Penetration Testing Professional

Architecture Fundamentals

Section 1: System Security – Module 1

v5.0

1.0 Architecture Fundamentals

MAP REF

1.1. Introduction

1.2. Architecture Fundamentals

1.3. Security Implementations

Introduction

Welcome to the **System Security** section of the PTP course.

The purpose of this section is to give you the fundamental concepts needed to help you improve your skills in topics such as fuzzing, exploit development, buffer overflows, debugging, reverse engineering and malware analysis.

If you choose to continue your cybersecurity career as a reverse engineer, malware analyst, penetration tester, forensics investigator, etc., this will provide a solid foundation in this area.

During the course, we will begin with important topics such as x86 and x64 architectures, assembly, compilers, security implementations such as **ASLR** and **DEP**, to name a few.

We will focus on attacking techniques such as the buffer overflows (BOF). We explore how BOF works and how we can fuzz and reverse engineer applications in order to find vulnerabilities. Finally, utilizing those vulnerabilities, we will see how to exploit them to execute arbitrary code on the target machine.

Although some of the topics and the techniques can be applied to different operating systems, we will focus mainly on Windows.

During the course, we will provide the tools and instructions for you to build your own lab.

Architecture Fundamentals

Let’s begin with some important concepts and terms that will help you understand the attacks and exploits that we will create and use later.

We will discuss the CPU, instructions, registers, machine code, assembly language, memory, the stack and much more. If you are comfortable with these topics, feel free to skip this section.

The Central Process Unit (**CPU**) is the device in charge of executing the **machine code** of a program.

The machine code, or machine language, is the set of instructions that the CPU processes.

Each **instruction** is a primitive command that executes a specific operation such as move data, changes the execution flow of the program, perform arithmetic or logic operations and others.

CPU instructions are represented in hexadecimal (HEX) format. Due to its complexity, it is impossible for humans to utilize it in its natural format.

Therefore, the same machine code gets translated into mnemonic code (a more readable language); this is called the **assembly** language (**ASM**). The two most popular are NASM (Netwide Assembler) and MASM (Microsoft Macro Assembler). The assembler we are going to use is **NASM**.

https://www.nasm.us/
https://www.microsoft.com/en-us/download/details.aspx?id=12654

The example in the next slide is the machine code of a program called *sample1_helloworld.exe*.

We explain this later, but for now, it is useful to understand the differences between the machine language and the assembly language.

You can download all the samples by opening the Resource tab in the member's area and downloading the module archive.

1.2.1 CPU, ISA and Assembly

MAP REF

```
helloworld.exe:
Disassembly of section .text:
00401500 <_main>:
  401500:       55                      push   ebp
  401501:       89 e5                   mov    ebp,esp
  401503:       83 e4 f0                and    esp,0xfffffff0
  401506:       83 ec 10                sub    esp,0x10
  401509:       e8 72 09 00 00          call   401e80 <___main>
  40150e:       c7 04 24 00 40 40 00    mov    DWORD PTR
[esp],0x404000
  401515:       e8 de 10 00 00          call   4025f8 <_puts>
  40151a:       b8 00 00 00 00          mov    eax,0x0
  40151f:       c9                      leave
  401520:       c3                      ret
```

Machine language

Assembly

IMPORTANT

Each CPU has its own **instruction set architecture** (**ISA**). The **ISA** is the set of instructions that a programmer (or a compiler) must understand and use to write a program correctly for that specific CPU and machine.

In other words, **ISA** is what a programmer can see: memory, registers, instructions, etc. It provides all the necessary information for who wants to write a program in that machine language.

One of the most common **ISA** is the **x86** instruction set (or architecture) originated from the **Intel 8086**.

The **x86** acronym identifies 32-bit processors, while **x64** (aka **x86_64** or **AMD64**) identifies the 64-bit versions.

1.2.2 Registers

The number of bits, **32** or **64**, refers to the width of the **CPU registers**.

Each CPU has its fixed set of registers that are accessed when required. You can think of registers as temporary variables used by the CPU to get and store data.

1.2.2 Registers

Registers are a key component of this module. Although almost all registers are small portions of memory in the CPU and serve to store data temporarily, it is important to know that some of them have specific functions, while some others are used for general data storage.

For the purpose of this course, we will focus on a specific group of registers: The General Purpose Registers (**GPRs)**.

The following table summarizes the eight general purpose registers. Notice that the naming convention refers to the x86 architecture. We will see how the names differ for 64-bit, 32-bit, 16-bit and 8-bit.

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

Although this information may seem overwhelming, everything will be more clear once we will talk about the stack.

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

- A low byte, identified by an **L** at the end of the name, and
- A high byte, identified by an **H** at the end of the name.

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

1.2.2 Registers

The following tables summarize the naming conventions. Although we will mainly use the 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	Destination
64-bit	RSP	RBP	RSI	RDI
32-bit	ESP	EBP	ESI	EDI
16-bit	SP	BP	SI	DI
8-bit	SPL	BPL	SIL	DIL

1.2.2 Registers

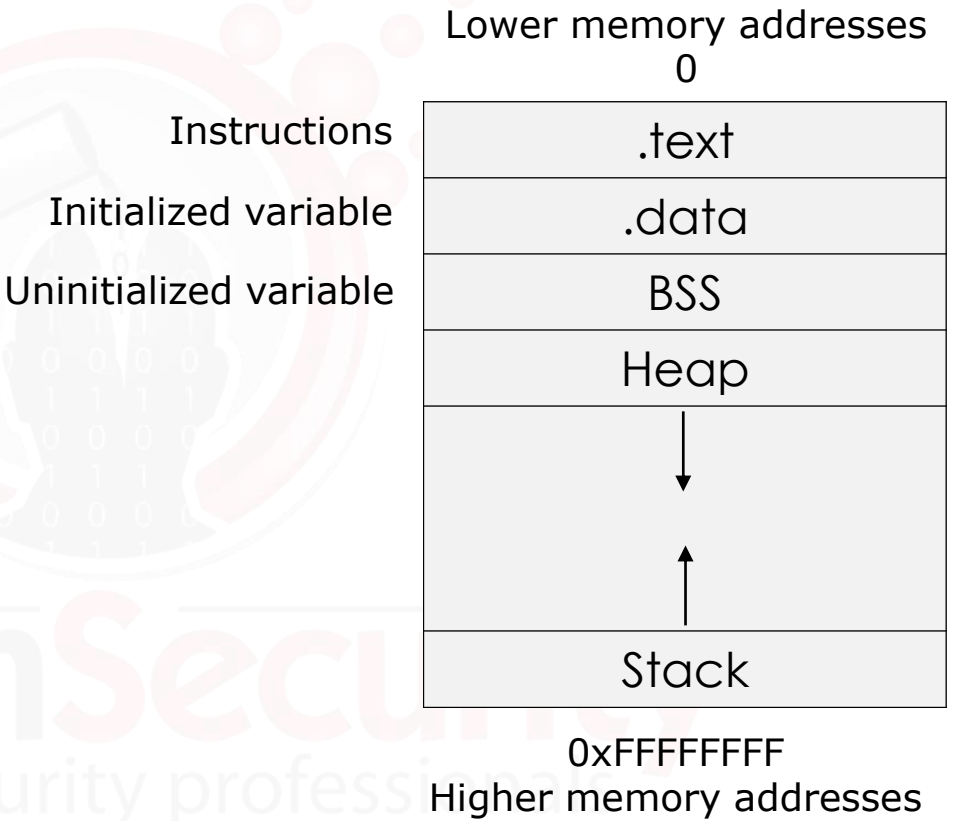
In addition to the eight general purposes registers, there is also another register that will be important for our purposes, the **EIP** (x86 naming convention). The Instruction Pointer (**EIP)** controls the program execution by storing a pointer to the address of the next instruction (machine code)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

IMPORTANT

It tells the CPU where the next instruction is.

1.2.3 Process Memory

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



The process is divided into four regions: **Text**, **Data, the Heap,** and the **Stack**.

The **Text** region, or instruction segment, 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.

The **Data** region is divided into initialized data and uninitialized data. Initialized data includes items such as static and global declared variables that are pre-defined and can be modified.

The uninitialized data, named Block Started by Symbol (**BSS**), also initializes variables that are initialized to zero or do not have explicit initialization (ex. `static int t`).

