Course Brief

Monday, January 23, 2023 9:01 AM

Tool list

https://x64dbg.com/
https://sbytestream.pythonanywhere.com/software/peinfo
http://www.rohitab.com/apimonitor
https://www.virtualbox.org/

In VM

Win 8 32bit
Ubuntu 22.04 64bit

Motives of Reverse engineering:

1. To look into skeletons and research on Zero day exploits
2. To reverse engineer Antivirus and design payload to avoid detection
3. To update firewalls according to malwares
4. To find the roots
5. To find how it works
6. To understand code/software psychology

Reverse Engineering Training Handbook

Prepared by: Arishti Security

Authors: Ravi Rajput, Adhokshaj Mishra, Animesh Roy

Version: 1.0.0

Website: arishtisecurity.com

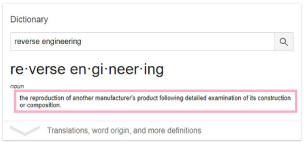
Email: info@arishtisecurity.com

Training Inquiry: training@arishtisecurity.com, (+91) 99 55 00 1636

1. Introduction to Reverse Engineering

1.1 What is Reverse Engineering and Why do we need it?

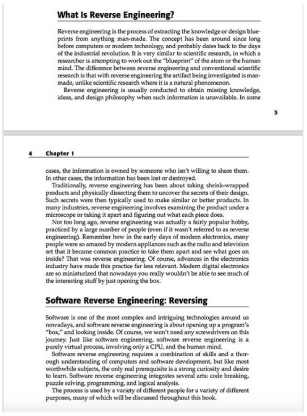
Let us Google the definition:



(Thanks Google.)

Reverse engineering in this context refers to the process of copying another person's source code after carefully analysing the executable machine code. In other words, by looking at the instructions a programme "gives" to the CPU, we may deduce what a programme performs.

Here's some more information on Reverse Engineering in general and Software RE in particular (taken from Reversing: Secrets of Reverse Engineering by Eldad Eilam):



What Is Reverse Engineering?

Reverse engineering is the process of extracting the knowledge or design blueprints from anything man-made. The concept has been around since long before computers or modern technology, and probably dates back to the days of the industrial revolution. It is very similar to scientific research, in which a researcher is attempting to work out the "blueprint" of the atom or the human mind. The difference between reverse engineering and conventional scientific research is that with reverse engineering the artifact being investigated is man-made, unlike scientific research where it is a natural phenomenon.

Reverse engineering is usually conducted to obtain missing knowledge, ideas, and design philosophy when such information is unavailable. In some

3

4 Chapter 1

cases, the information is owned by someone who isn't willing to share them. In other cases, the information has been lost or destroyed.

Traditionally, reverse engineering has been about taking shrink-wrapped products and physically dissecting them to uncover the secrets of their design. Such secrets were then typically used to make similar or better products. In many industries, reverse engineering involves examining the product under a microscope or taking it apart and figuring out what each piece does.

Not too long ago, reverse engineering was actually a fairly popular hobby, practiced by a large number of people (even if it wasn't referred to as reverse engineering). Remember how in the early days of modern electronics, many people were so amazed by modern appliances such as the radio and television set that it became common practice to take them apart and see what goes on inside? That was reverse engineering. Of course, advances in the electronics industry have made this practice far less relevant. Modern digital electronics are so miniaturized that nowadays you really wouldn't be able to see much of the interesting stuff by just opening the box.

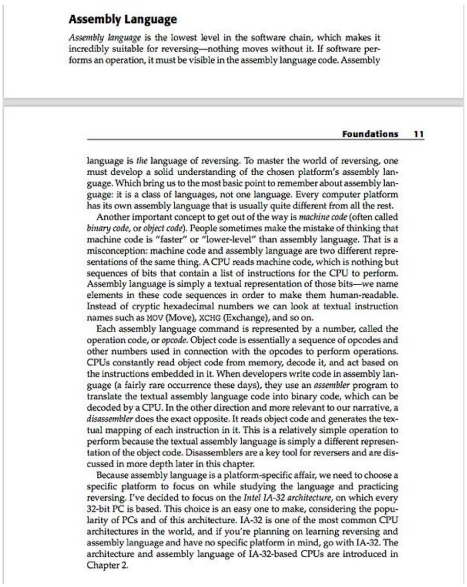
Software Reverse Engineering: Reversing

Software is one of the most complex and intriguing technologies around us nowadays, and software reverse engineering is about opening up a program's "box," and looking inside. Of course, we won't need any screwdrivers on this journey. Just like software engineering, software reverse engineering is a purely virtual process, involving only a CPU, and the human mind.

Software reverse engineering requires a combination of skills and a thorough understanding of computers and software development, but like most worthwhile subjects, the only real prerequisite is a strong curiosity and desire to learn. Software reverse engineering integrates several arts: code breaking, puzzle solving, programming, and logical analysis.

