COSC 366
Intro to Computer Security

Lecture 04
Software Security

Dr. Suya

Fall 2024

Review what we've learned last week.

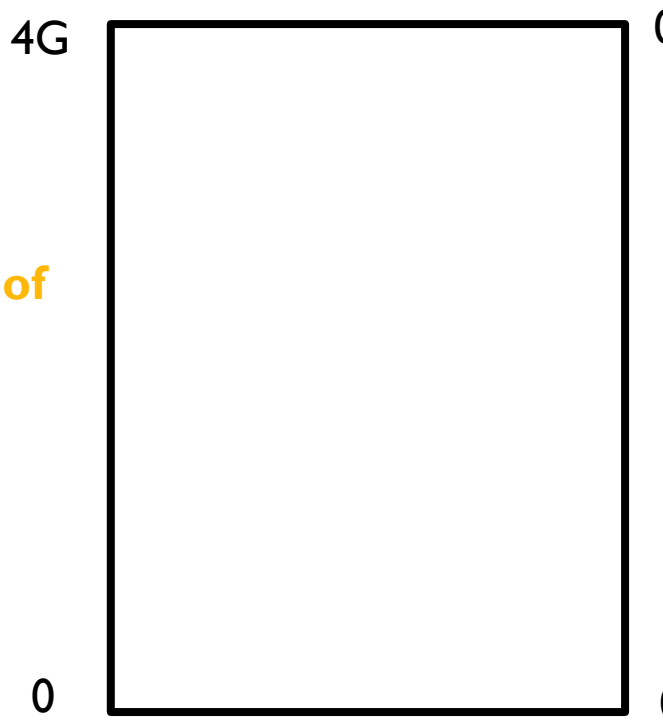
- Function
- Memory layout
- Stack layout

Function

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```

- Function name
  - Main
- Arguments
  - none
- Local variables
  - E.g., a, b
- Return address
  - Invisible
- Return value
  - 1

