**Architecture Fundamentals**

Let’s begin with some important concepts and terms that will help for understanding the architecture, attacks and exploits that will create and use later.

**CPU**

CPU is standing for Central Process Unit. Is charging of executing the machine code of the program.

**Machine code**

The machine code (Machine language) is the set of instructions that the CPU processes.

**Instructions**

Instruction is a primitive command that executes a specific operation such as move data, changes the execution flow of the program, preform arithmetic of logical operations and others.

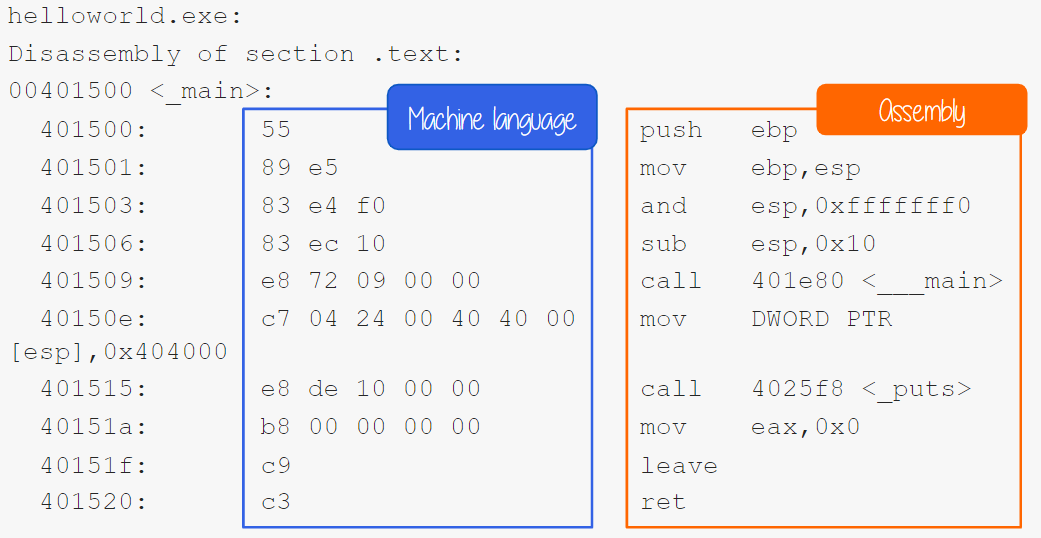
**Note:** CPU instructions are represented in Hexadecimal (HEX) format, Due to its complexity.

**Mnemonic code**

The same machine code gets translated into mnemonic code (a more readable language); this is called assembly language (ASM).

**Assembly language**

Assembly language has two popular are NASM (Netwide assembler) and MASM (Microsoft Macro Assembler).

[Example on machine code and assembly language]  
  
**Note:** Machine codes are represented in HEX

**ISA**

ISA is standing for Instruction set architecture. The ISA is set of instructions that a programmer (or compiler) must understand and use to write a program correctly for the specific CPU and machine.

One of command ISA is x86 instruction set originated by the intel 8086.

**Note:** the x86 acronym identifies 32-bit processors, while x64 (aka x86\_64 or AMD64) identifies 64-bit processors.

**Note:** the number of bits, 32 or 64, refer to width of CPU registers.

Each CPU has its fixed set of registers that are accessed when required.

**Registers**

Registers are small portions of memory in the CPU and serve to store data temporary, it is important to know that some of them have specific functions. While some others are used for general data storage.

**General Purpose Registers**

General Purpose registers (GPRs), we are going to summarizes the 8 general purpose registers.

[8 General purpose registers]

|  |  |  |
| --- | --- | --- |
| X86 Naming convention | Name | Purpose |
| EAX | Accumulator | Used in arithmetic operation |
| EBX | Base | Used as a pointer to data |
| ECX | Counter | Used in shift/rotate instruction |
| EDX | Data | Used in arithmetic operation and I/O |
| ESP | Stack pointer | Pointer to the top of the stack |
| EBP | Base pointer | Pointer to the base of the stack (aka stack base pointer, frame pointer) |
| ESI | Source index | Used as pointer to a source in stream operation |
| EDI | Destination Index | Used as pointer to a destination in stream operation |

**Note:** the naming convention refers to the x86 architecture.

The naming convention of the old 8-bit CPU had 16-bit register divided into two parts:

* A low byte: identified by a **L** at the end of the name.
* A high byte: Identified by a **H** at the end of the same.

The 16-bit naming convention combines the **L** and the **H** and replaces it with an **X**. while for stack pointer, base pointer, source and destination registers it simply removes the **L**.

In the 32-bits representation, the register acronym is prefixed with an **E.** meaning extended. Whereas, in the 64-bit representation, the **E** is replaced with the **R**.