Next is the **Heap**, which 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 a more detailed process, these may be topics to do your own research on.

The last region of the memory is the **Stack**. For our purposes, this is the most important structure we will deal with.

http://man7.org/linux/man-pages/man2/brk.2.html
http://man7.org/linux/man-pages/man3/malloc.3.html
http://man7.org/linux/man-pages/man3/realloc.3p.html
http://man7.org/linux/man-pages/man1/free.1.html

1.2.4 The Stack

The Stack is a Last-in-First-out (**LIFO)** block of memory. It is located in the higher part 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 local variables.

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

Before seeing how the stack works and how to operate on it, it is important to understand how the stack grows.

Common sense would make you think that the stack grows upwards, towards higher memory addresses, but as you saw in the previous memory structure diagram, the **stack grows downward**, towards the lower memory addresses.

This is probably due to historical reasons when the memory in old computers was limited and divided into two parts: *Heap* and *Stack*.

Knowing the limits of the memory allowed the programmer to know how big the heap and/or the stack would be.

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 downward.

As previously mentioned, the stack is a **LIFO** structure, and the most fundamental operations are the **PUSH** and **POP**.

The main pointer for these operations is the **ESP**, which contains the memory address for the top of the stack and changes during each **PUSH** and **POP** operation.

In the majority of these modules, when a new topic is introduced like the `PUSH` process, we will provide you with two examples.

The first will be of the concept. It will not be overly specific, but it will provide you with a good overview. The second example will be an actual breakdown of the previous example with specific languages and terms.

While the first examples are fairly simple, understanding them will lay a good foundation for the more specific concepts we will explain later.

We will now give you a visual representation of what we have been talking about.

The first example of how the stack changes is the execution of the following instruction: `PUSH E`.

The second example is the execution of the following instruction: `POP E`.

PUSH Instruction:

- **PUSH** E

PUSH Process:

- A **PUSH** is executed, and the **ESP** register is modified.

Starting value:

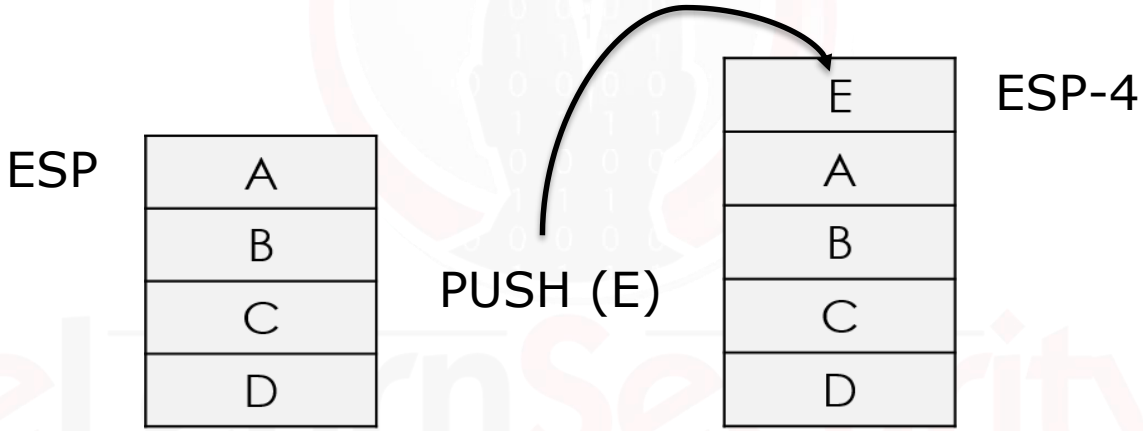
- The **ESP** points to the top of the stack.

Process:

- A **PUSH** instruction subtracts 4 (in 32-bit) or 8 (in 64-bit) from the **ESP** and writes the data to the memory address in the **ESP**, and then updates the **ESP** to the top of the stack. Remember that the Stack grows backward. Therefore the **PUSH** subtracts 4 or 8, in order to point to a lower memory location on the stack. If we do not subtract it, the **PUSH** operation will overwrite the current location pointed by **ESP** (the top) and we would lose data.

Ending value:

- The **ESP** points to the top of the stack -4.



Now for a more detailed example of the **PUSH** instruction.

Starting value: (ESP contains the address value)

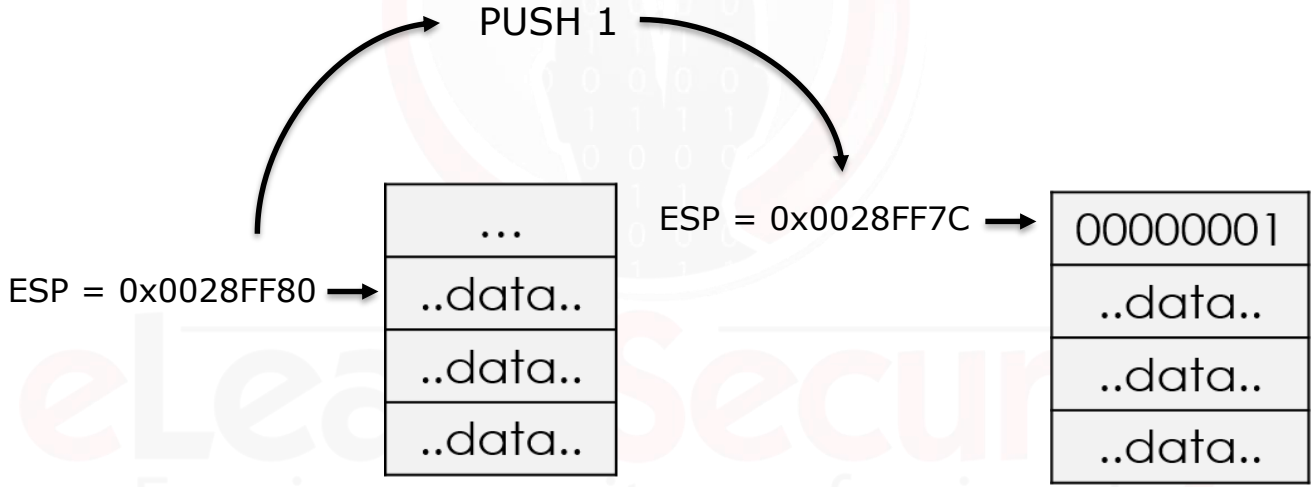
- **ESP** points to the following memory address: `0x0028FF80`.

Process:

- The program executes the instruction `PUSH 1`. **ESP** decreases by `4`, becoming `0x0028FF7C`, and the value `1` will be pushed on the stack.

Ending value:

- **ESP** points to the following memory address: `0x0028FF7C`.



1.2.4.2 POP Instruction

POP Process:

- A **POP** is executed, and the **ESP** register is modified.

Starting value:

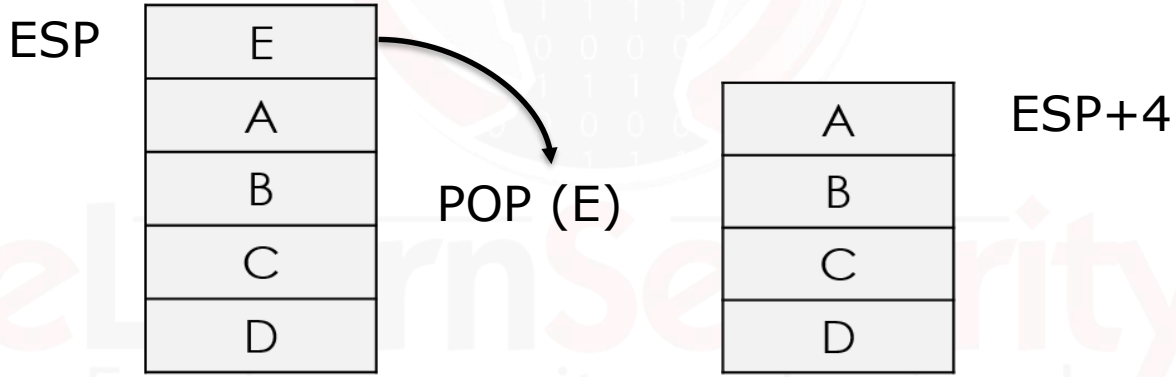
- The **ESP** points to the top of the stack. (Previous **ESP** +4)

Process:

- The **POP** operation is the 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.

