Ingeniería Inversa
Clase 10

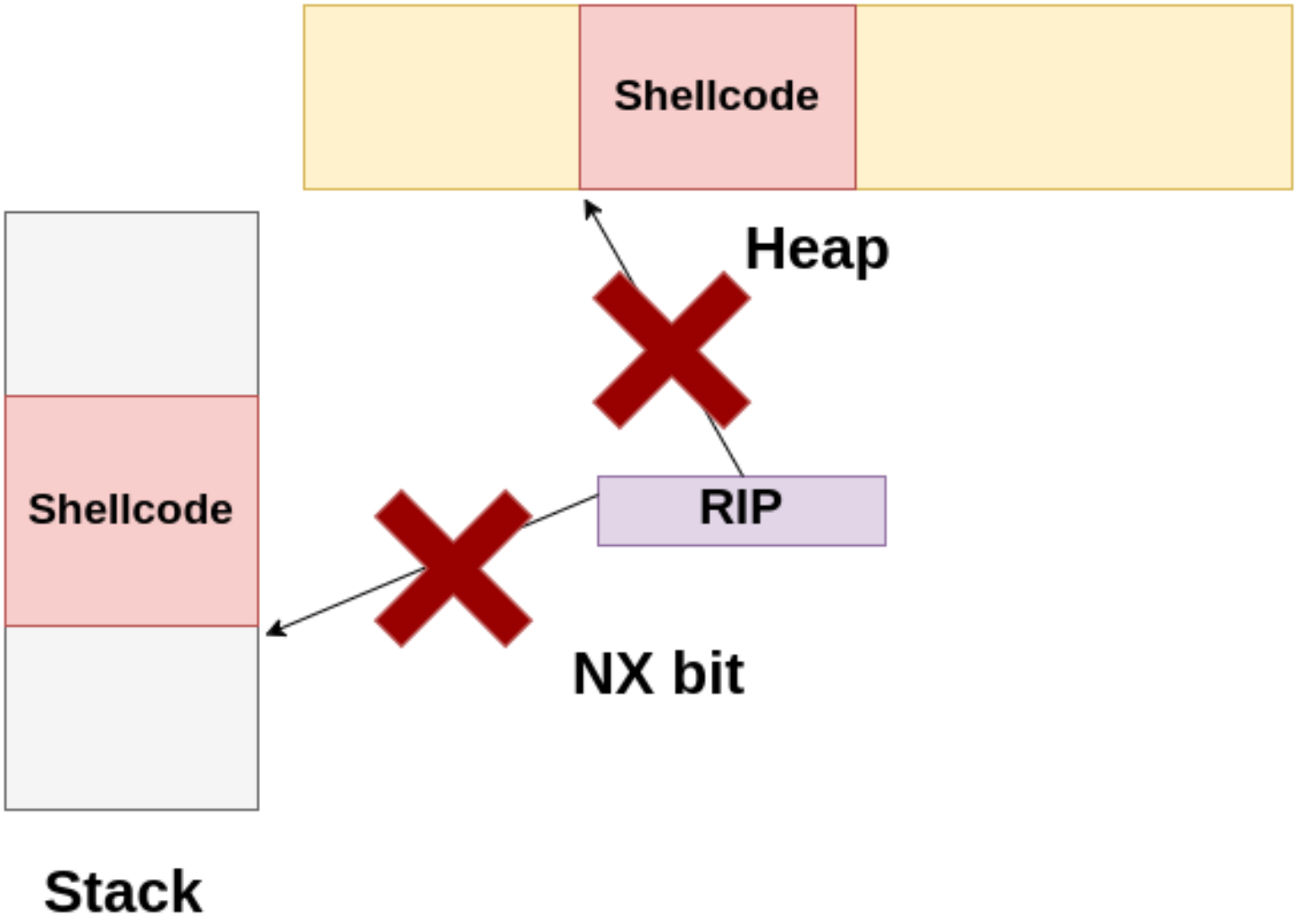
Exploit Writing III
Return Oriented Programming (ROP)

martin.uy
Open by default.

ROP

- ROP: Return Oriented Programming
  - Se controla RIP (instruction pointer), pero:
  - No se puede saltar a ejecutar el shellcode en el stack, data o heap
    - Datos ya no son ejecutables (DEP → Data Execution Prevention)
    - NX bit (x86)
    - Esto aplica tanto a espacio de usuario como a espacio de kernel

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
```

La sección GNU_STACK (del binario “main”) tiene el flag RW

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
  - Llamar a *system* (Linux) o *WinExec* (Windows)
    - Invocar un comando o aplicación (ej. shell)
  - Llamar a *dlopen* (Linux) o *LoadLibrary* (Windows)
    - Ejecutar código al cargar una librería
  - En x86, una corrupción del stack puede permitir controlar todos los parámetros de estas llamadas (ABI)

ROP

- Return to libc
  - En x86_64, la ABI supone cargar registros para enviar parámetros
  - Randomización del espacio virtual de direcciones (ASLR): ¿en qué dirección virtual están las funciones *system*, *dlopen*, *WinExec* y *LoadLibrary*?

ROP

- Return to libc
  - Return to strcpy/memcpy/sprintf/etc
  - Copiar shellcode a un lugar escribible y ejecutable
  - W^X: protección contra segmentos escribibles y ejecutables

Lab

Ejercicio 10.1: return to Libc

ROP

- Return Oriented Programming (ROP)
  - El control del stack es requerido para hacer ROP
    - Pivotear el stack a un lugar bajo control si es necesario
  - Encadenar múltiples llamadas a pequeñas secuencias de assembly: gadgets
    - Cada “llamada” es un retorno a lo que haya en el tope del stack
    - Todos los gadgets terminan en la instrucción RET (o una equivalente) que permita seguir controlando el flujo de ejecución a través del stack
    - Se va modificando convenientemente el estado de los registros y la memoria en cada llamada

ROP

- Return Oriented Programming
  - Objetivos: desproteger memoria (syscall *mprotect* en Linux o *VirtualProtect* en Windows) para saltar al shellcode o ejecutar un binario (syscall *execve*)
    - Otra alternativa puede ser alocar memoria nueva con permisos de escritura y ejecución, y copiar el payload a ese lugar

ROP

- Return Oriented Programming
  - ¿En qué dirección está el shellcode?
  - Ejemplo: randomización del stack

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
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

```
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
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

```
kernel_dev [C/C++ Attach to Application] gdb (7.12.1)
(gdb) print/x $rsi
$2 = 0x7ffd59c41000
(gdb)
```

ROP

- Return Oriented Programming
  - ¿En qué address está el shellcode?
    - Un ptr leak o un heap spray pueden ser necesarios
  - ¿En qué address están los gadgets?
    - Las librerías mapeadas pueden randomizarse (PIC) pero algunas no
    - La imagen binaria puede randomizarse (PIE) o no

ROP

- Return Oriented Programming
  - ¿En qué address están los gadgets?
    - Ejemplo de 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
               0x00000000001d2ac 0x00000000001d2ac  R E    200000
LOAD           0x00000000001dfc8 0x00000000021dfc8 0x00000000021dfc8
               0x0000000000001280 0x0000000000001fc0  RW     200000
DYNAMIC        0x00000000001ea18 0x00000000021ea18 0x00000000021ea18
               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
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

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)
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

```
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
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Prog

```
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)
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

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
  - Los binarios ELF x86 solían no ser PIE, y en el program header se especificaba la virtual address para ser mapeados