The following tables summarize the naming conventions, we will mainly use 32-bit name convention it is useful to understand the 64-bit name convention as well.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Register | Accumulator | | | | Counter | | | | Data | | | | Base | | | |
| 64-bit | RAX | | | | RCX | | | | RDX | | | | RBX | | | |
| 32-bit |  | EAX | | |  | ECX | | |  | EDX | | |  | EBX | | |
| 16-bit |  |  | AX | |  |  | CX | |  |  | DX | |  |  | BX | |
| 8-bit |  |  | AH | AL |  |  | CH | CL |  |  | DH | DL |  |  | BH | BL |

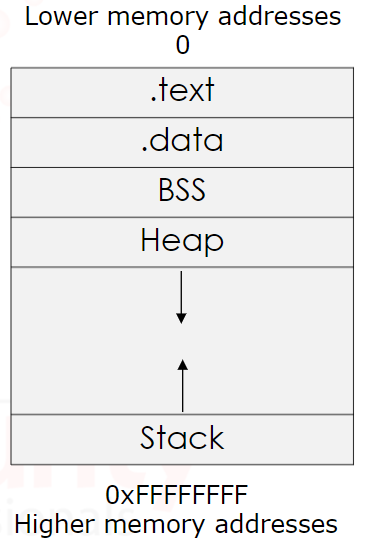
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Register | Stack Pointer | | | | Base Pointer | | | | Source Index | | | | Destination Index | | | |
| 64-bit | RSP | | | | RBP | | | | RSI | | | | RDI | | | |
| 32-bit |  | ESP | | |  | EBP | | |  | ESI | | |  | EDI | | |
| 16-bit |  |  | SP | |  |  | BP | |  |  | SI | |  |  | DI | |
| 8-bit |  |  |  | SPL |  |  |  | BPL |  |  |  | SIL |  |  |  | DIL |

In addition to the eight general purposes registers, there is also another register that will be important for our purposes, the **EIP** (x86 naming convention). The instruction pointer (EIP) controls the program execution by storing a pointer to the address of the next instruction (machine code) that will be executed.

**Note:** EIP tells the CPU where the next instruction.

**Process Memory**

When a process runs, it is typically organized in memory as shown in the figure below.



The process is divided into four regions:

1. Text.
   1. Text region is fixed by the program and contains the program code (instructions). This region is marked as read-only since the program should not change during execution.
2. Data.
   1. Data region is divided into initialized data and uninitialized data.
      1. Initialized data includes items such as statics and global declared variables that are pre-defined and can be modified.
      2. Uninitialized data is named block started by symbol **(BSS)**, also initializes variables that initialized to zero or do not have explicit initialization (ex: static int num1)
3. Heap.
   1. Heap starts right after the BSS segment. During the execution, the program can request more space in memory via brk and sbrk system calls, used by mlloc, realloc and free. Hence, the size of the data region can be extended; this is not vital, but if you are very interested in more detailed process, **these may be topics to do your own research on.**
4. Stack.
   1. It is located in the higher prat of the memory. You can think of the stack as an array used for saving a function’s return addresses, passing function arguments and storing variables.

**The Stack**

The stack is a **LIFO** (Last-in-first-out) block of memory. The purpose of **ESP** register (Stack Pointer) to identify the top of stack and it is modified each time a value is pushed in (**PUSH**)or popped out (**POP**)**.**

**Note:** common sense would make you think that the stack grows upwards, towards higher memory addresses, but as you saw the previous memory structure diagram, the stack grows downward, toward the lower memory addresses. (this is probably due to historical reasons)

**Note:** it was decided that the Heap would start from lower addresses and grow upwards and the stack would start from the end of the memory and grow downwards.

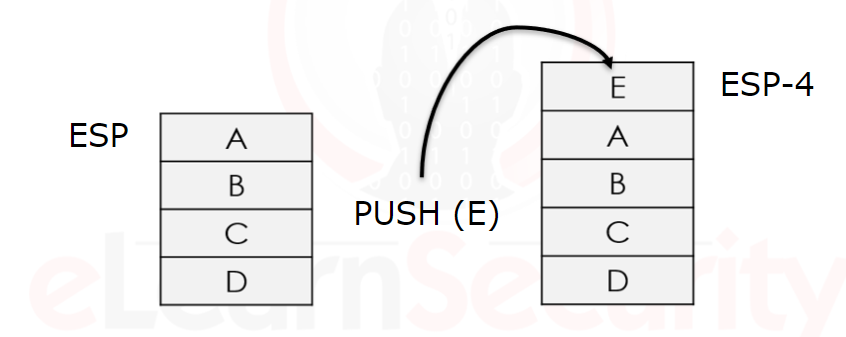
**PUSH Instruction**

At first, we are going to execute following instruction: PUSH E.

* PUSH instruction: PUSH E
* Push process: A PUSH is executed, and the **ESP** register is modified.
* Starting value: the **ESP** points to the top of the stack.