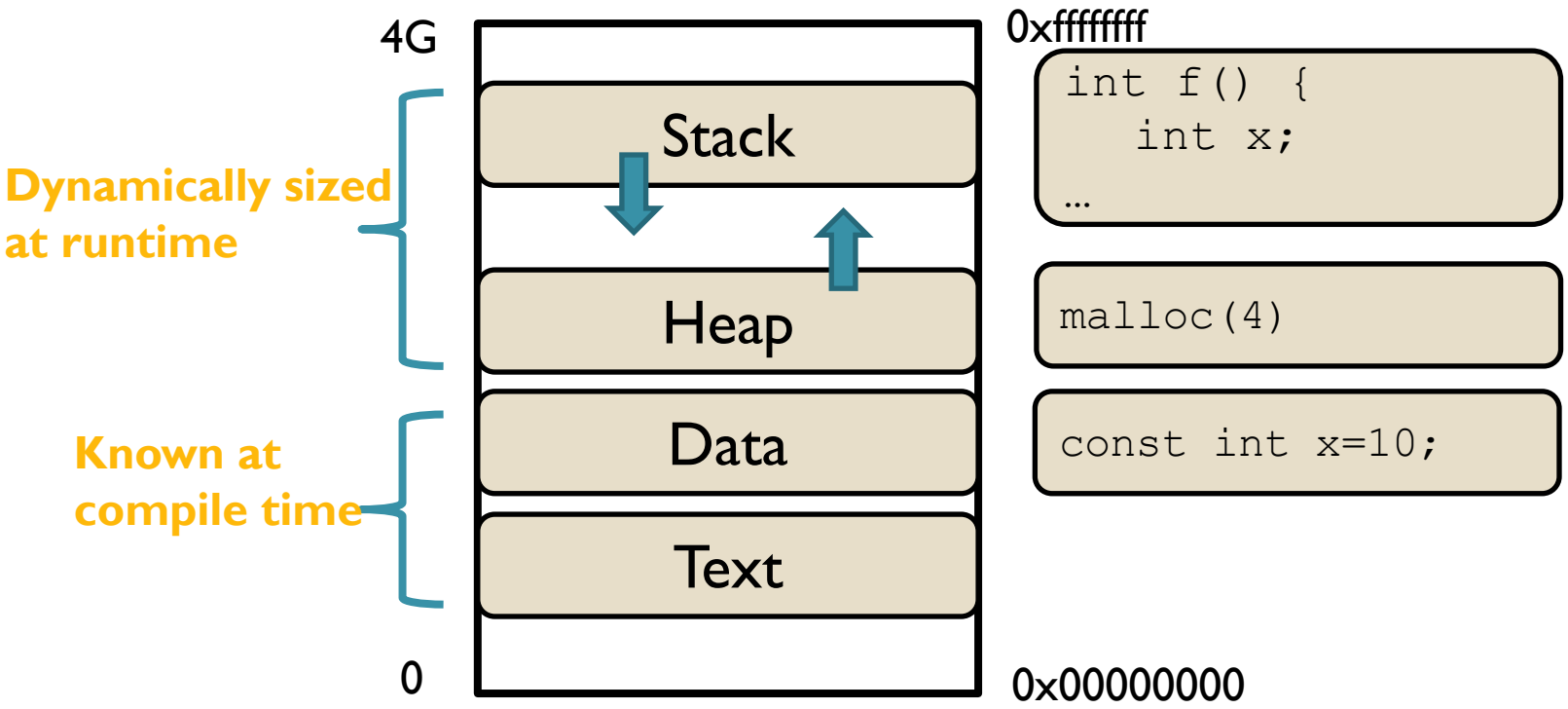
All programs are stored in memory



The process's view of memory is that it owns all of it

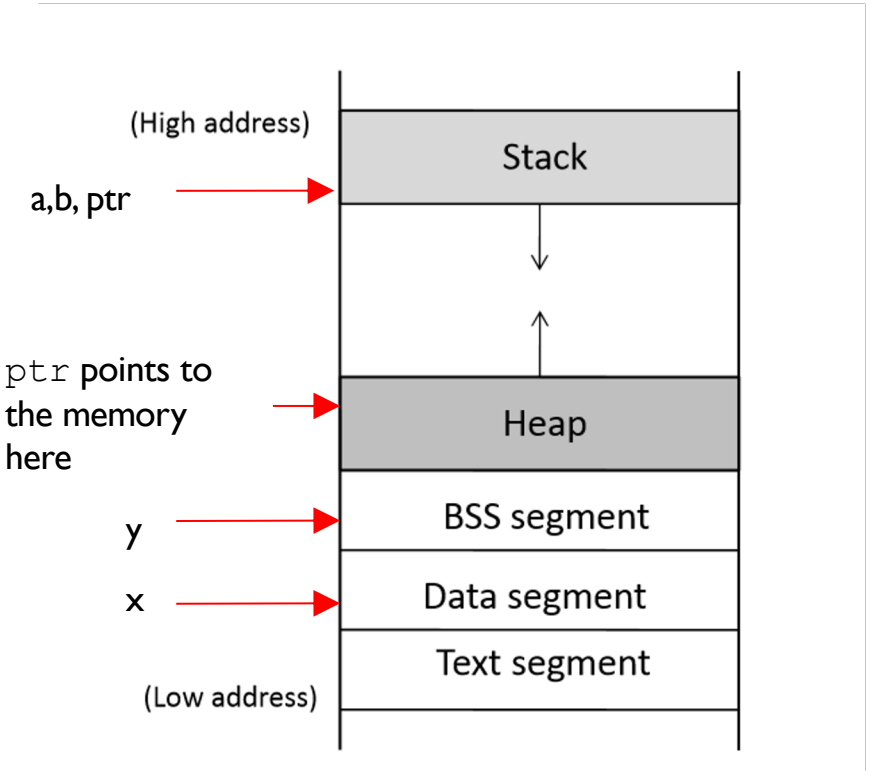
In reality, these are virtual address; the OS/CPU map them to physical addresses.

Stack & Heap grow in opposite directions



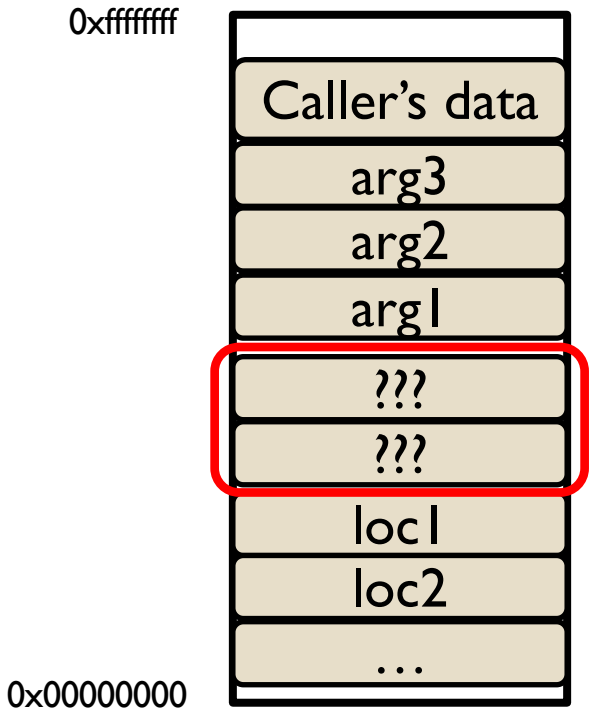
Program Memory Stack

```
int x = 100;
int main()
{
   // data stored on stack
   int   a=2;
   float b=2.5;
   static int y;

   // allocate memory on heap
   int *ptr = (int *) malloc(2*sizeof(int));

   // values 5 and 6 stored on heap
   ptr[0]=5;
   ptr[1]=6;

   // deallocate memory on heap
   free(ptr);

   return 1;
}
```



Stack layout when calling function

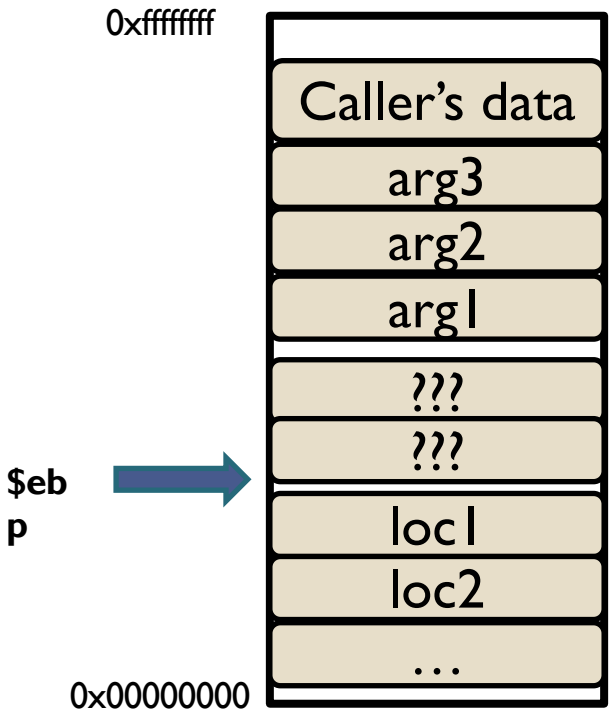
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