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)
```

Console | Tasks | Problems | Executables | Debugger Console | Memory | Progress | Search

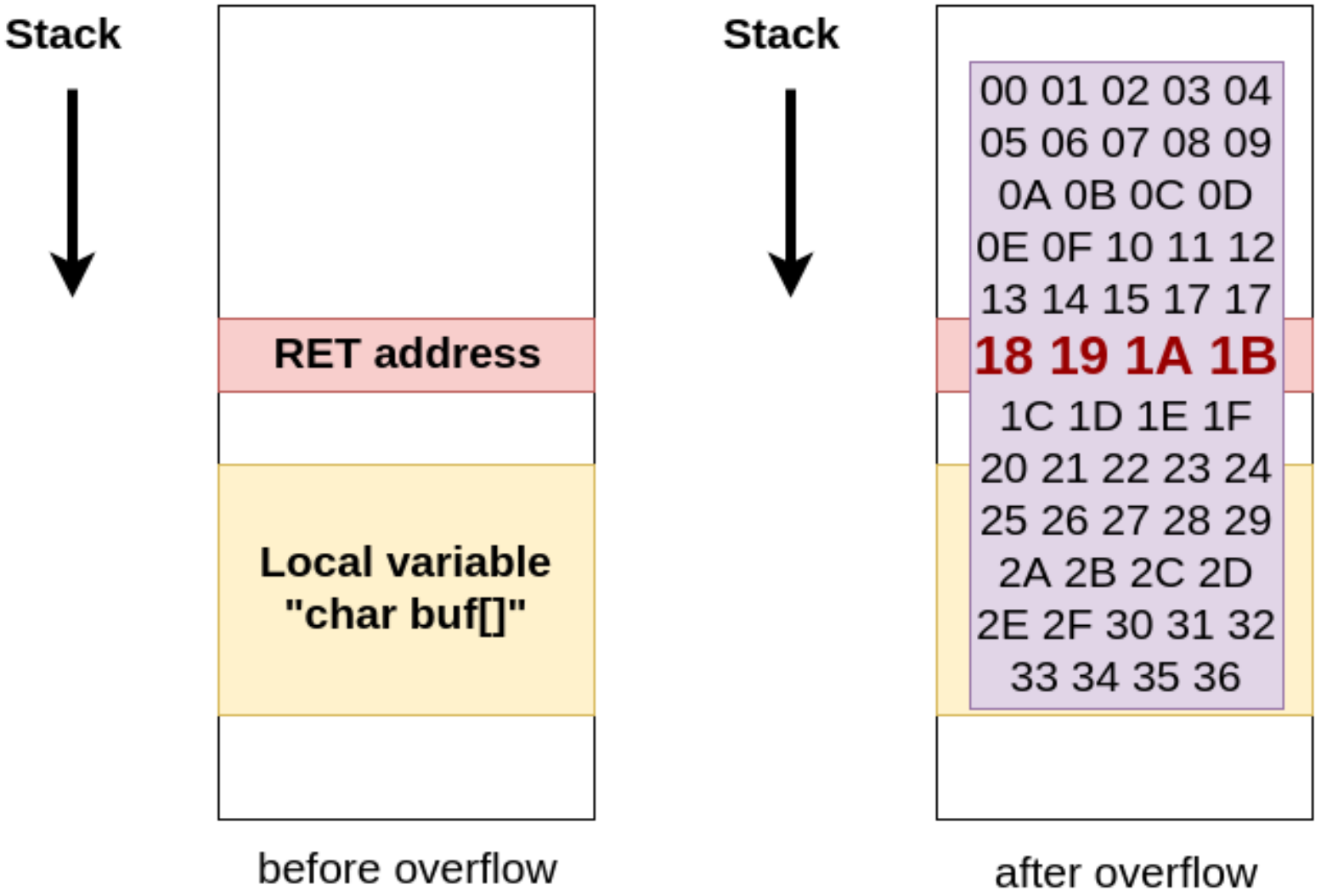
```
kernel_dev [C/C++ Attach to Application] gdb (7.12.1)
(gdb) print/x $rsi
$1 = 0x8048000 → addr
```

fs/binfmt_elf.c (Linux kernel)

ROP

- Ej: supongamos que este binario main-static (ELF 32, no PIE) tiene un stack overflow y podemos controlar EIP
  - Stack canary → no
  - DEP → sí (stack no ejecutable)
  - ASLR → sí para las librerías, no para la imagen ejecutable

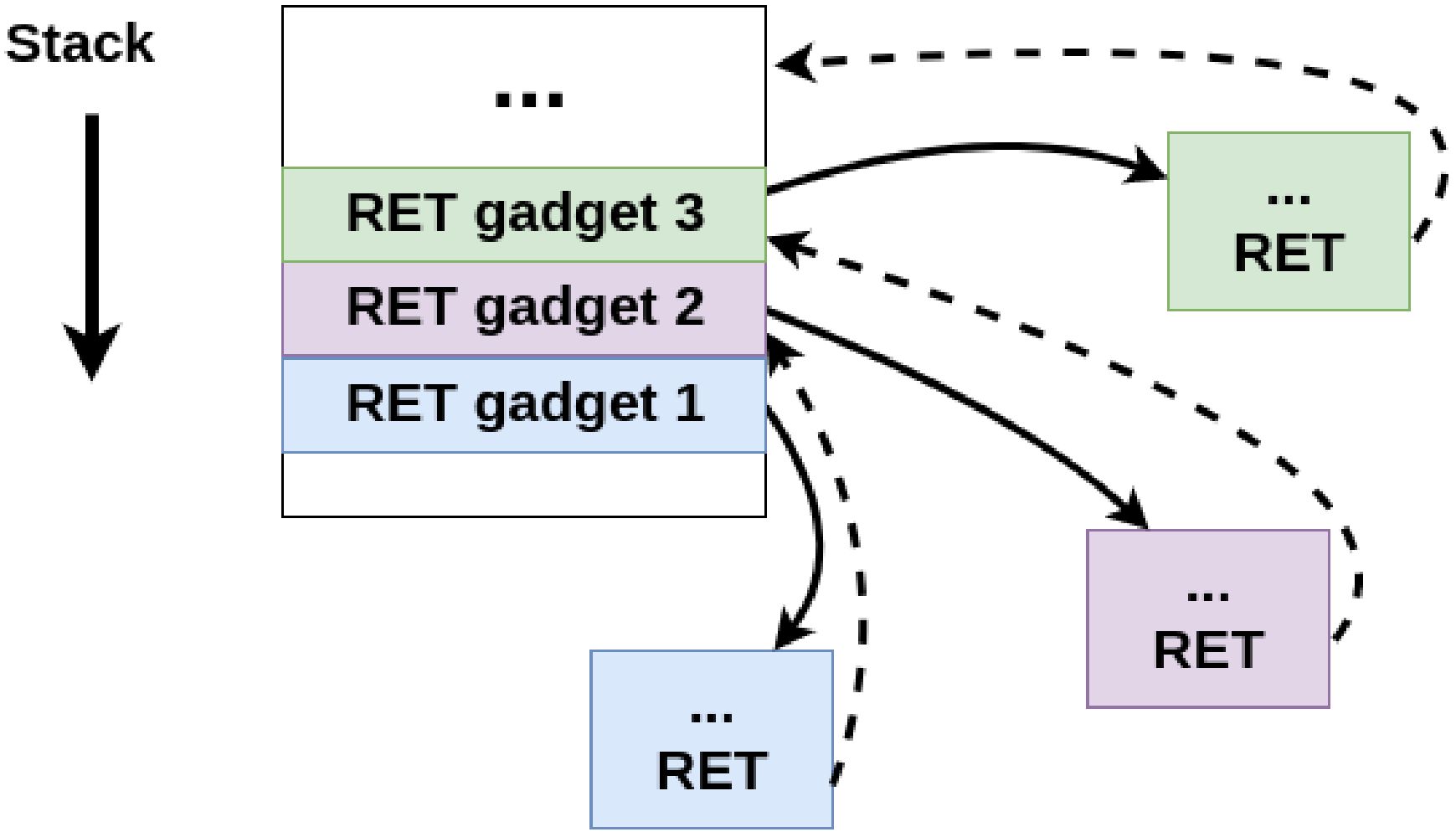
ROP


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

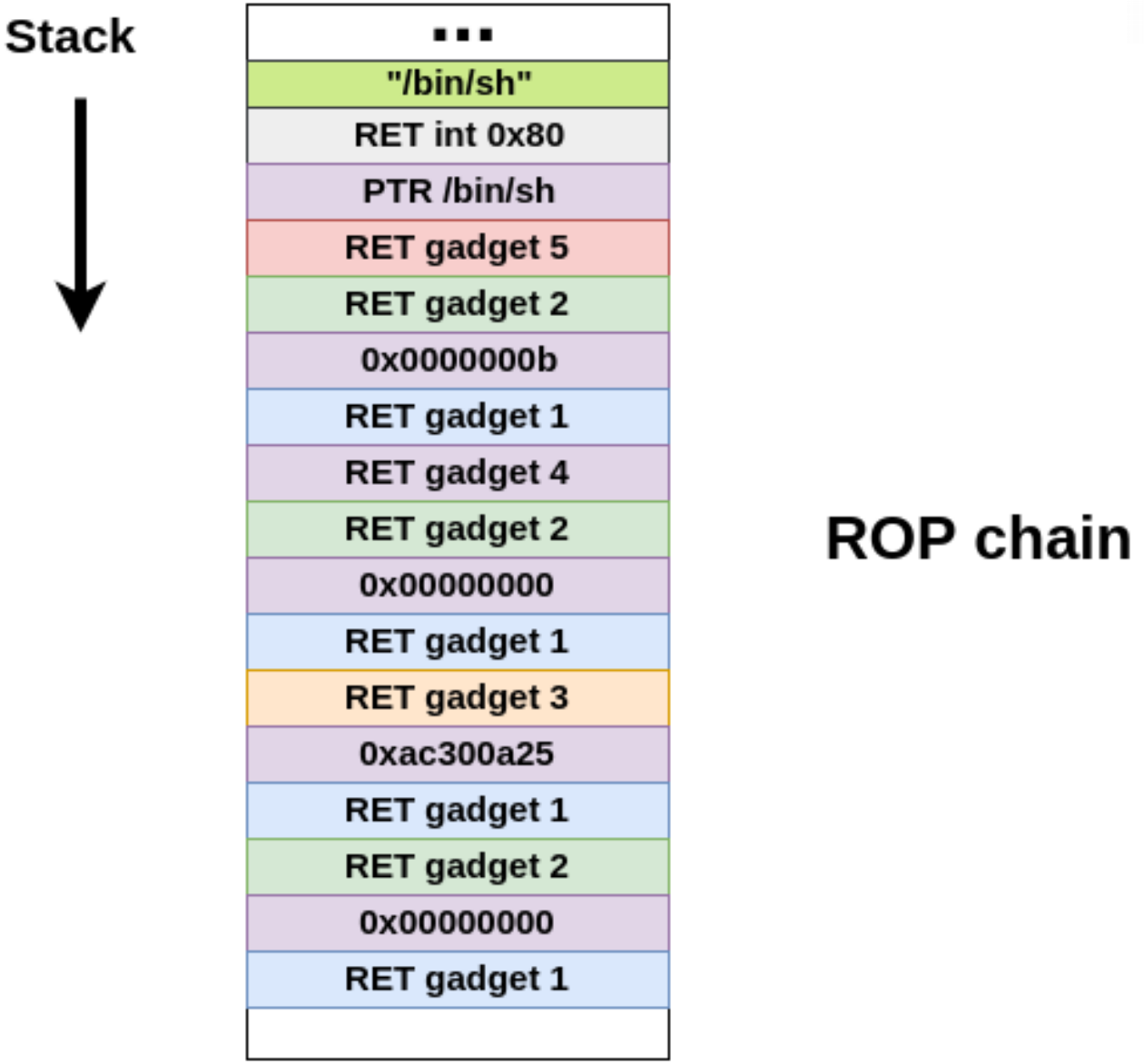
ROP

- Ej: si queremos llamar a *sys_execve* y ejecutar */bin/bash* en Linux x86, ¿qué debemos hacer según la ABI de syscalls?
  - eax = 0xb (número de syscall)
  - ebx = puntero a “/bin/bash” (parámetro 1)
  - ecx = puntero a null (parámetro 2 - argv)
