x0000000b

RET gadget 1

RET gadget 4

RET gadget 2

0x00000000

RET gadget 1

RET gadget 3

0xac300a25

RET gadget 1

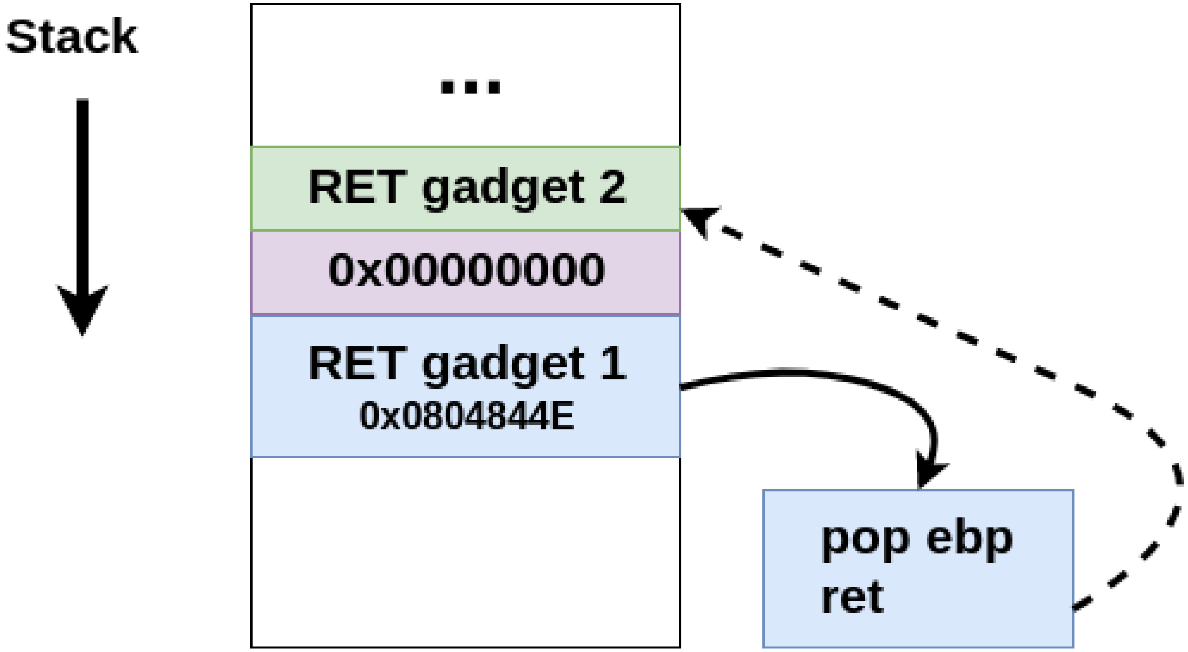
RET gadget 2

0x00000000

RET gadget 1

ROP chain

ROP



Stack

...

RET gadget 2

0x00000000

RET gadget 1
0x0804844E

pop ebp
ret

State

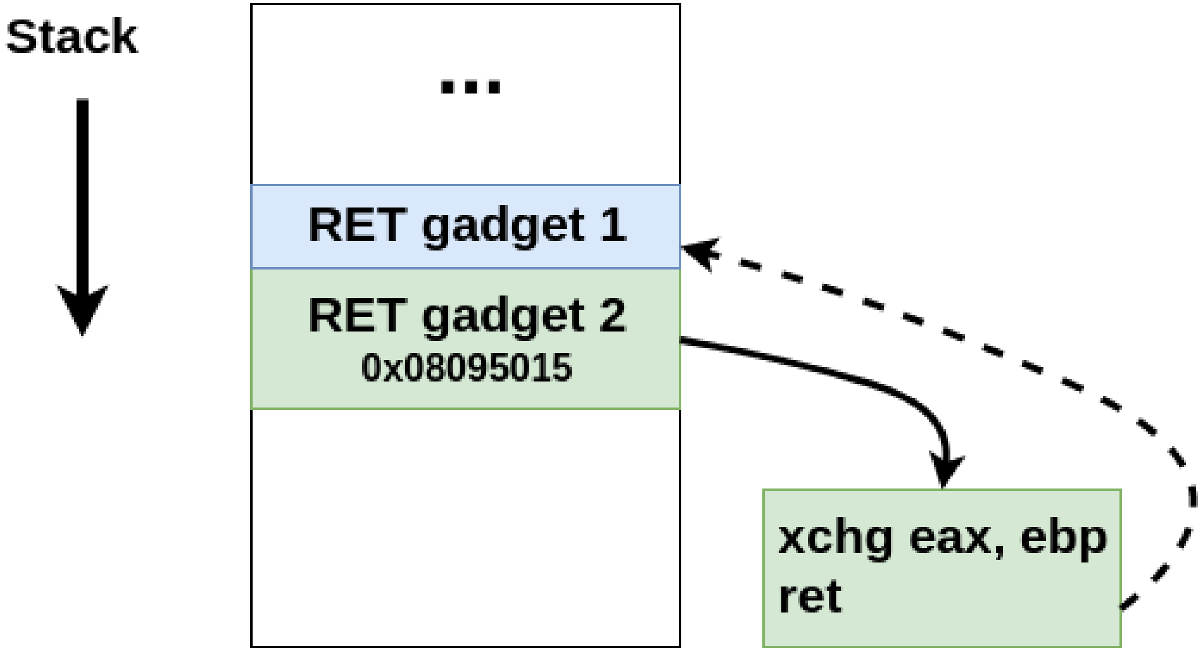
ebp = ????????
eax = ????????
ebx = ????????
ecx = ????????
edx = ????????

Before

ebp = 0x00000000
eax = ????????
ebx = ????????
ecx = ????????
edx = ????????

After

ROP



State

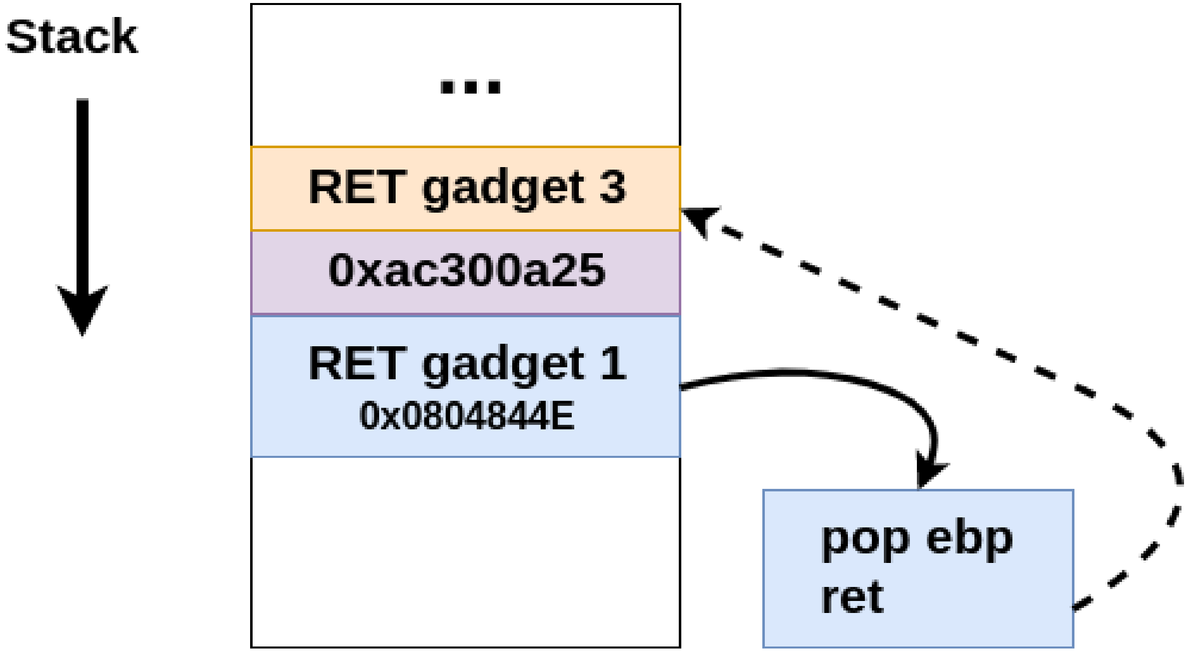
```
ebp = 0x00000000
eax = ????????
ebx = ????????
ecx = ????????
edx = ????????
```