Ending value:

- The **ESP** points to the top of the stack. (Same as the previous location before the **PUSH**)



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

Starting value: (**ESP** contains the address value)

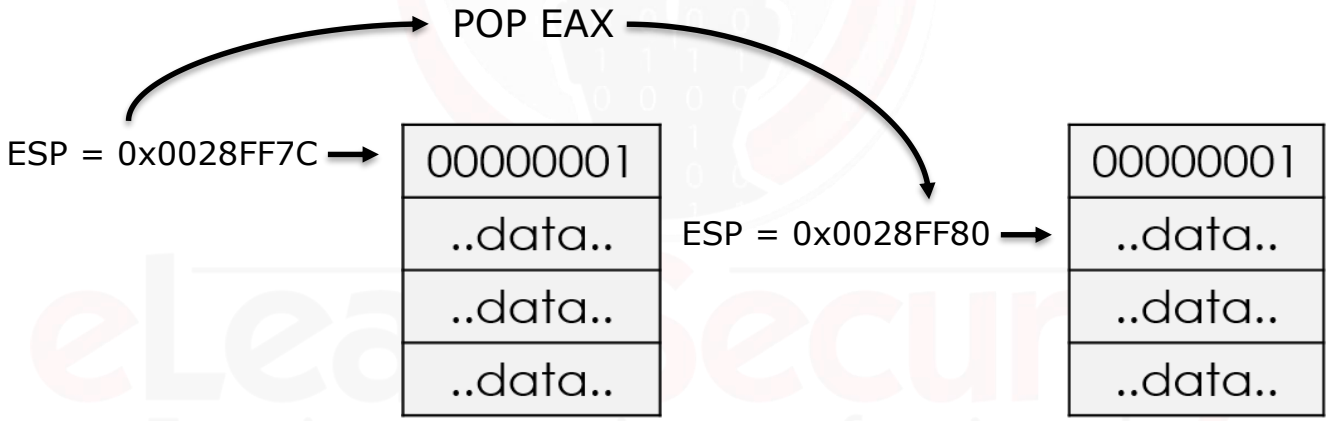
- After the `PUSH 1`, the **ESP** points to the following memory address: `0x0028FF7C`.

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 and becoming `0x0028FF80`.

Ending value:

- **ESP** points to the following memory address: 0x0028FF8. It returns to its original value.



It is important to understand that the value popped from the stack is not deleted (or zeroed). It will stay in the stack until another instruction overwrites it.

Now that we know more about the Stack, 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 or a function terminates, it returns control to the statement or instruction that called the function.

1.2.4.4 Stack Frames

Functions contain two important components, the **prologue** and the **epilogue,** which we will discuss later, but here is a very quick overview. The **prologue** prepares the stack to be used, similar to putting a bookmark in a book. When the function has completed, the **epilogue** resets the stack to the prologue settings.

The Stack consists of logical **stack frames** (portions/areas of the Stack), that are **PUSH**ed when calling a function and **POP**ped 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 time of the initial call.

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

We will break down this process in a more specific example for you to better understand how stack frames work. First, we will explain the operations, and then we will illustrate how it happens in an actual program. When this program is run, the following process occurs.

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 call (the next statement in the code).

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

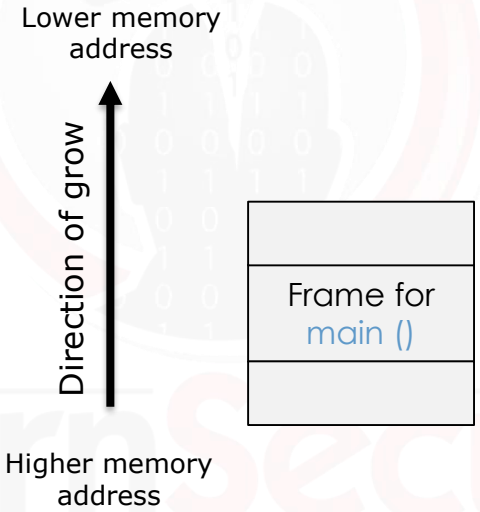
```
int b(){ //function b
        return 0;
}
int a(){ // function a
        b();
        return 0;
}
int main (){//main function
        a();
        return 0;
}
```

STEP 1

The entry point of the program is `main()`.

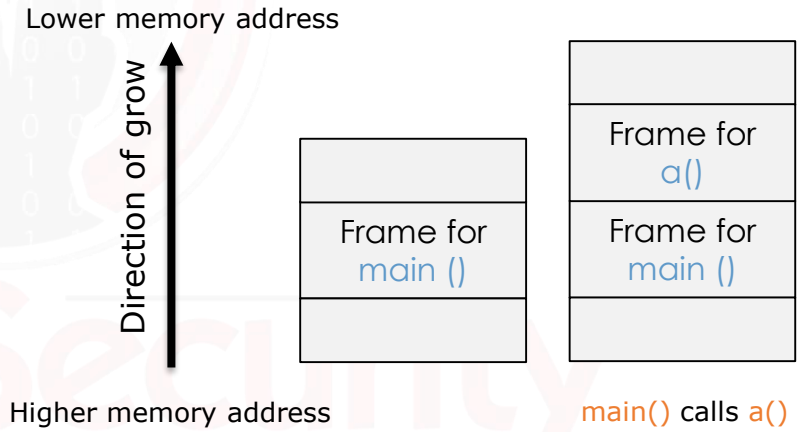
The first stack frame that needs to be pushed to the Stack is the `main()` stack frame. Once initialized, the stack pointer is set to the top of the stack and a new `main()` stack frame is created.

Our stack will then look like the following:



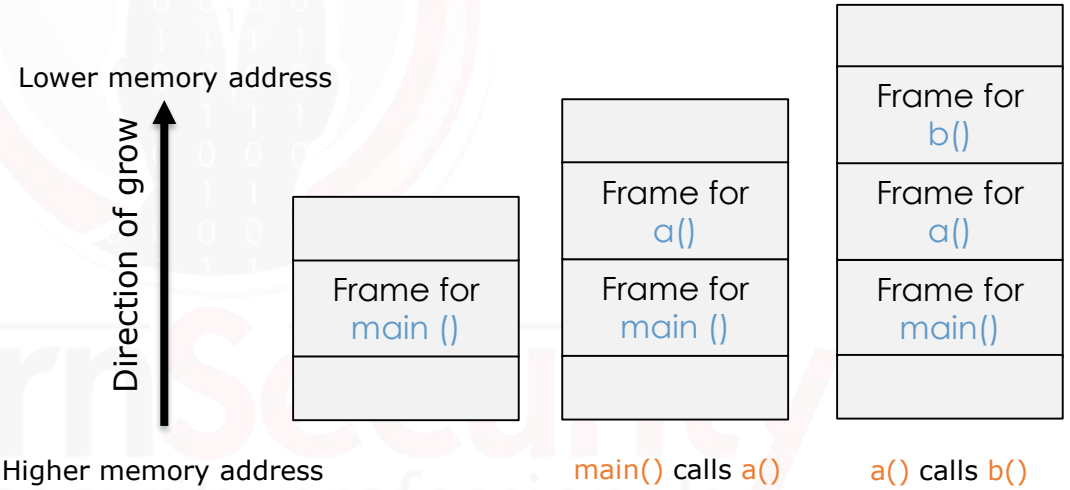
STEP 2

Once inside `main()`, the first instruction that executes 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.