  - edx = puntero a null (parámetro 3 - envp)
  - eip = puntero a instrucción “int 80”

ROP

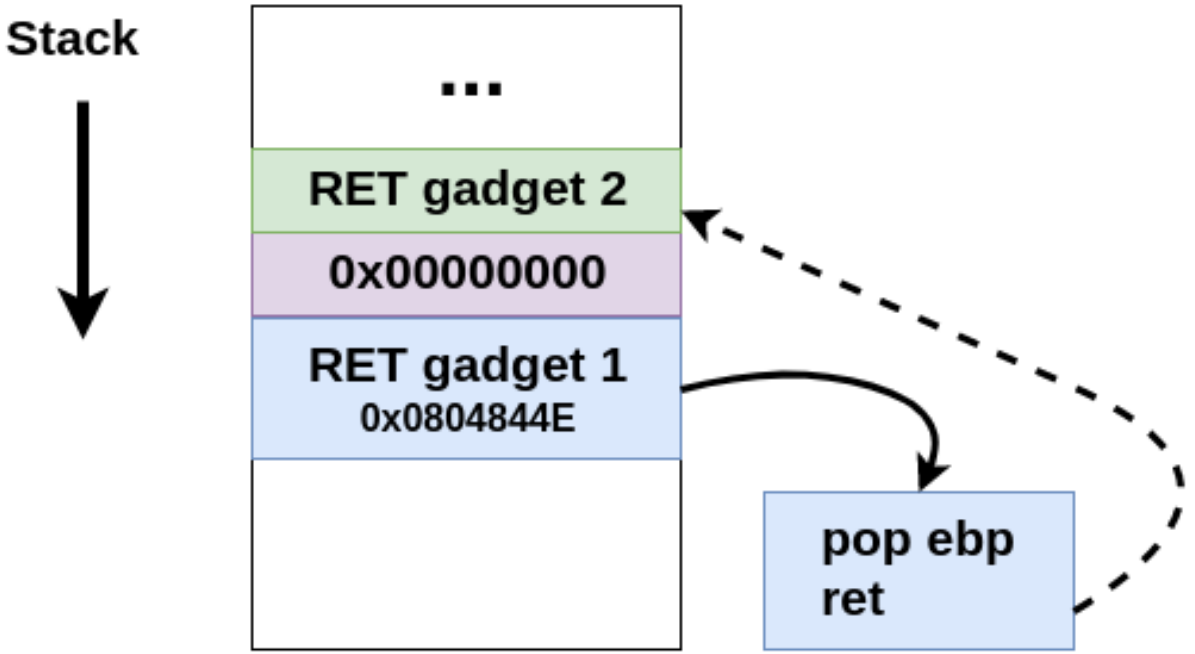


ROP



ROP chain

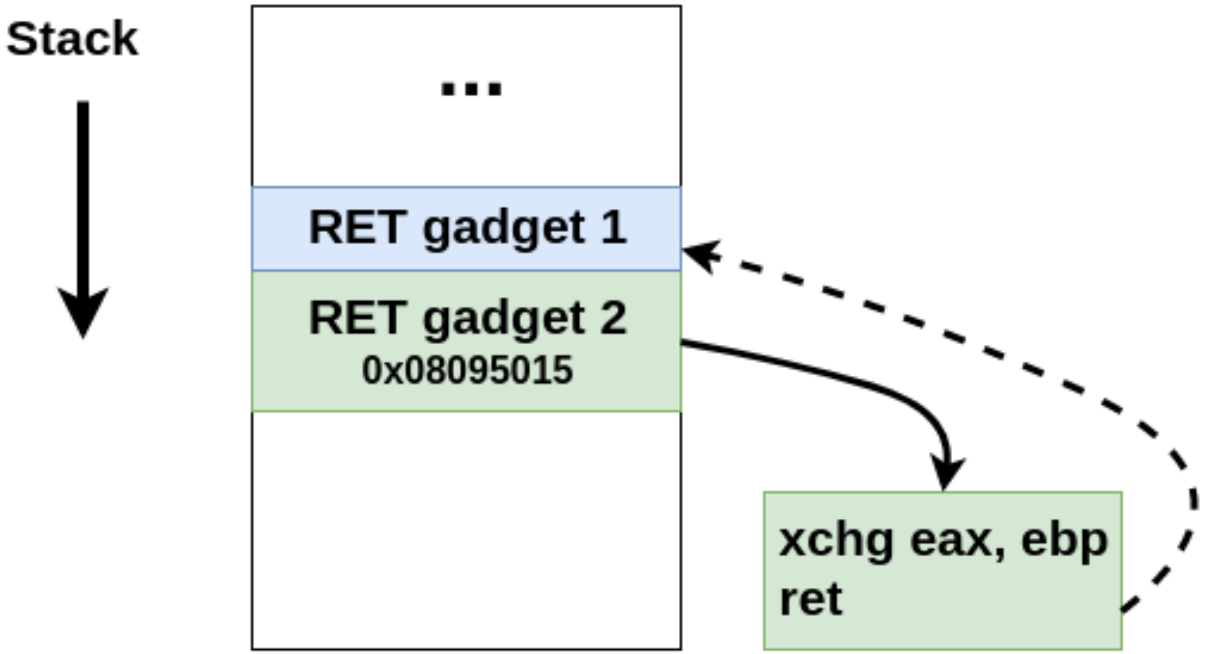
ROP



State

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

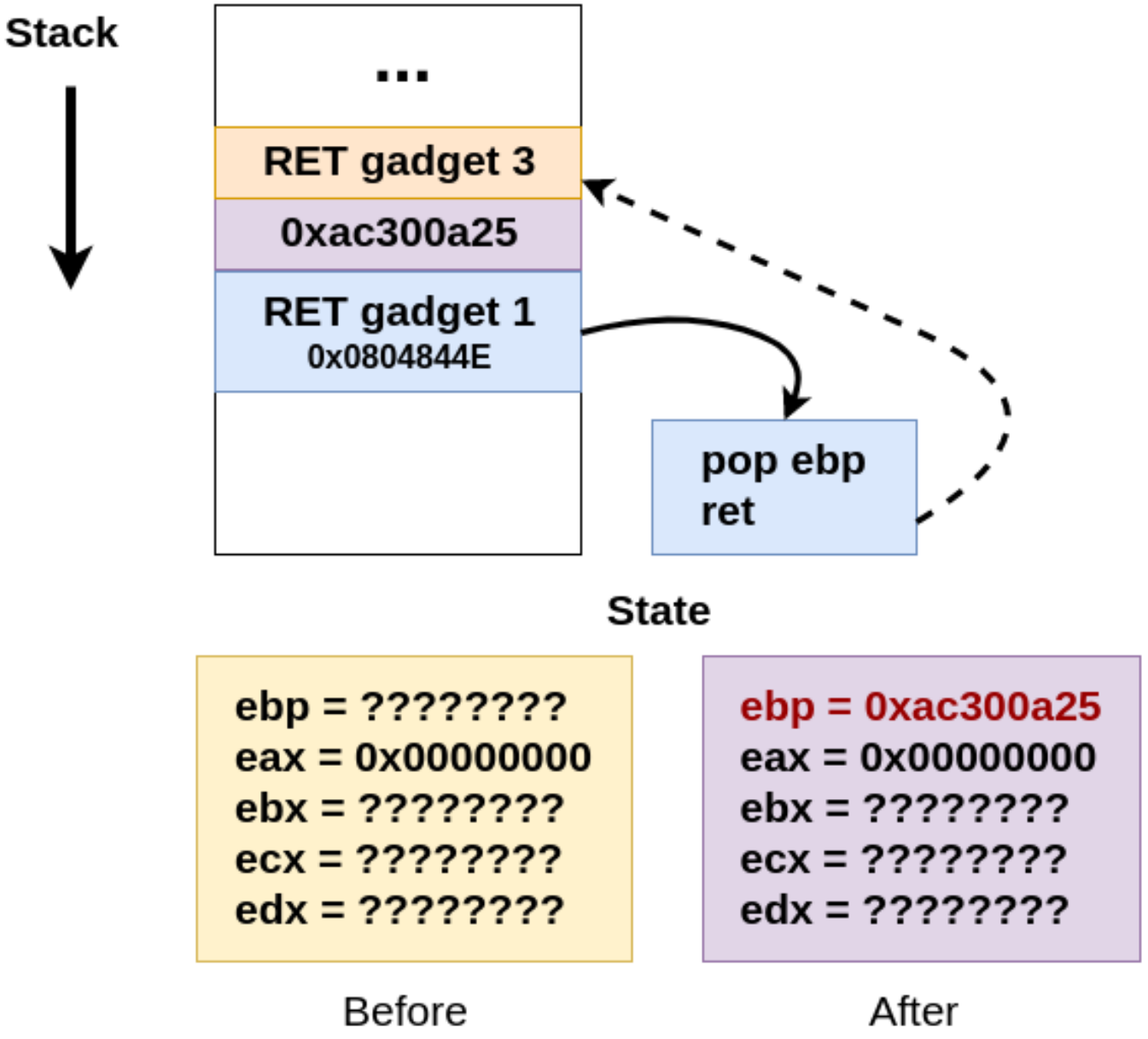
ROP



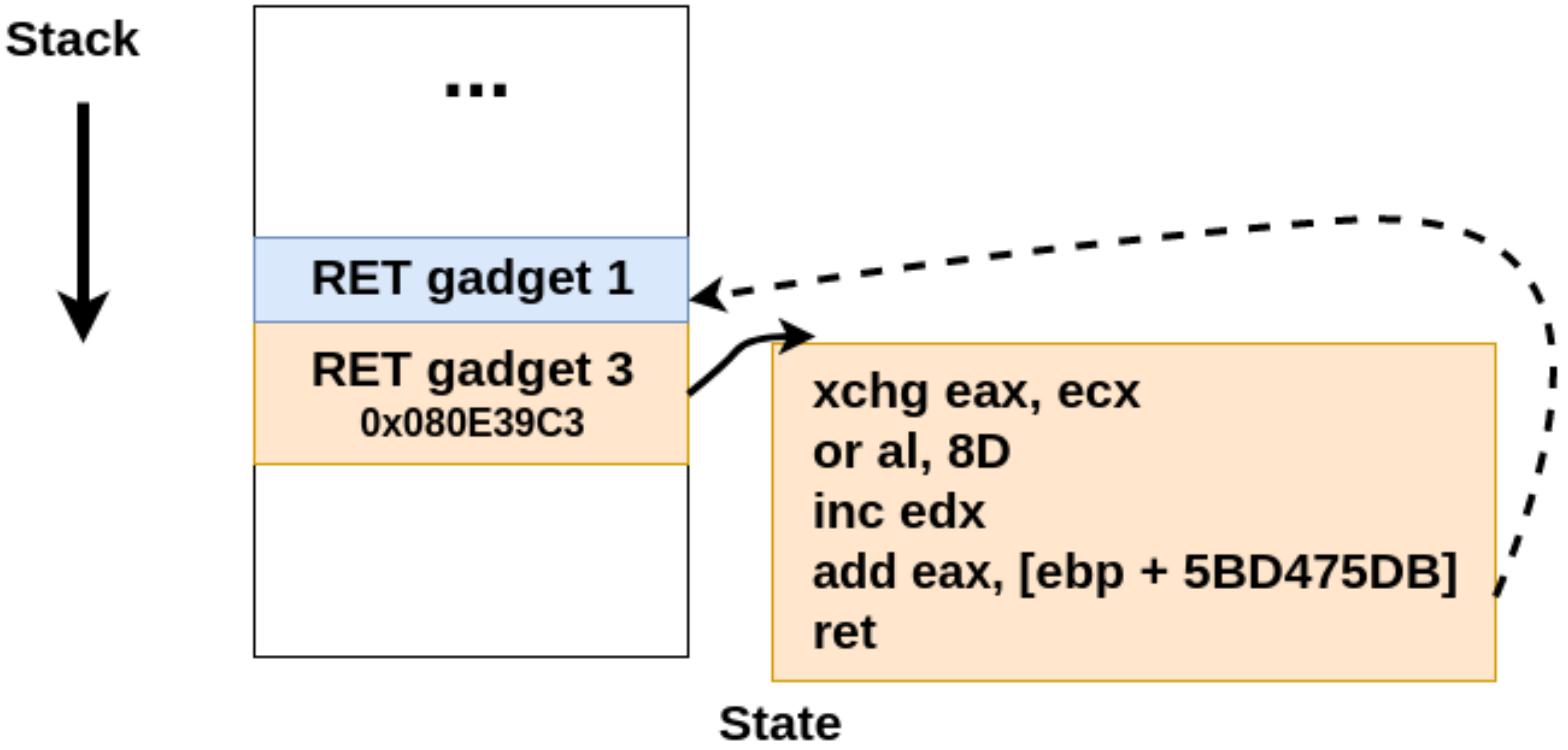
State

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

ROP


Stack
...
RET gadget 3
0xac300a25
RET gadget 1
0x0804844E
pop ebp
ret
State
ebp = ????????
eax = 0x00000000
ebx = ????????
ecx = ????????
edx = ????????
ebp = 0xac300a25
eax = 0x00000000
ebx = ????????
ecx = ????????
edx = ????????
Before
After

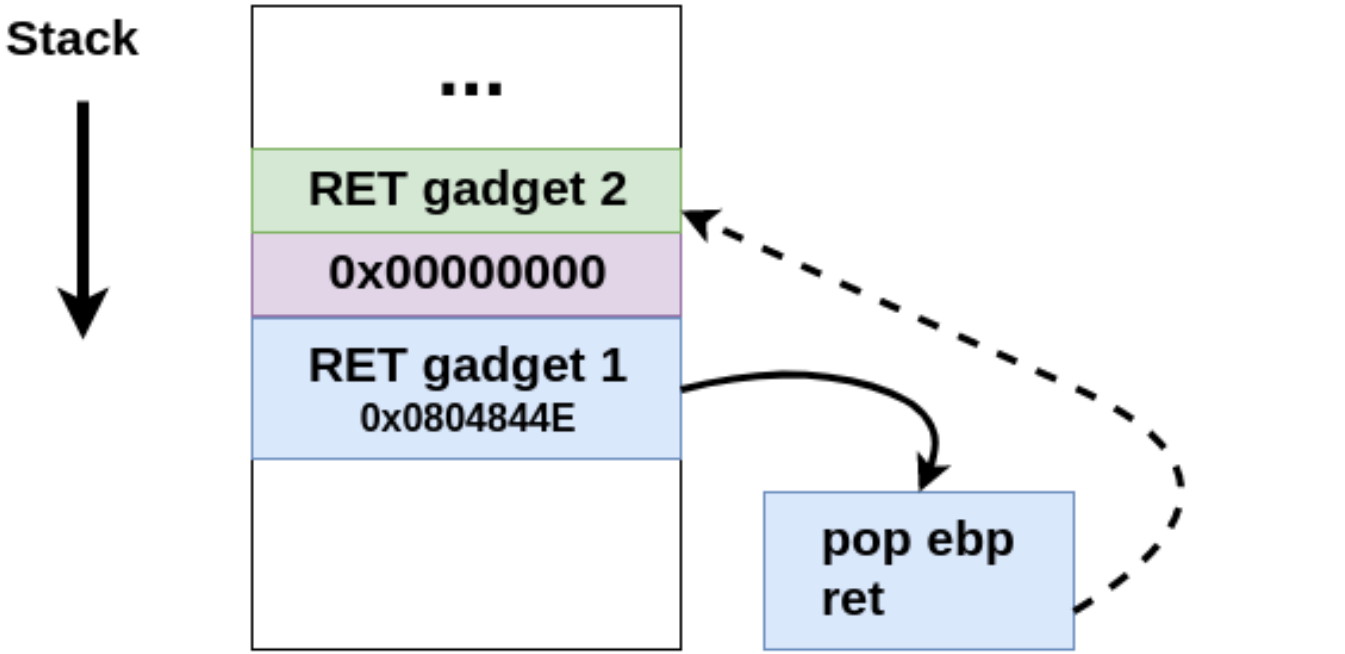
ROP



State

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

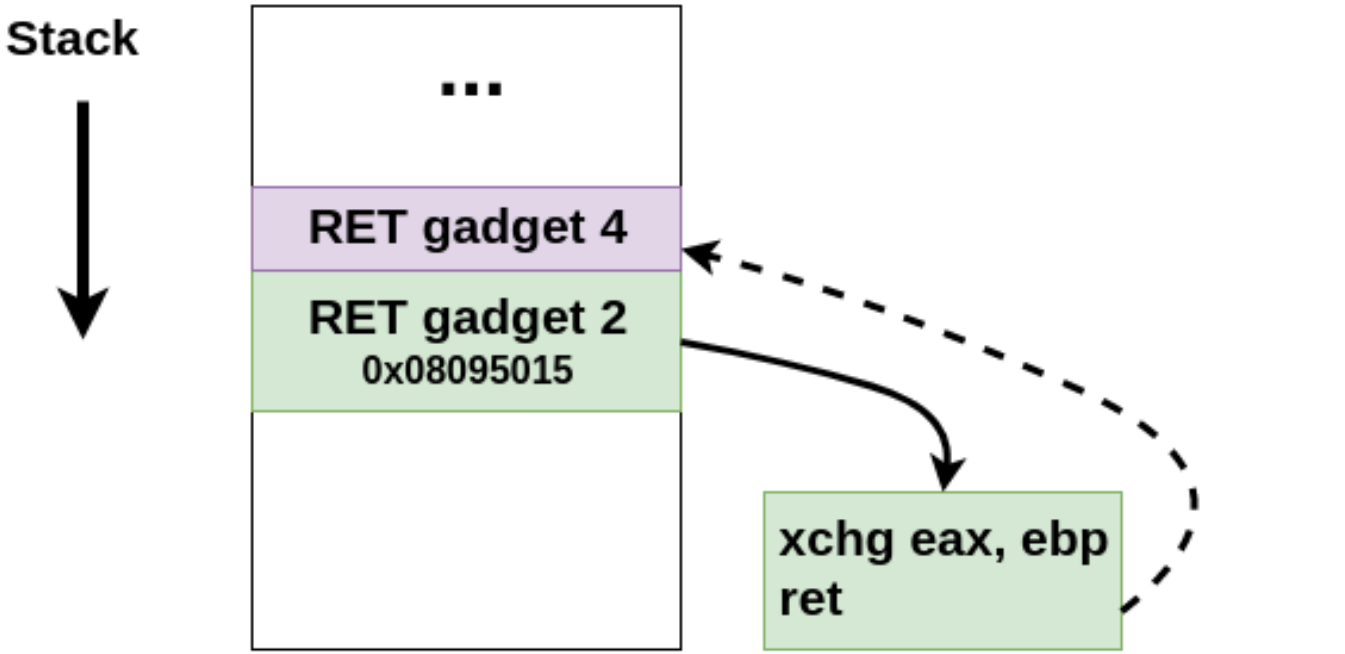
ROP



State

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

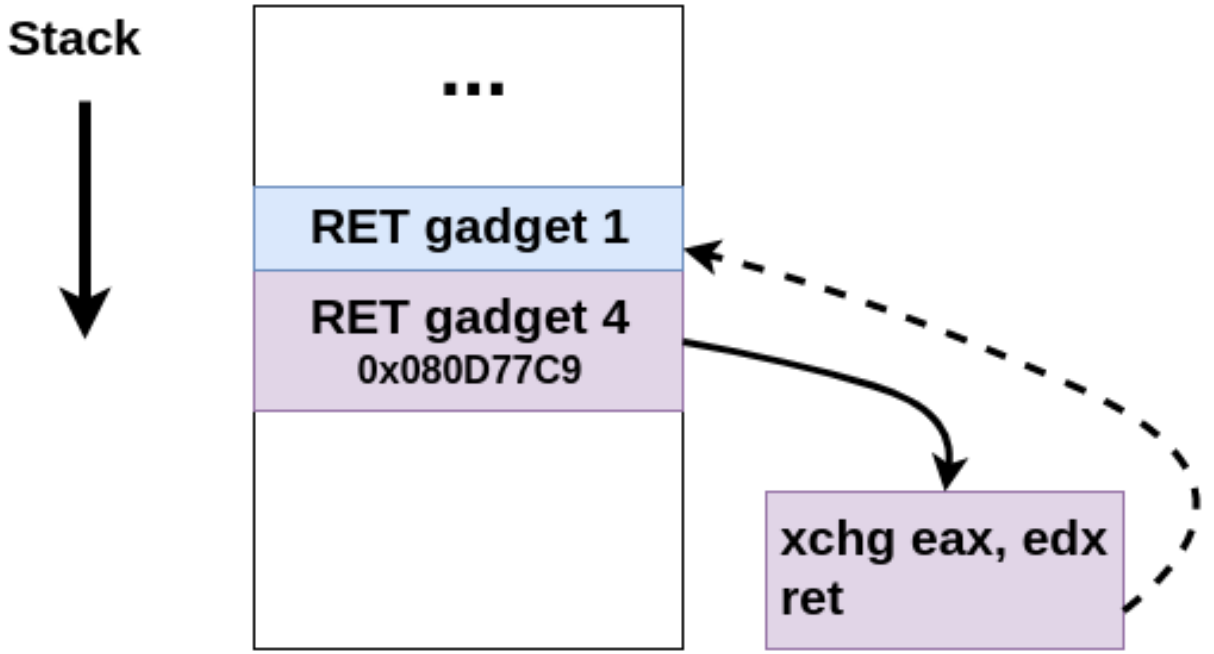