Before

```
ebp = ????????
eax = 0x00000000
ebx = ????????
ecx = ????????
edx = ????????
```

After

ROP



Stack

...

RET gadget 3

0xac300a25

RET gadget 1
0x0804844E

pop ebp
ret

State

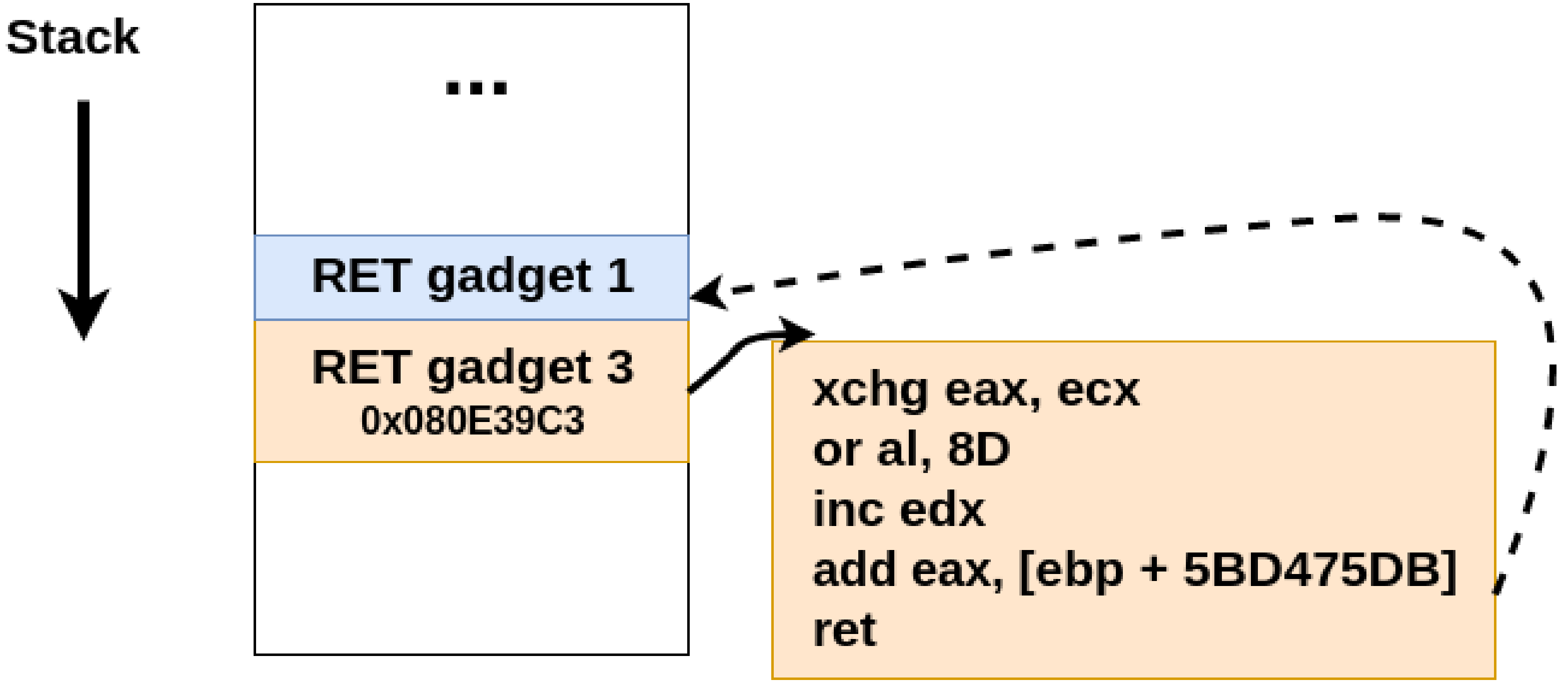
ebp = ????????
eax = 0x00000000
ebx = ????????
ecx = ????????
edx = ????????

Before

ebp = 0xac300a25
eax = 0x00000000
ebx = ????????
ecx = ????????
edx = ????????

After

ROP

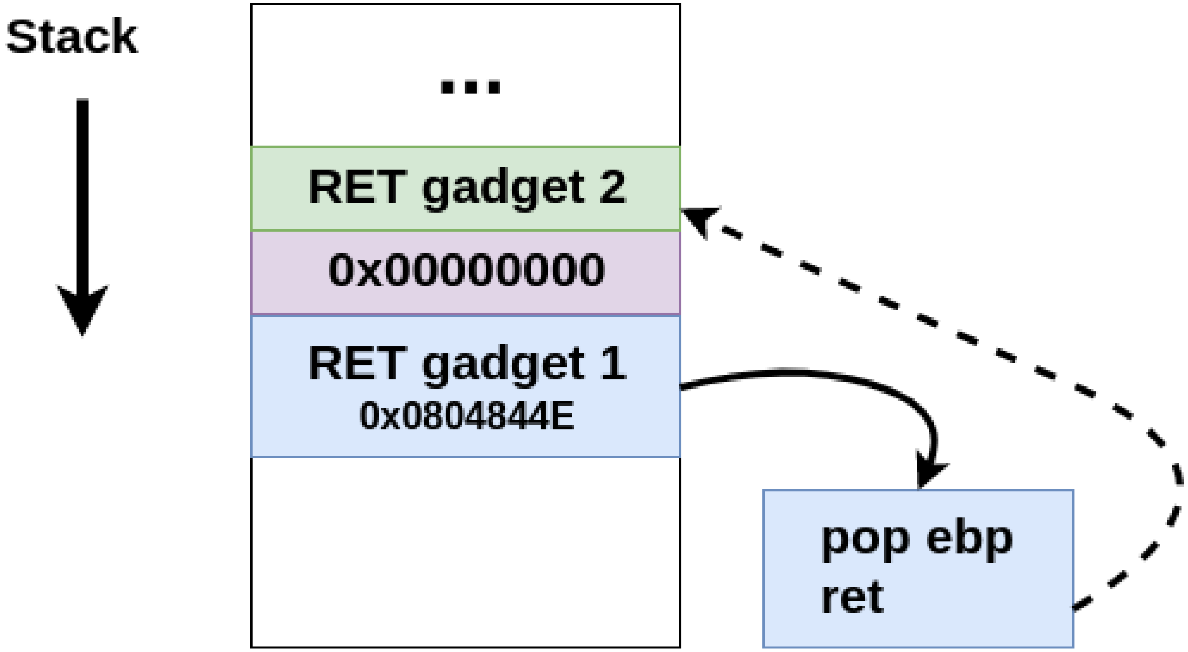


```
xchg eax, ecx
or al, 8D
inc edx
add eax, [ebp + 5BD475DB]
ret
```