EBP (Base Pointer)

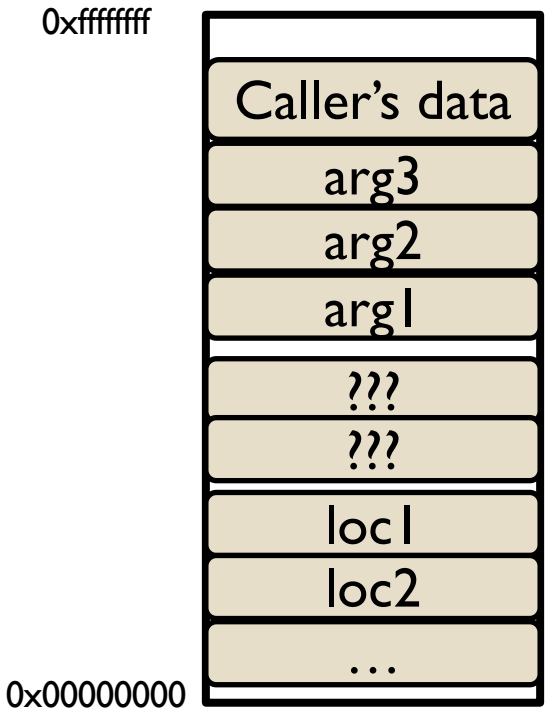
```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```

Q) Where is loc2?
What's the specific address?
A) But we can know loc2 is always 8bytes before "???"s ➔ addr of ??? - 8B



Stack layout when calling function

```
void func(char *arg1, int arg2, int arg3)
{
    char loc1[4]
    int  loc2;
    int  loc3;
    ...
}
```



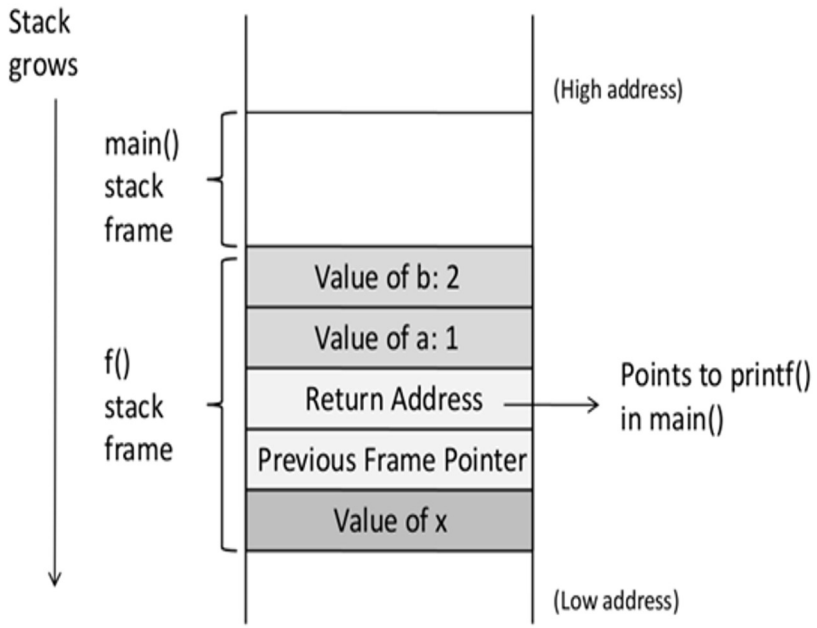
Q) What are "???" ?

First, we need $ebp

Second, we need a return address

Function Call Stack

