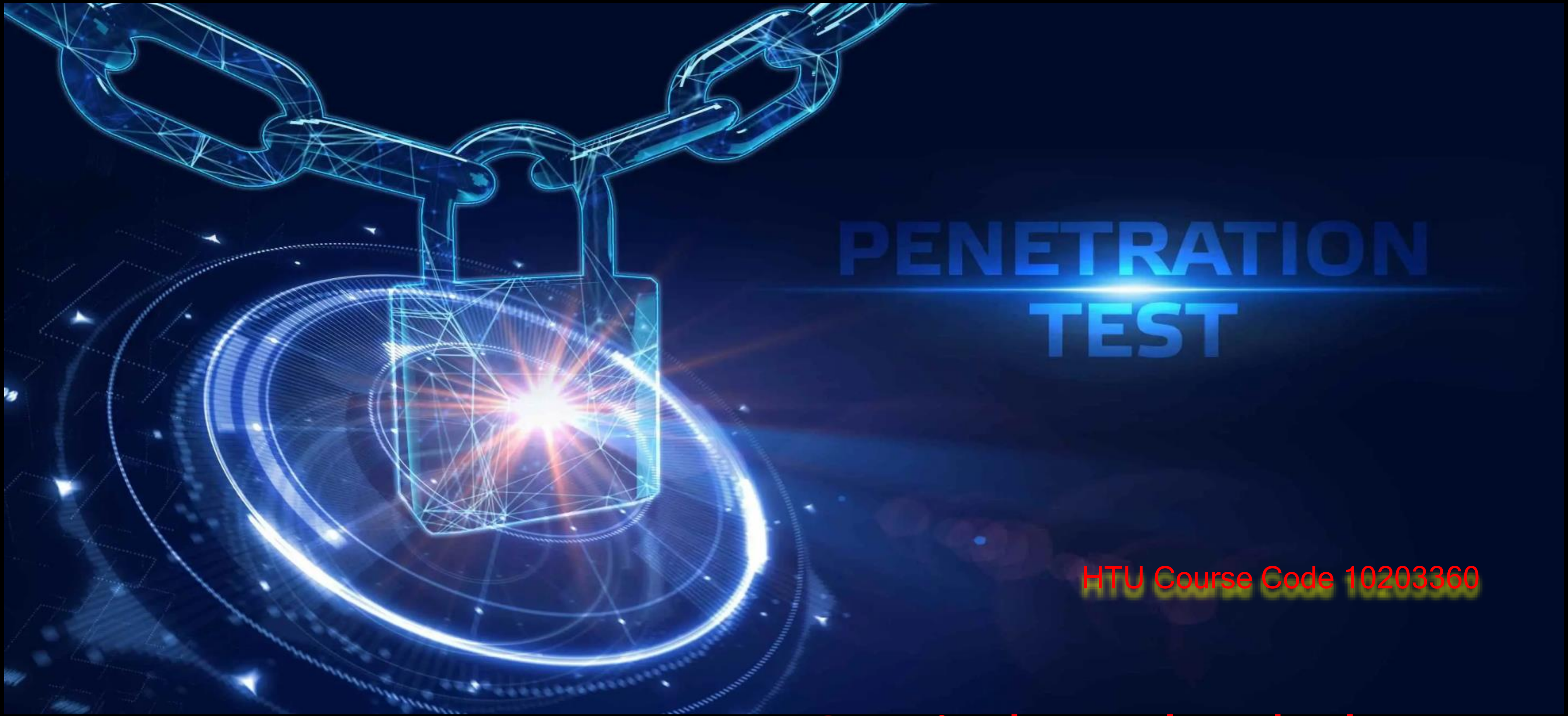


HTU Course Code 10203360

By: Sami Al-mashaqbeh

LEARNING OUTCOMES

Gain comprehensive knowledge of CPU instructions, memory addressing mechanisms, and vulnerability exploitation techniques, while mastering Python and Bash scripting for automating tasks and executing complex penetration testing scripts effectively.

- Demonstrate understanding of CPU instructions by identifying and explaining the operation of common instruction sets and their impact on system performance.
- Comparing in detail between CPU pointers and their importance in controlling the running program inside the memory
- Discuss the stack buffer overflow and how it can crash or control the running software inside the memory.

CPU INSTRUCTIONS & REGISTERS

SOFTWARE EXPLOITATION INTRO.

- A program is made of a set of rules following a certain execution flow that tells the computer what to do.
- Exploiting the program (Goal):
  - Getting the computer to do what you want it to do, even if the program was designed to prevent that action *[The Art of Exploitation, 2nd Ed]*.
- First documented attack 1972 (US Air Force Study).
- Even with the new mitigation techniques, software today is still exploited!

CPU INSTRUCTIONS & REGISTERS

- The CPU contains many registers depending on its model & architecture.
- In this section, we are interested in three registers: EBP, ESP, and EIP which is the instruction pointer.
- (Instruction) is the lowest execution term for the CPU. (Statement) is a high level term that is compiled and then loaded as one or many instructions.
- Assembly language is the human friendly representation of the instructions machine code.

CPU REGISTERS OVERVIEW

16 Bits	32 Bits	64 Bits	Description
AX	EAX	RAX	Accumulator
BX	EBX	RBX	Base Index
CX	ECX	RCX	Counter
DX	EDX	RDX	Data
BP	EBP	RBP	Base Pointer
SP	ESP	RSP	Stack Pointer
IP	EIP	RIP	Instruction Pointer
SI	ESI	RSI	Source Index Pointer
DI	EDI	RDI	Destination Index Pointer

- Some registers can be accessed using there lower and higher words. For example, AX register; lower word AL and higher word AH can be accessed separately.
- The above is not the complete list of CPU registers.

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A function consist of:

- Name
- Parameters (or arguments)
- Body
- Local variable definitions
- Return value type

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
      myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions, High Level View

```
void myfun2(char *x) {
   printf(“You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf(“No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

PUSH position into the Stack

`myfun1(argv[1]);`

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun1(argv[1]);`

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

```
myfun1(argv[1]);
```

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

```
myfun1(argv[1]);
```

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
      myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun1(argv[1]);`

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
      myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

PUSH position into the Stack

`myfun2(buffer);`

`myfun1(argv[1]);`

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
      myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

myfun2(buffer);
myfun1(argv[1]);

Functions, High Level View

```
void myfun2(char *x) {
   printf(“You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
      myfun1(argv[1]);
   else printf(“No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

```
myfun2(buffer);
myfun1(argv[1]);
```

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun2(buffer);`
`myfun1(argv[1]);`

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

POP Position out of the Stack

myfun2(buffer);

myfun1(argv[1]);

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

```
myfun1(argv[1]);
```

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
        myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

POP Position out of the Stack

`myfun1(argv[1]);`

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions, High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions, High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

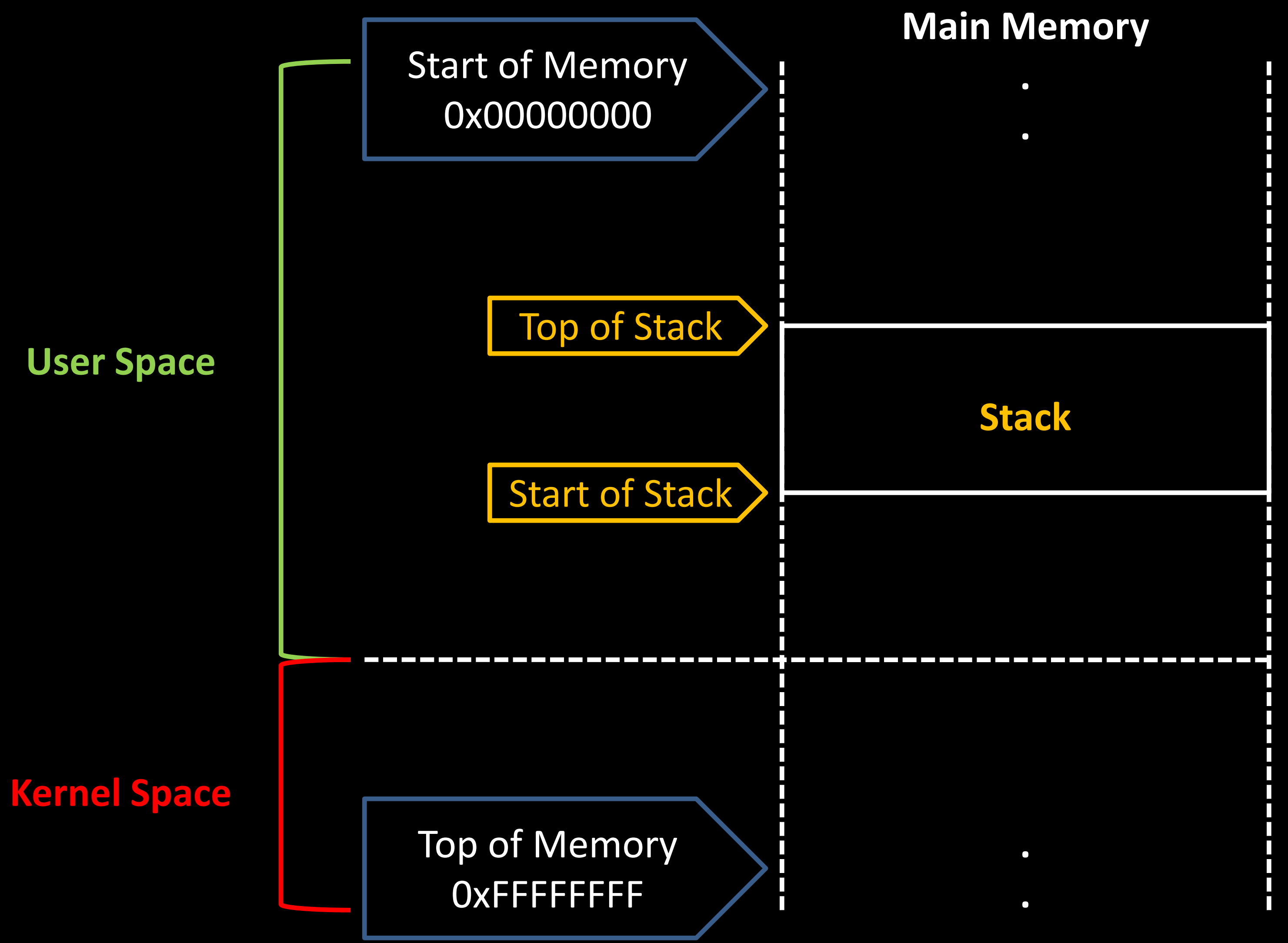
End of Execution

Saved Return Positions

Stack & Stack Frames

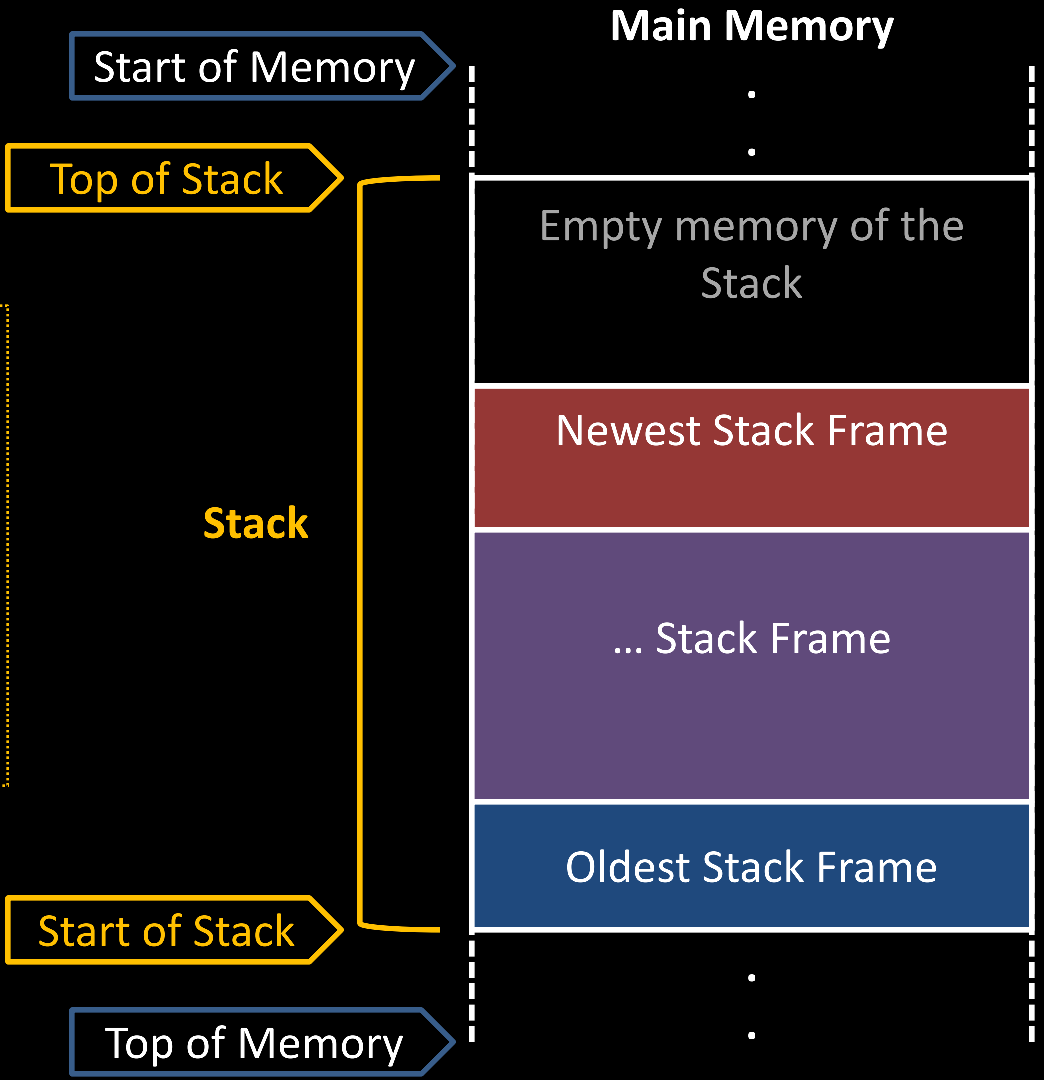
- There is no "physical" stack inside the CPU. Instead; the CPU uses the main memory to represent a "logical" structure of a stack.
- The operating system reserves a contiguous raw memory space for the stack. This stack is logically divided into many Stack Frames.
- The stack and all stack frames are represented in the memory upside-down.
- A stack frame is represented by two pointers:
  - Base pointer (saved in EBP register): the memory address that is equal to (EBP-1) is the first memory location of the stack frame.
  - Stack pointer (saved in ESP register): the memory address that is equal to (ESP) is the top memory location of the stack frame.
- When Pushing or Popping values, ESP register value is changed (the stack pointer moves)
- Base Pointer (value of EBP) never change unless the current Stack Frame is changed.
- The stack frame is empty when EBP value = ESP value.

Memory Addressing



Stack & Stack Frames inside the Main Memory

Note 1:
The newest stack frame is indexed as **Stack Frame 0**, the older one **Stack Frame 1**, And the Oldest Stack Frame is indexed **Stack Frame (count-1)**

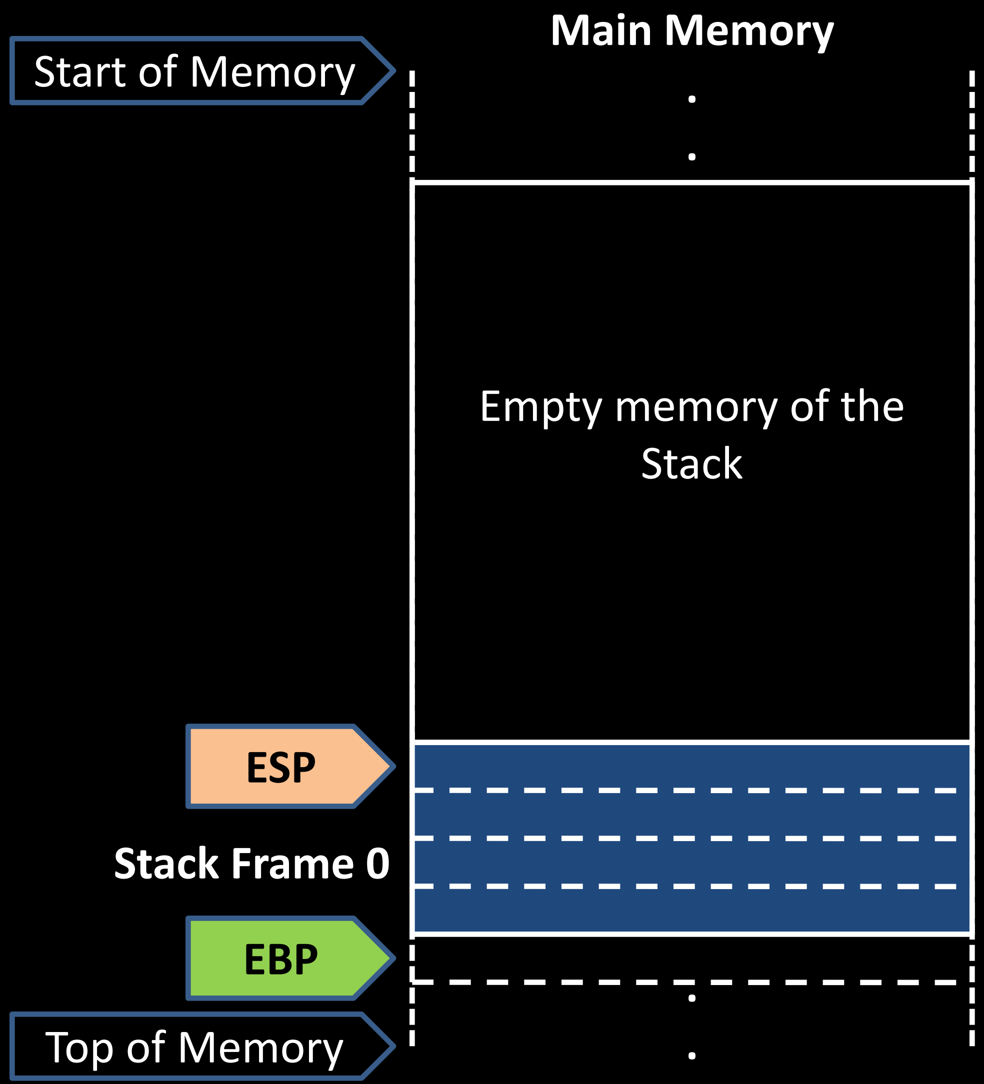


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

ESP points to the top of the current Stack Frame. And it points to the top of the **Stack** as well.