State

Before	After
ebp = 0xac300a25	ebp = 0xac300a25
eax = 0x00000000	eax = ????????
ebx = ????????	ebx = ????????
ecx = ????????	ecx = 0x00000000
edx = ????????	edx = ????????

ROP



State

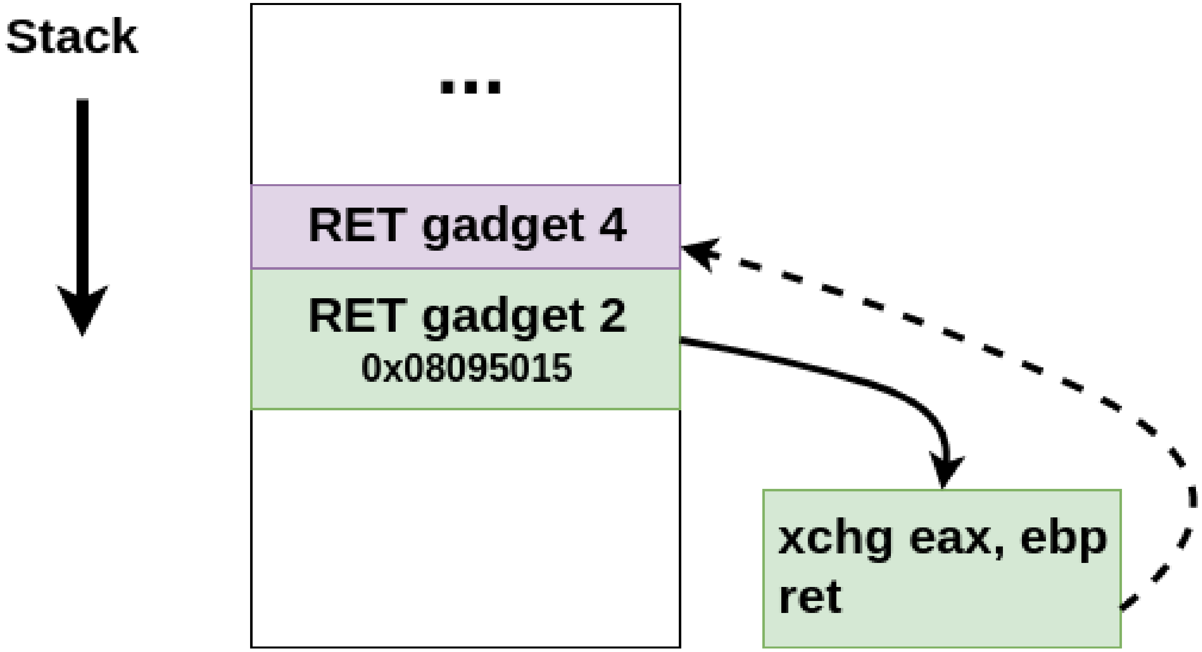
```
ebp = 0xac300a25
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = ????????
```

Before

```
ebp = 0x00000000
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = ????????
```

After

ROP



Stack

...

RET gadget 4

RET gadget 2
0x08095015

xchg eax, ebp
ret

State

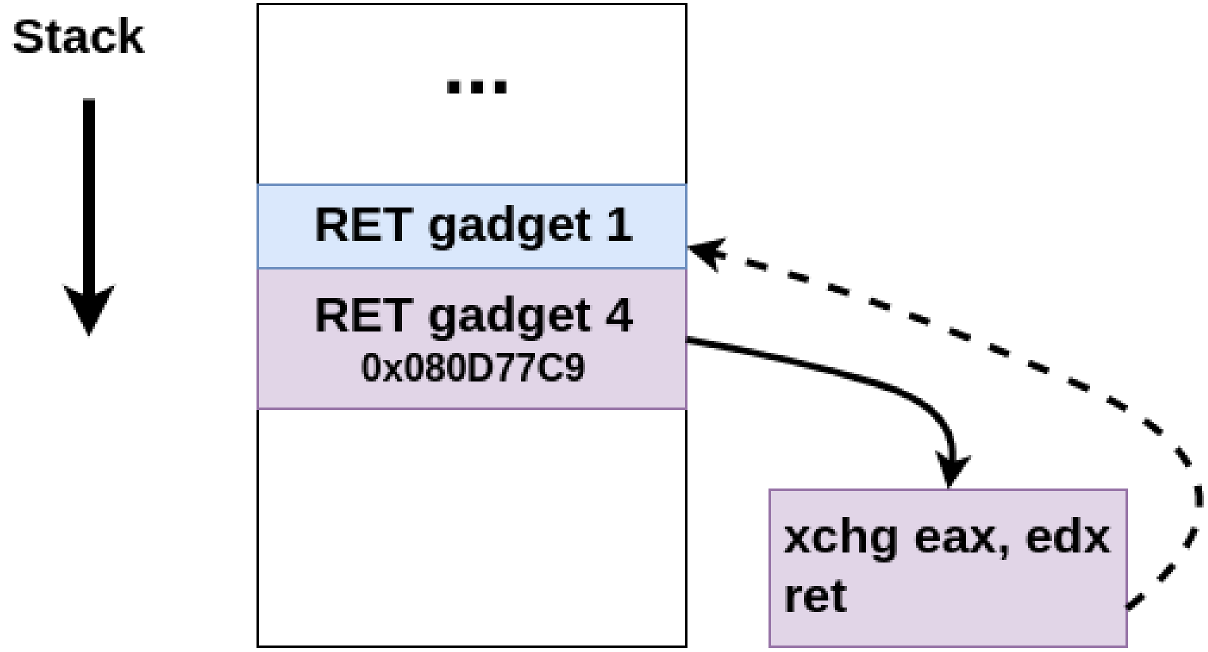
ebp = 0x00000000
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = ????????

Before

ebp = ????????
eax = 0x00000000
ebx = ????????
ecx = 0x00000000
edx = ????????