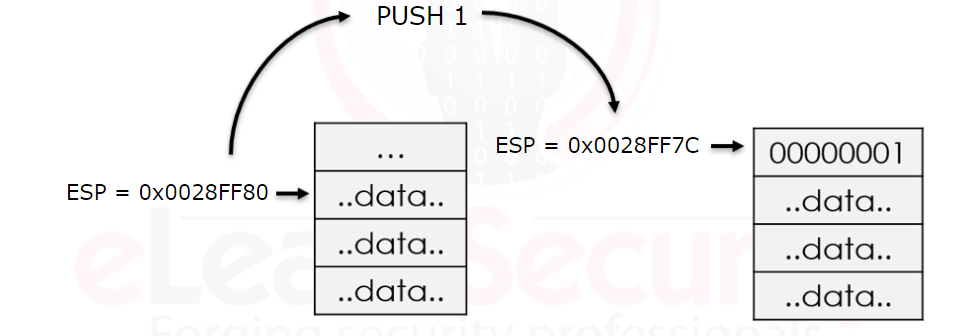
Process: a **PUSH** instruction subtracts 4 (in 32-bit) or 8 (in 64-bit) from the **ESP** and writes the data to memory address in the **ESP** and then updates the **ESP** to the top of the stack.

**Note:** the stack grows backward. Therefore, the **PUSH** subtracts 4 or 8, in order to point to a lower location on the stack. If we do not subtract it, the **PUSH** operation will overwrite the current pointed by **ESP** (the top) and we would lose data.



Now for a more detailed example of **PUSH** INSTRUCTION:

* STARTING VALUE: **ESP** contains the address value.
  + **ESP** point to the following address: 0x028ff80
* PROCESS: the program executes the instruction **PUSH** 1. **ESP** decreases by 4, become 0x0028ff7c and value 1 will be pushed on the stack.
* ENDING VALUE: **ESP** points to the following memory address: 0x0028ff7c

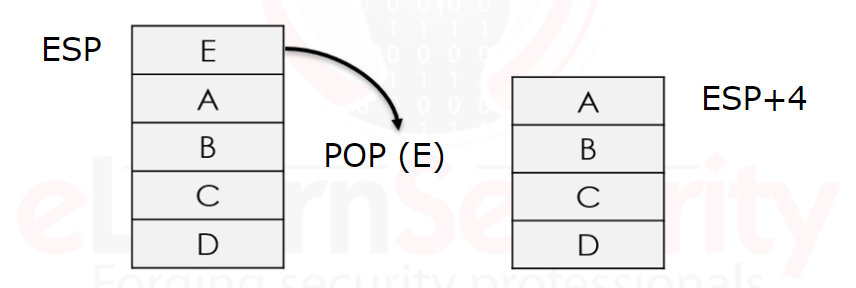


**POP Instruction**

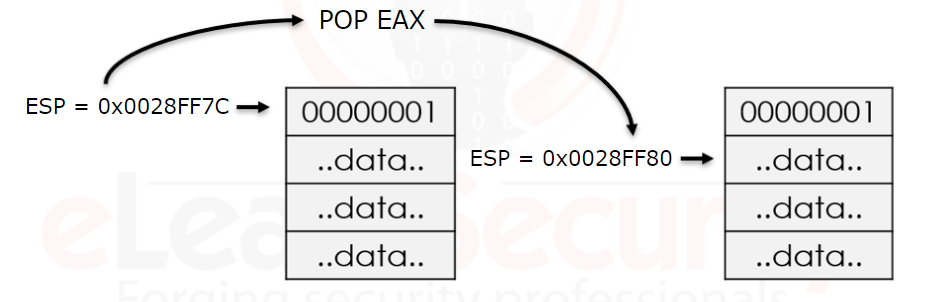
At first, we are going to execute following instruction: POP E

* POP instruction: **POP E**.
* POP process: a **POP** is executed, and the **ESP** register is modified.
* Starting value: the **ESP** points to the top of the stack. (Pervious **ESP** +4).

Process: the **POP** operation is opposite of **PUSH,** and it retrieves data from the top of the stack. Therefore, the data contained at the address location in **ESP** (the top of the stack) is retrieved and stored (usually in another register). After a **POP** operation, the **ESP** value is incremented, in **x86** by 4 or in **x64** by 8



Here is a more detailed example of **POP**.

* STARTING VALUE: **ESP** contains the address value.
  + After the **PUSH** 1,the **ESP** points to the following address: 0x0028ff7c.
* PROCESS: the program executes the inverse instruction, **POP EAX**. The value (00000001) contained at the address of the **ESP** (0x0028ff7c = the top of the stack) will be popped out from the stack and will be copied in the **EAX** register. Then, **ESP** is updated by adding 4 becoming 0x28ff80.