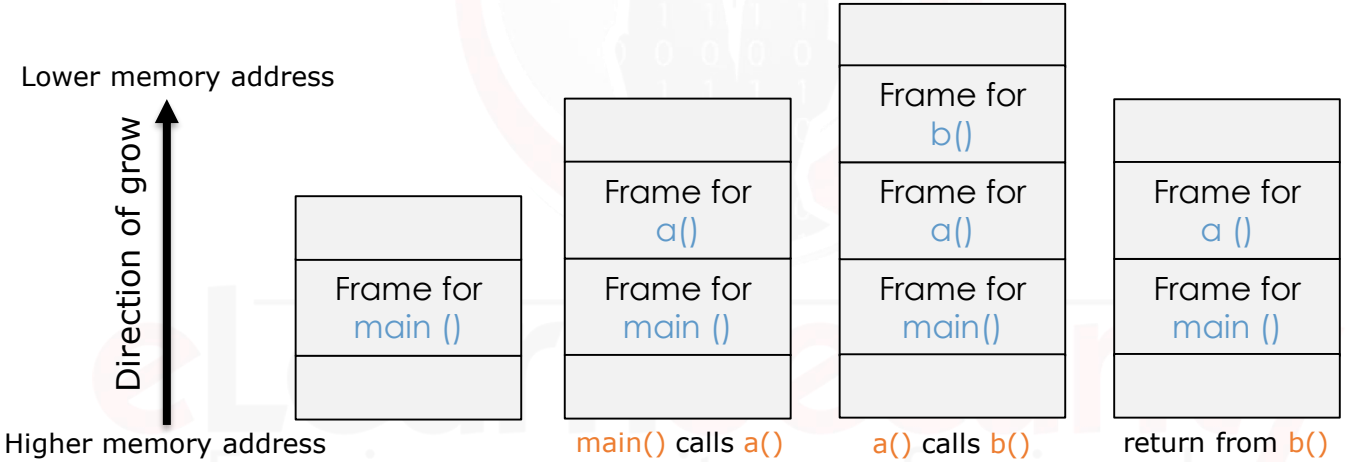
STEP 3

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()` will be pushed on the top of the stack.



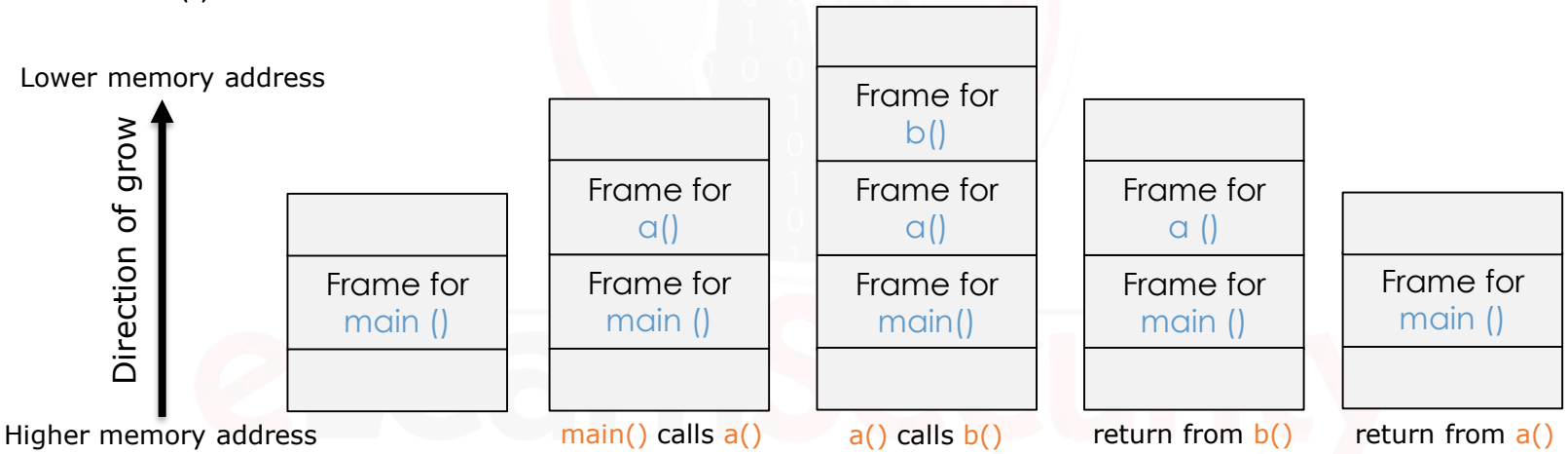
STEP 4

The function `b()` does nothing and just returns. When the function completes, the stack pointer is moved to its previous location, and the program returns 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 was a quick overview of how stack frames work, but for buffer overflows, we need to go into more detail as to what information is stored, where it is stored and how the registers are updated.

This second example is also written in C.

The most important thing to observe is how the stack changes. Let's inspect the steps in the next slides.

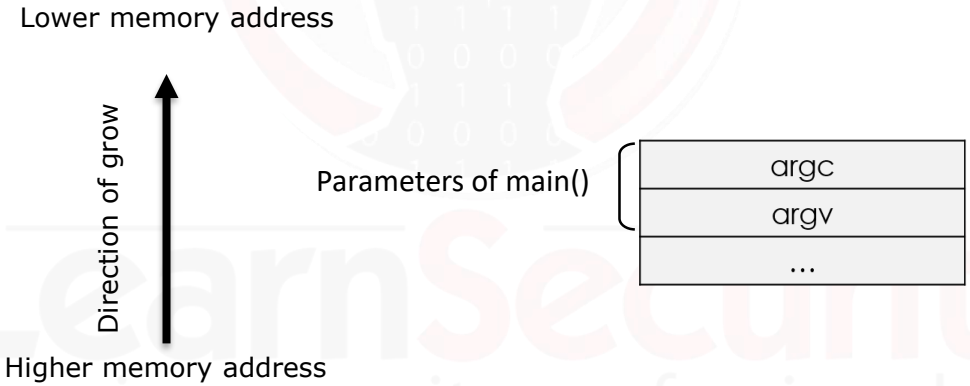
```
void functest(int a, int b, int c) {
 int test1 = 55;
 int test2 = 56;
}
int main(int argc, char *argv[]) {
 int x = 11;
 int z = 12;
 int y = 13;
 functest(30,31,32);
 return 0;
}
```

Step 1

Step 2

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:



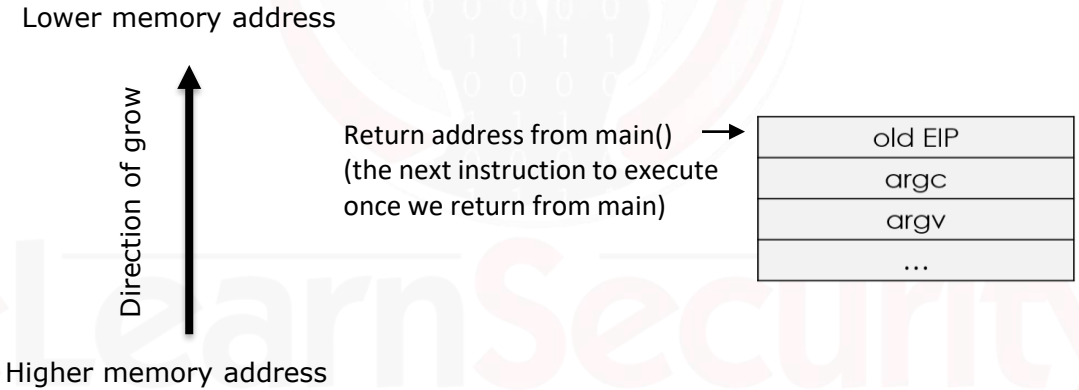
STEP 2

CALL the function `main()`. Then, the processor **PUSH**es the content of the **EIP** (Instruction Pointer) to the stack and points to the first byte after the `CALL` instruction.

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

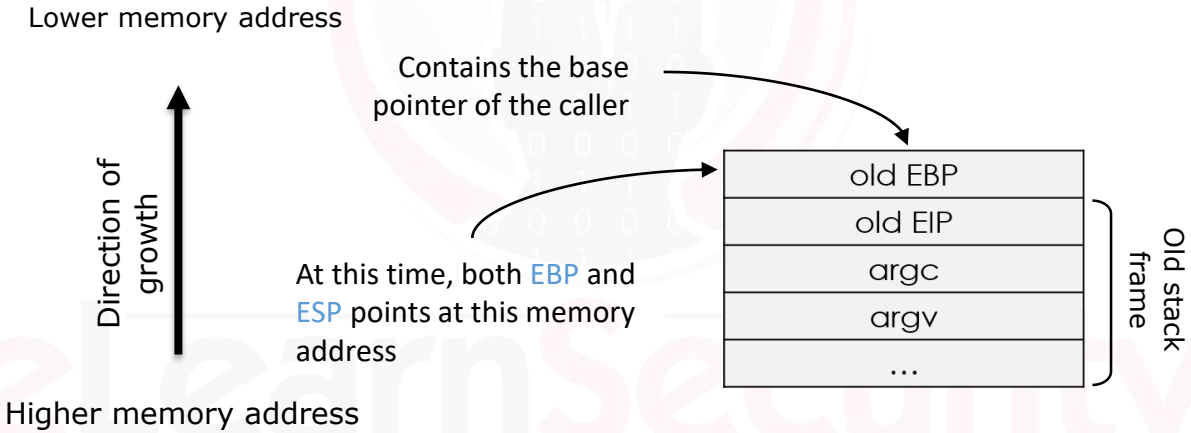
The caller (the instruction that executes the function calls - the OS in this case) loses its control, and the callee (the function that is called - the main function) takes control.