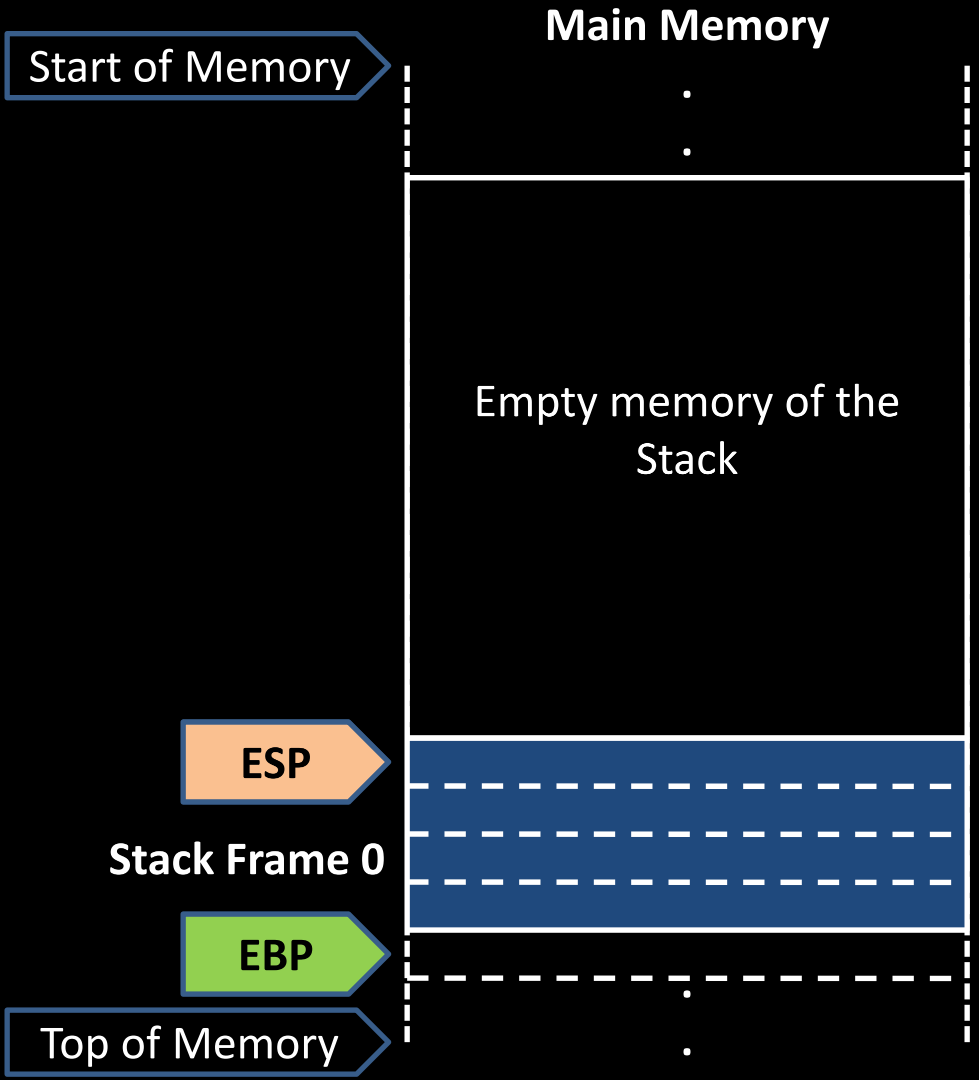
Whenever a function is called, a new Stack Frame is created. Local variables are also allocated in the bottom of the created Stack Frame.



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

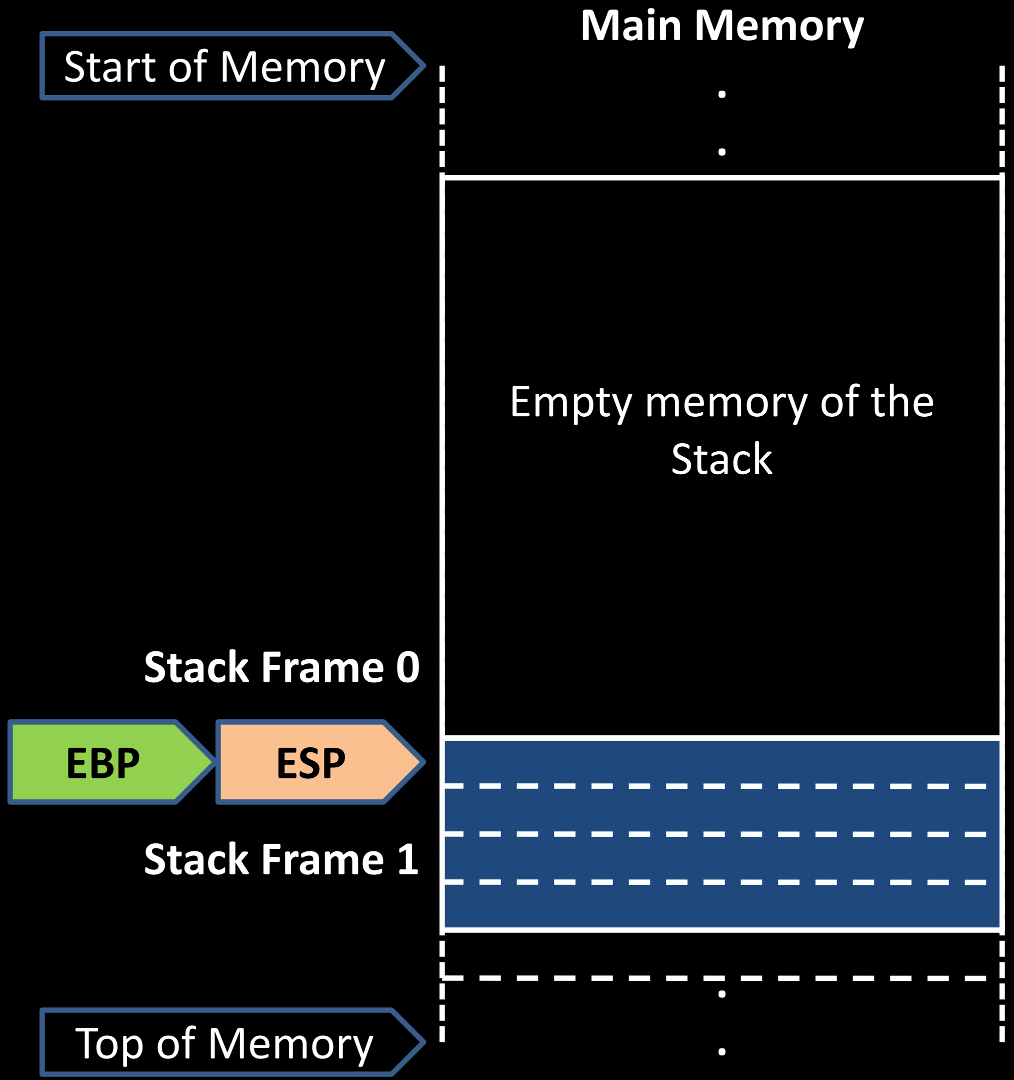
To create a new Stack Frame, simply change EBP value to be equal to ESP.



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

Now EBP = ESP, this means that the Newest Stack Frame is empty. The previous stack frame now is indexed as Stack Frame 1

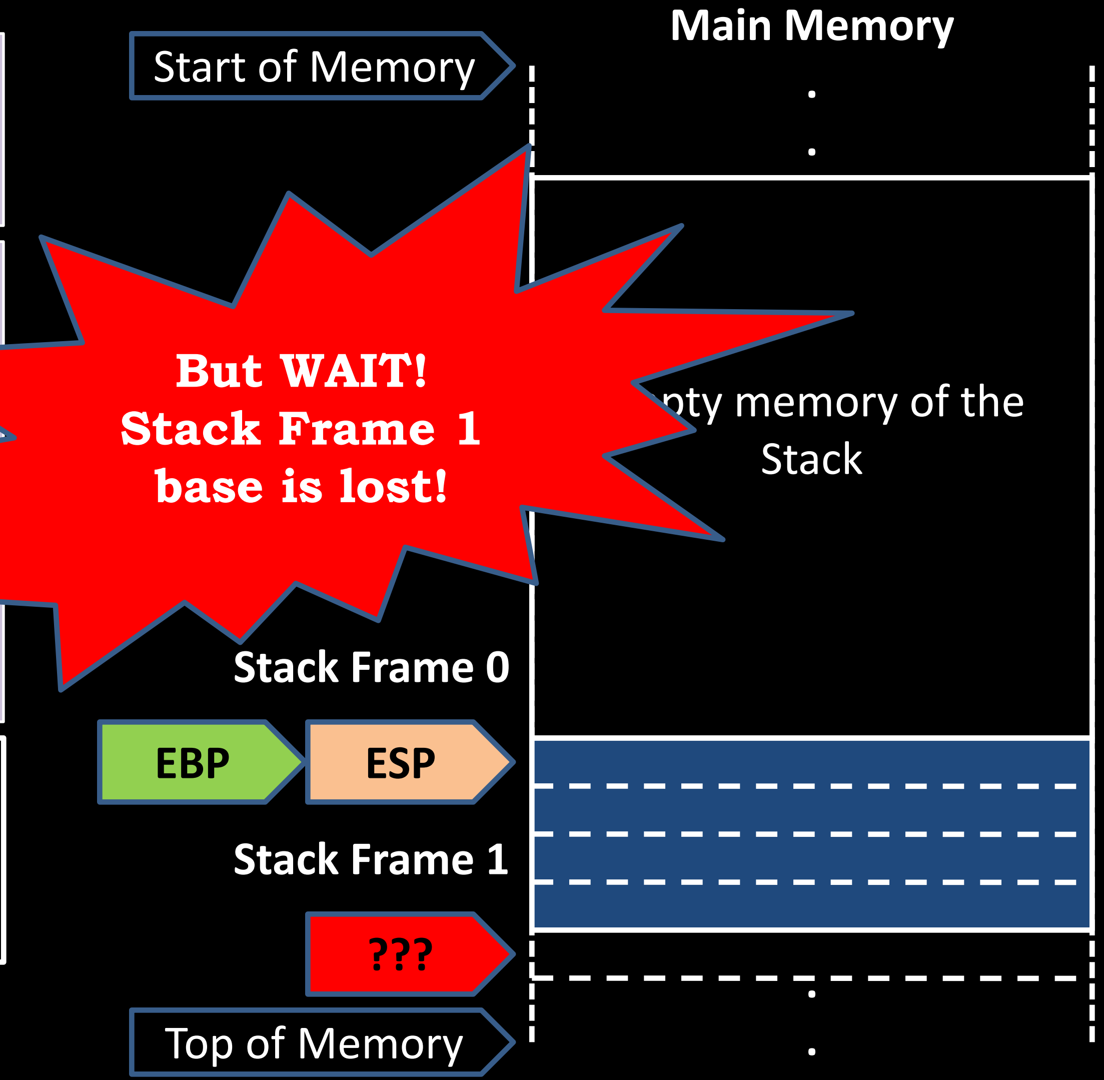


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

Now EBP = ESP, this means that the Newest Stack Frame is empty. The previous stack frame now is indexed as Stack Frame 1

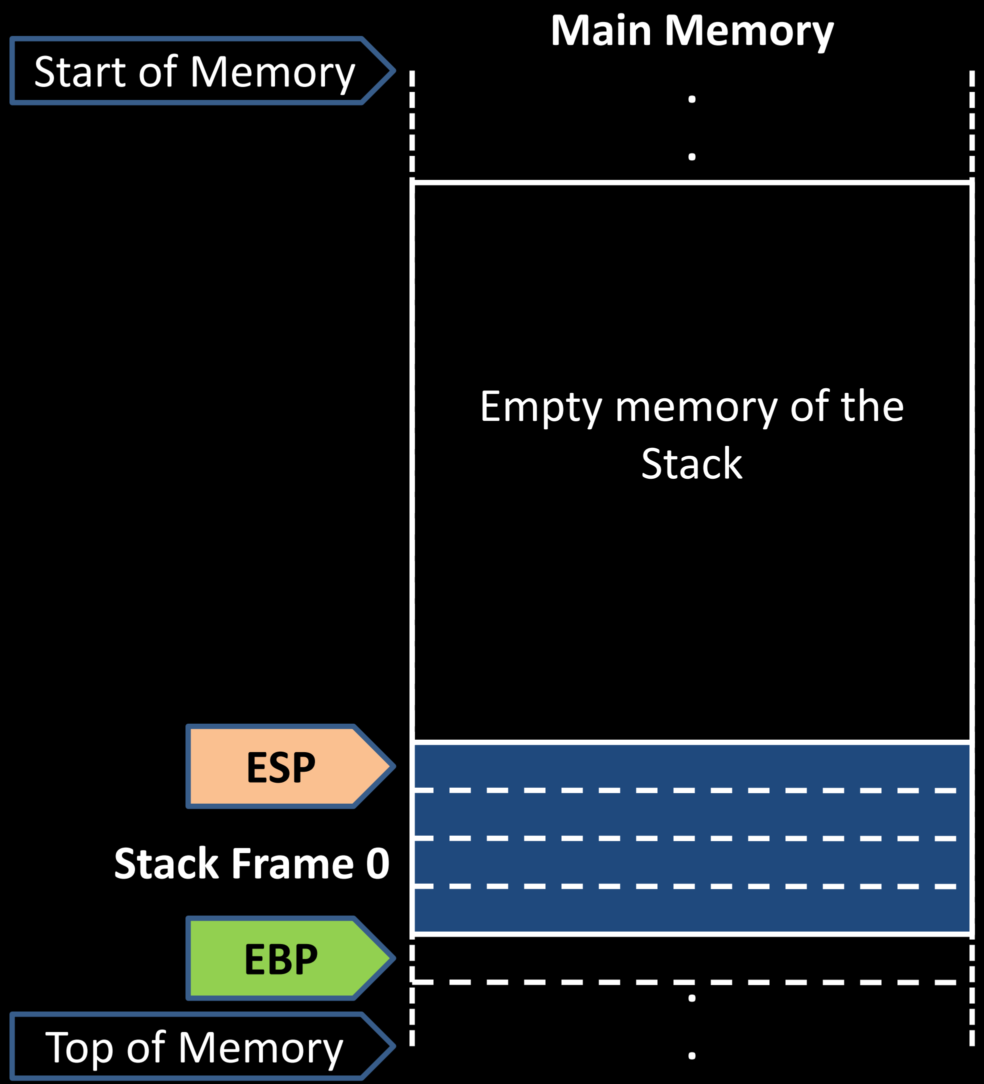
Let’s try again. This time we should save EBP value before changing it.



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

First, PUSH value of EBP to save it.

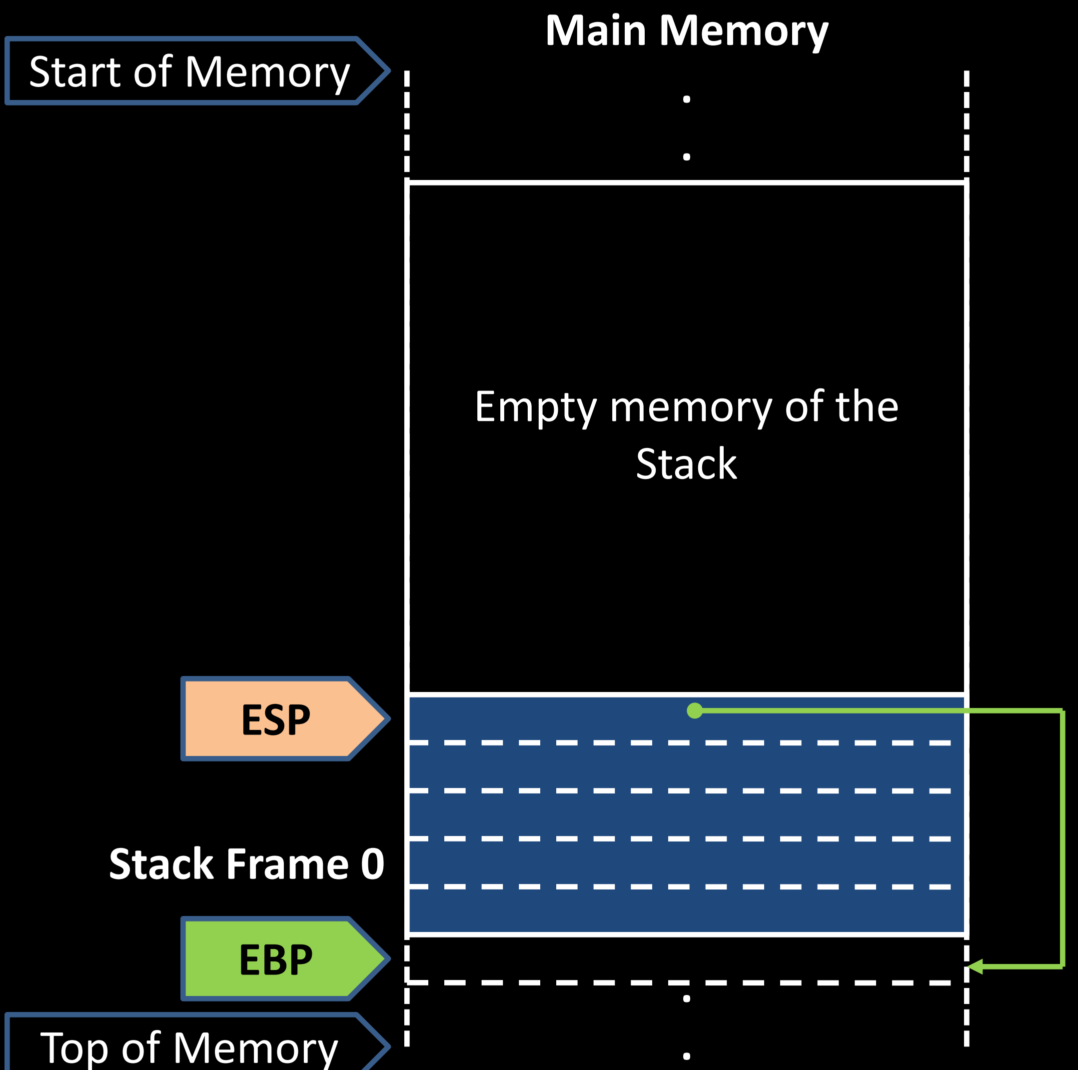


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

First, PUSH value of EBP to save it.