**V.IMP Note:** the value popped from the stack is not deleted (or zeroed). It will stay in the stack until another instruction overwrites it.

**Procedures and Functions**

We will investigate how procedures and functions work. It is important to know that procedures and functions alter the normal flow of the process. When a procedure of a function terminates, it returns control to the statement or instruction that called the function.

**Stack frames**

Functions contain two important components, the **prologue** and the **epilogue**, the **prologue** prepares the stack to be used, when the function has completed, the **epilogue** reset the stack to the prologue settings.

**Note:** the stack consists of logical **stack frames** (portions/areas of the stack), that are pushed when calling a function and popped when returning a value.

When a subroutine, such as a function or procedure, is started, a stack frame is created and assigned to the current **ESP** location (top of the stack); this allows the subroutine to operate independently in its own location in the stack.

When the subroutine ends, two things happen:

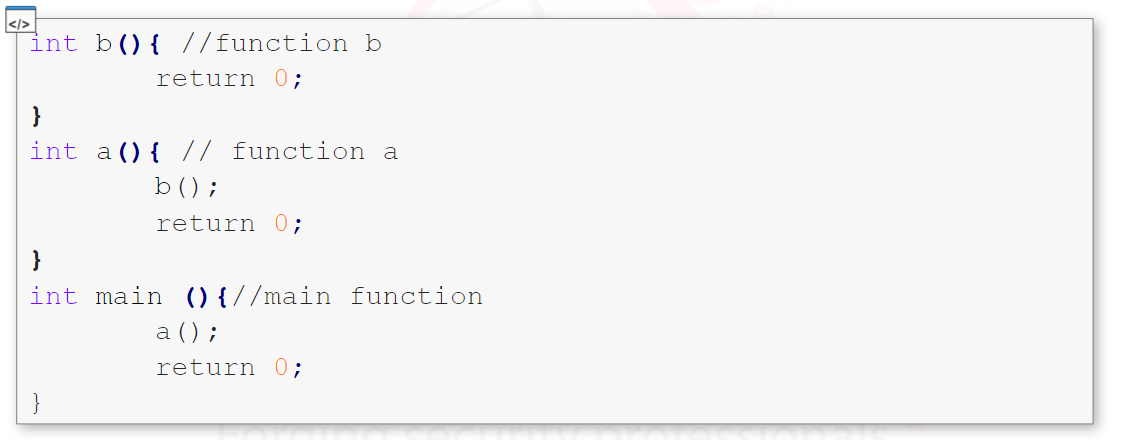
1. The program receives the parameters passed from the subroutine.
2. The instruction pointer (**EIP)** is reset to the location at the end tine of the initial call.

**Note:** In other words, the stack frame keeps track of the location where each subroutine should return control when it terminates.

This process has three main operations:

1. When a function is called, the arguments [(in brackets)] need to be evaluated.
2. The control flow jumps to the body of the function, and the program executes its code.
3. Once the function ends, a return statement is encountered, the program returns to the function called (the next statement in the code).

The following diagram explains how this works in the stack. This example is written in **C**:

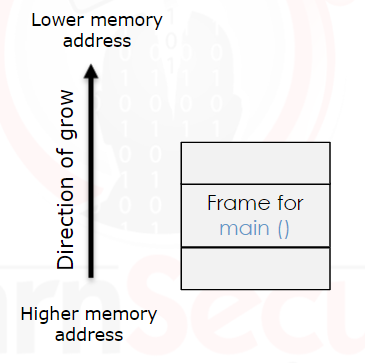


STEP 1:

The entry point of the program is main ().

The first stack frame that need to be pushed to the stack is the main () stack frame. Once initialized, the stack pointer is set to the top of the stack and a new main () stack frame is created.

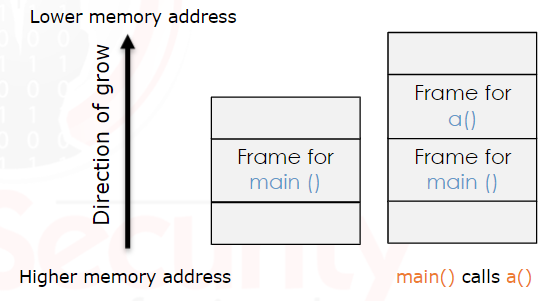
[Our stack will then look like the following]



STEP 2:

Once inside main (), the first instruction that execute is a call to the function named a (). Once again, the stack pointer is set to the top of the stack of main () and a new stack frame for a () is created on the stack.