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, we have to save the current **EBP** on the Stack. If we did not do this, when we returned, we will not know that this 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.

From this point, the new stack frame starts on the top of the old one.



1.2.4.5 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 gets the control, it will execute the following instructions:

```
1 push ebp
2 mov ebp, esp
3 sub esp, X    // X is a number
```

```
1 push ebp
```

The first 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.

```
2 mov ebp, esp
```

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
- **Important:** 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`.

```
3 sub esp, X     // X is a number
```

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.

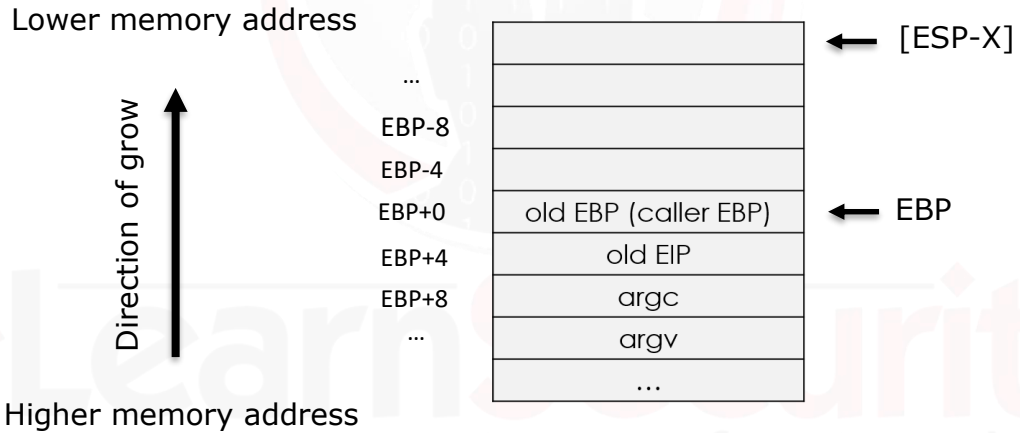
- Similar to the previous instruction, `X` is the source and `esp` is the destination. Therefore, the instruction subtracts X from `esp` (this `X` is not the `int` variable `X` from the program).

Let’s inspect the last instruction in more detail.

The third instruction creates enough space in the stack to copy local variables. Variables are allocated by decreasing the stack pointer (top of the stack) by the amount of space required.

Remember that the stack grows backward. Therefore, we have to decrease its value to expand the stack frame.

The following image represents the Stack once the prologue has happened:



Notice that since the main function contains other variables, and a function call, the actual stack frame for the `main()` subroutine is slightly bigger.

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

The variable is a hexadecimal value that is an offset from the base pointer (**EBP**) or the stack pointer (**ESP**).

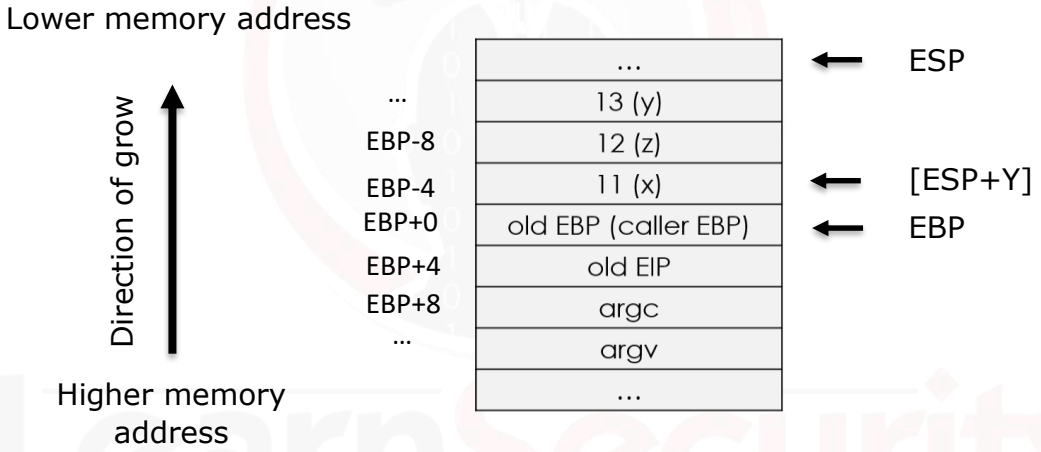
1.2.4.5 Prologue

The instructions after the prologue are like the following:

```
MOV DWORD PTR SS:[ESP+Y],0B
```

This instruction means: move the value `0B` (hexadecimal of `11` - the first local variable) into the memory address location pointed at `ESP+Y`. Note that `Y` is a number and `ESP+Y` points to a memory address between **EBP** and **ESP**.

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()`.

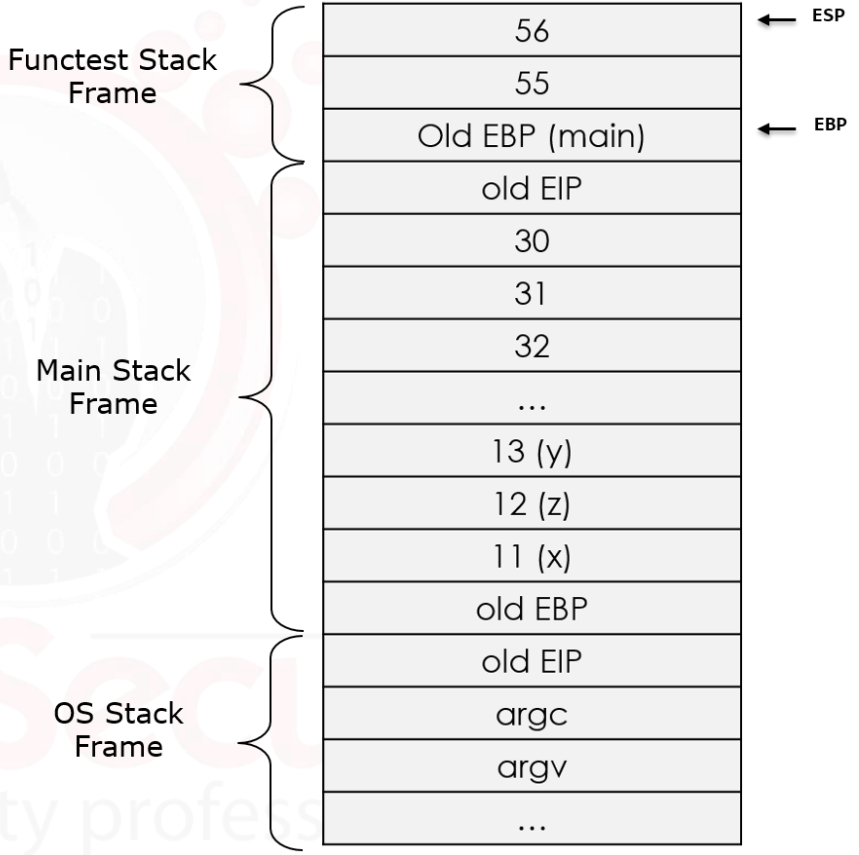
The whole process will be executed again. This time a new stack frame will be created for the function `functest()`.

1.2.4.5 Prologue

The process looks like the following:

- **PUSH** the function parameters in the stack.
- Call the function `functest()`.
- Execute the prologue (which will update **EBP** and **ESP** to create the new stack frame).
- Allocate local variables onto the stack.

The following is how the stack looks like at the end of the entire process.



So far we have seen only half of the process: how the stack frames are created. Now, we have to understand how they are destroyed.

What happens when the code executes a return statement, and the control goes back to the previous procedure (and stack frame)?

1.2.4.6 Epilogue

When the program enters a function, the **prologue** is executed to create the new stack frame.