The process is used by a variety of different people for a variety of different purposes, many of which will be discussed throughout this book.

However, there are already several fairly good introductions to Assembly language online, so I would be delighted to give you one. So, here is another passage from Secrets of Reversing:



Assembly Language

Assembly language is the lowest level in the software chain, which makes it incredibly suitable for reversing—nothing moves without it. If software performs an operation, it must be visible in the assembly language code. Assembly

language is *the* language of reversing. To master the world of reversing, one must develop a solid understanding of the chosen platform's assembly language. Which bring us to the most basic point to remember about assembly language: it is a class of languages, not one language. Every computer platform has its own assembly language that is usually quite different from all the rest.

Another important concept to get out of the way is *machine code* (often called *binary code*, or *object code*). People sometimes make the mistake of thinking that machine code is "faster" or "lower-level" than assembly language. That is a misconception: machine code and assembly language are two different representations of the same thing. A CPU reads machine code, which is nothing but sequences of bits that contain a list of instructions for the CPU to perform. Assembly language is simply a textual representation of those bits—we name elements in these code sequences in order to make them human-readable. Instead of cryptic hexadecimal numbers we can look at textual instruction names such as `MOV` (Move), `XCHG` (Exchange), and so on.

Each assembly language command is represented by a number, called the operation code, or *opcode*. Object code is essentially a sequence of opcodes and other numbers used in connection with the opcodes to perform operations. CPUs constantly read object code from memory, decode it, and act based on the instructions embedded in it. When developers write code in assembly language (a fairly rare occurrence these days), they use an *assembler* program to translate the textual assembly language code into binary code, which can be decoded by a CPU. In the other direction and more relevant to our narrative, a *disassembler* does the exact opposite. It reads object code and generates the textual mapping of each instruction in it. This is a relatively simple operation to perform because the textual assembly language is simply a different representation of the object code. Disassemblers are a key tool for reversers and are discussed in more depth later in this chapter.

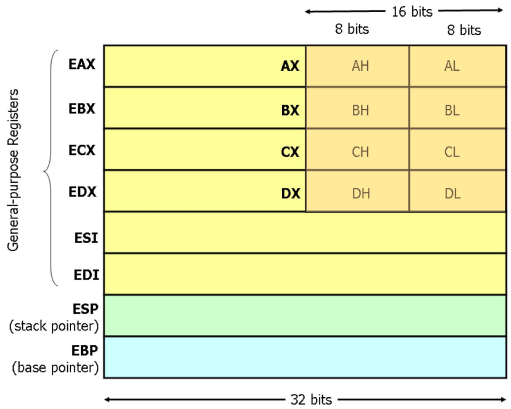
Because assembly language is a platform-specific affair, we need to choose a specific platform to focus on while studying the language and practicing reversing. I've decided to focus on the *Intel IA-32 architecture*, on which every 32-bit PC is based. This choice is an easy one to make, considering the popularity of PCs and of this architecture. IA-32 is one of the most common CPU architectures in the world, and if you're planning on learning reversing and assembly language and have no specific platform in mind, go with IA-32. The architecture and assembly language of IA-32-based CPUs are introduced in Chapter 2.

1.2 x86 Assembly Guide

We do not cover the entire x86 instruction set in this handbook because it is a vast and complicated collection (Intel's x86 instruction set manuals are over 2900 pages long). For instance, the x86 instruction set has a 16-bit subset. The 16-bit programming paradigm may be quite difficult to use. The memory model is segmented, there are additional limitations on register usage, etc. In this manual, we will focus on x86 programming's more contemporary features while only covering the instruction set in enough depth to obtain a fundamental understanding of x86 programming.

Registers

Eight 32-bit general purpose registers are present in modern (i.e., 386 and later) x86 processors, as seen in Figure below. Most of the registration names are historical. For instance, ECX was referred to as the counter because it was used to store a loop index, but EAX was known as the accumulator since it was utilised by several arithmetic operations. The stack pointer (ESP) and the base pointer are two registers that are reserved for particular reasons, despite the fact that most of the registers in the current instruction set have lost their special functions (EBP).



Memory and Addressing Modes

You can declare static data regions (analogous to global variables) in x86 assembly using special assembler directives for this purpose. Data declarations should be preceded by the .DATA directive. Following this directive, the directives

- DB → One-byte data locations
- DW → Two-byte data locations
- DD → Four-byte data locations

Declared locations can be labelled with names for later reference — this is like declaring variables by name but abides by some lower-level rules. For example, locations declared in sequence will be in memory next to one another.