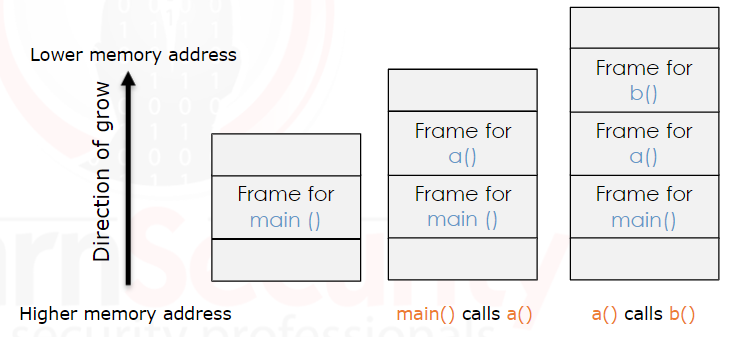
[Our stack will then look like the following]



STEP 3:

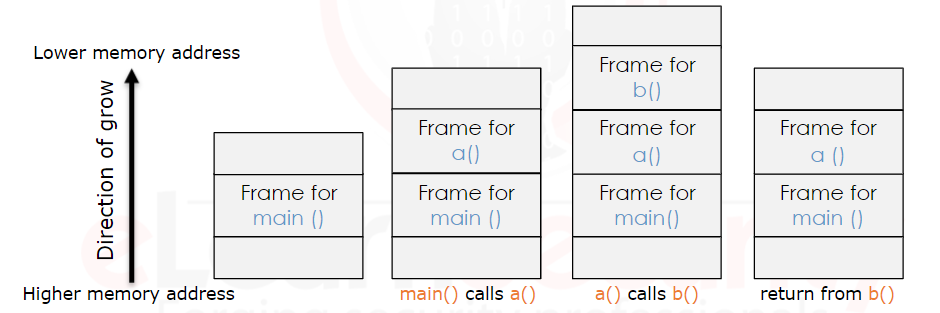
Once the function a () starts, the first instruction is a call to the function named b (). Here again, the stack pointer is set, and a new stack frame for b 9) will be pushed on the top of the stack.

[Our stack will then look like the following]



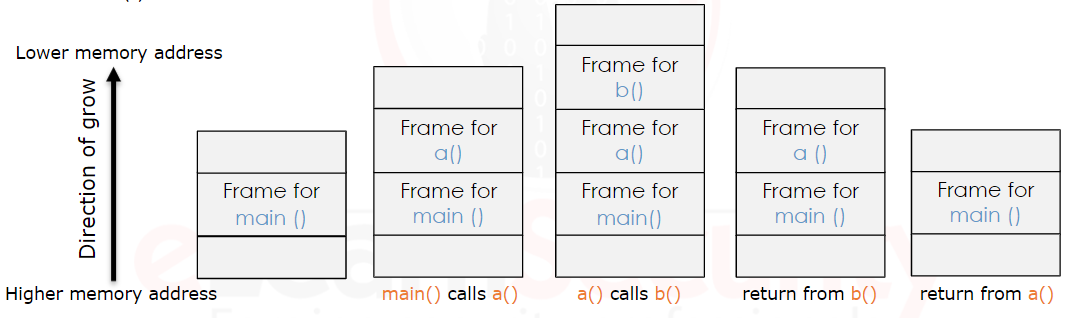
STEP 4:

The function b () does nothing and just returns. When the function completes, the stack pointer is moved to its pervious location, and the program return to the stack frame of a () and continues with the next instruction.

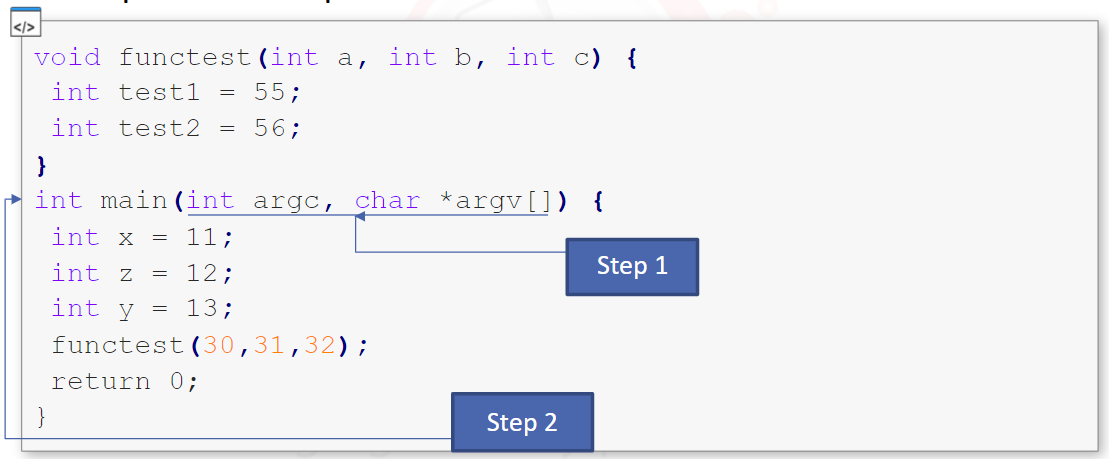


STEP 5:

The next instruction executed is the return statement contained in a (). The a () stack frame is popped, the stack pointer is reset, and we will get back in the main () stack frame.