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

1.2.4.6 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 by **POP**ping the base pointer from the stack.
- Returns to the caller by **POP**ping the instruction pointer from the stack (stored in the stack) and then it jumps to it.

1.2.4.6 Epilogue

The following code represents the epilogue:

```
leave
ret
```

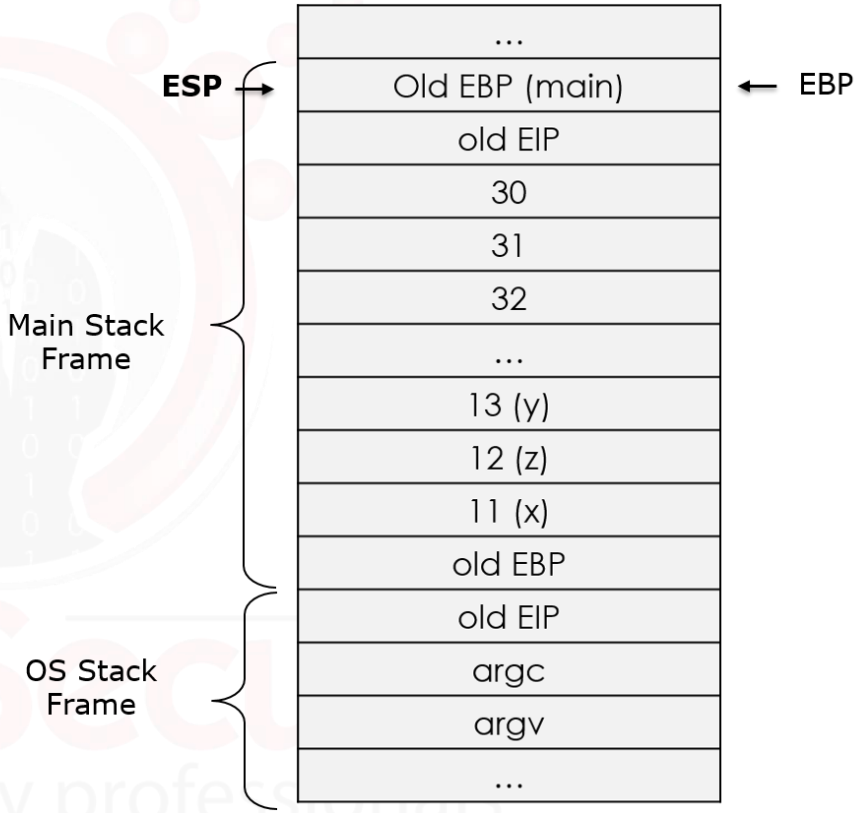
The instructions can also be written as follows:

```
mov esp, ebp
pop ebp
ret
```

1.2.4.6 Epilogue

Here is what happens to the previous stack when the function `functest()` ends.

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



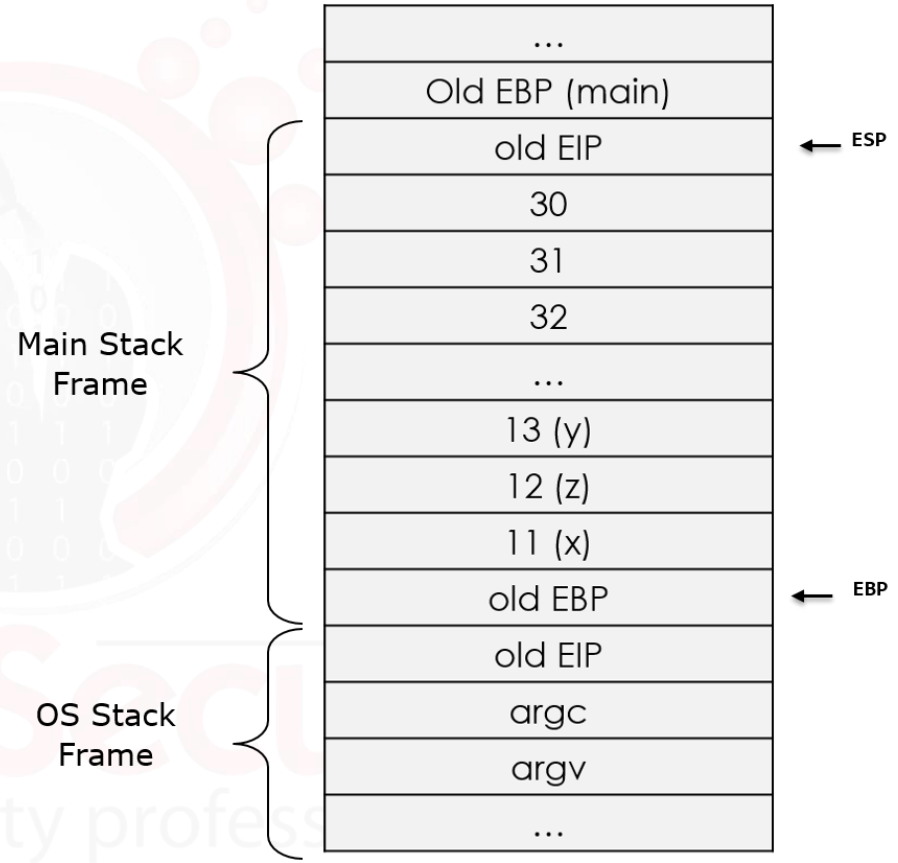
1.2.4.6 Epilogue

The first instruction in the epilogue is `mov esp, ebp`. After it gets executed, both **ESP** and **EBP** point to the same location.

The next instruction is `pop ebp`, which simply **POPS** 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.

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

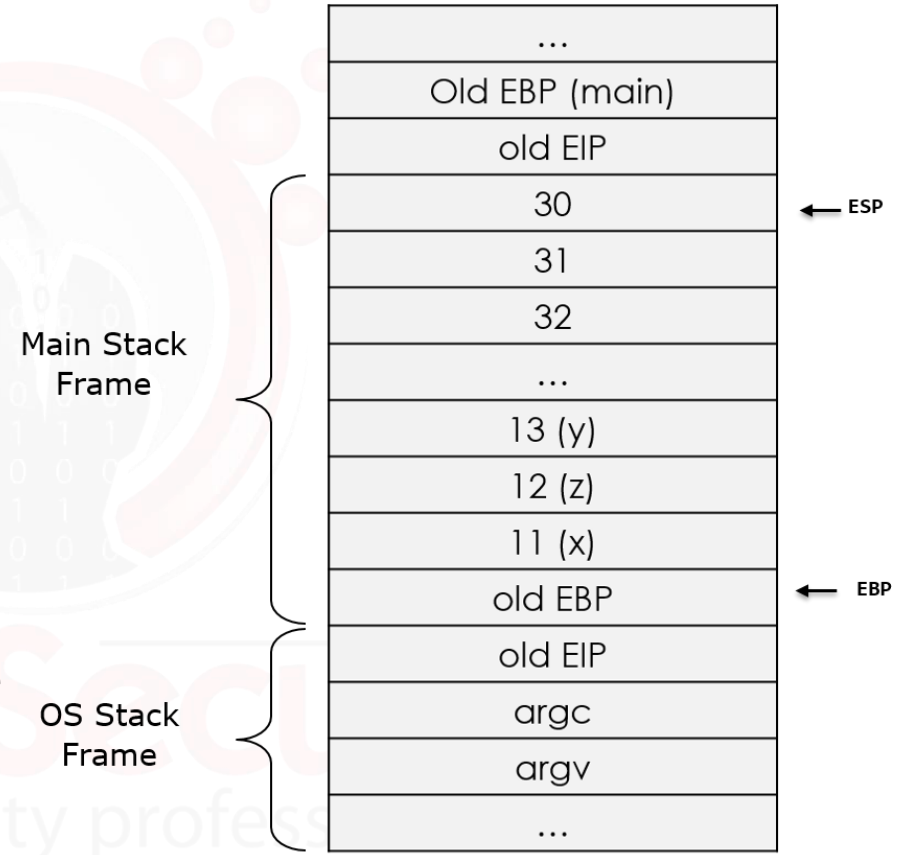
Therefore, **ESP** now points to the old **EIP** previously stored.