```
void f(int a, int b)
{
  int x;
}
void main()
{
  f(1,2);
  printf("hello world");
}
```



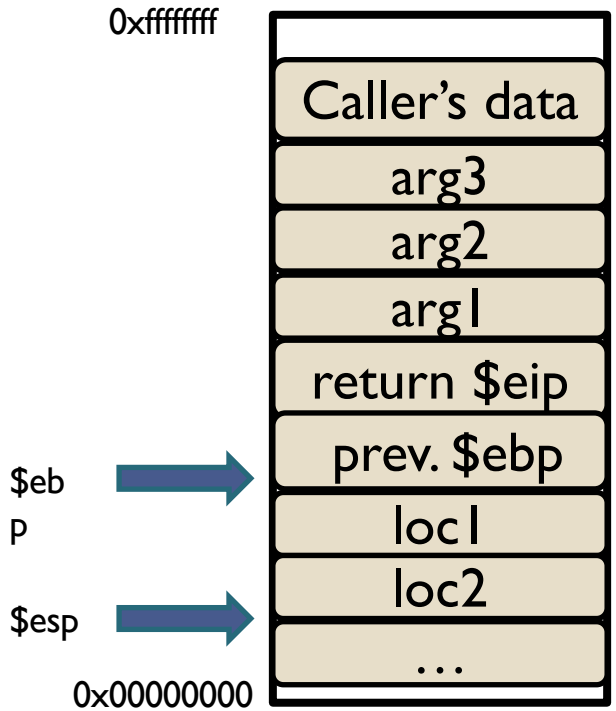
Returning from functions

In C

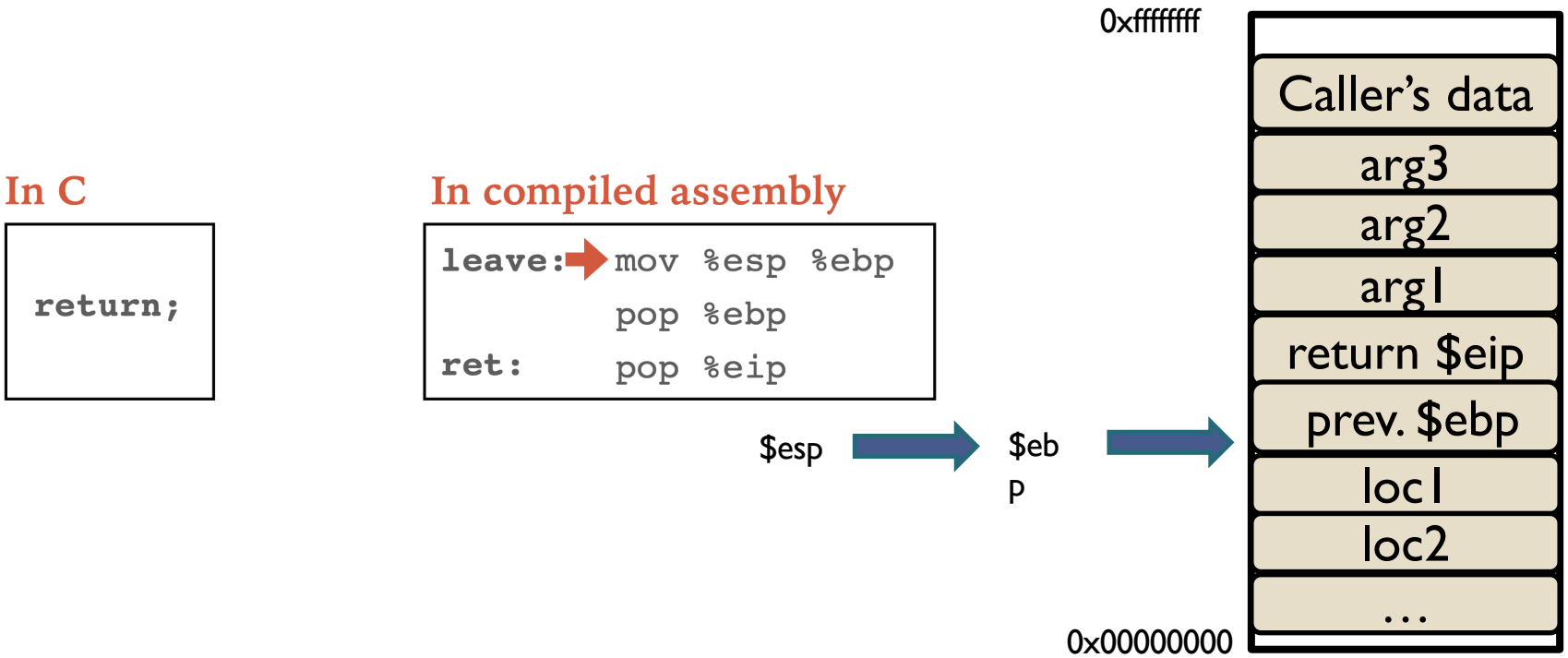
```
return;
```

In compiled assembly

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



Returning from functions



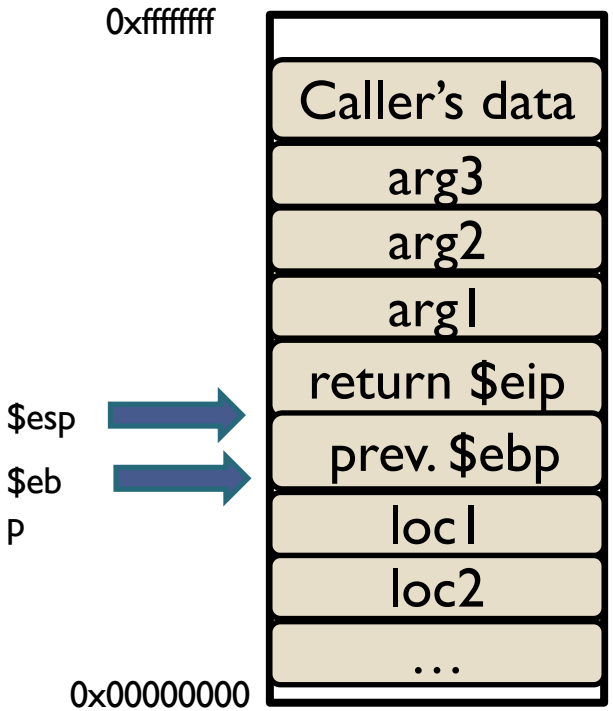
Returning from functions

In C