ROP



State

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

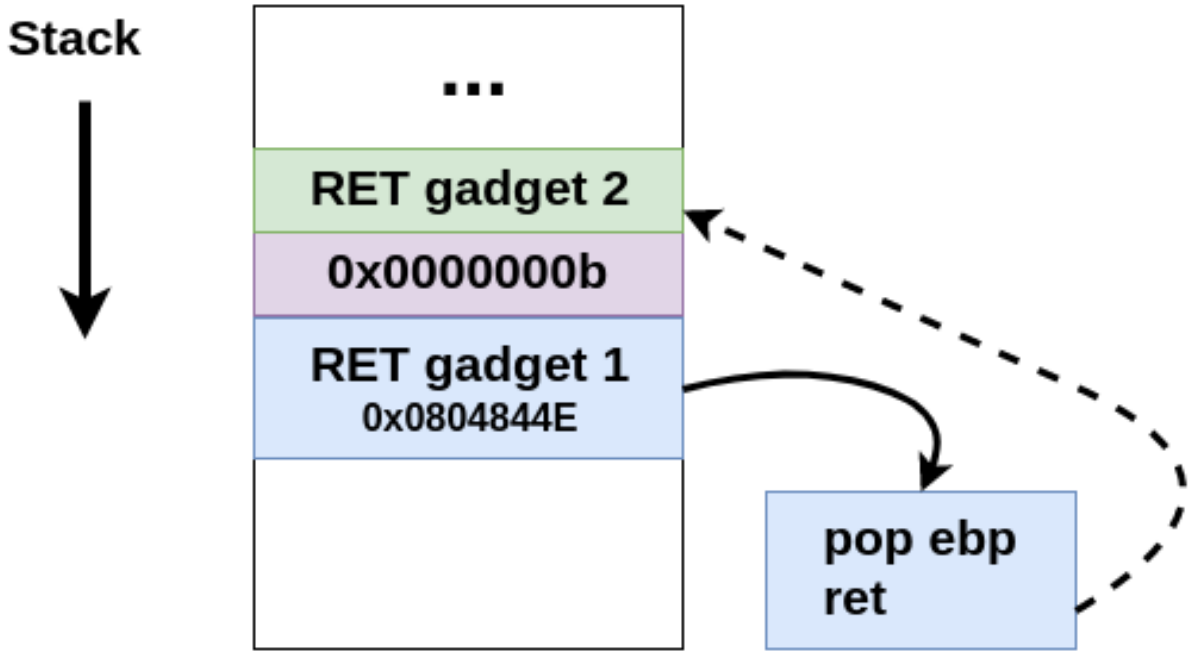
ROP



State

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

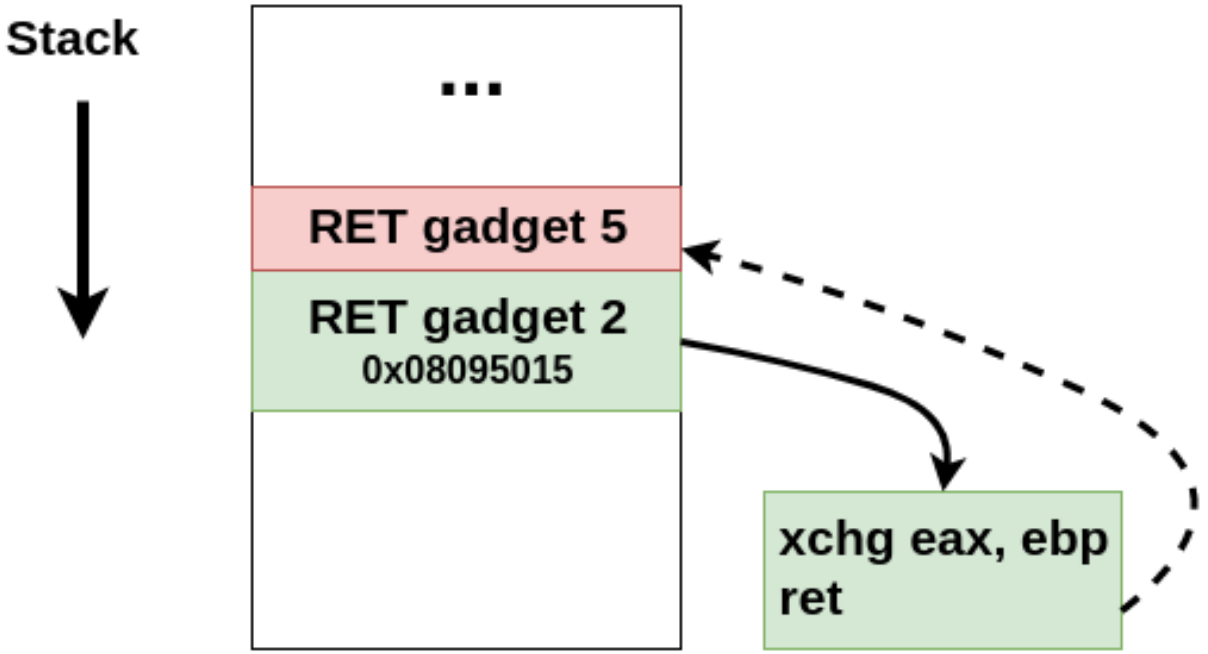
ROP



State

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

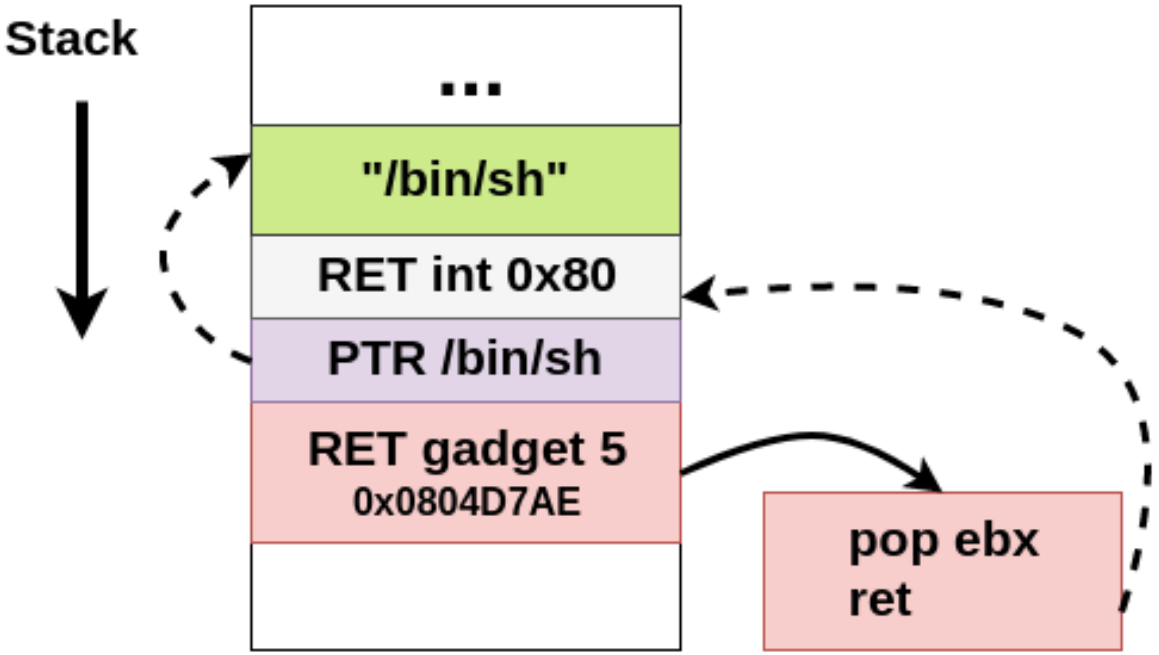
ROP



State

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

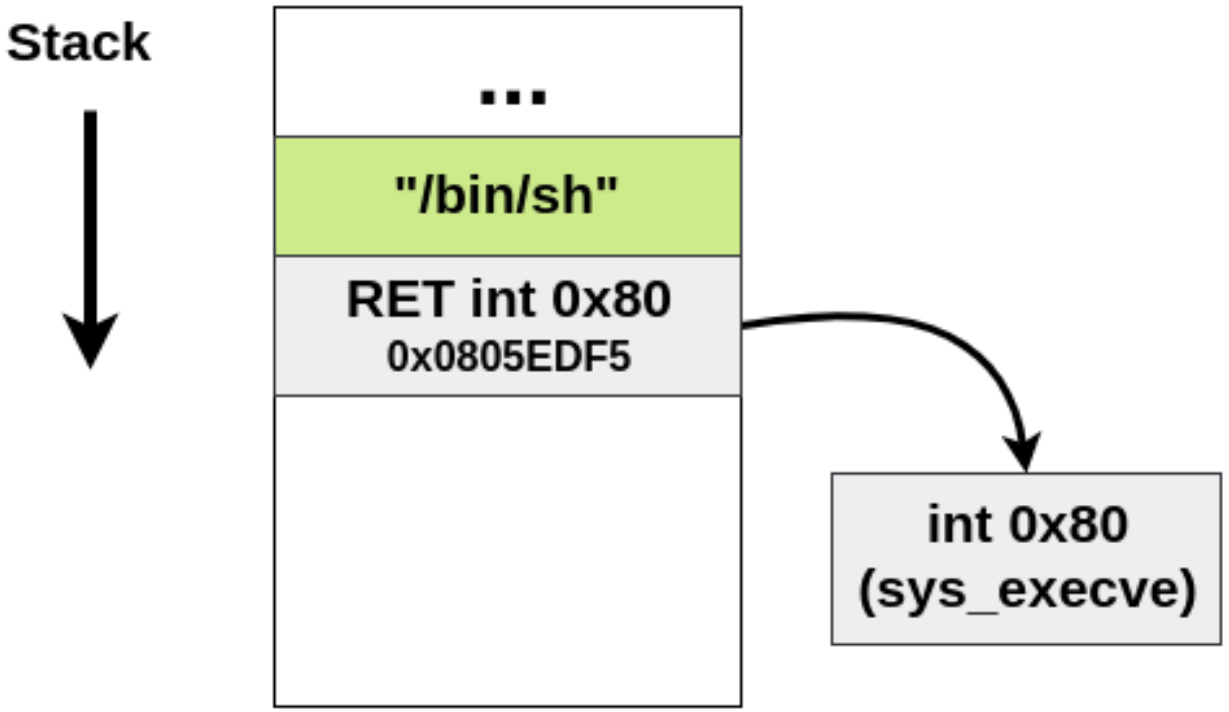
ROP



State

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

ROP



State

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

ROP

- ¿Cómo encontrar gadgets?
  - Herramientas de análisis estático
  - Herramientas de análisis dinámico: AGAFI
- Problema de satisfacción de restricciones
  - Efectos colaterales de algunos gadgets
  - Balanceo para reads/writes indirectos
- Instrucciones de pocos bytes son preferibles (Ej: xchg + ret son 2 bytes y no hay efectos colaterales)