Now change the value of EBP.



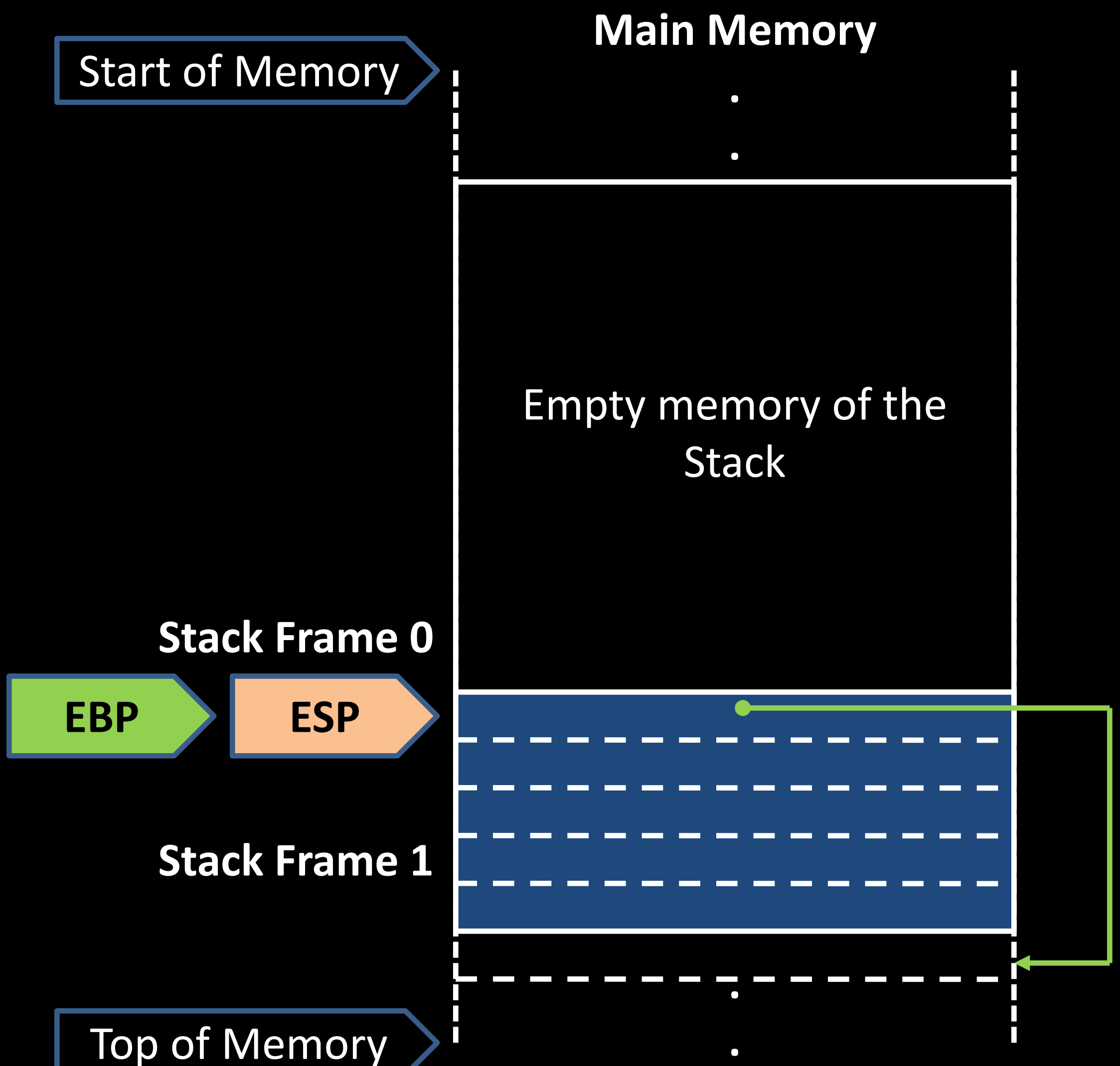
Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

First, PUSH value of EBP to save it.

Now change the value of EBP.

PROLOGUE is: Creating new Stack Frame then allocating space for local variables.

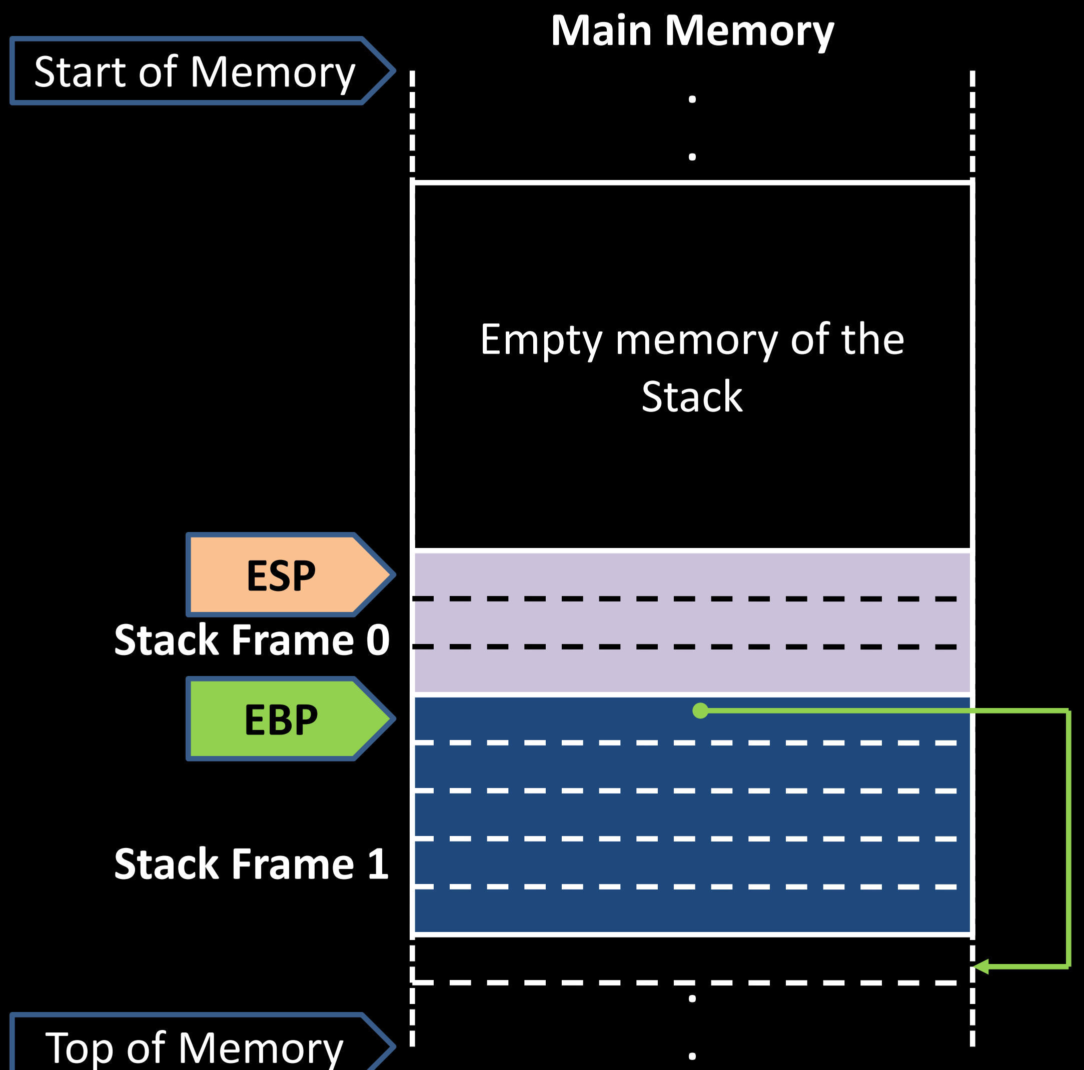


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

PUSH and POP operations affect ESP value only.

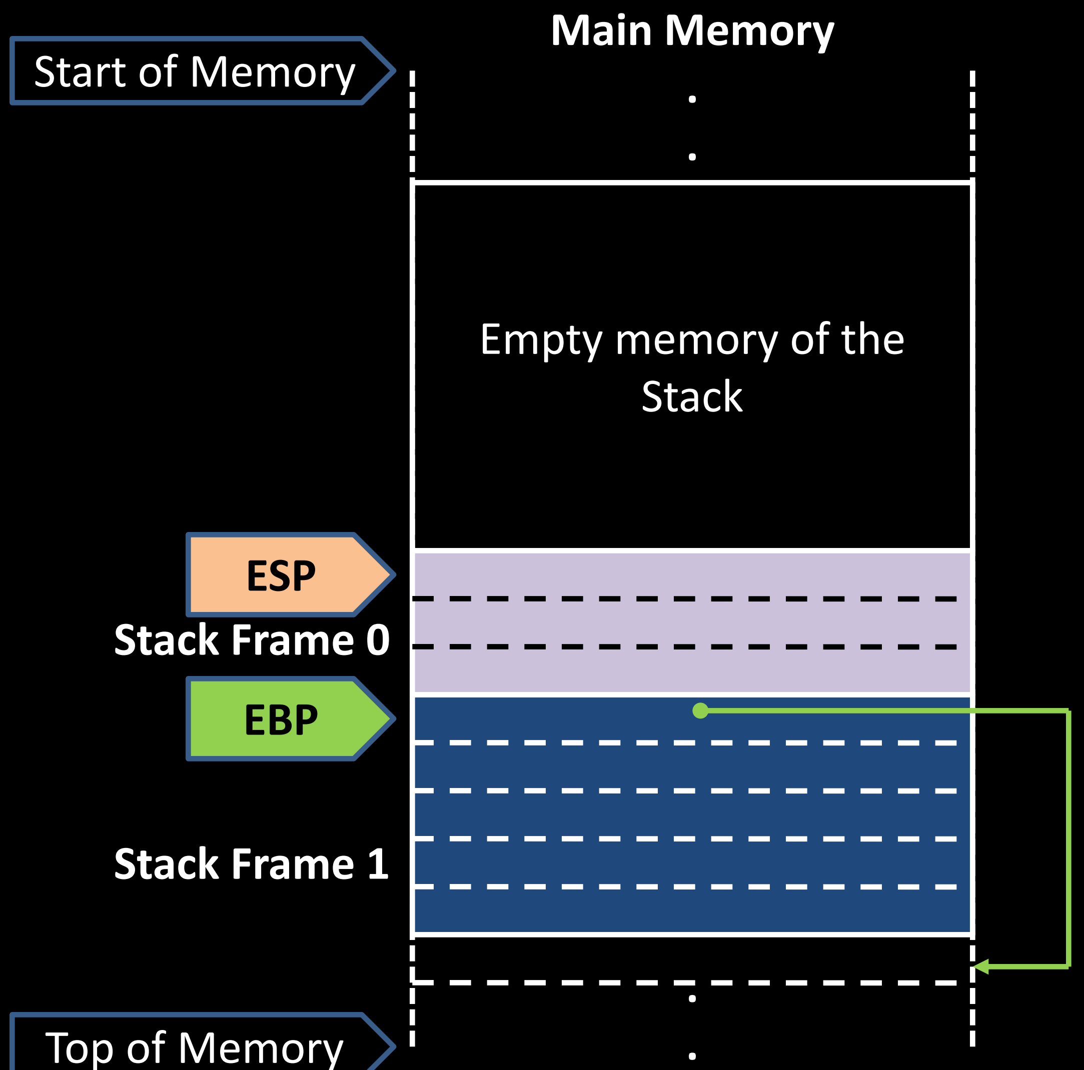
We don't need to save ESP value for the previous stack frame, because it is equal to the current EBP value



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

To empty out the current Stack Frame, ESP value should be set to the same value of EBP

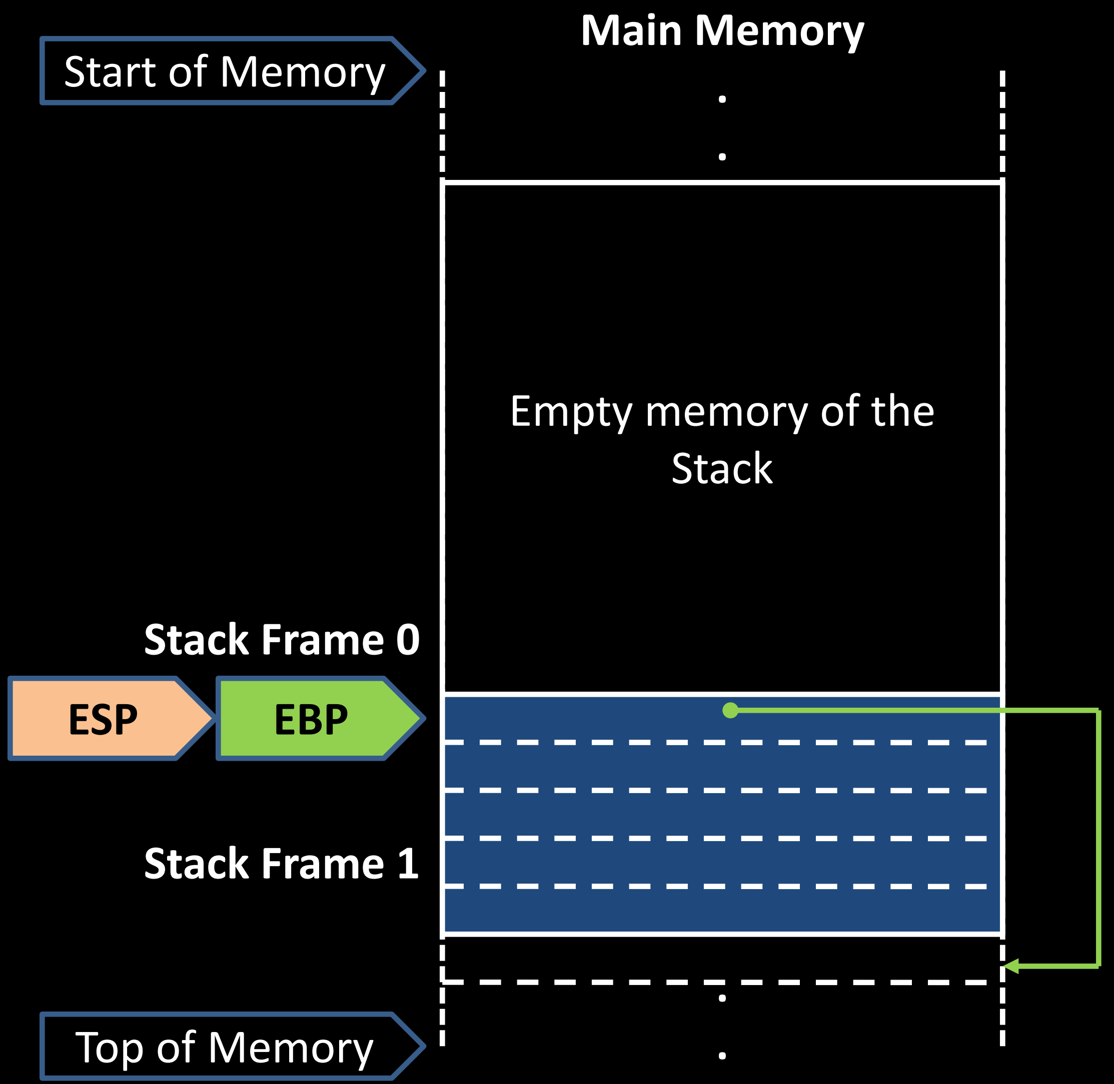


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

To empty out the current Stack Frame, ESP value should be set to the same value of EBP

To delete the current Stack Frame and return back to the previous one, we should POP out the top value from the **Stack** into EBP.