```
return;
```

In compiled assembly

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



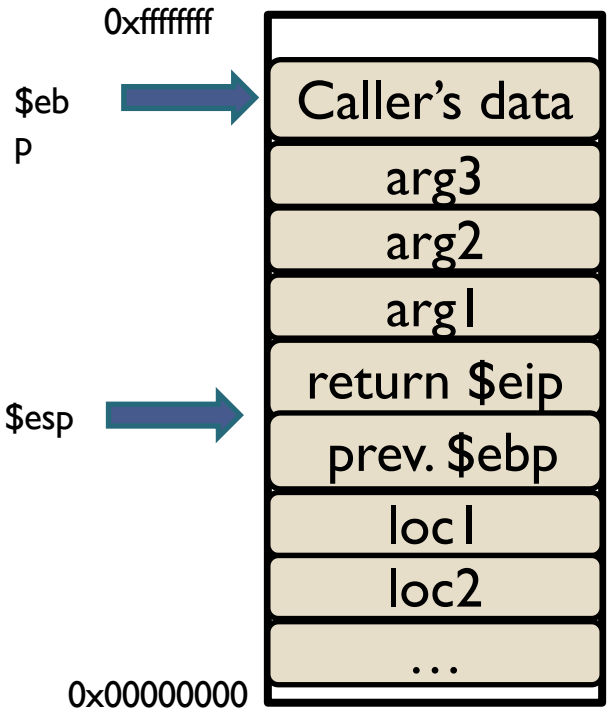
Returning from functions

In C

```
return;
```

In compiled assembly

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



Returning from functions

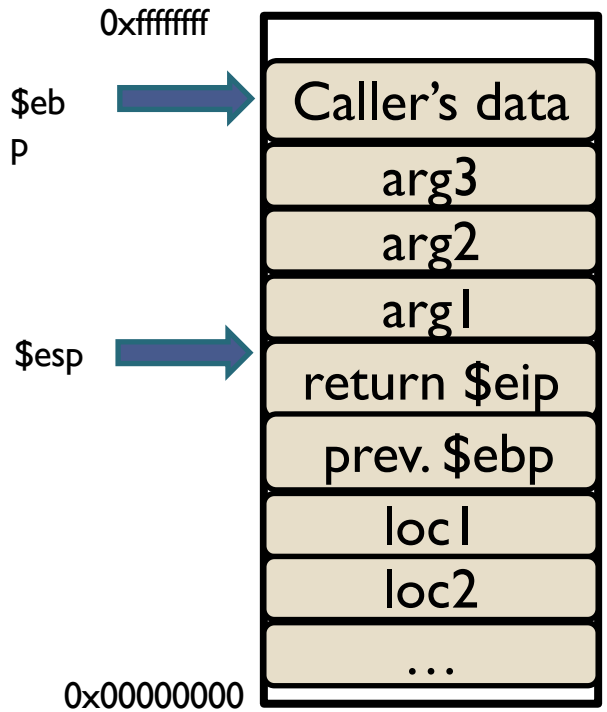
In C

```
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**

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

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)

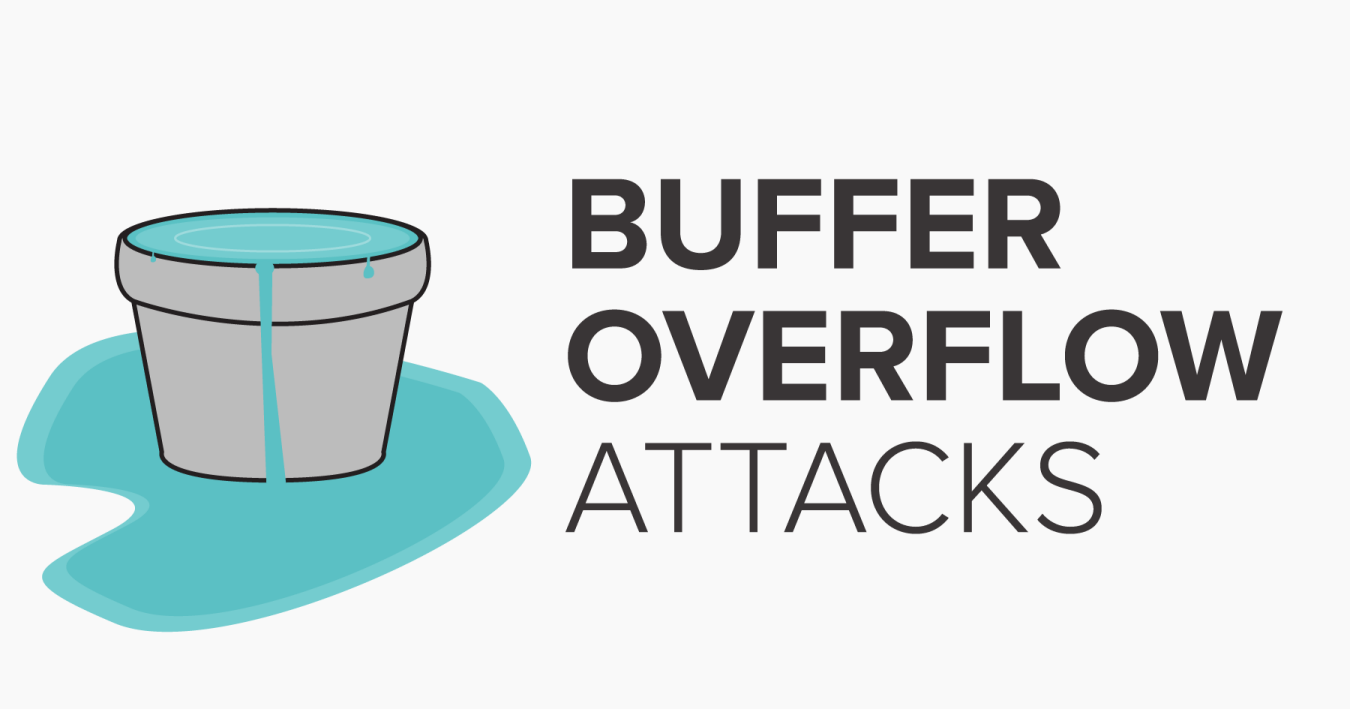
1. **Reset the previous stack frame**: %esp = $ebp; pop %ebp
2. **Jump back to return address**: pop %eip