After

ROP



State

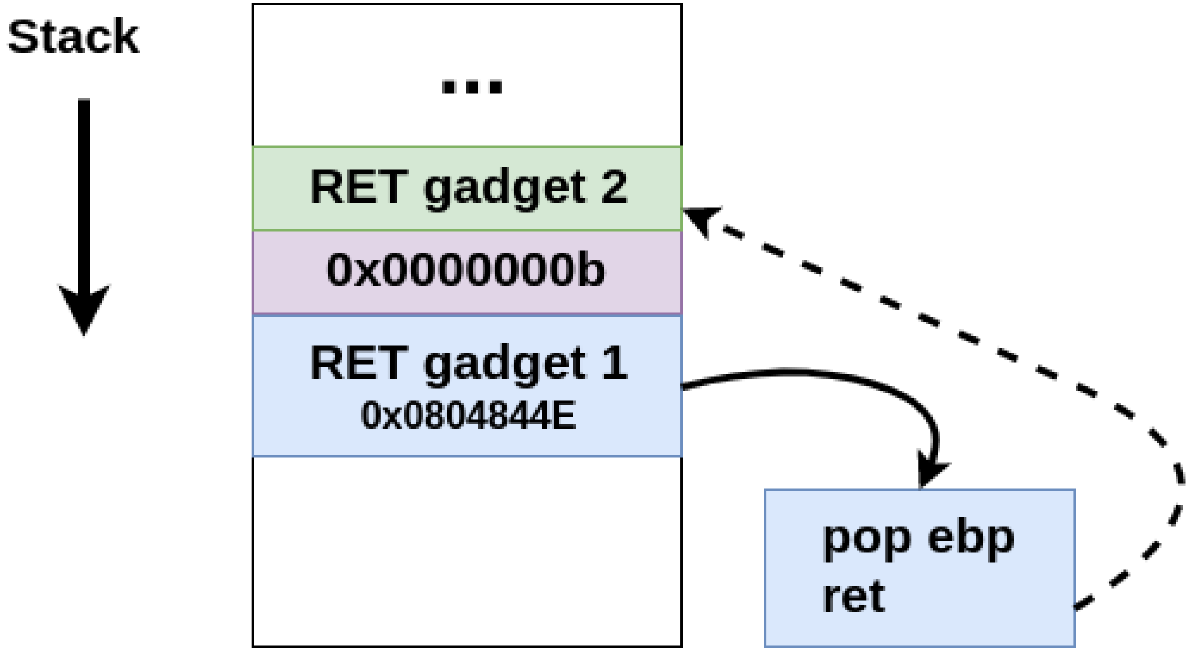
ebp = ????????
eax = 0x00000000
ebx = ????????
ecx = 0x00000000
edx = ????????

Before

ebp = ????????
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = 0x00000000

After

ROP



State

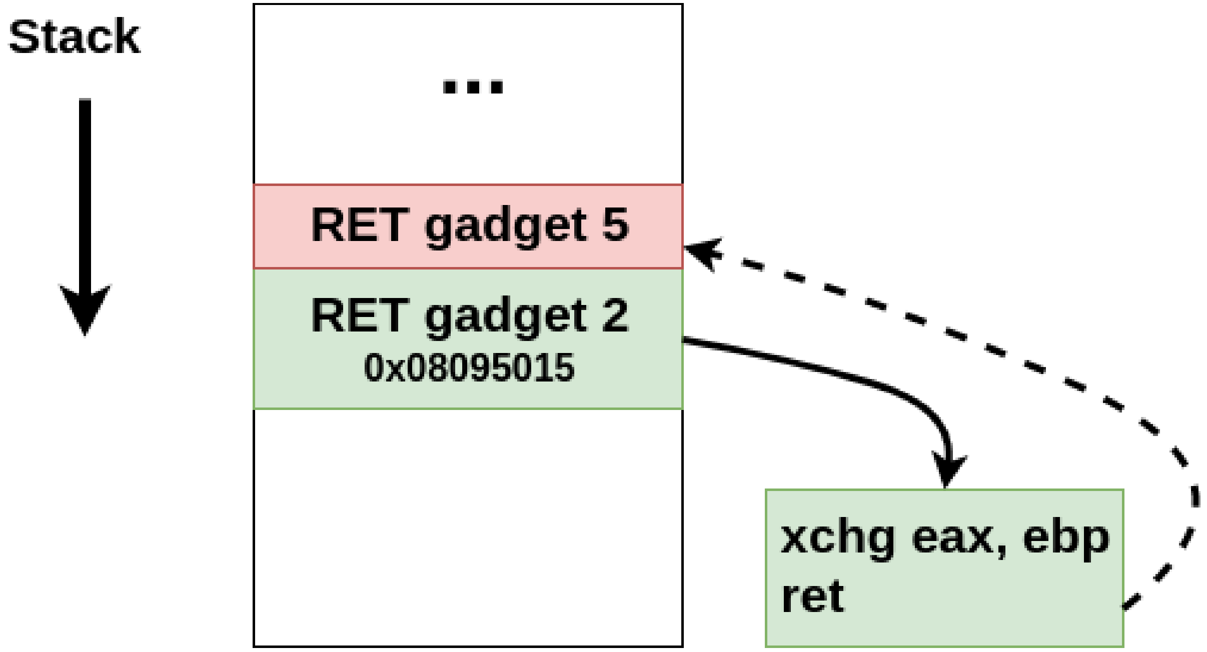
ebp = ????????
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = 0x00000000

Before

ebp = 0x0000000b
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = 0x00000000

After

ROP



Stack

...

RET gadget 5

RET gadget 2
0x08095015

xchg eax, ebp
ret

State

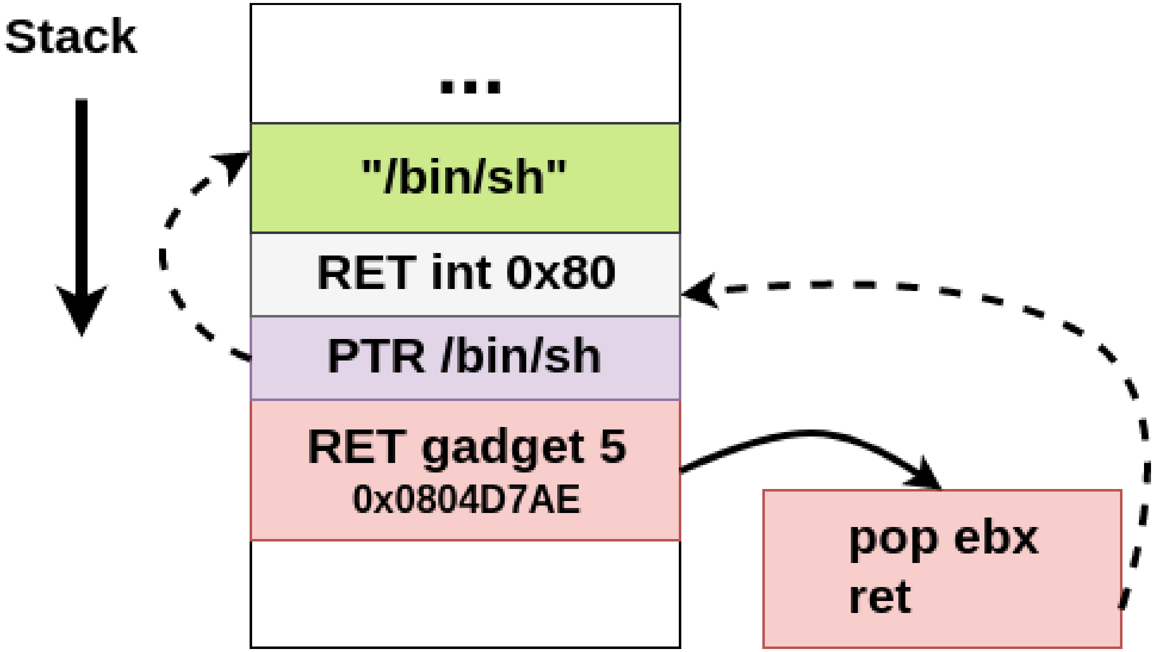
ebp = 0x0000000b
eax = ????????
ebx = ????????
ecx = 0x00000000
edx = 0x00000000