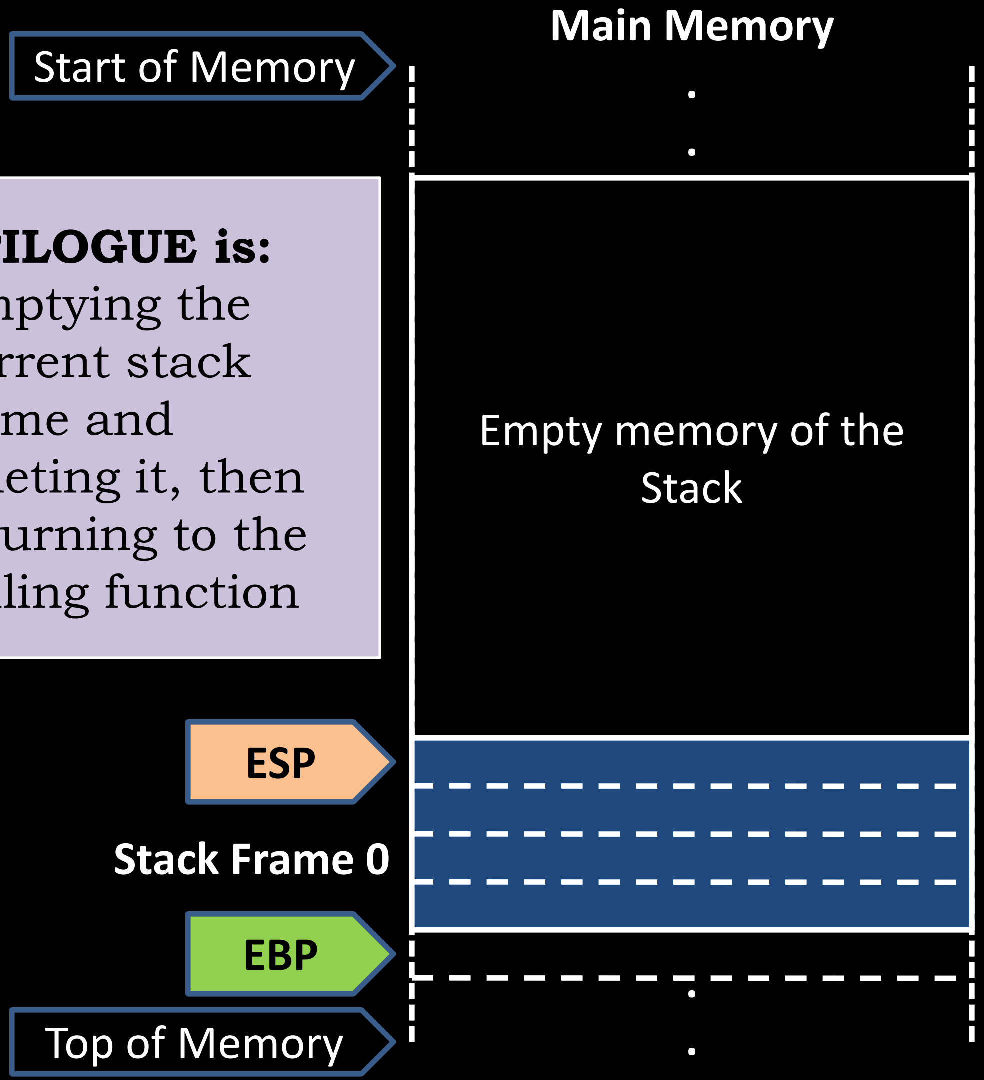
Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

To empty out the current Stack Frame, ESP value should be set to the same value of EBP

To delete the current Stack Frame and return back to the previous one, we should POP out the top value from the **Stack** into EBP.

EPILOGUE is: Emptying the current stack frame and deleting it, then returning to the calling function



Functions, Low Level View
- Understanding the Process -

A simple function call in a high level language is not a simple operation as it seems.

`add(x, y);`

PUSH arguments (if any)	PUSH arguments (if any)
Call the function	PUSH EIP Jump to function's first instruction
PROLOGUE	PUSH EBP Set EBP = ESP PUSH local variables (if any)
Execute the function	Execute the function
EPILOGUE	POP out all local variable POP EBP POP EIP
POP arguments	POP arguments

Functions, Low Level View
- Understanding the Process -

Each PUSH operation must be reversed by a POP operation somewhere in the execution

Performing (PUSH arguments) is done by the caller function. Arguments are pushed in a reverse order.

Performing (POP arguments) can be done by the caller or the callee function. This is specified by the (call type) of the callee function

Return value of the callee is saved inside EAX register while executing the function's body

- PUSH arguments (if any)
- PUSH EIP
- Jump to function's first instruction
- PUSH EBP
- Set EBP = ESP
- PUSH local variables (if any)
- Execute the function
- POP out all local variable
- POP EBP
- POP EIP
- POP arguments

Functions, Low Level View
- Call Types -

- Programming languages provide a mechanism to specify the call type of the function.
- (Call Type) is not (Return Value Type).
- The caller needs to know the call type of the callee to specify how arguments should be passed and how Stack Frames should be cleaned.
- There are many call types; two of them are commonly used in most programming languages:
  - `cdecl`: the default call type for C functions. The caller is responsible of cleaning the stack frame.
  - `stdcall`: the default call type for Win32 APIs. The callee is responsible of cleaning the stack frame.
- Some call types use deferent steps to process the function call. For example, `fastcall` send arguments within Registers not by the stack frame. (Why?)

Functions, Low Level View
- Assembly Language -

Each of these steps are processed by one or many instructions.

As like as other programming languages; assembly provides many ways to perform the same operation. Therefore, the disassembled code can vary from one compiler to another.

Now we are going to introduce the default way for performing each of these steps using assembly language.

PUSH arguments (if any)
PUSH EIP
Jump to function's first instruction
PUSH EBP
Set EBP = ESP
PUSH local variables (if any)
Execute the function
POP out all local variable
POP EBP
POP EIP
POP arguments

Functions, Low Level View
- Assembly Language -

	cdecl	stdcall	
caller	push <arg2> push <arg1>	push <arg2> push <arg1>	PUSH arguments (if any)
caller	call <callee>	call <callee>	PUSH EIP Jump to function's first instruction
callee	push ebp mov ebp, esp push <default value of local variable>	push ebp mov ebp, esp push <default value of local variable>	PUSH EBP Set EBP = ESP PUSH local variables (if any)
			Execute the function
callee	pop ecx pop ebp ret	pop ecx pop ebp ret <args size>	POP out all local variable POP EBP POP EIP
caller	pop ecx pop ecx		POP arguments

Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

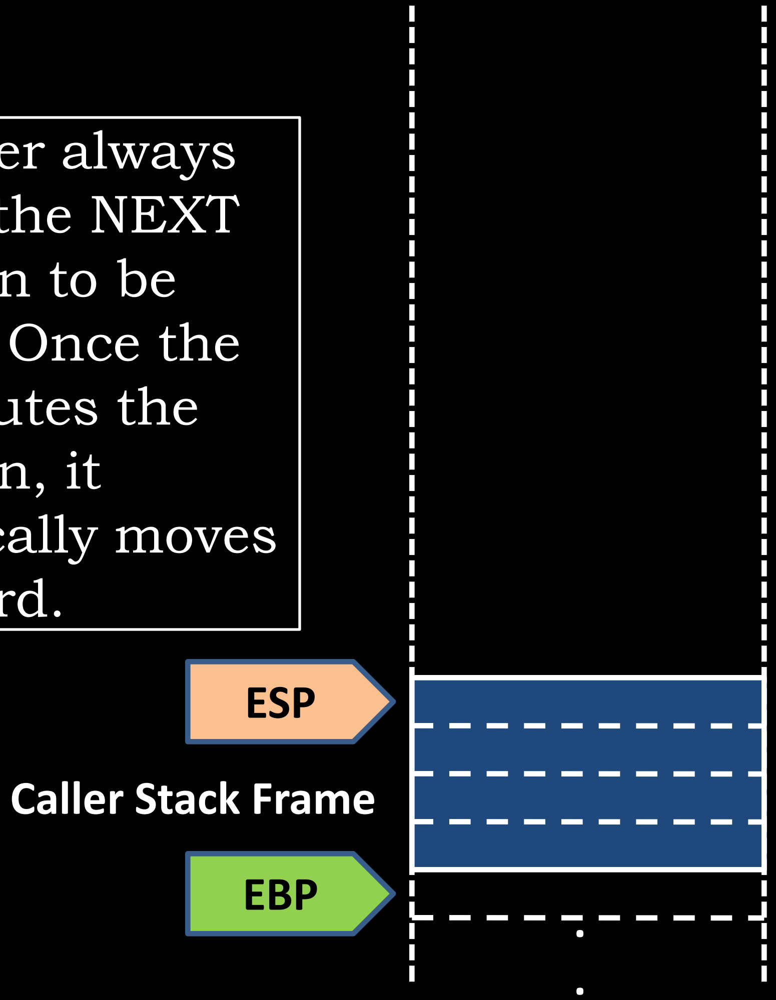
```
push    ebp
mov     ebp, esp
push   <default
        value of
        local
        variable>
```

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

EIP

EIP register always points to the NEXT instruction to be executed. Once the CPU executes the instruction, it automatically moves EIP forward.



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

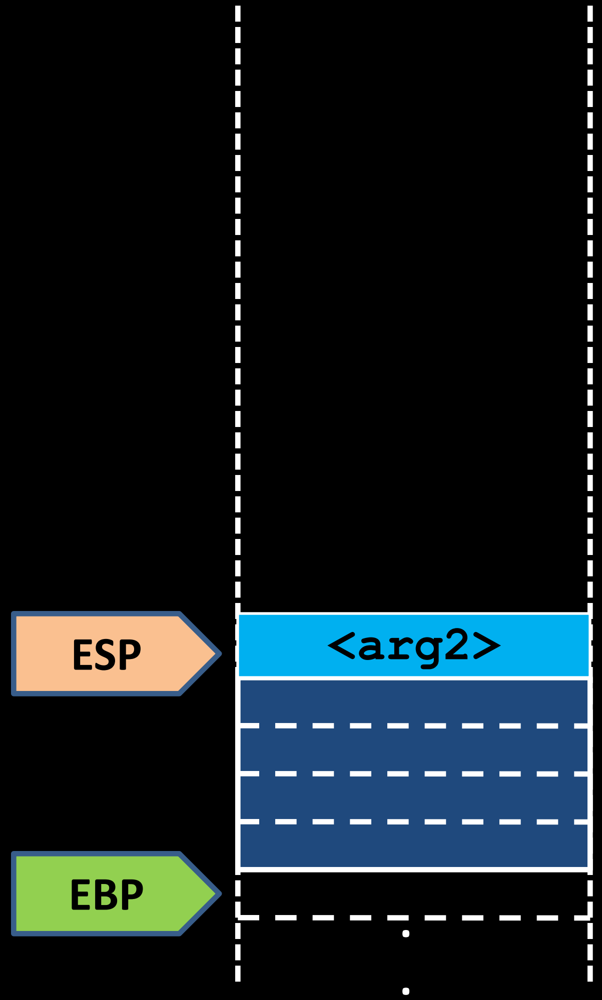
EIP

```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```



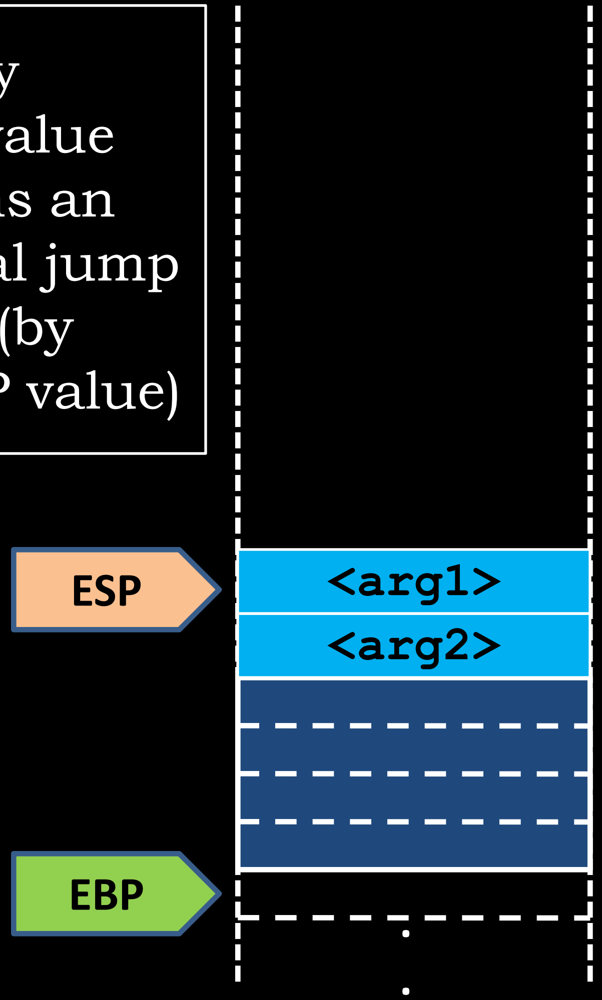
Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>

call    <callee>

push    ebp
mov     ebp, esp
push   <default
        value of
        local
        variable>

pop     ecx
pop     ebp
ret

pop     ecx
pop     ecx
```

EIP

(**call**) actually pushes EIP value then performs an unconditional jump to the callee (by changing EIP value)



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

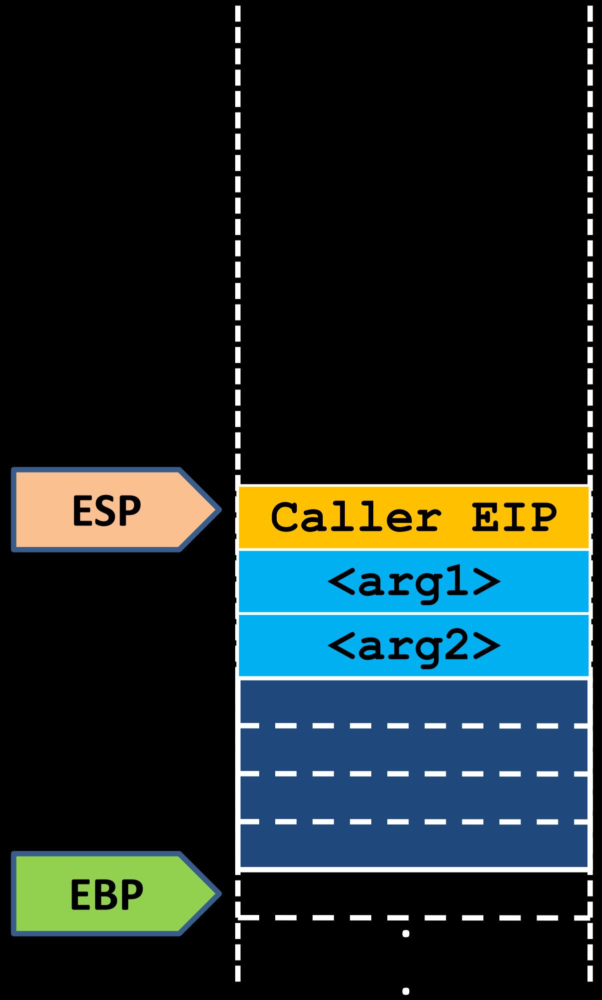
```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push    <default
        value of
        local
        variable>
```

EIP

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

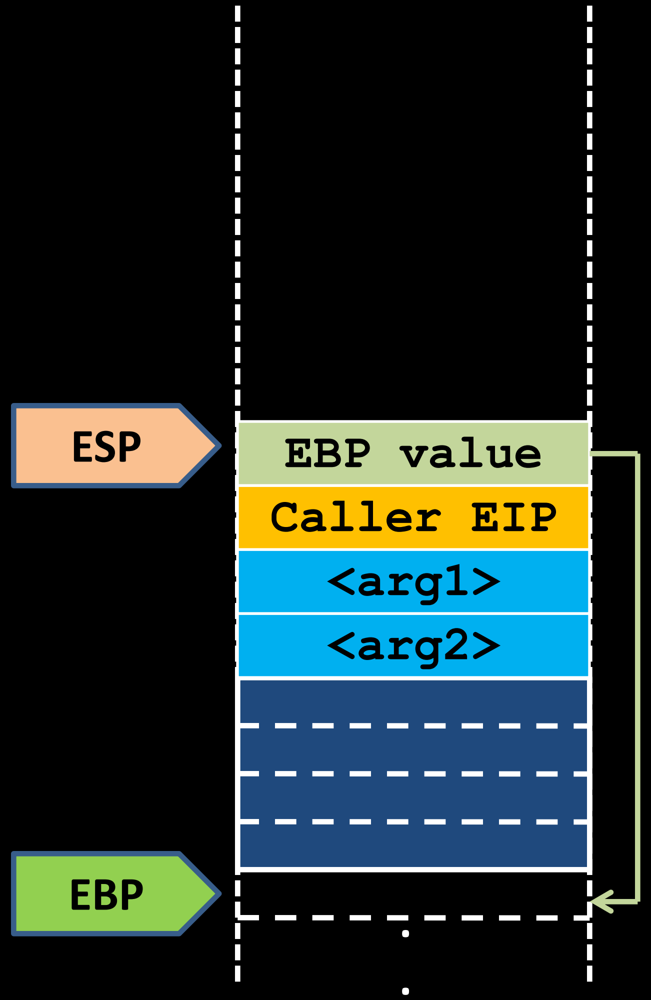


Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>

call    <callee>

push    ebp
mov     ebp, esp
push   <default
        value of
        local
        variable>