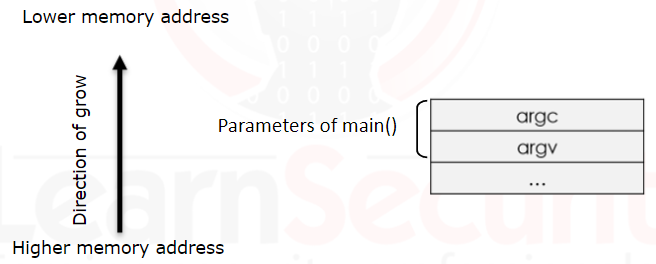


This second example is also written in **C**:



STEP 1:

When the program starts, the function main () parameters (argc, argv) will be pushed on the stack, from right to left. Our stack will look like this:



STEP 2:

**CALL** the function main (). Then, the processor pushes the content of the **EIP** (instruction pointer) to the stack and points to the first byte after call instruction.

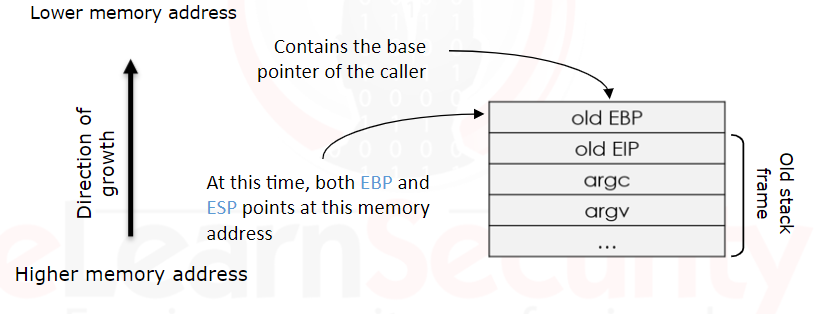
**Note:** this process is important because we need to know the address of the next instruction in order to proceed when we return from the function called.

STEP 3:

The caller (the instruction that executes the function calls – the OS in this case) loses its control, and the callee (the function that is called – the main function) takes control.

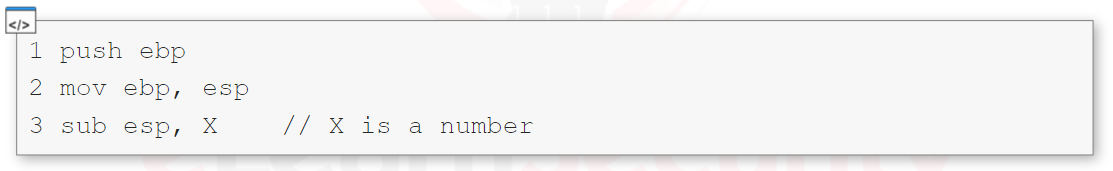
STEP 4:

Now that we are in the main () function, a new stack frame needs to be created. The stack frame is defined by the **EBP** (Base pointer) and the **ESP** (Stack pointer). Because we don’t want to lose the old stack frame information belonged to the previous stack frame, the function that called main (). Once its value is stored, the **EBP** is updated, and it points to the top of the stack.



**Prologue**

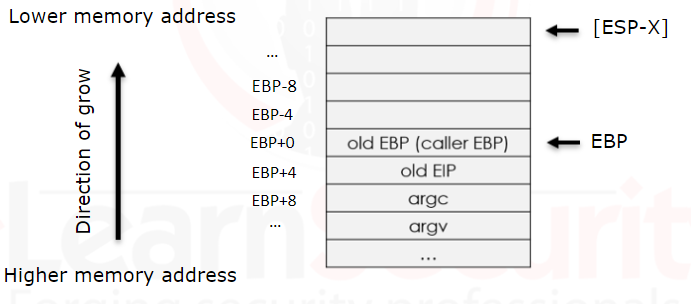
The previous step is known as the **prologue**: it is a sequence of instructions that take place at the beginning of a function. This will occur for all functions. Once the callee get the control, it will execute the following instructions:



* 1st instruction:
  + Push ebp saves the old base pointer onto the stack, so it can be restored later on when the function returns.
  + **EBP** is currently pointing to the location of the top of the previous stack frame.
* 2nd instruction:
  + The second instruction (mov ebp, esp) copies the value of the stack pointer (**ESP** – top of the stack) into the base pointer (**EBP**); this creates a new stack frame on top of the stack.
    - The base of the new stack frame is on top of the old stack frame.