Before

ebp = ????????
eax = 0x0000000b
ebx = ????????
ecx = 0x00000000
edx = 0x00000000

After

ROP



State

```
ebp = ????????
eax = 0x0000000b
ebx = ????????
ecx = 0x00000000
edx = 0x00000000
```

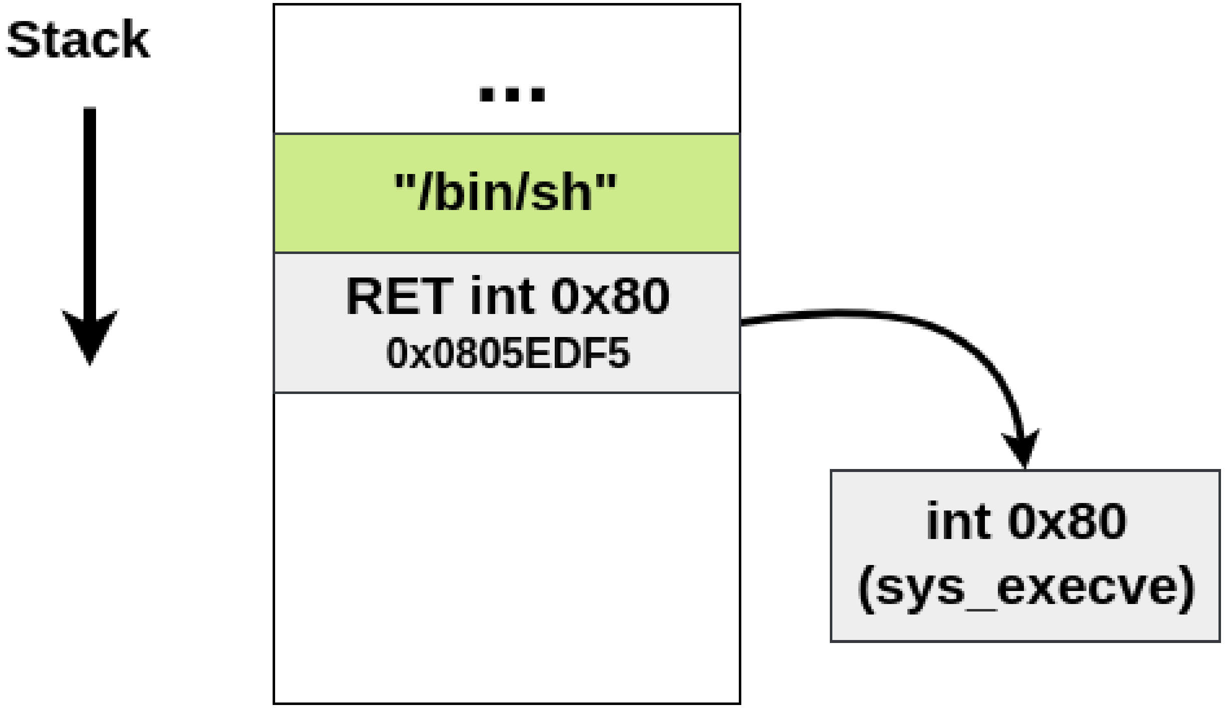
Before

```
ebp = ????????
eax = 0x0000000b
ebx = PTR /bin/sh
ecx = 0x00000000
edx = 0x00000000
```

✓

After

ROP



State

ebp = ????????
eax = 0x0000000b
ebx = PTR /bin/sh
ecx = 0x00000000
edx = 0x00000000

✓

ROP

- How to find gadgets?
  - Static analysis tools
  - Dynamic analysis tools: AGAFI
- Constraints satisfaction problem
  - Side-effects of some gadgets
  - Compensation for indirect reads/writes
- Instructions of a few bytes are preferable (I.e: xchg + ret is 2 bytes long and there are no side-effects)

ROP

- Unaligned jumps to find gadgets
  - In x86/x86_64 is possible to jump unaligned
  - CISC architectures have multiple valid instructions, which is an advantage
- POPAD instruction is interesting
  - 1 byte long (0x61)
  - Load multiple registers with values from the stack at once

ROP

- Multiple ways of achieving the desired state. Example: set eax to 0:
  - Is eax already 0?
  - pop eax
  - xor eax, eax
  - mov eax, 0x0
  - dec eax
  - xchg eax, r (r = 0)
  - etc.

ROP

- PTR leaks: is there any register pointing to a known place at crash time?
- Jump Oriented Programming: instead of RETs, use indirect jumps
- Call Oriented Programming: instead of RETs, use indirect calls
- In kernel space ROP works exactly the same way