Today's class

- Buffer Overflow Attack
  - We will focus on only Stack (it happens in heap though)

BUFFER OVERFLOW

Buffer overflows: High level



Buffer overflows: High level

- Buffer
  - Contiguous set of a given data type
  - Common in C
    - E.g., all strings are buffers of *char*'s
- Overflow
  - Put more into the buffer than it can hold
- Where does the extra data go?
  - Now, you are experts in memory layouts…

Buffer Overflow

```
char           A[8] = "";
unsigned short B    = 1979;
```

variable name	A								B	
value	[null string]								1979	
hex value	00	00	00	00	00	00	00	00	07	BB

How can we overwrite a value in an adjacent buffer?

variable name	A								B	
value	'e'	'x'	'c'	'e'	's'	's'	'i'	'v'	25856	
hex	65	78	63	65	73	73	69	76	65	00

How about this?

→ A[8] = 'A'

→ A[9] = 'B'

https://onlinegdb.com/Bj0Y0JwI-

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 (not applicable for other data types)

```
1 #include <string.h>
2 #include <stdio.h>
3 void main () {
4     char src[40]="Hello world \0 Extra string";
5     char dest[40];
6
7     // copy to dest (destination) from src (source)
8     strcpy (dest, src);
9 }
```

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`**

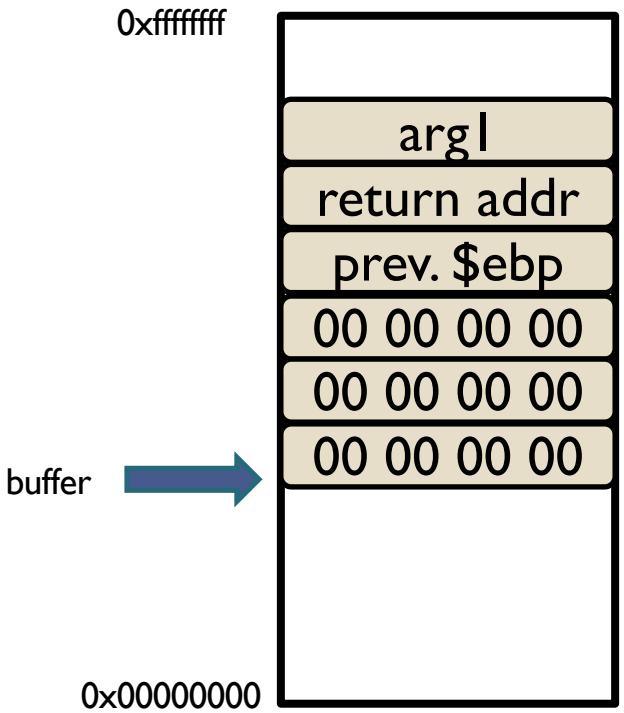
Common functions that cause overflow

- What if the string to copy is larger than the size of buffer?

```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;

}
```

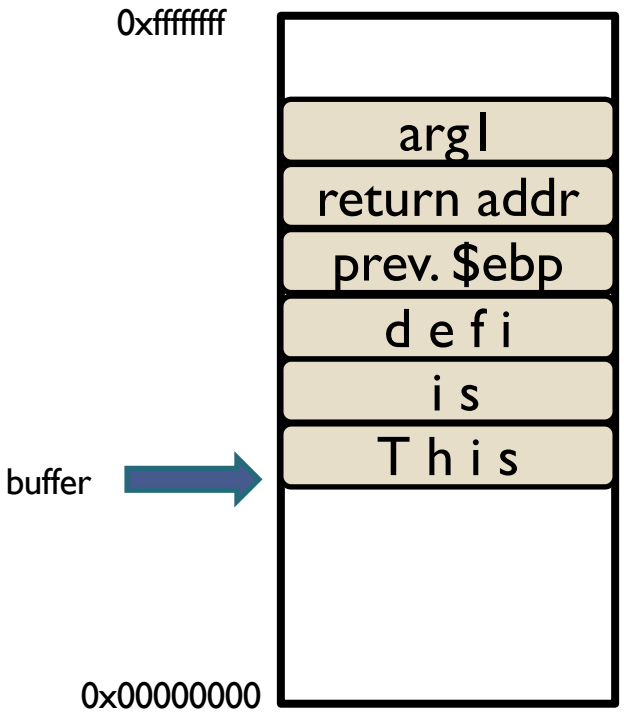
Common functions that cause overflow

```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;

}
```