pop     ecx
pop     ebp
ret

pop     ecx
pop     ecx
```

EIP



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

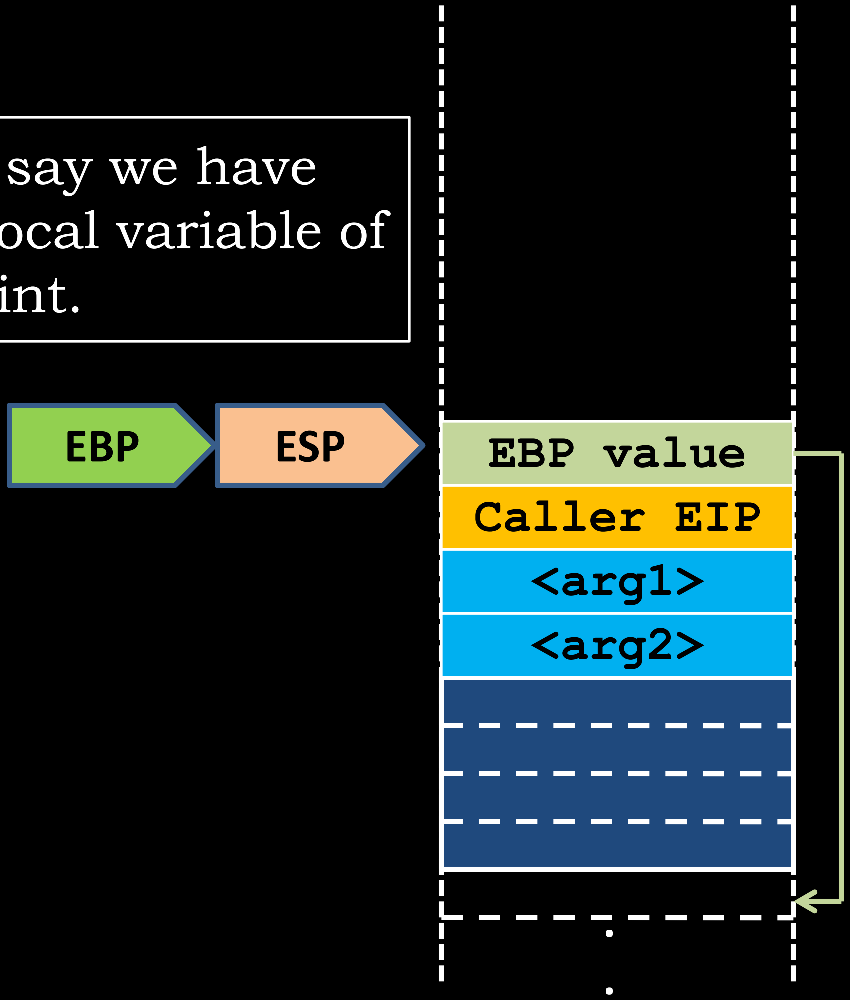
```
push    ebp
mov     ebp, esp
push   <default
        value of
        local
        variable>
```

EIP

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

Let’s say we have one local variable of type int.



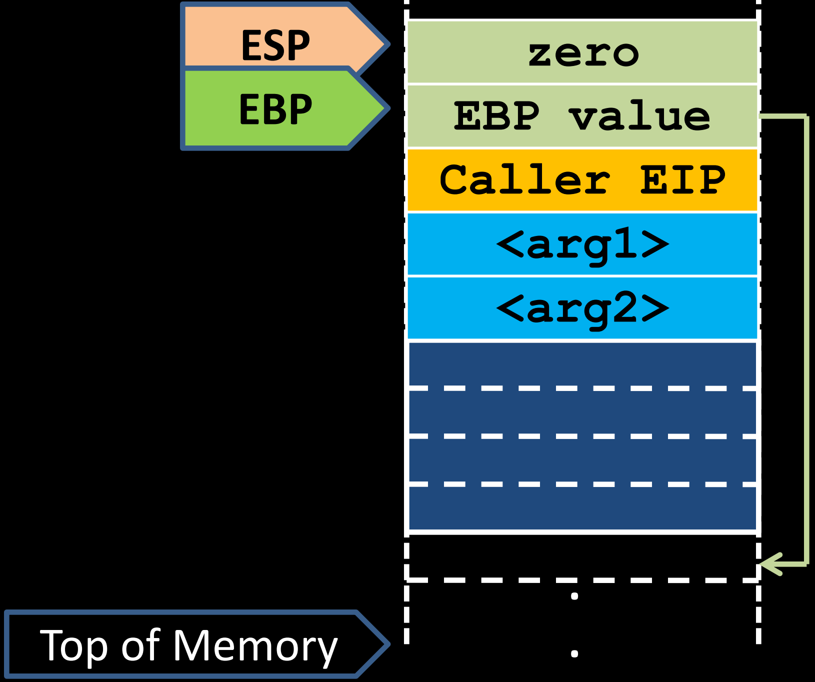
Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>

call    <callee>

push    ebp
mov     ebp, esp
push   <default
        value of
        local
        variable>

pop     ecx
pop     ebp
ret

pop     ecx
pop     ecx
```

ESP may change inside the callee body, but EBP does not change. Therefore, EBP location is used to locate variable and arguments.

Start of Memory



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>
```

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

Start of Memory

ESP can change in the callee body, but EBP does not change. Therefore, EBP location is used to locate variable and arguments.

Remember that each row of this stack graph is 32bits (4 bytes)

- EBP - 4 → zero
- EBP → EBP value
- Caller EIP
- EBP + 8 → <arg1>
- EBP + 12 → <arg2>

Top of Memory

Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

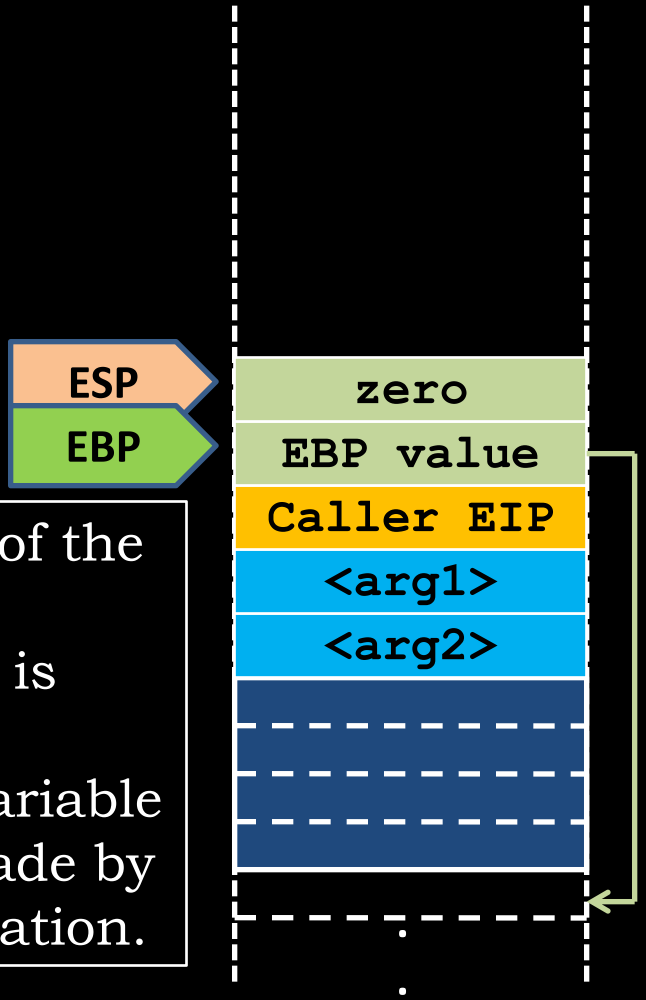
```
push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>
```

```
pop     ecx
pop     ebp
ret
```

EIP

```
pop     ecx
pop     ecx
```

At the end of the callee, EPILOGUE is processed. Cleaning variable space is made by a POP operation.



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

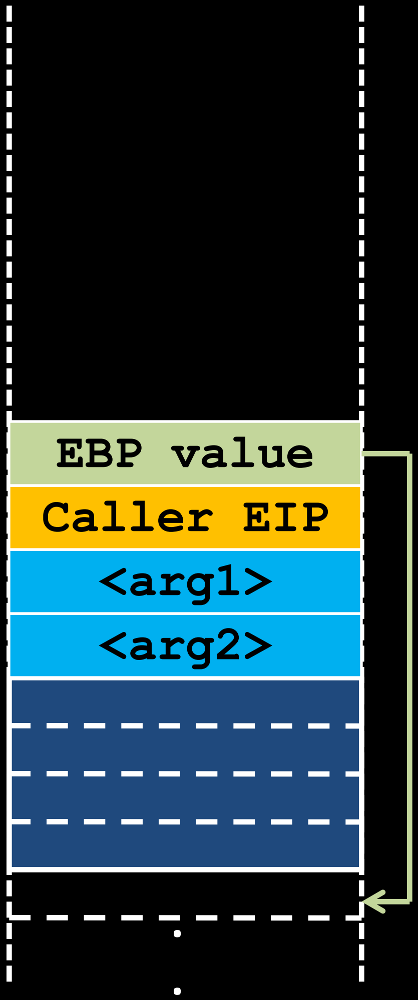
```
push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>
```

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

EIP

ESP EBP



Now caller base EBP should be retrieved

Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push   <default
       value of
       local
       variable>
```

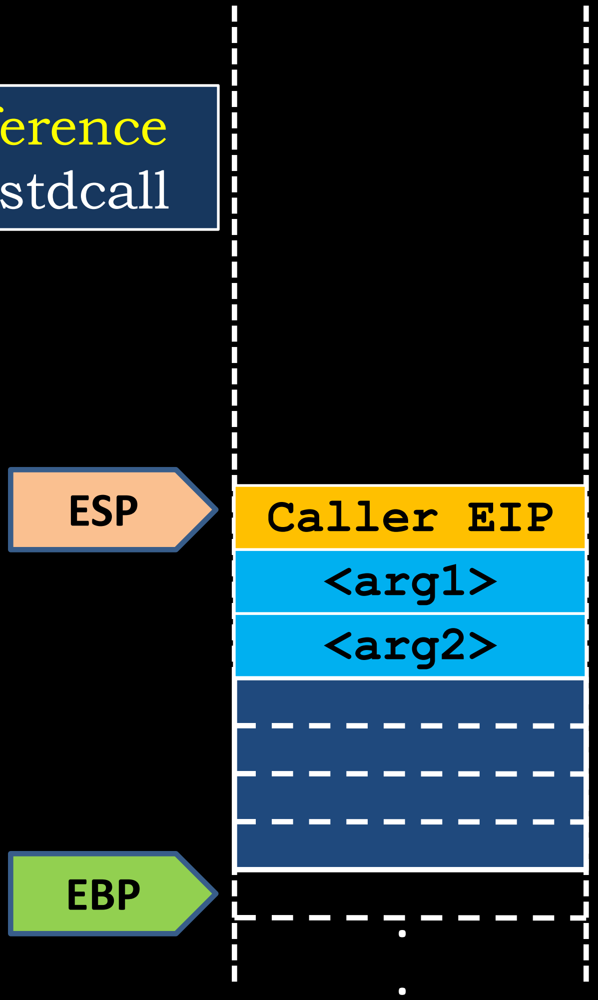
```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

EIP

Here comes the deference between cdecl and stdcall

ret instruction simply pops a value from the stack and save it in EIP, that should direct the execution back to the caller



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push   <default
        value of
        local
        variable>
```

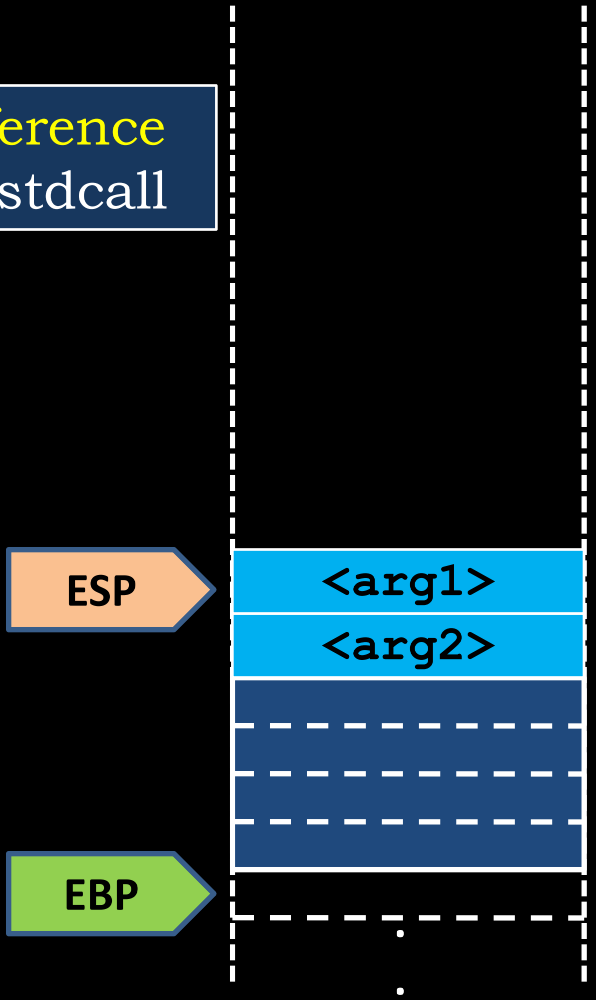
```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

EIP

Here comes the deference between cdecl and stdcall

Now the caller is responsible of cleaning the stack from passed arguments using POP operations.



Functions, Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>
```

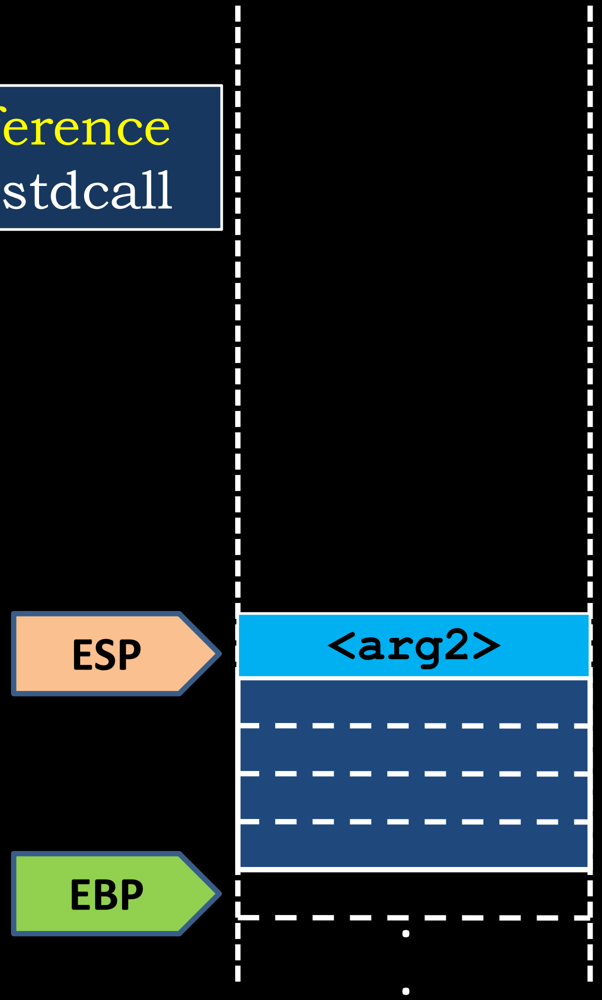
```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

EIP

Here comes the deference between cdecl and stdcall

Now the caller is responsible of cleaning the stack from passed arguments using POP operations.



Functions, Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

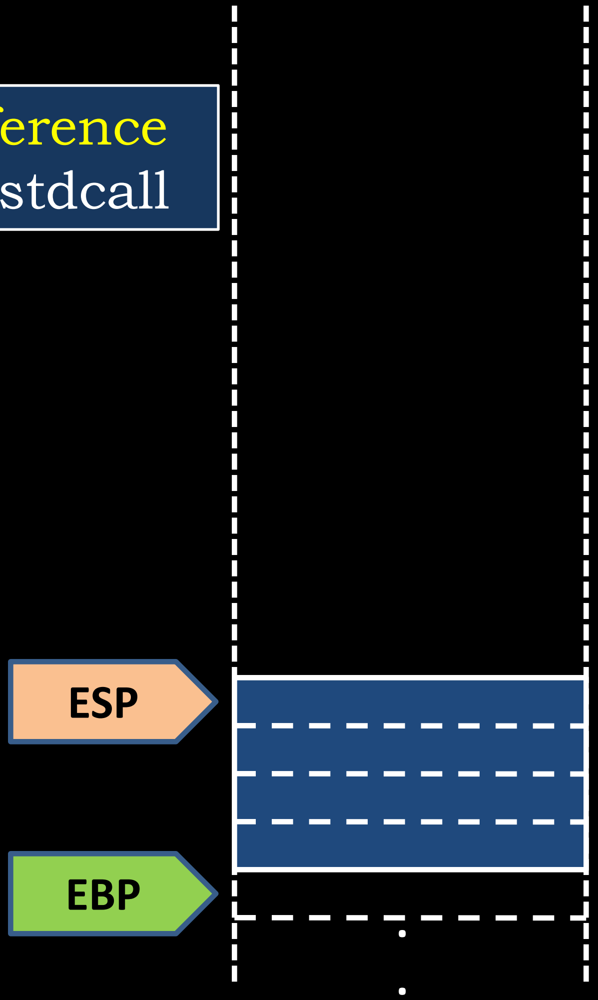
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

Here comes the deference between cdecl and stdcall



Functions, Low Level View
- General Trace -

stdcall

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

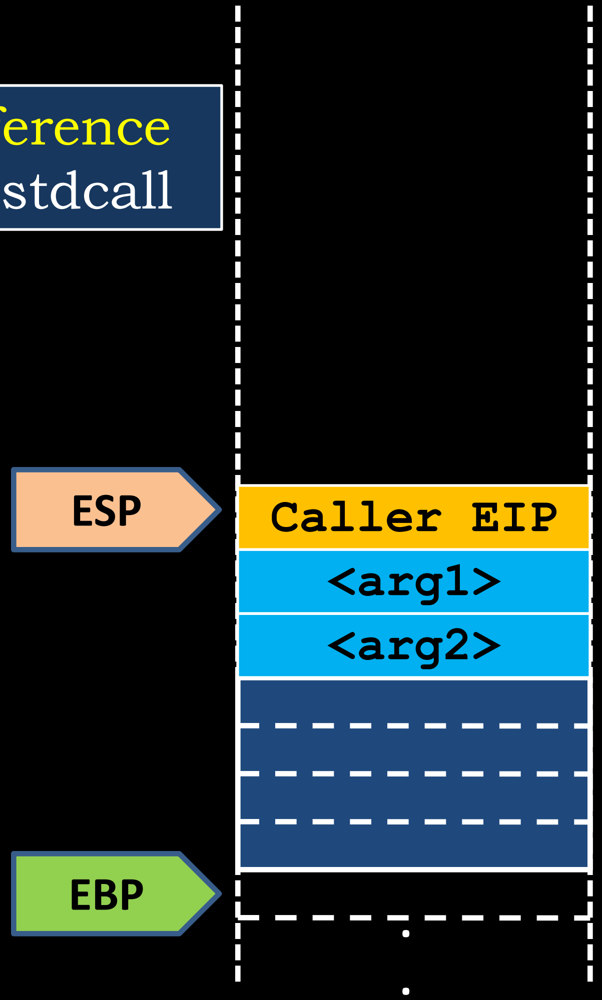
```
push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>
```