Demo 10.1

ROP chain in user space

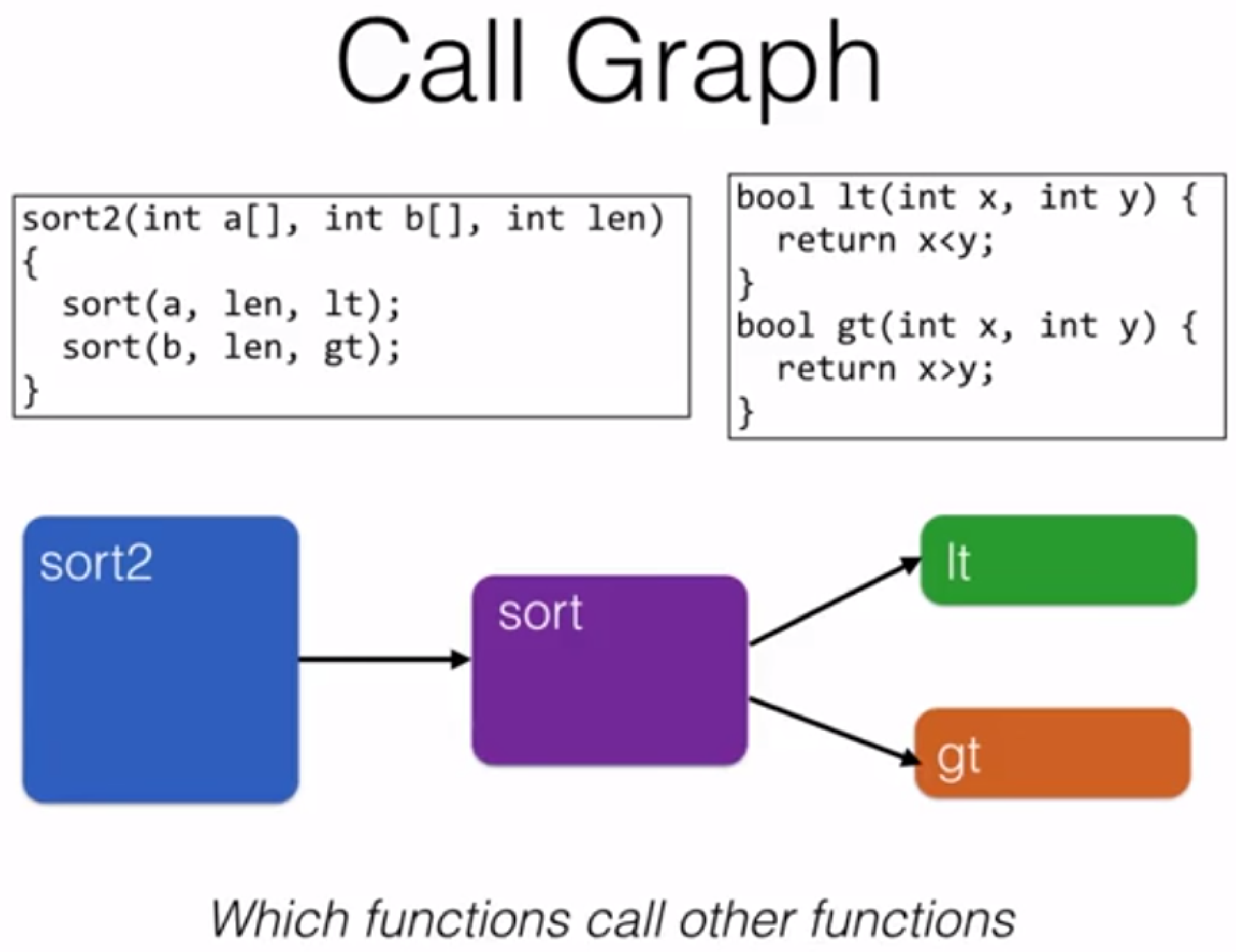
Control Flow Integrity

- A program has expected execution flows, defined by a graph in compilation/linking time
- A ROP attack makes the program execute a anomalous or unexpected flow
- Can the program detect when the expected flow is broken? This would be a good compromise indicator

Control Flow Integrity

- If DEP (Data Execution Prevention) is taken for granted, how can the attacker corrupt flows?
  - CALL 0xAABBCCDD cannot be corrupted: memory where relative call parameter is located is in the code segment (.text) and it’s not writable
  - Execution flows that can be corrupted are those that depend on data (indirect): CALL [REG] or JMP [REG] (being REG a register loaded with a value from memory), RET

Control Flow Integrity



Call Graph

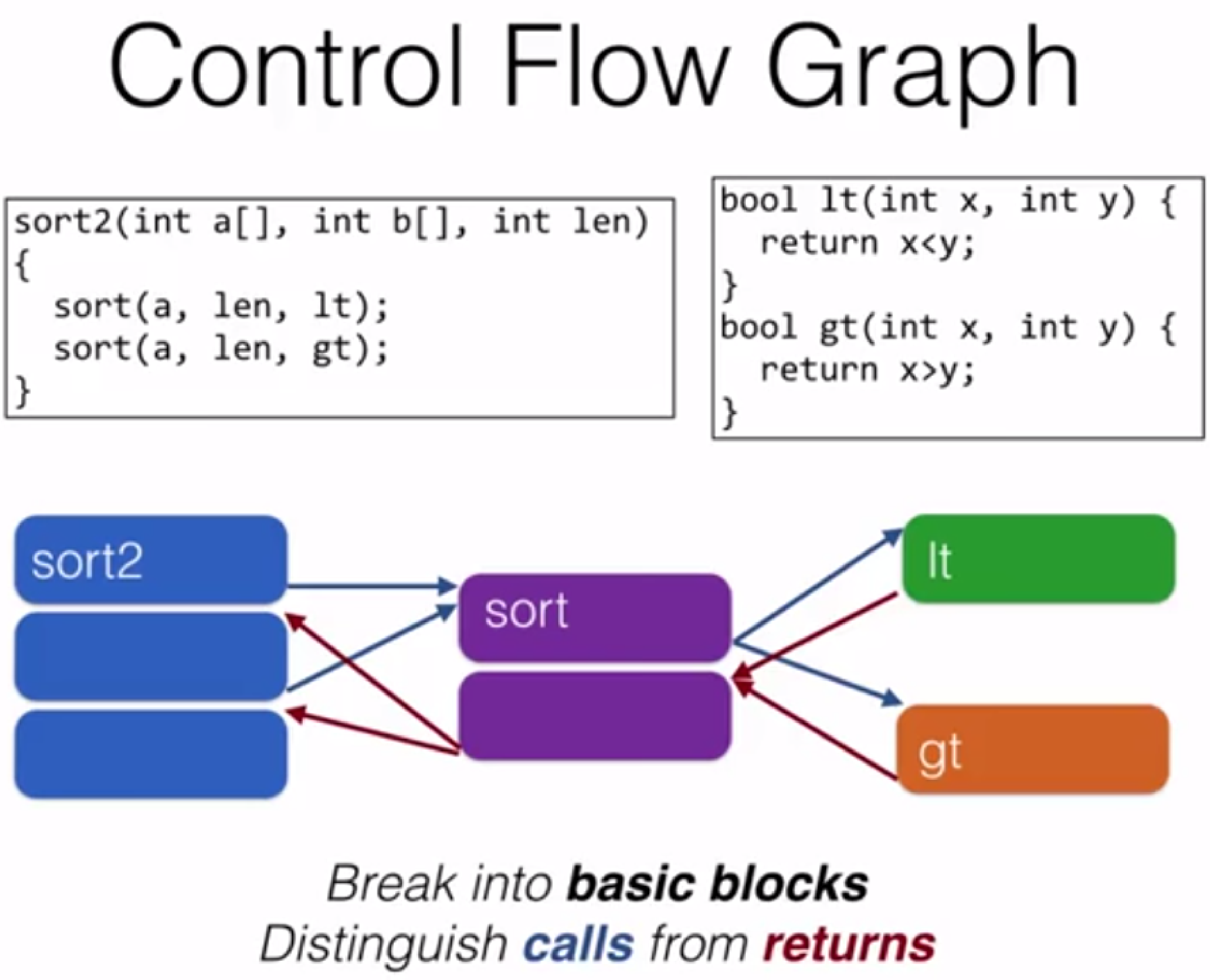
```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(b, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

Which functions call other functions

Image from Software Security course (University of Maryland)

Control Flow Integrity



Control Flow Graph

```
sort2(int a[], int b[], int len)
{
  sort(a, len, lt);
  sort(a, len, gt);
}
```