1.2.4.6 Epilogue

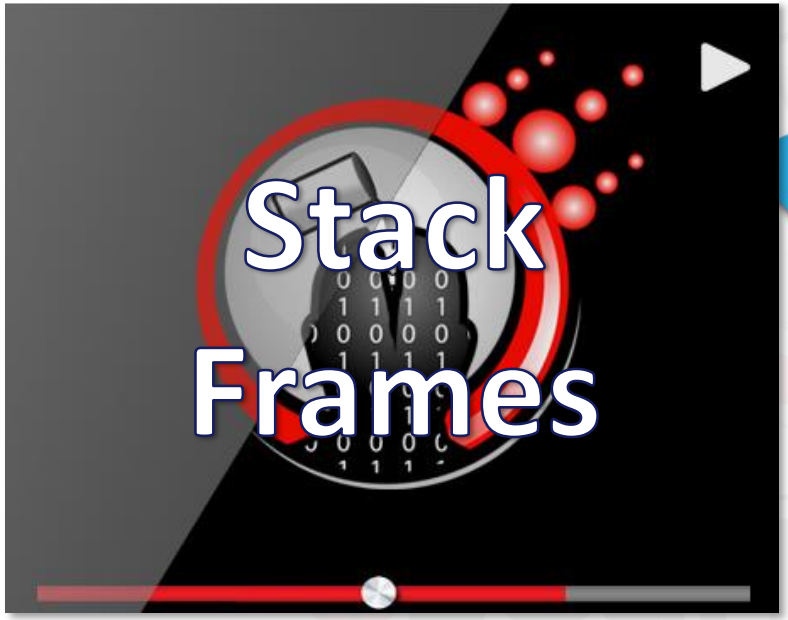
The last instruction that the epilogue will execute is ret.

RET pops the value contained at the top of the stack to the old **EIP** – the next instruction after the caller, and jumps to that location. This gives control back to the caller. **RET** affects only the **EIP** and the **ESP** registers.



At this point, everything is restored correctly, and the program can continue with the next instruction. Although all this information may seem overwhelming right now, in the following video we will see the entire process in a debugger.

This will help you to understand how stack and registers work.



If you have a FULL or ELITE plan you can click on the image on the left to start the video

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

Even though there are three types of endianness, we will explain only two of them, the most important ones: big-endian and little-endian.

1.2.5 Endianness

First, it is important to know these two concepts:

- The most significant bit (**MSB**) in a binary number is the largest value, usually the first from the left. So, for example, considering the binary number 100 the **MSB** is 1.

- The least significant bit (**LSB**) in a binary number is the lowest values, usually the first from the right. So, for example, considering the binary number 110 the **LSB** is 0.

1.2.5 Endianness

In the 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: the `0x12345678` value is represented as:

	Address in memory	Byte value
Highest memory		
	+0	0x12
	+1	0x34
	+2	0x56
Lowest memory	+3	0x78

Respectively, in the little-endian representation, the least significant byte (**LSB**) is stored at the lower memory address, while the most significant byte is at the highest memory address.
Example: `0x12345678` is represented in memory as:

	Address in memory	Byte value
Highest memory		
	+0	0x78
	+1	0x56
	+2	0x34
Lowest memory	+3	0x12

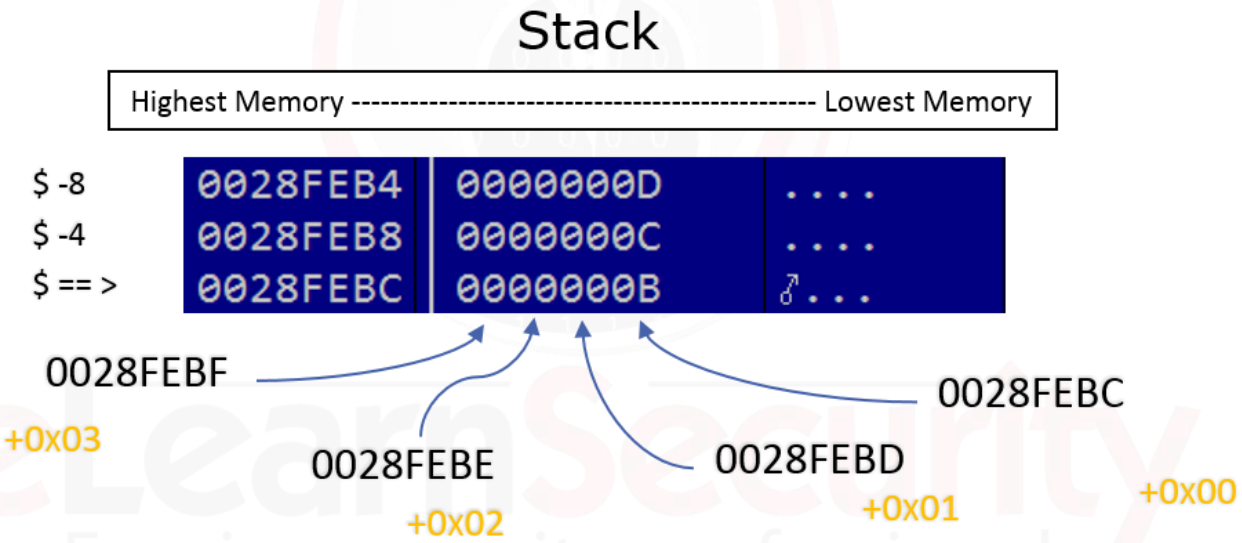
Here's another example. Let us consider the value 11 (`0B` in hexadecimal).

The example system is using little-endian representation; therefore, the **LSB** is stored in the lower memory address, or **MSB** is stored at the highest memory address.

Using the previous table, we will have the following

Highest memory	Address in memory	Byte value
	+0	0x0B
	+1	0x00
	+2	0x00
Lowest memory	+3	0x00

Accordingly, the value will be represented on the stack like the following image.



Remember that the most significant byte is stored at the highest memory address and since the stack grows backward (towards lower addresses), the most significant byte (`0B` in this case) will be stored on the "*left*" (`0028FEBF` – the highest memory address).

You will see later on that knowing the difference between little-endian and big-endian is important to write our payloads correctly and successfully exploit Buffer Overflows vulnerabilities.

Another important topic is the No Operation instruction (**NOP)**.

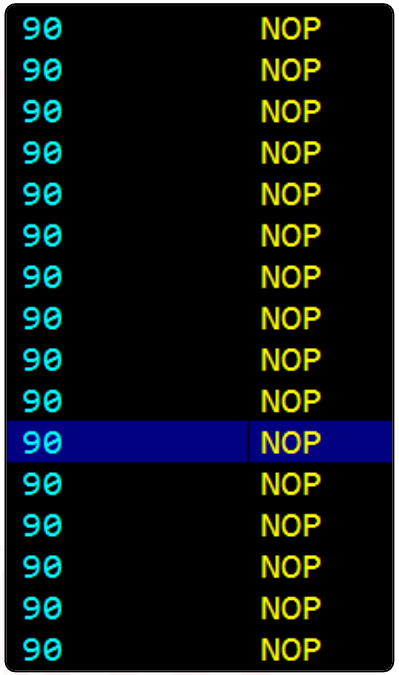
NOP is an assembly language instruction that does nothing. When the program encounters a **NOP**, it will simply skip to the next instruction. In Intel x86 CPUs, **NOP** instructions are represented with the hexadecimal value `0x90`.

NOP-sled is a 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.

The reason is because Buffer Overflows have to match a specific size and location that the program is expecting.

The following image represents what is called **NOP**-sled.

We will cover this in more detail in the next modules.