```
pop     ecx
pop     ebp
ret     <args size>
```

EIP

Here comes the deference between cdecl and stdcall

ret instruction proceeded by an integer value will add that value to ESP <u>immediately after</u> performing POP EIP



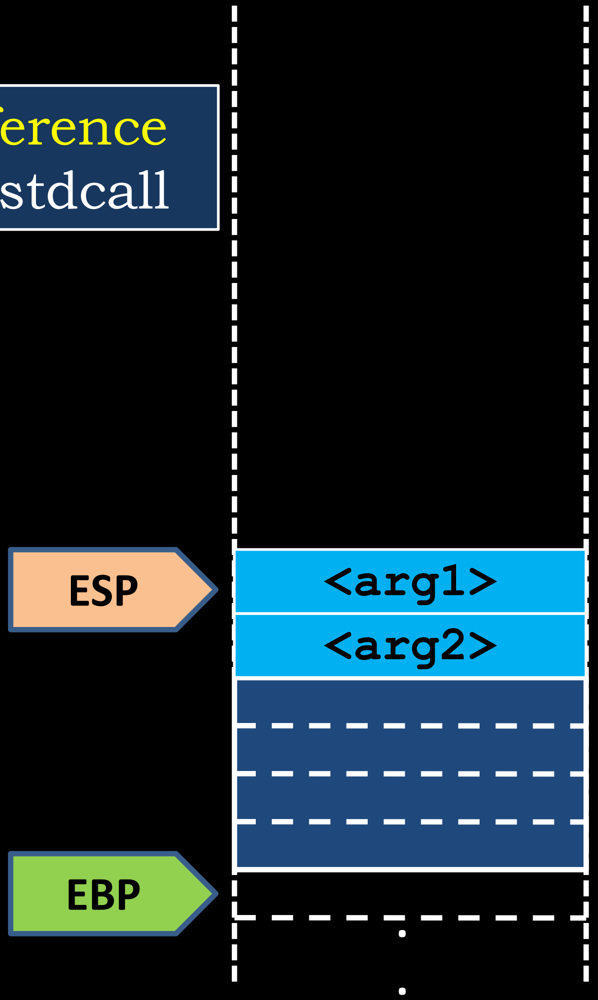
Functions, Low Level View
- General Trace -

stdcall

```
push    <arg2>
push    <arg1>

call    <callee>

push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>

pop     ecx
pop     ebp
ret     <args size>
```

Here comes the deference between cdecl and stdcall

Now EIP is changed, but the CPU did not finish executing the instruction. It will add <args size> value to ESP.
In this example, we have two 32bits arguments (8 bytes)



Functions, Low Level View
- General Trace -

stdcall

```
push   <arg2>
push   <arg1>
```

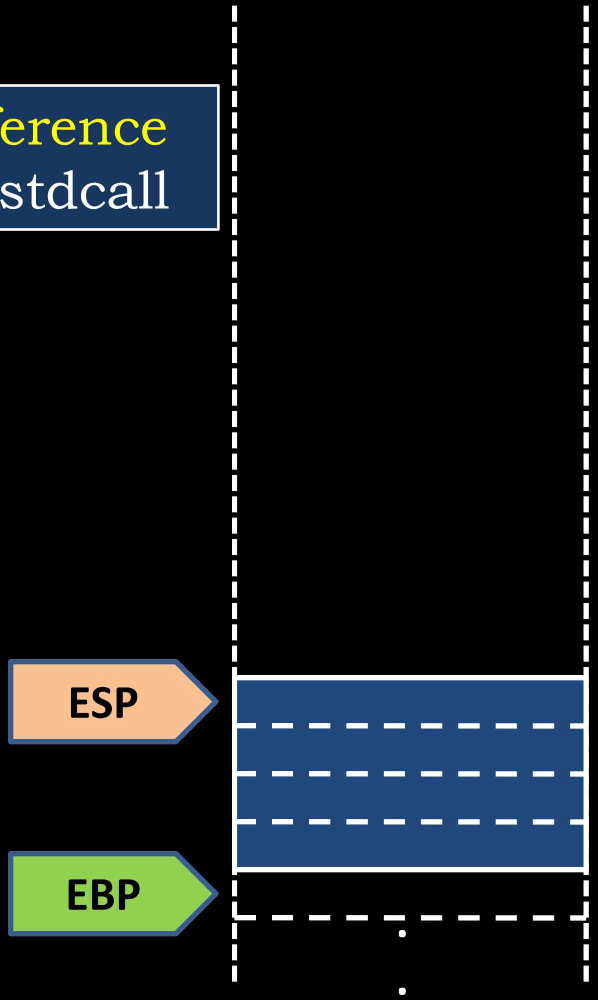
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

```
pop    ecx
pop    ebp
ret    <args size>
```

Here comes the deference between cdecl and stdcall

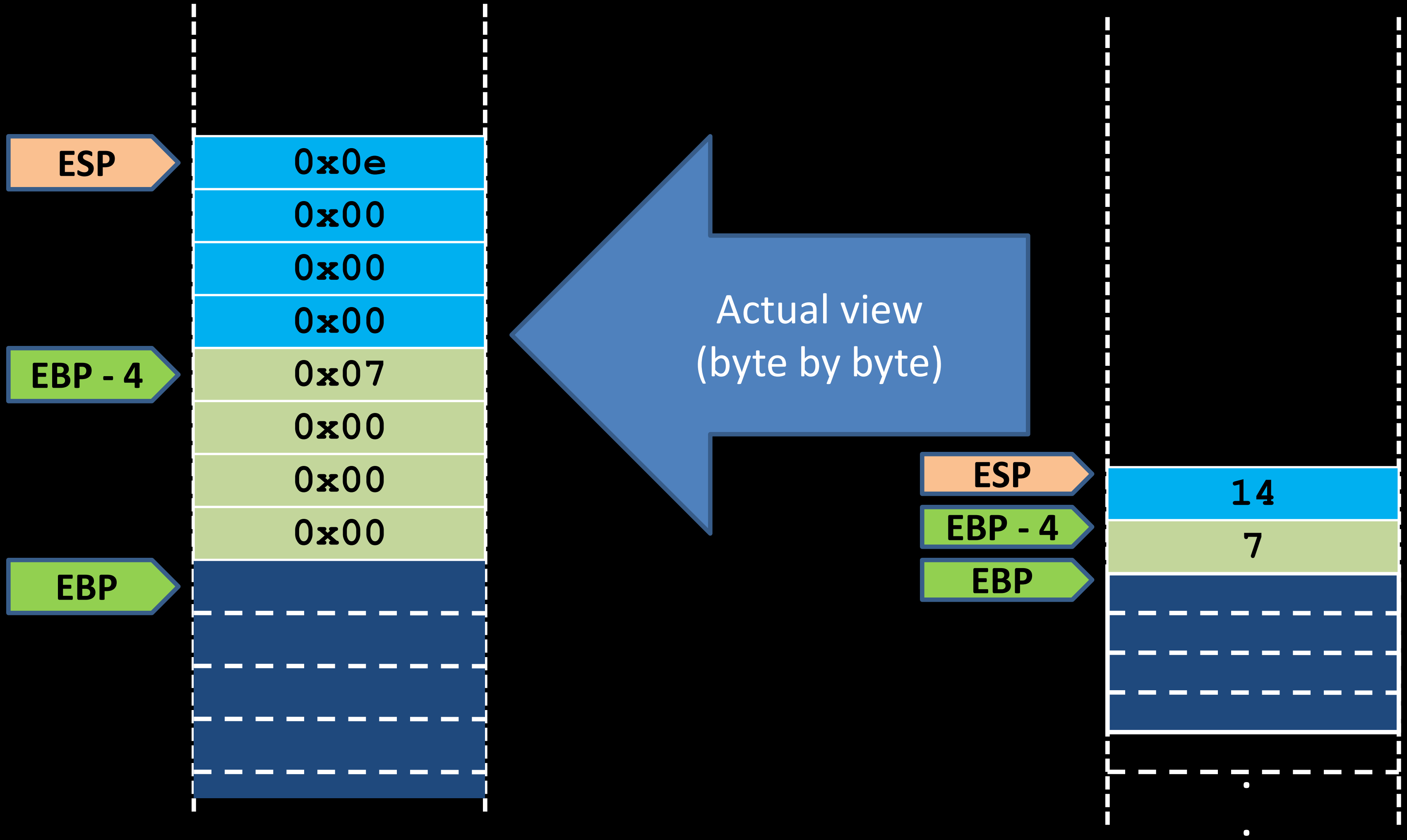
The stack has been cleaned by the callee. Now execution is back to the caller.



Functions, Low Level View
- Code Optimization -

- Compilers do not generate the default code like previous example. They use intelligent methods to optimize the code to be smaller and faster.
- For example, instructions `mov` and `xor` can be used to set EAX register to zero, but xor is smaller as a code byte. Therefore, compilers use `xor` instead of `mov` for such scenarios:
  - mov eax, 0 → code bytes: `B8 00 00 00 00`
  - xor eax, eax → code bytes: `3C 00`
- Discussing code optimization is out of the scope of this course, but we are going to discuss few tricks that you will see in the code generated by GCC for our examples.

Functions, Low Level View
- Hint about Endianness -



Functions, Low Level View
- Hint about Endianness -

Start of Memory

	little-endian	big-endian	
ESP	0x0e	0x00	ESP
	0x00	0x00	
	0x00	0x00	
	0x00	0x0e	
EBP - 4	0x07	0x00	EBP - 4
	0x00	0x00	
	0x00	0x00	
	0x00	0x07	
EBP			EBP

In little-endian architect (like intel processors); multi-byte values are filled starting from the least significant byte. In big-endian (like SPARC processors) they are filled in a reverse order (starting from most significant byte).

Top of Memory

Functions, Low Level View
- Example from GCC -

```
void myfun1(char *str) {
push    ebp
mov     ebp,esp
char buffer[16];
sub     esp, 0x18
strcpy(buffer, str);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80482c4 <strcpy@plt>
myfun2(buffer);
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80483b4 <myfun2>
}
leave
ret
```

The function myfun1 require 16 bytes for the local array.

strcpy require 8 bytes for it's arguments

myfun2 require 4 bytes for it's arguments

The compiler made a reservation for 24 bytes (0x18) which is 16 for array + 8 for **maximum** arguments space

Functions, Low Level View
- Example from GCC -

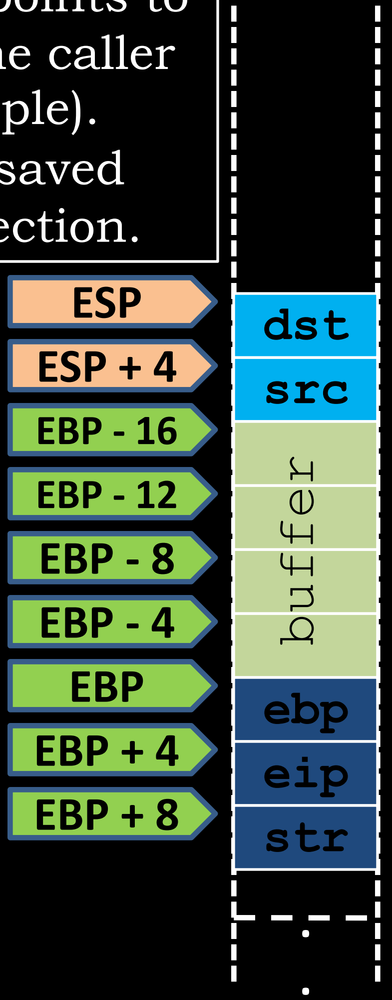
```
void myfun1(char *str) {
push    ebp
mov     ebp,esp
char buffer[16];
sub     esp,0x18
strcpy(buffer, str);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80482c4 <strcpy@plt>
myfun2(buffer);
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80483b4 <myfun2>
}
leave
ret
```

EIP

By default, EBP+4 points to the saved EIP of the caller (`main` in this example). EBP points to the saved EBP by epilogue section.

strcpy takes two arguments, destination dst then source src.

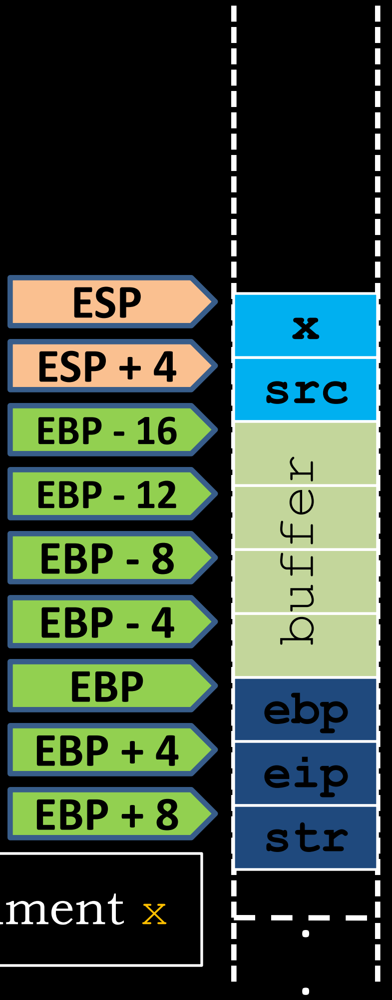
EBP+8 is the sent value by the caller `main` to the callee `myfun1` that is named `str` in this code.



Functions, Low Level View
- Example from GCC -

```
void myfun1(char *str) {
push    ebp
mov     ebp,esp
char buffer[16];
sub     esp,0x18
strcpy(buffer, str);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80482c4 <strcpy@plt>
myfun2(buffer);
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80483b4 <myfun2>
}
leave
ret
```

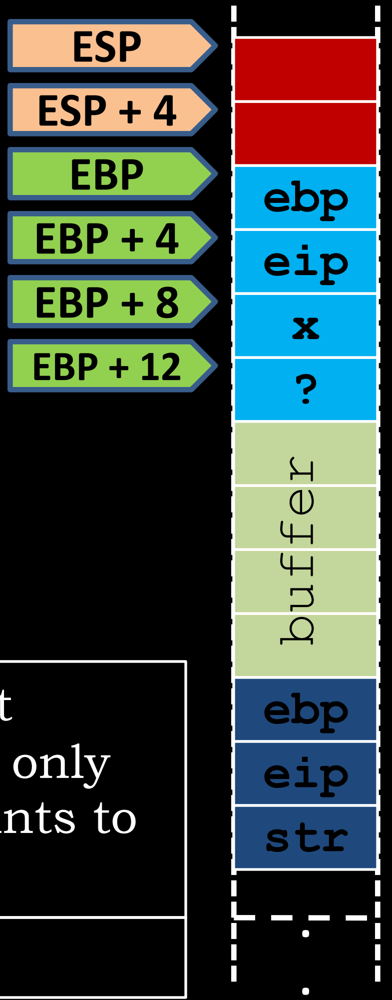
EIP



myfun2 takes one argument x

Functions, Low Level View
- Example from GCC -

```
void myfun2(char *x) {
push    ebp
mov     ebp,esp
sub     esp,0x8
printf(" You entered: %s\n", x);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
mov     DWORD PTR [esp],0x8048520
call    0x80482d4 <printf@plt>   EIP
}
leave
ret
```



EPB+8 points to the first argument sent to the current function. EBP+12 points is the second and so on. But only one argument used by `myfun2`. Therefore, EBP+12 points to an irrelevant location as `myfun2` can see.

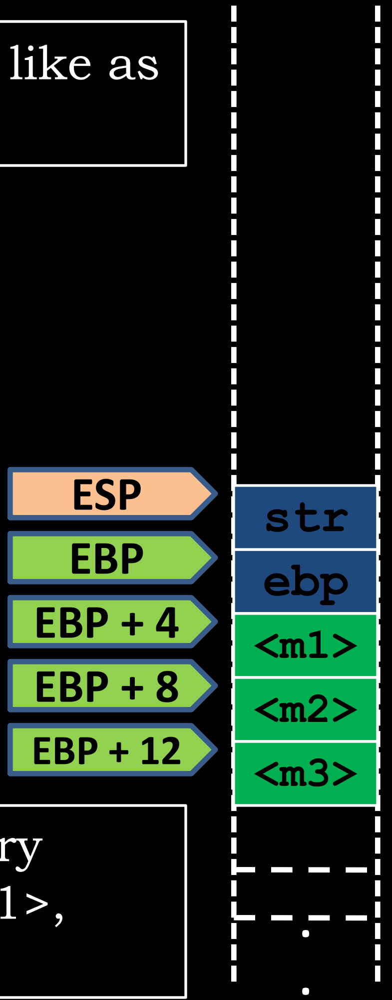
Can you guess what is currently saved in [EBP+12] ?

Functions, Low Level View
- Example from GCC -