```
bool lt(int x, int y) {
  return x<y;
}
bool gt(int x, int y) {
  return x>y;
}
```

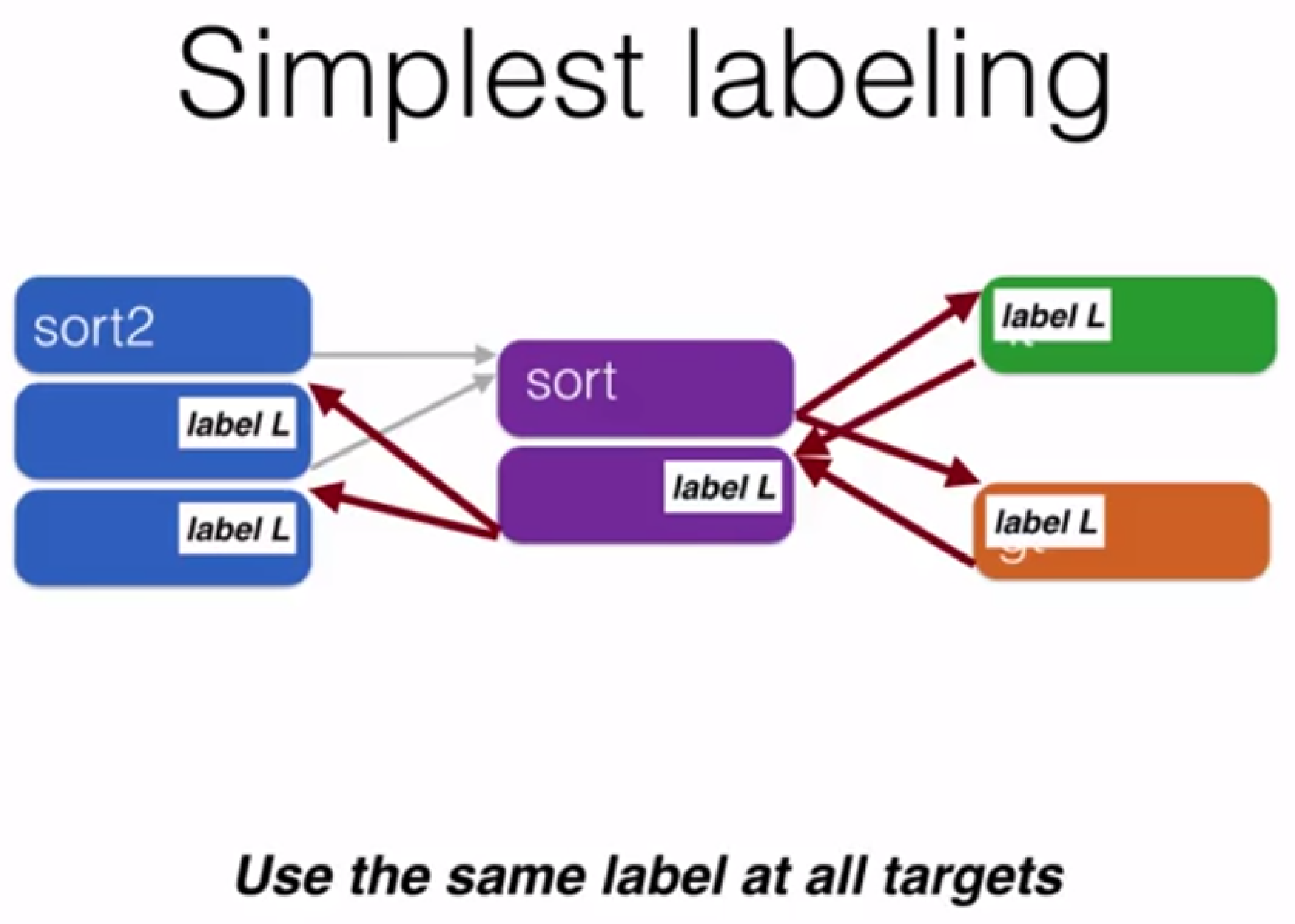
Break into ***basic blocks***
Distinguish ***calls*** *from* ***returns***

Image from Software Security course (University of Maryland)

Control Flow Integrity

- It’s possible to label destinations for indirect jumps. This is: add label bytes (not executable) previous to the jump target
- Before jumping, verify the existence of a correct label in those bytes previous to the jump target
- If label is correct, proceed to the jump Otherwise, an anomalous flow has been detected
- This has a performance hit

Control Flow Integrity

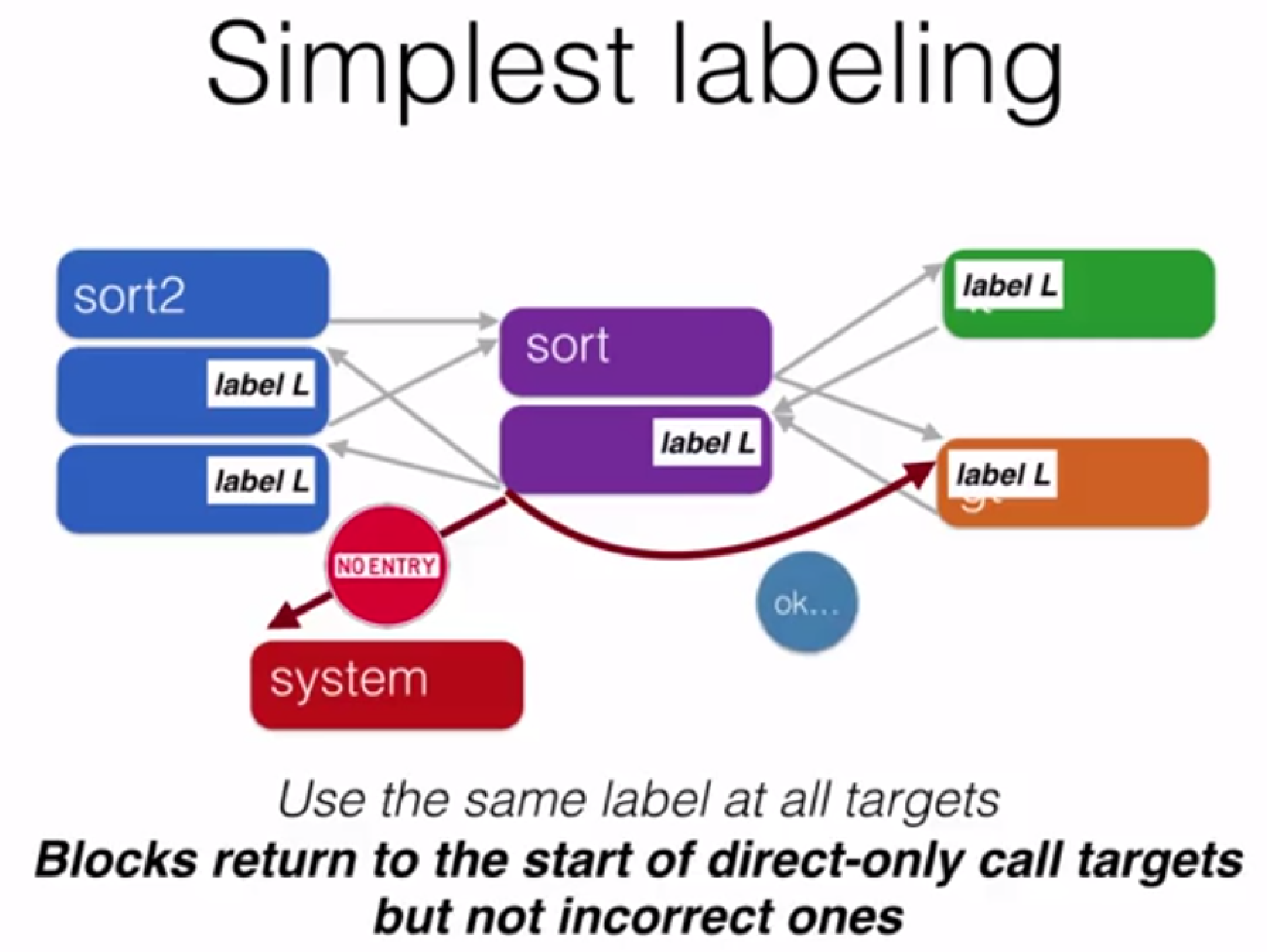


Simplest labeling

Use the same label at all targets

Image from Software Security course (University of Maryland)

Control Flow Integrity



Simplest labeling

Use the same label at all targets

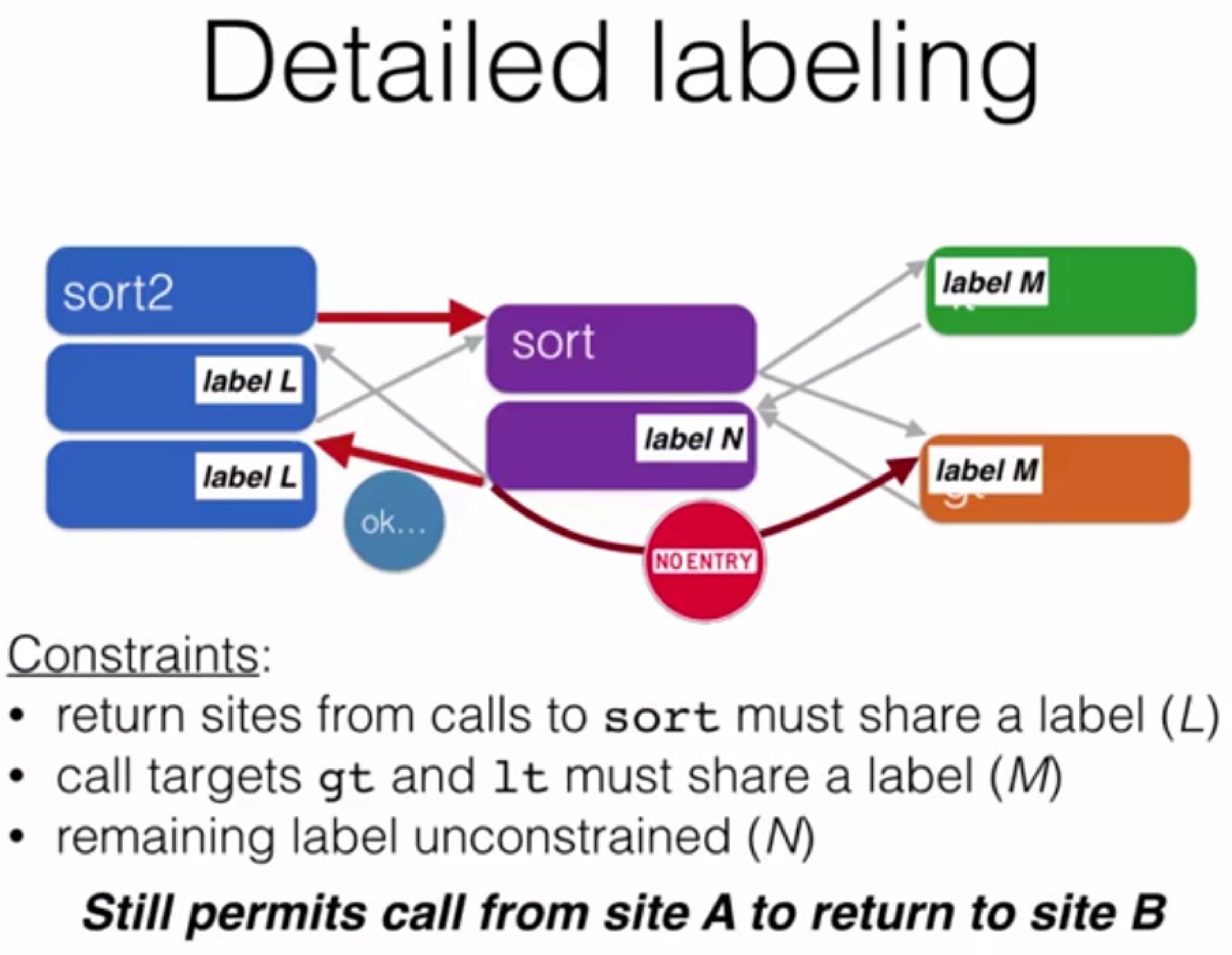
Blocks return to the start of direct-only call targets but not incorrect ones

Image from Software Security course (University of Maryland)

Control Flow Integrity

- This technique does not prevent from jumping to a place with the same label, despite not being a possible flow in the graph
- Greater label granularity is needed to prevent these cases

Control Flow Integrity



Detailed labeling

Constraints:

- return sites from calls to `sort` must share a label (*L*)
- call targets `gt` and `lt` must share a label (*M*)
- remaining label unconstrained (*N*)

Still permits call from site A to return to site B

Image from Software Security course (University of Maryland)

Control Flow Integrity