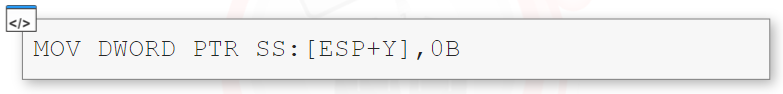
**Note V.IMP:** Notice that in assembly, the second operand of the instruction (esp in this case) is the source, while the first operand (ebp in this case) is the destination. Hence, esp is moved into ebp.

* 3rd instruction:
  + The last instruction (sub esp, X) moves the stack pointer (top of the stack) by decreasing its value; this is necessary to make space for the local variables.
    - Similar to the previous instruction, X is the source and esp is the destination, therefor, the instruction subtracts X from esp and the result will be stored at esp.

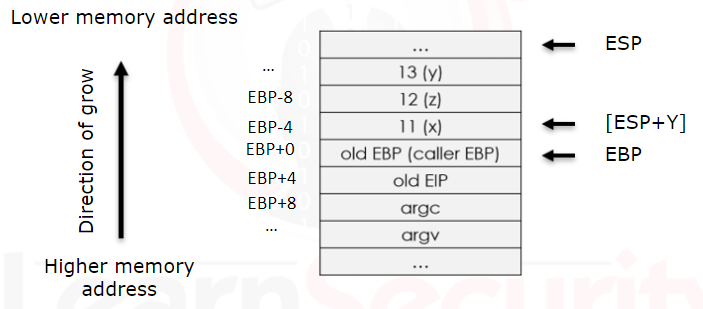


Once the prologue ends, the stack frame for main () is complete, and the local variables are copied. Since **ESP** is not pointing to the memory address right after **EBP**, we cannot use **PUSH** operation, since **PUSH** stores the value on top of the stack (the address pointed by **ESP**).

The instruction after prologue are like the following:



This process will repeat through all the variables, and once the process completes, the stack will look like the following:

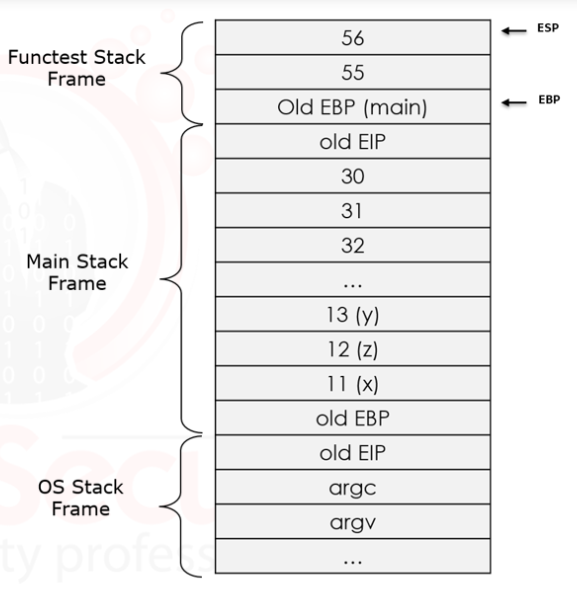


Then the main () continues executing its instructions.

STEP 5:

Looking back at the source code from the second example, we can see that the next instruction calls the function functest ().

The whole process will be executed again. This time a new stack frame will be created for the function functest ().



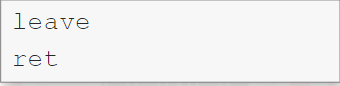
**Epilogue**

When the program executes a return statement, the previous stack frame is restored thanks to **epilogue.**

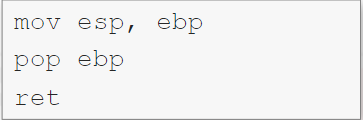
The operations executed by the **epilogue** are the following:

* Return the control to the caller.
* Replace the stack pointer with the current base pointer. It restores its value to before the prologue; this is done my popping the base pointer from the stack.
* Return to the caller by popping the instruction pointer from the stack (stored in the stack) and then it jumps to it.

The following code represents the epilogue:



The instructions can also be written as following:

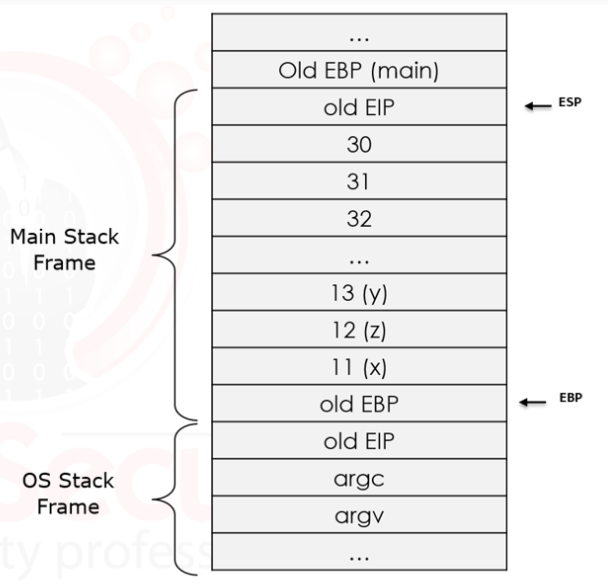


**Note:** even if the code does not contain a return, when the program leaves a subroutine it will still run the epilogue.