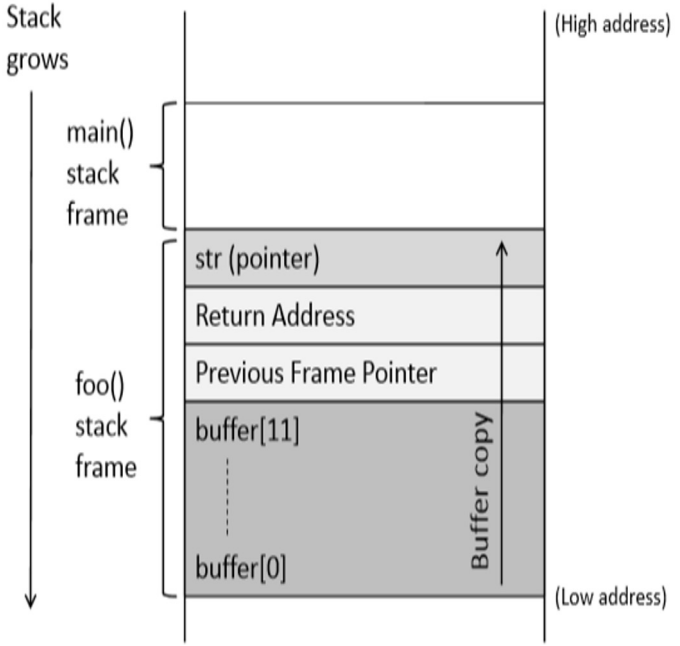
Common functions that cause overflow

```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

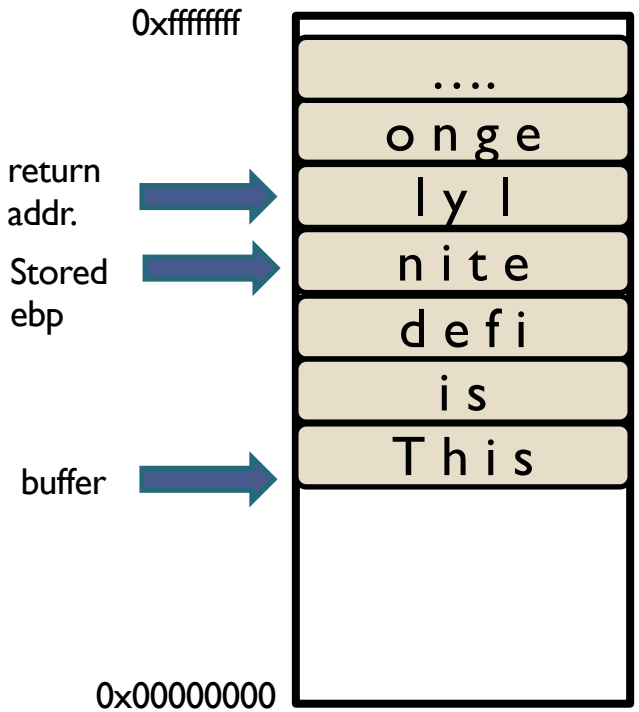
```c
#include <string.h>
void foo(char *str)
{
    char buffer[12];
    /* The following statement will result in a buffer overflow */
    strcpy(buffer, str);
}
int main() {
    char *str = "This is definitely longer than 12";
    foo(str);
    return 1;

}
```



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

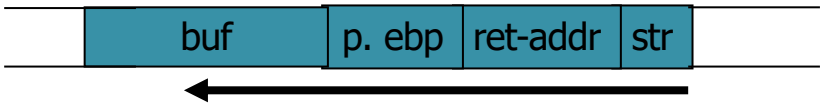


Example of a Stack-based Buffer Overflow

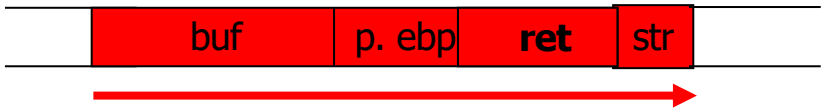
- Suppose a web server contains a function:

```
void func(char *str) {
                        char buf[128];
                        strcpy(buf, str);
                        do_something(buf);
                      }
```

- When the function is invoked, the stack looks like:



- What if `*str` is 136 bytes long? After `strcpy`:



What is common in these examples?

- Functions does not check the length.

Some Unsafe C Lib Functions

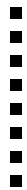
strcpy (char *dest, const char *src)

strcat (char *dest, const char *src)

gets (char *s)

scanf (const char *format, …)

sprintf (char *str, const char *format, …)



What's the common things?

- Functions does not check the length.
- 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

```
#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;
}
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAA\xfe\xca\xef\xbe
(little-endian system)

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
    - Malicious code to gain access

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 (e.g., in C) that does what you want
  - open a network connection
- Dump the machine code
  - machine code is what is injected as shellcode
- Need to adjust so there are no null bytes in it
  - (C) null bytes are interpreted as string terminators and stop early
- In practice there are repositories of this stuff on the Internet
  - Alphanumeric shellcode exists: if restricted to alphanumereics
  - "English" shellcode exists: if filters suspicious code

Shellcode

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

Clears %eax, set to 0, act as NULL terminator for string code "/bin/sh"