```
int main(int argc, char *argv[]){
push    ebp
mov     ebp,esp
sub     esp,0x4
if (argc > 1)
cmp     DWORD PTR [ebp+8],0x1
jle     0x8048412
myfun1(argv[1]);
mov     eax,DWORD PTR [ebp+12]
add     eax,0x4
mov     eax,DWORD PTR [eax]
mov     DWORD PTR [esp],eax
call    0x80483cf <myfun1>   EIP
jmp     0x804841e
else printf("No arguments!\n");
mov     DWORD PTR [esp],0x8048540
call    0x80482d4 <printf@plt>
}
leave
ret
```

main is a function as like as any other function.

Can you tell what these instructions do?



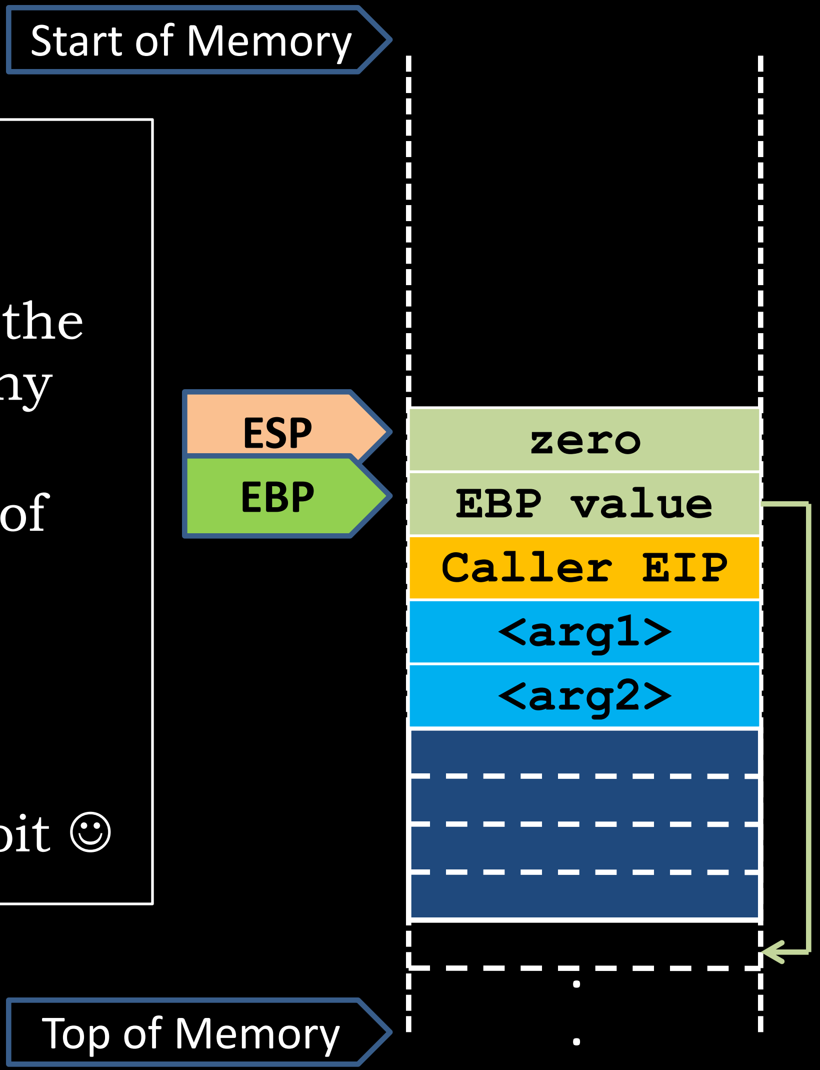
What do these memory locations contain <m1>, <m2>, and <m3>?

Functions, Low Level View
- Stack Reliability -

So,

What if we can locate Caller EIP in the stack and change it using `mov` or any other instruction?
What if the new value is a location of another block of code?
What if the other block of code is harmful (security wise)?

Bad for the user, good for the Exploit ☺



Memory Corruption

Memory Corruption

- Memory corruption is when a programming error causes a program to access memory in an invalid way, resulting in undefined behavior
  - Overwriting memory reserved for a different variable
  - Overwriting memory reserved for programming language runtime control structures
  - Access uninitialized or freed memory
- Memory corruption may allow an attacker to inject his *shellcode* to do malicious activity
- In other words ***Arbitrary code execution***”

Memory Corruption Classes

- Buffer overflows (Stack, Heap, Data segment, etc)
- Format string injection
- Out-of-bounds array accesses
- Integer overflows (can lead to buffer overflows or out-of-bounds array access)
- Uninitialized memory use
- Dangling/stale pointers (i.e. use-after-free)

Memory Corruption Exploits

- Usually the goal is to inject a machine code payload (“shellcode”) and get the target program to run it
  - Usually we just want it to give us a remote or higher-privileged shell (/bin/sh or cmd.exe)
  - Not all exploits will use a payload that runs a shell
- Not all memory corruption exploits execute shellcode

History of Memory Corruption

- "Multics Security Evaluation: Vulnerability nalysis" (1974)
- Morris Worm, 1988
- "Vulnerability in NCSA HTTPD 1.3", Thomas Lopatic, 1995
- "Smashing the Stack for Fun and Profit", Aleph One, 1996
- "Getting around non-executable stack (and fix)", Solar Designer, 1997
- "JPEG COM Marker Processing Vulnerability", Solar Designer, 2000

Vulnerability Analysis

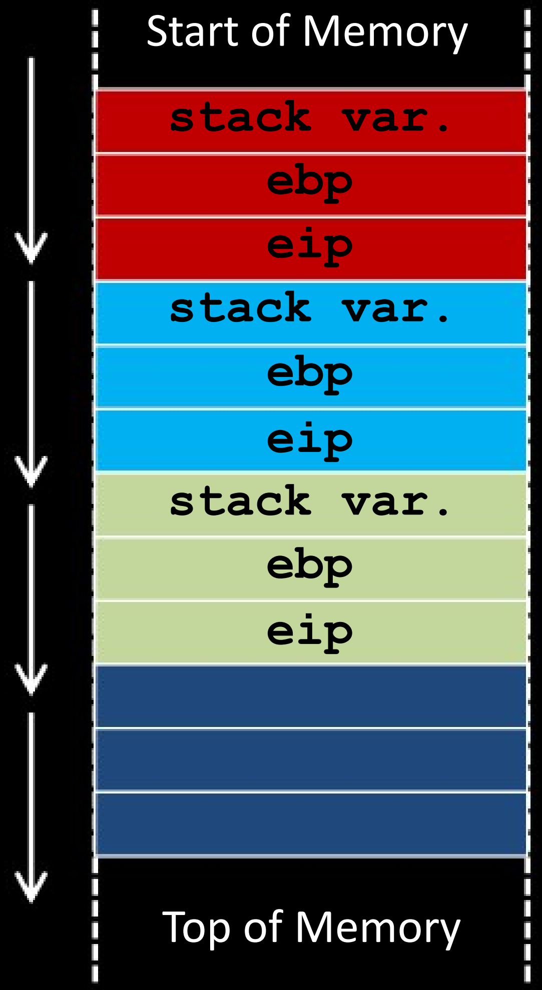
- A program crashes, is it repeatable and reproducible?
- Memory is corrupted, is it controllable?
- Memory corruption can be controlled, is it exploitable?
- Some tools are available to help
  - !exploitable (WinDbg)
  - Crash Wrangler (Mac OS X)
  - Mona.py (ImmunityDebugger)

Stack Buffer Overflows

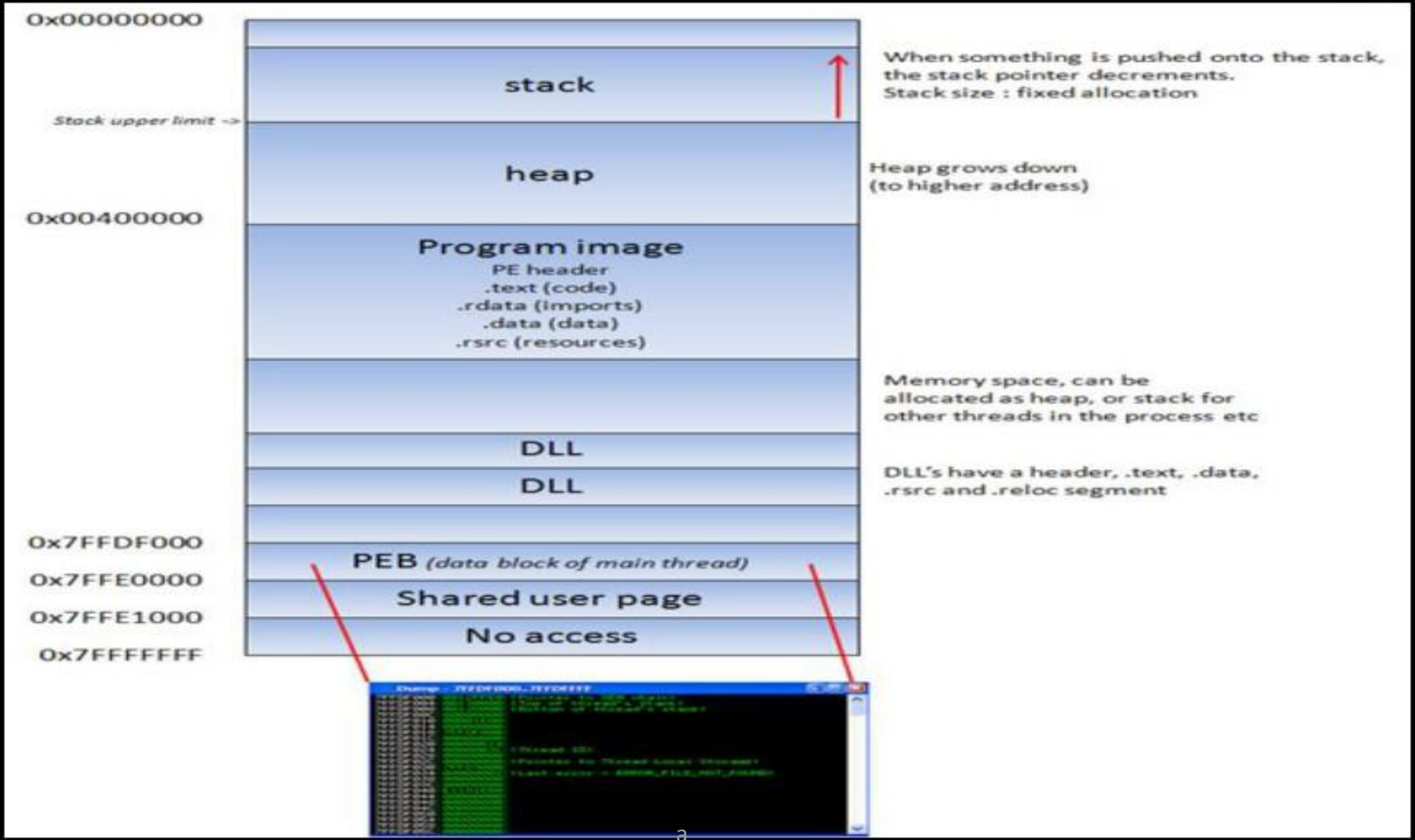
- The canonical, simplest type of memory corruption to understand and exploit
- First publicly used by Robert Morris worm in 1988
  - Used a stack buffer overflow in VAX BSD in.fingerd
- Are *still* exploitable on many systems today !!!
- Many operating systems and compilers include defenses against these now (more on this later)

Stack BoF - Cont.

- Stack variable overflows, overwriting the return address
- The attacker writes a memory address in the stack for the return address
- The subroutine returns into payload on stack



Win32 Process Memory Map

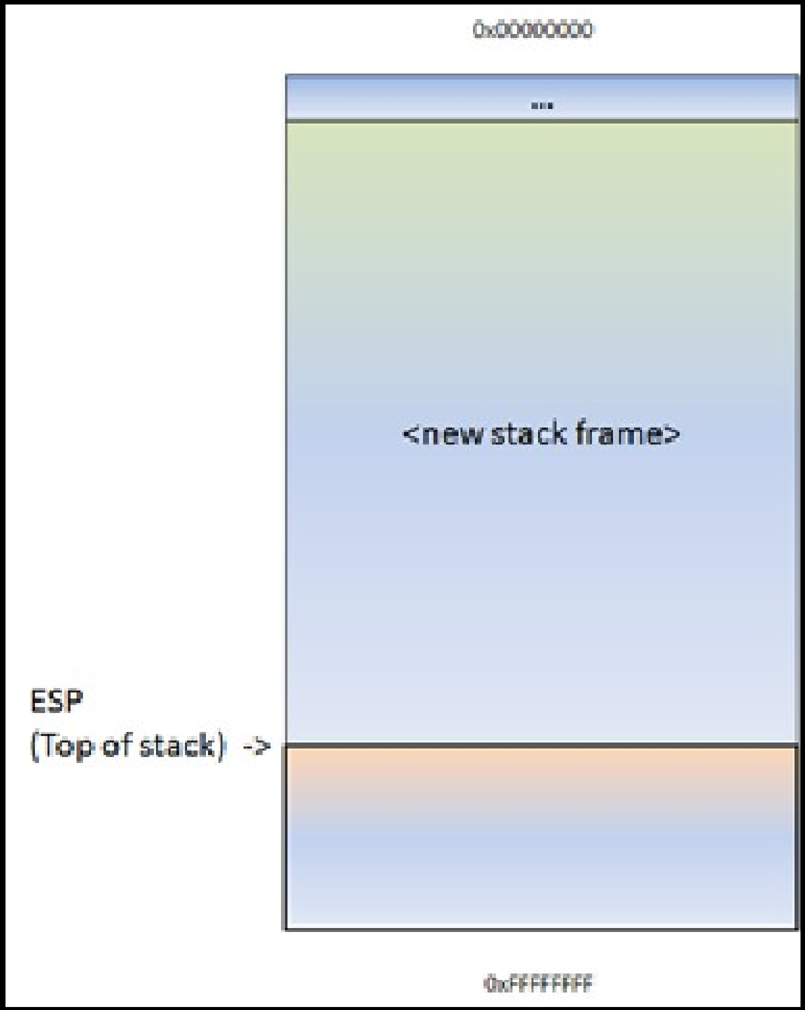


Simple Code Demonstration

```
#include <string.h>
void do_something(char *Buffer)
{
    char MyVar[128];
    strcpy(MyVar,Buffer);
}