* 1st instruction:
  + (mov esp, ebp) after it gets executed, both **ESP** and **EBP** point to the same location.
* 2nd instruction:
  + (pop ebp) which simply pop the value from the top of the stack into **EBP**. Since the top of the stack points to the memory address location where the old **EBP** is stored (the **EBP** of the caller), the caller stack frame is restored.

**Note:** it is important to know that a **POP** operation automatically updates the **ESP** (same as the **PUSH**).

* + Therefore, **ESP** now points to the old **EIP** previously stored.



* 3rd instruction:
  + **RET** pops the vaule contained at the top of the stack to the old **EIP** – the next instruction after the caller, and jump to that location. This gives control back to the caller. **RET** affects only the **EIP** and the **ESP** registers.

**Endianness**

Endianness is the way of representing (storing) values in memory.

Even through there are three type of endianness, we are going to explain 2 of them:

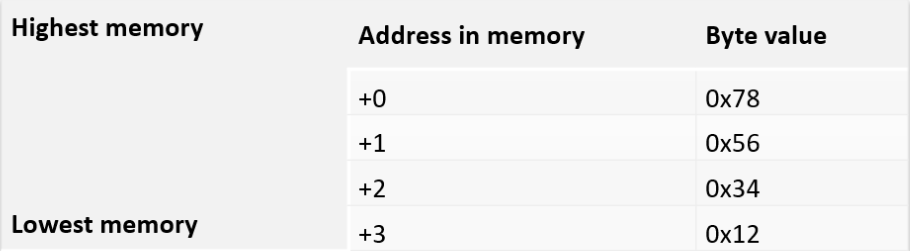
* Big-endian
* Little-endian

First it is important to know these two concepts:

* **MSB** (most significant bit) in a binary is the largest value, usually the first from the left.
* **LSB** (least significant bit) in a binary is the lowest values, usually the first from the right.

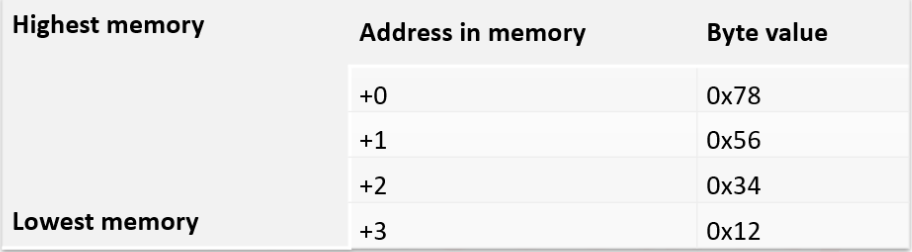
However, big-endian representation, the least significant byte (LSB) is stored at the highest memory address. While the most significant byte (MSB) is at the lowest memory address.

[Example: 0x12345678]

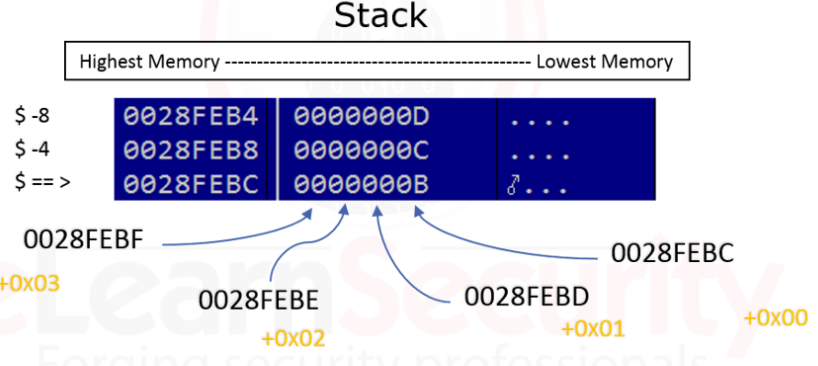


Respectively, the little-endian representation, the most significant byte (MSB) is at the highest memory address. While least significant byte (LSB) is stored at the lowest memory address.

[Example: 0x12345678]



Example on little-endian, the value 11 (0B in hexadecimal => 0x0000000B).



**NOPs**

NOP is an assembly language instruction that does nothing, when the program encounters a **NOP**, it will simply skip to the next instruction. In intel x86 CPUs, **NOP** instructions are represented with the hexadecimal value 0x90.

NOP-sled:

Technique used during the exploitation process of Buffer Overflows. Its only purpose is to fill a large (or small) portion of the stack with **NOPs**; this will allow us to slide down to the instruction we want to execute, which is usually put after the NOP-sled.