ROP

- Saltos desalineados para encontrar gadgets
  - En x86/x86_64 se puede saltar desalineado
  - Arquitectura CISC tiene muchas instrucciones válidas, esto es una ventaja
- Instrucción POPAD es interesante
  - 1 byte de largo (0x61)
  - Carga múltiples registros con valores tomados del stack

ROP

- Múltiples formas de lograr el estado deseado. Ejemplo: poner eax en 0:
  - ¿eax ya es 0?
  - pop eax
  - xor eax, eax
  - mov eax, 0x0
  - dec eax
  - xchg eax, r (r = 0)
  - etc.

ROP

- PTR leaks: ¿hay algún registro apuntando a un lugar conocido en el momento del crash?
- Jump Oriented Programming: en lugar de RETs, jumps indirectos
- Call Oriented Programming: en lugar de RETs, calls indirectos
- En espacio de kernel ROP funciona exactamente de la misma forma

Demo 10.1

ROP chain en espacio de usuario

Control Flow Integrity

- Un programa tiene flujos de ejecución esperados, determinados por un grafo en tiempo de compilación/linkeo
- Un ataque ROP hace al programa ejecutar un flujo anómalo o inesperado
- ¿Puede el programa detectar cuando se rompe el flujo de ejecución esperado? Este sería un buen indicador de compromiso

Control Flow Integrity

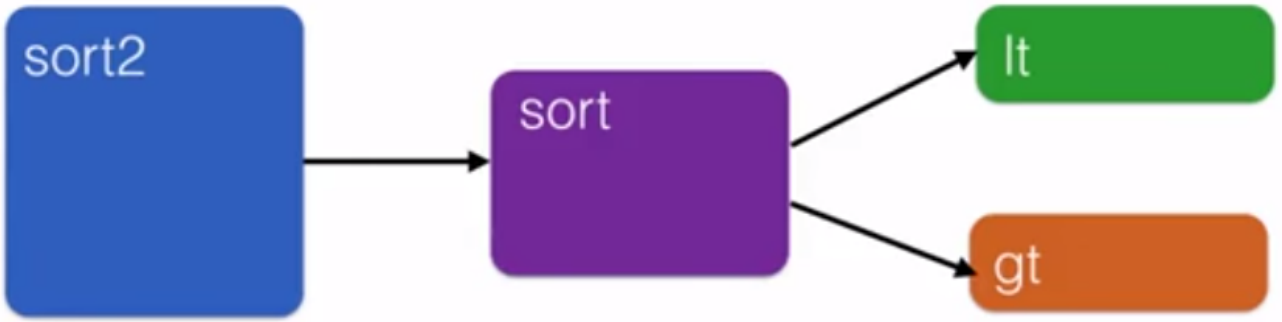
- Si asumimos DEP (Data Execution Prevention), ¿cómo puede el atacante corromper flujos?
  - CALL 0xAABBCCDD no se puede corromper: la memoria donde está el parámetro del call relativo está en el segmento de código (.text) y no es escribible
  - Se pueden corromper los flujos de código que dependen de datos (indirectos): CALL [REG] o JMP [REG] (siendo REG un registro cargado con un valor de la memoria), 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

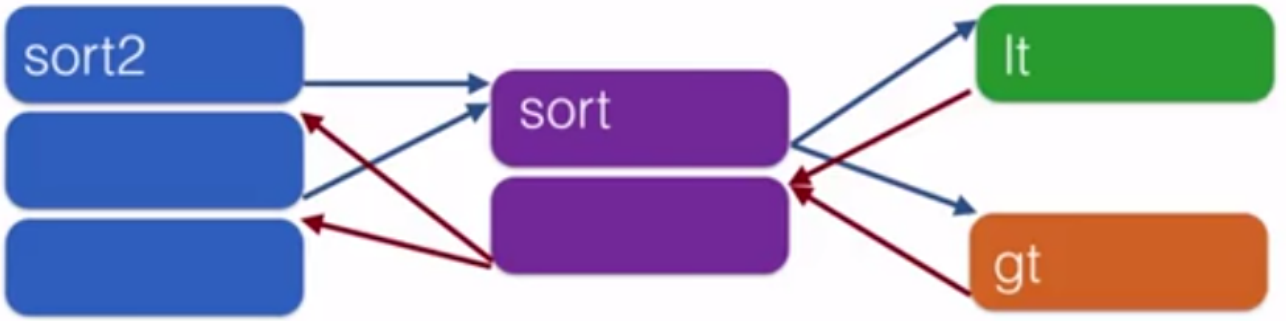
Imagen del curso Software Security (University of Maryland)

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***

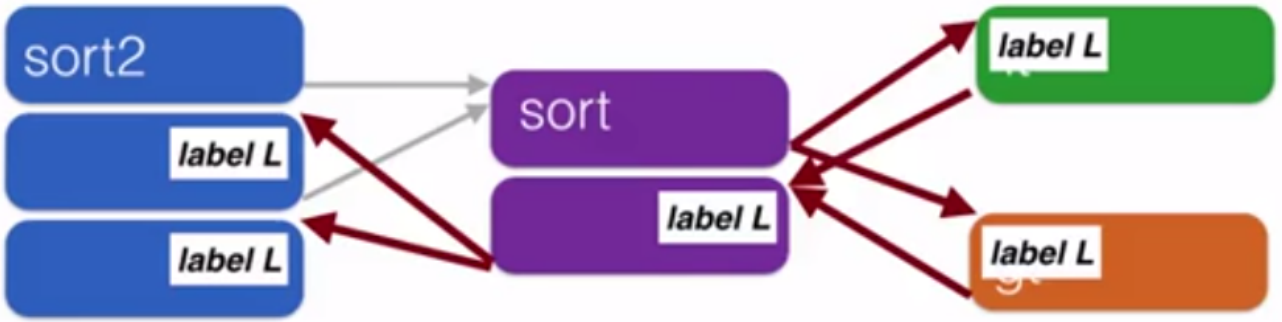