int main (int argc, char **argv)
{
    do_something(argv[1]);
}
```

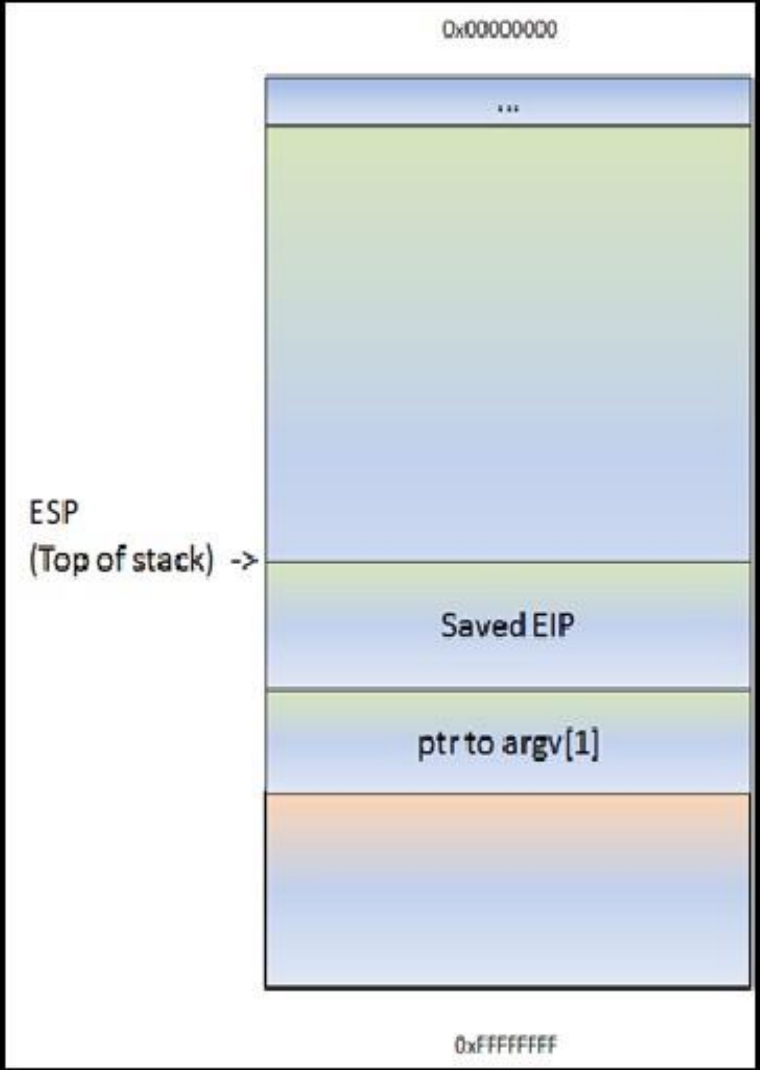
Simple Code Demonstration - Cont.

- When function "do_something(param1)" gets called from inside main(), the following happen :
  - A new stack frame will be created, on top of the 'parent' stack.
  - The stack pointer (ESP) points to the highest address of the newly created stack.
  - This is the "top of the stack".



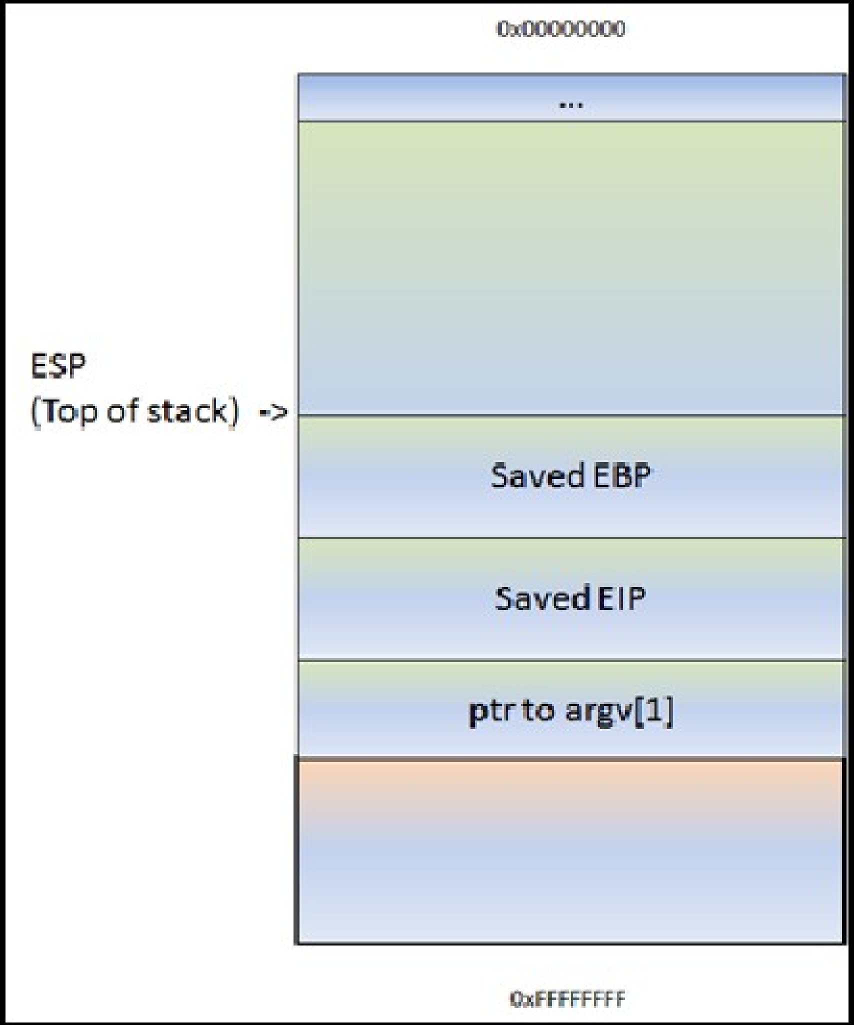
Demonstration - Cont.

- Before do_something() is called, a pointer to the argument(s) gets pushed to the stack (this is a pointer to argv[1]).
- Next, function do_something is called.
- The CALL instruction will first put the current instruction pointer onto the stack (so it knows where to return to if the function ends) and will then jump to the function code.



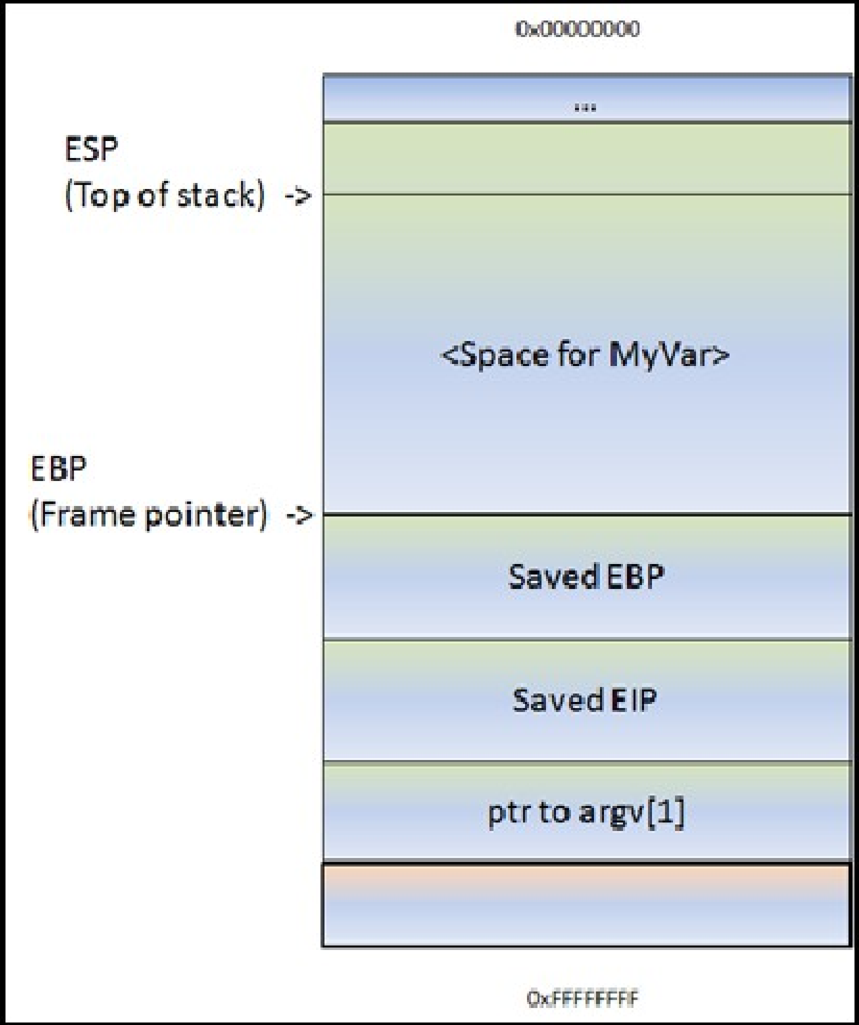
Demonstration - Cont.

- Next, the functio *prolog* executes.
- This basically saves the frame pointer (EBP) onto the stack, so it can be restored as well when the function returns.
- The instruction to save the frame pointer is "*push ebp*".
- ESP is decremented again with 4 bytes.



Demonstration - Cont.

- Next, we can see how stack space for the variabl MyVa (128bytes) is declared/allocated.
- In order to hold the data, some space is allocated on the stack to hold data in this variable ESP is decremented by a number of bytes.
- This number of bytes wil most likely be more than 128 bytes, because of an allocation routine determined by the compiler.
  - In the case of Dev-C++, this is 0×98 bytes.
  - So you will see *SUB ESP, 98* instruction.
- That way, there will be space available for this variable.



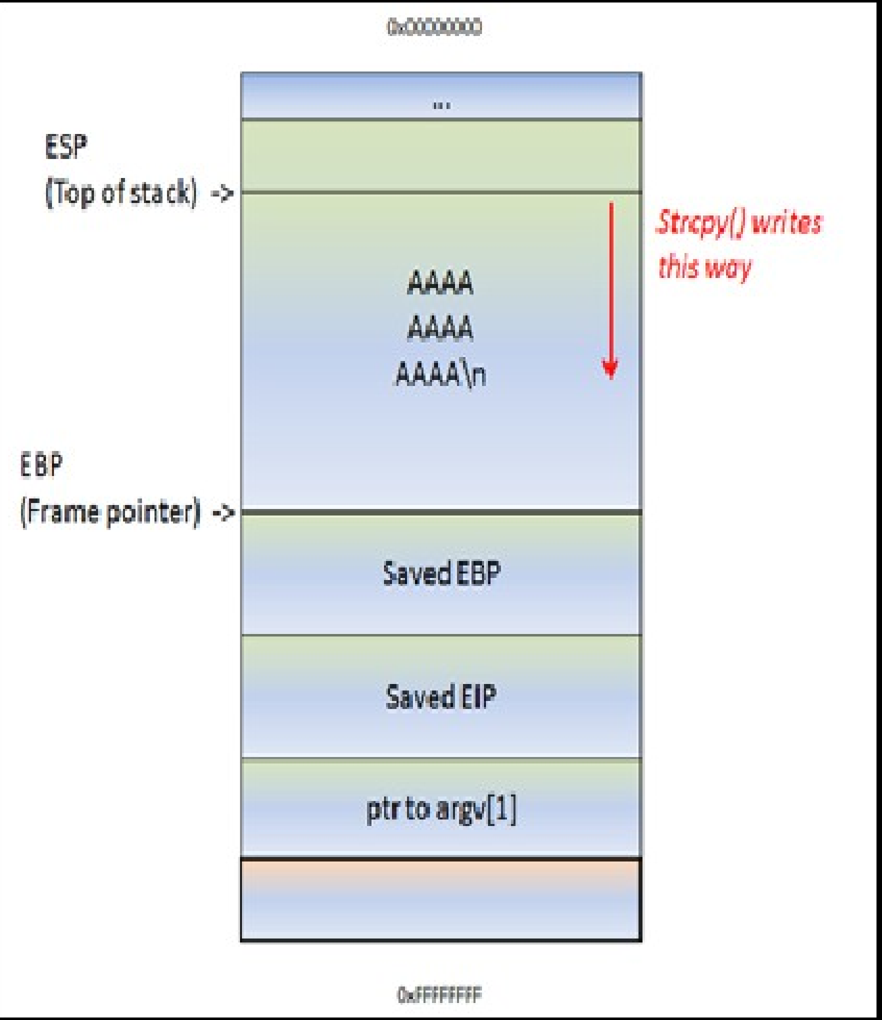
Demonstration - Cont.

- The disassembly of the function looks like this :

Memory	Opcode	Instruction
00401290	55	PUSH EBP
00401291	89E5	MOV EBP,ESP
00401293	81EC 98000000	SUB ESP,98
00401299	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
0040129C	894424 04	MOV DWORD PTR SS:[ESP+4],EAX
004012A0	8D85 78FFFFFF	LEA EAX,DWORD PTR SS:[EBP-88]
004012A6	890424	MOV DWORD PTR SS:[ESP],EAX
004012A9	E8 72050000	CALL ; \strcpy
004012AE	C9	LEAVE
004012AF	C3	RETN

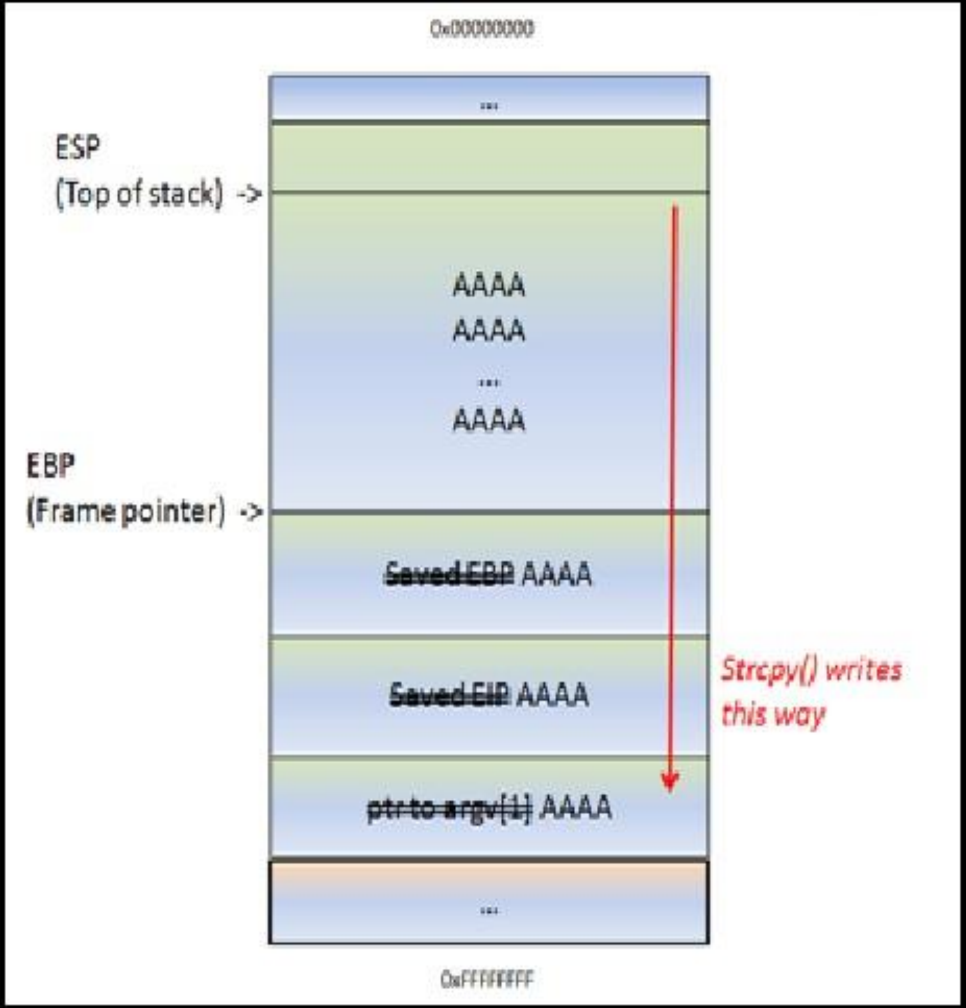
Demonstration - Cont.

- The strcpy() function will read data, from the address pointed to by [Buffer], and store it in <space for MyVar>, reading all data until it sees a null byte (string terminator).
- While it copies the data, ESP stays where it is.
- The strcpy() does not use PUSH instructions to put data on the stack it basically reads a byte and writes it to the stack, using an index (for example ESP, ESP+1, ESP+2, etc).
- So after the copy, *ESP still points at the begin of the string*.



Demonstration - Cont.

- That means If the data in [Buffer] is somewhat longer than 0×98 bytes, the strcpy() will overwrite saved EBP and eventually saved EIP (and so on).
- After all, it just continues to read & write until it reaches a null byte in the source location



Demonstration - Cont.

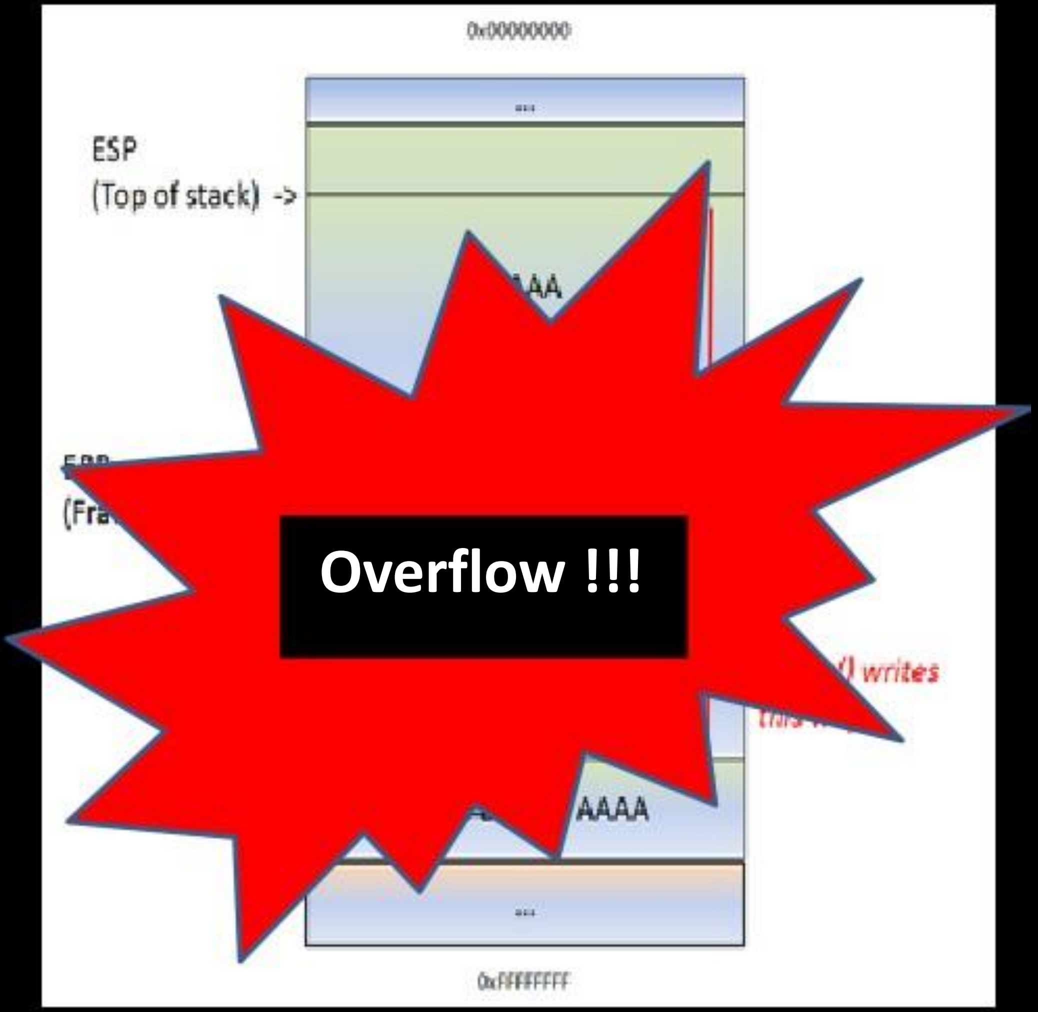
- *ESP* still points at the begin of the string.
- The strcpy() completes as if nothing is wrong.
- After the strcpy(), the function ends, and this is where things get interesting. The functio *epilog* kicks in.
- Basically, *it will move ESP back to the location where saved EIP was stored, and it will issue a RET*.
- It will take the pointer (AAAA or 0×41414141 in our case, since it got overwritten), and will jump to that address.

Demonstration - Cont.

- So yo *control EIP* !!!
- By controlling EIP, you basically change the return address that the function will uses in order to “resume normal flow”.
- If you change this return address by issuing a buffer overflow, it’s not a “normal flow” anymore!
- Suppose you can overwrite the buffer in MyVar, EBP, EIP and you have’s (your own code) in the area before and after saved EIP think about it.
- After sending the buffer (**[MyVar][EBP][EIP][your code]**), ESP will/should point at the beginning of [your code].

Demonstration - Cont.

If you can make EIP go to your code? Then you're in control



Exploit By Numbers

1. Trigger the vulnerability
2. Identify usable characters for attack string
3. Identify offsets and significiant elements in attack string
4. Fill in jump addresses, readable/writable addresses, etc
5. Identify amount of usable space for the payload
6. Drop in payload

Identify Usable Characters

- The attack string is the part of the input that triggers the vulnerability and contains values for overwritten memory (and possibly the payload also)
- Certain characters in the attack string may cause the application to parse the input differently and not trigger the vulnerability (“bad bytes”)
  - NULL bytes (any ASCII string)
  - Whitespace (\t\n\r)

Identify Offsets

- Use a pattern string to identify offsets into your attack string of data placed into registers or written to memory
- We are going to use Metasploit' **pattern_create.rb**
  - Can write your own too
- \# pattern_create.rb 32

  Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab

- \# pattern_offset.rb 0x41366141

  18

Fill in Memory Addresses

- For an exploit to function, certain parts of the attack string may need to readable, writable, or executable memory addresses
  - In particular, we want to overwrite the return address with the memory address of executable code
  - This memory address will redirect execution into our attack string
  - Spend quality time in your target's address

Identify Usable Space

- We need to know how much room we have for our payload
- We will size it out by placing increasingly large numbers of NOPs followed by a debug interrupt (int 3)
- If the target generates a breakpoint exception, we have that much usable space
- If the target crashes in another way, we may need to shrink the payload space

Drop in Payload

- The payload must also not use any bad bytes or else it may get truncated and not execute properly
- For simple payloads and vulnerabilities, avoiding NULL bytes in the instruction encodings may be enough
- For more complex payloads and vulnerabilities, a payload decoder may be used to decode the payload before executing

Useful Tools

- GCC: gcc -c shellcode.s
- Objdump: objdump -d shellcode.o
- LD: ld binary.o -o binary
- NASM: nasm -f elf64 shellcode.asm
- strace: trace system calls and signals
- Corelan’s pveWritebin.pl, pveReadbin.pl, and mona.py
- BETA3 --decode
- Ndisasm
- OllyDBG, Immunity Debugger, WinDBG
- GDB