Security Implementations

1.3 Security Implementations

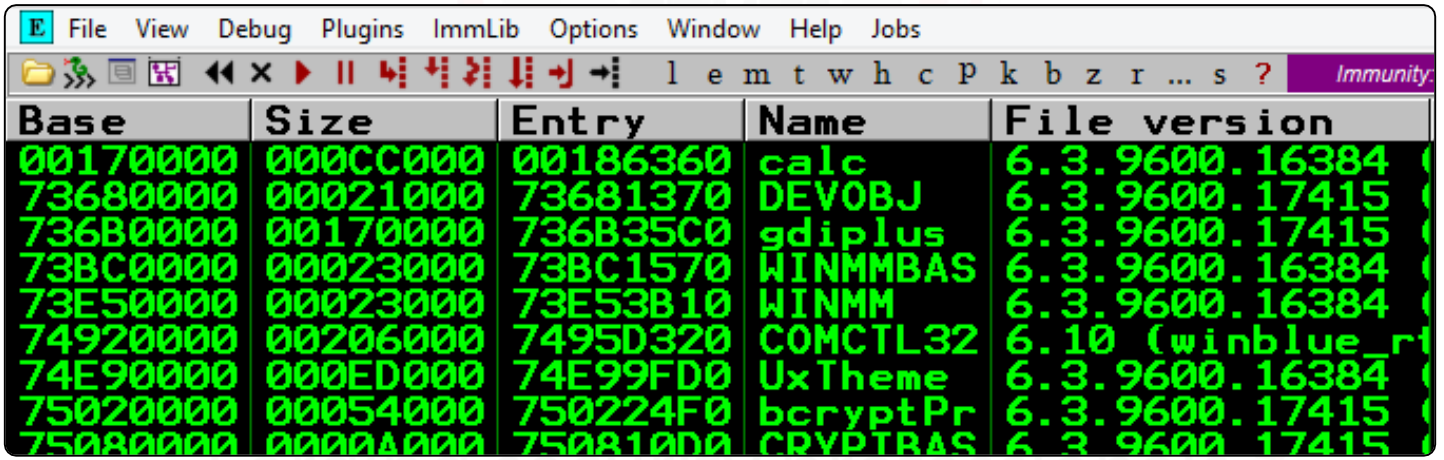
Here is an overview of the security implementations that have been developed during the past several years to prevent, or impede, the exploitation of vulnerabilities such as Buffer Overflow.

- Address Space Layout Randomization (**ASLR**)
- Data Execution Prevention (**DEP**)
- Stack Cookies (Canary)

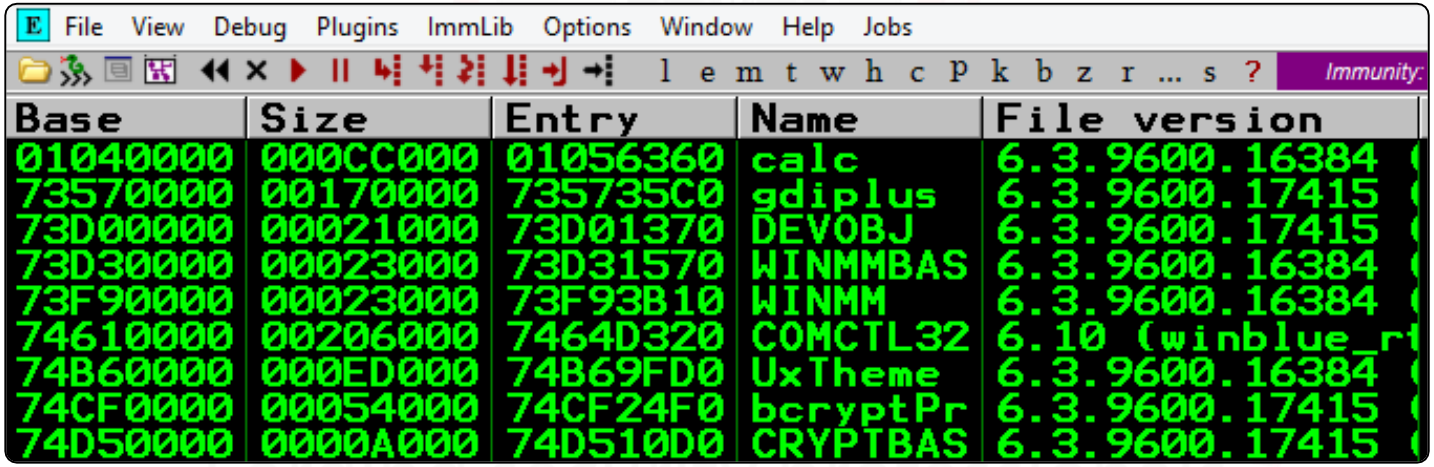
The goal of Address Space Layout Randomization (**ASLR**) is to introduce randomness for executables, libraries, and stacks in the memory address space; this makes it more difficult for an attacker to predict memory addresses and causes exploits to fail and crash the process.

When **ASLR** is activated, the OS loads the same executable at different locations in memory every time.

Here is an example. The application calculator (`calc.exe`) is opened in the debugger and has the base address of `00170000`.



After rebooting the machine and reloading calculator (`calc.exe`) again, we can see that the base address has changed to: `01040000`.



Base	Size	Entry	Name	File version
01040000	000CC000	01056360	calc	6.3.9600.16384
73570000	00170000	735735C0	gdiplus	6.3.9600.17415
73D00000	00021000	73D01370	DEVOBJ	6.3.9600.17415
73D30000	00023000	73D31570	WINMMBAS	6.3.9600.16384
73F90000	00023000	73F93B10	WINMM	6.3.9600.16384
74610000	00206000	7464D320	COMCTL32	6.10 (winblue_r
74B60000	000ED000	74B69FD0	UxTheme	6.3.9600.16384
74CF0000	00054000	74CF24F0	bcryptPr	6.3.9600.17415
74D50000	0000A000	74D510D0	CRYPTBAS	6.3.9600.17415

This happens because of **ASLR**. Once we dig deeper into Buffer Overflow vulnerabilities, we will see why **ASLR** may be a problem to successful exploitation.

There will be an exhaustive explanation of debugging tools in the next modules, so don't worry if you don't understand these screenshots.

It is important to note that **ASLR** is not enabled for all modules.

This means that, even if a process has **ASLR** enabled, there could be a *DLL* in the address space without this protection which could make the process vulnerable to the **ASLR** bypass attack.

Software: To verify the status of ASLR on different programs, download the Process Explorer from here, and verify yourself. In our system, we can see that not all the processes use **ASLR**:

explorer.exe	0.04	46,600 K	98,764 K	2100 Windows Explorer	ASLR
ImmunityDebugger.exe	0.09	18,688 K	31,376 K	1276 Immunity Debugger, 3...	
calc.exe		1,112 K	5,404 K	1516 Windows Calculator	ASLR
vmtoolsd.exe	0.13	9,692 K	14,084 K	1964 VMware Tools Core S...	ASLR
chrome.exe	0.02	31,340 K	68,104 K	1228 Google Chrome	ASLR
chrome.exe		1,348 K	4,504 K	1288 Google Chrome	ASLR
chrome.exe	0.01	44,948 K	52,824 K	3036 Google Chrome	ASLR
chrome.exe	0.10	34,416 K	40,132 K	1004 Google Chrome	ASLR
procexp.exe		2,204 K	7,544 K	1968 Sysinternals Process ...	ASLR
procexp64.exe	0.76	14,988 K	35,660 K	1920 Sysinternals Process ...	ASLR
devcpp.exe		4,228 K	16,508 K	2440 Dev-C++ IDE	

https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

Windows provides another tool that helps solve the problem of exploitation, the **Enhanced Mitigation Experience Toolkit** (EMET).

It provides users with the ability to deploy security mitigation technologies to all applications.

https://blogs.technet.microsoft.com/srd/2010/09/02/the-enhanced-mitigation-experience-toolkit-2-0-is-now-available/

Data Execution Prevention (**DEP**) is a defensive hardware and software measure that prevents the execution of code from pages in memory that are not explicitly marked as executable. The code injected into the memory cannot be run from that region; this makes buffer overflow exploitations even harder.

If you want to read more about DEP, here is a good resource.

https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

The canary, or stack cookie, is a security implementation that places a value next to the return address on the stack.

The function prologue loads a value into this location, while the epilogue makes sure that the value is intact. As a result, when the epilogue runs, it checks that the value is still there and that it is correct.

If it is not, a buffer overflow has probably taken place. This is because a buffer overflow usually overwrites data in the stack.

We will talk again about security implementations once we study how buffer overflows work.

References

References

NASM
https://www.nasm.us/

MASM
https://www.microsoft.com/en-us/download/details.aspx?id=12654

Brk
http://man7.org/linux/man-pages/man2/brk.2.html

Malloc
http://man7.org/linux/man-pages/man3/malloc.3.html

Ralloc
http://man7.org/linux/man-pages/man3/realloc.3p.html

Process Explorer
https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

EMET
https://blogs.technet.microsoft.com/srd/2010/09/02/the-enhanced-mitigation-experience-toolkit-2-0-is-now-available/

DEP
https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in

Stack Frames