Imagen del curso Software Security (University of Maryland)

Control Flow Integrity

- Podemos etiquetar los destinos de los saltos indirectos. Esto es: agregar bytes de la etiqueta (no ejecutables) previo al destino del salto
- Antes de saltar, verificar la existencia de una etiqueta correcta en los bytes previos al destino del salto
- Si la etiqueta es correcta, se procede al salto. De lo contrario, se detecta un flujo anómalo
- Esto tiene una penalización de performance

Control Flow Integrity

Simplest labeling

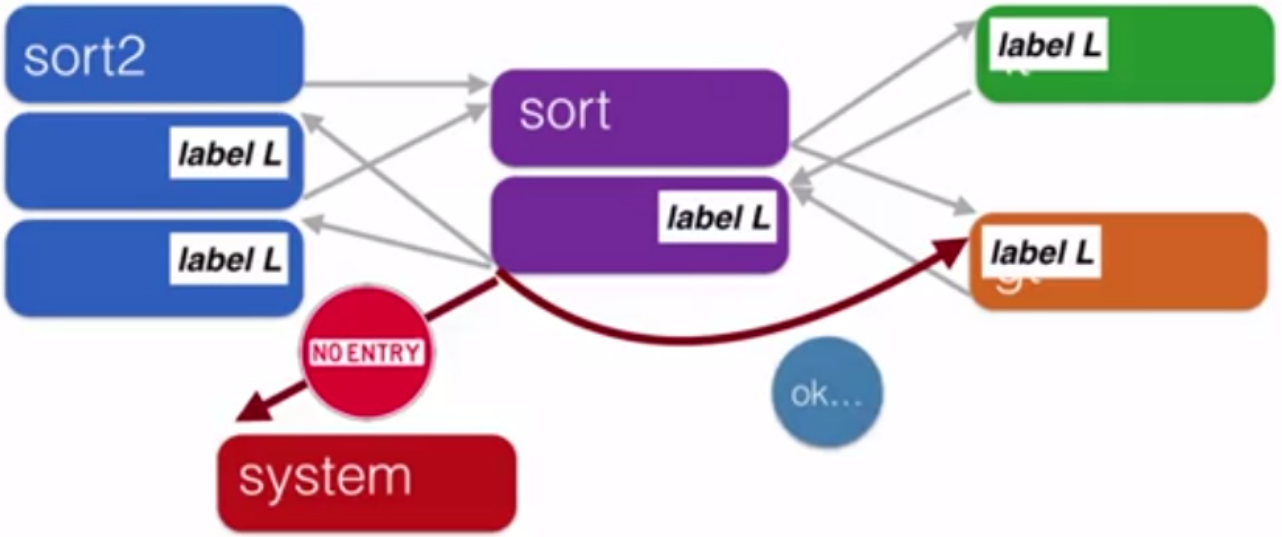


Use the same label at all targets

Imagen del curso Software Security (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

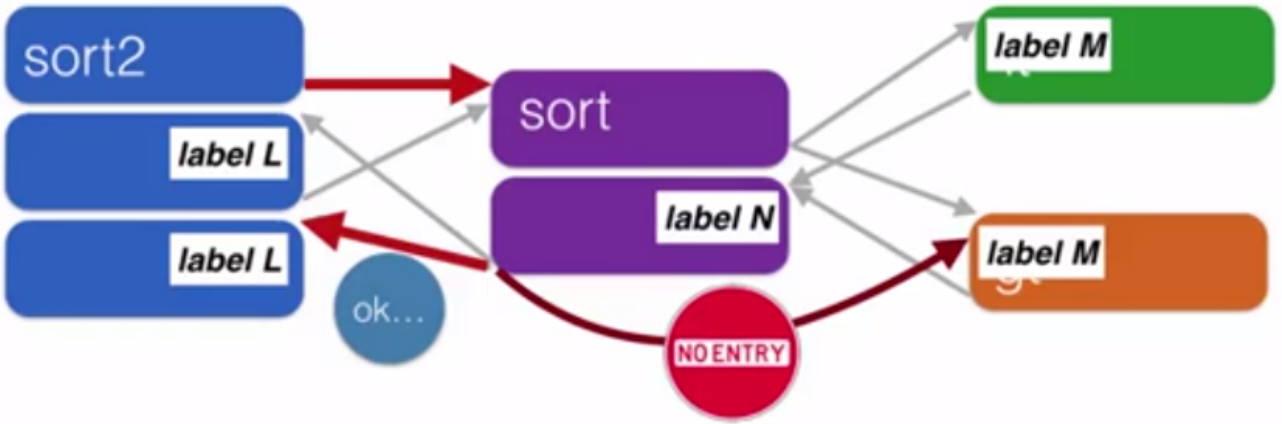
Imagen del curso Software Security (University of Maryland)

Control Flow Integrity

- Esta técnica no impide saltar hacia un lugar con la misma etiqueta, más allá de que este no sea un flujo posible en el grafo
- Es necesaria mayor granularidad de etiquetas para restringir estos casos

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

Imagen del curso Software Security (University of Maryland)

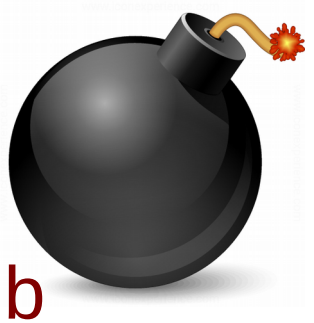
Control Flow Integrity

```
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;
}
```

Control Flow Integrity

clang++



```
movq  -48(%rbp), %rax   → puntero a objeto b
movq  (%rax), %rdi      → objeto b
movq  -56(%rbp), %rax   → puntero a objeto c
movq  (%rdi), %rcx      → vtable B
movq  %rcx, %rdx
subq  %r15, %rdx
rolq $59, %rdx          → chequeo de integridad: ¿es una vtable válida?
cmpq  $3, %rdx
jae  46 <_main+99>      → si no lo es, error
movq  (%rax), %rbx      → objeto c
callq  *(%rcx)          → call al 1er método de la vtable B
```

Lab

Ejercicio 10.2

ROP chain en espacio de usuario

Ejecutar shellcode en stack

Referencias

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