Push NULL byte

//sh

/bin

Assembly

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

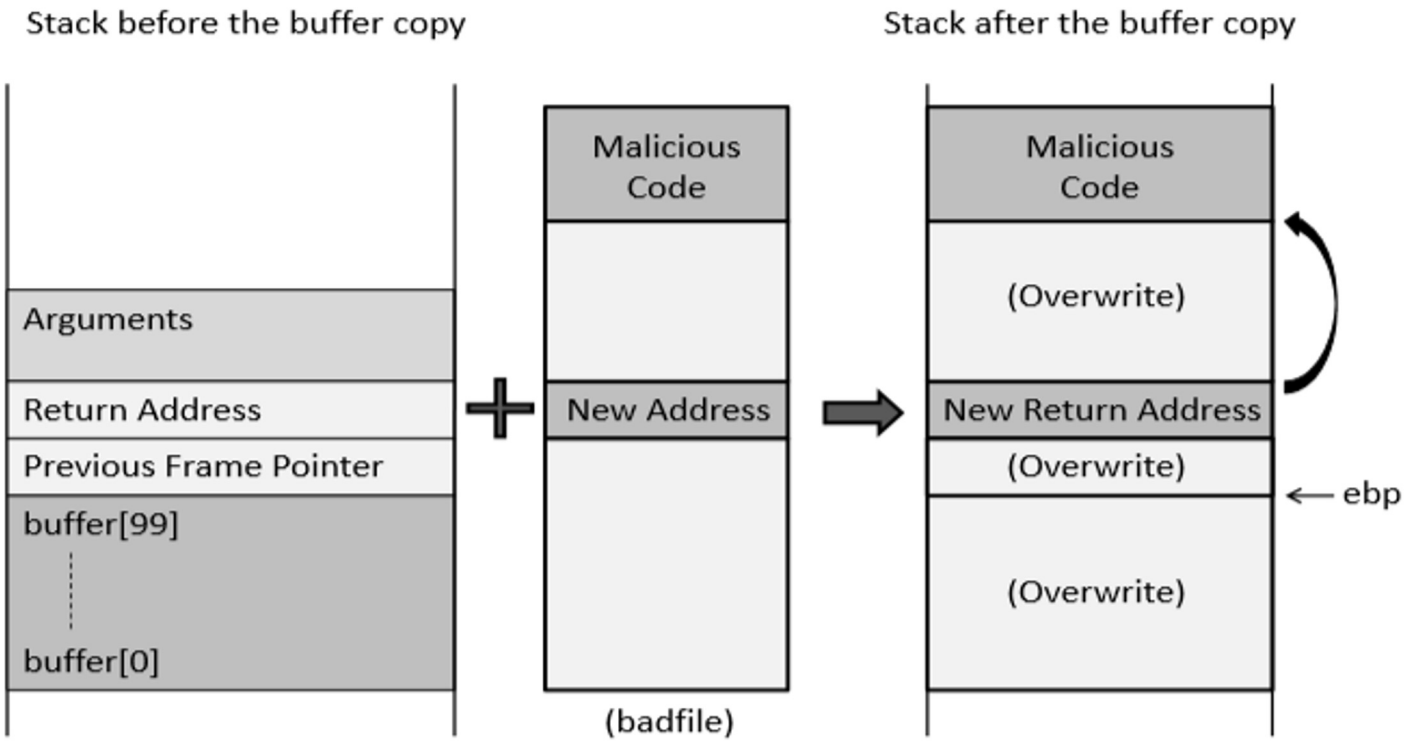
```
"\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
...
```

Machine code

Push NULL byte to indicate end of arguments

A register stores the address of first argument to a function

What if they are malicious code?



Environment Setup

1. Turn off address randomization (countermeasure)

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

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

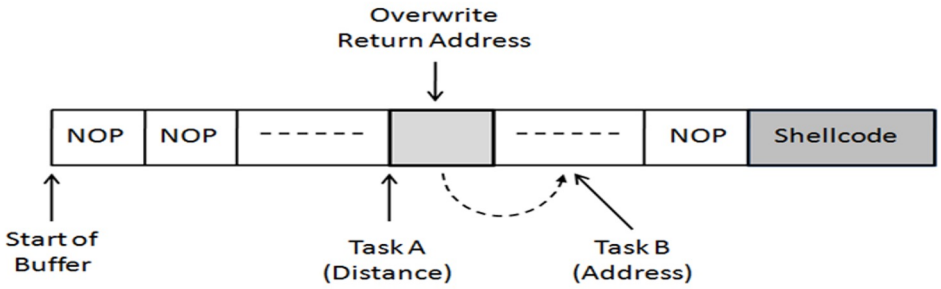
```
% gcc -o stack -z execstack -fno-stack-protector stack.c
% sudo chown root stack
% sudo chmod 4755 stack
```

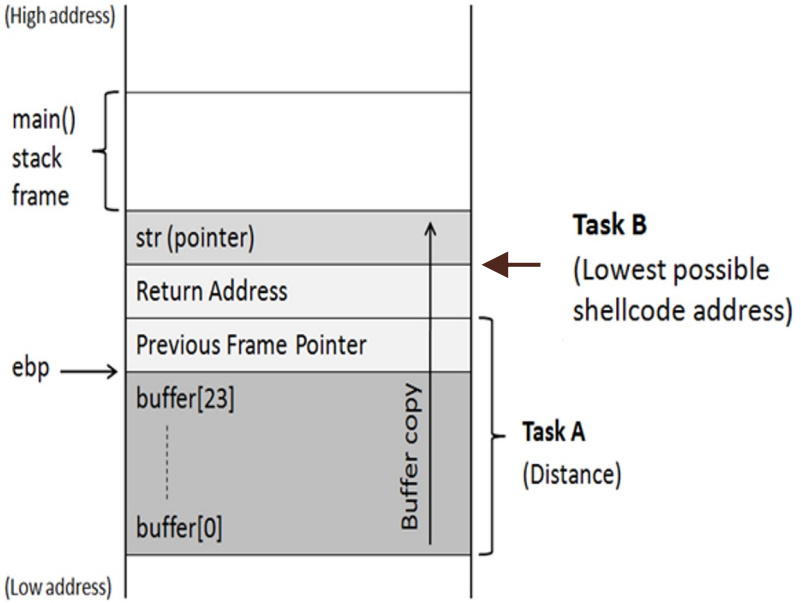
Use shell commands to setup

- have sufficient privilege
- exploit buffer overflow

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?
  - The memory layout can different from what was anticipated by the attacker (e.g. wrongly guessing user commands, additional environment variables)
- Also, may not exactly know where the return address is
- Solution?
  - NOP (0x90)
  - Using multiple (nearby) return addresses, combine with NOP

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.