```
class A {
public:
    virtual int m(void) = 0;
};
class B : public A {
public:
    int m(void);
};
class C : public A {
public:
    int m(void);
};
int B::m(void) {
    return 1;
}
int C::m(void) {
    return 2;
}
```

```
int main(void) {
    int res = 0;
    A* b = new B();
    A* c = new C();

    volatile unsigned long bu =
reinterpret_cast<unsigned long>(&b);
    volatile unsigned long cu =
reinterpret_cast<unsigned long>(&c);
    A* bb = *(reinterpret_cast<A**>(bu));
    A* cc = *(reinterpret_cast<A**>(cu));

    res += bb->m();
    res += cc->m();

    return res;
}
```

clang++

Control Flow Integrity

```
movq  -48(%rbp), %rax
movq  (%rax), %rdi
movq  -56(%rbp), %rax
movq  (%rdi), %rcx
movq  %rcx, %rdx
subq   %r15, %rdx
rolq $59, %rdx
cmpq  $3, %rdx
jae   46 <_main+99>
movq  (%rax), %rbx
callq   *(%rcx)
```

pointer to object b

object b

pointer to object c

vtable B

integrity check: is it a valid vtable?

if not, error

object c

call to 1st method from vtable B

Lab

Exercise 10.2

ROP chain in user space

Execute shellcode in the stack

References

- Software Security – University of Maryland
  - https://en.coursera.org/learn/software-security
- https://clang.llvm.org/docs/ControlFlowIntegrity.html