Example declarations:

```
.DATA
        var     DB 64    ; Declare a byte, referred to as location var, containing the value 64.
        var2    DB ?     ; Declare an uninitialized byte, referred to as location var2.
                DB 10    ; Declare a byte with no label, containing the value 10. Its location is var2 + 1.
        X       DW ?     ; Declare a 2-byte uninitialized value, referred to as location X.
        Y       DD 30000  ; Declare a 4-byte value, referred to as location Y, initialized to 30000.
```

Arrays in x86 assembly language are only a collection of cells that are close together in memory, unlike arrays in high level languages where they can have numerous dimensions and are accessible by indices. Just listing the values, like in the first example below, might be used to declare an array. The DUP directive and the usage of string literals are two more typical techniques for defining data arrays. With the DUP instruction, the assembler is instructed to repeat an expression a specific number of times. 4 DUP(2), for instance, equals 2, 2, 2, 2.

Some examples:

```
        Z       DD 1, 2, 3      ; Declare three 4-byte values, initialized to 1, 2, and 3. The
                value of location Z + 8 will be 3.

        bytes   DB 10 DUP(?)    ; Declare 10 uninitialized bytes starting at location bytes.

        arr     DD 100 DUP(0)   ; Declare 100 4-byte words starting at location arr, all
                initialized to 0

        str     DB 'hello',0    ; Declare 6 bytes starting at the address str, initialized to the
                ASCII character values for hello and the null (0) byte.
```

Addressing Memory

Memory addresses on contemporary x86-compatible CPUs may span up to 232 bytes, and they are 32 bits wide. When we used labels to refer to memory regions in the cases, the assembler substituted those labels with 32-bit amounts that specified memory locations. The x86 offers a versatile method for calculating and referring to memory addresses in addition to enabling the reference of memory regions by labels (i.e., constant values): up to two of the 32-bit registers and a 32-bit signed constant can be joined together to compute a memory address. Optionally, one of the registers may be pre-multiplied by 2, 4, or 8.

The addressing modes can be used with many x86 instructions (we will describe them in the next section). Here we illustrate some examples using the mov instruction that moves data between registers and memory. This instruction has two operands: the first is the destination and the second specifies the source.

Some examples of mov instructions using address computations are:

```
mov eax, [ebx]        ; Move the 4 bytes in memory at the address contained in EBX into EAX
mov [var], ebx        ; Move the contents of EBX into the 4 bytes at memory address var.
                      (Note, var is a 32-bit constant).
mov eax, [esi-4]      ; Move 4 bytes at memory address ESI + (-4) into EAX
mov [esi+eax], cl     ; Move the contents of CL into the byte at address ESI+EAX
mov edx,
[esi+4*ebx]           ; Move the 4 bytes of data at address ESI+4*EBX into EDX
```

Some examples of invalid address calculations include:

```
mov eax, [ebx-ecx]         ; Can only add register values
mov [eax+esi+edi], ebx     ; At most 2 registers in address computation
```

Size Directives

In general, the assembly code instruction in which the data item is addressed will reveal the intended size of the data item at a specific memory address. For each of the instructions, the size of the register operand might be used to estimate the size of the memory regions. The assembler deduced that the area of memory we were referring to was 4 bytes wide when we were loading a 32-bit register. The assembler may guess that we intended the address to point to a single byte in memory while we were putting the value of a one-byte register to memory.

The extent of a referred-to memory area is often unclear, though. Think about the command mov [ebx], 2. Should this instruction insert the value 2 at address EBX into a single byte? The 32-bit integer representation of 2 may be moved into the 4-byte region beginning at address EBX. The assembler needs to be specifically instructed as to which is the proper interpretation because both are legitimate possibilities. This is accomplished by using the size directives BYTE PTR, WORD PTR, and DWORD PTR, which denote sizes of 1, 2, and 4 bytes, respectively.

For example:

```
mov BYTE PTR [ebx], 2  ; Move 2 into the single byte at the address stored in EBX.


mov WORD PTR [ebx], 2  ; Move the 16-bit integer representation of 2 into the 2 bytes
starting at the address in EBX.


mov DWORD PTR [ebx], 2 ; Move the 32-bit integer representation of 2 into the 4 bytes
starting at the address in EBX.
```

Instructions

Machine instructions generally fall into three categories: data movement, arithmetic/logic, and control-flow. In this section, we will look at important examples of x86 instructions from each category. This section should not be considered an exhaustive list of x86 instructions, but rather a useful subset. For a complete list, see Intel's instruction set reference.
We use the following notation:

```
<reg32>         Any 32-bit register (EAX, EBX, ECX, EDX, ESI, EDI, ESP, or EBP)
<reg16>         Any 16-bit register (AX, BX, CX, or DX)
<reg8>          Any 8-bit register (AH, BH, CH, DH, AL, BL, CL, or DL)
<reg>           Any register
<mem>           A memory address (e.g., [eax], [var + 4], or dword ptr [eax+ebx])
<con32>         Any 32-bit constant
<con16>         Any 16-bit constant
<con8>          Any 8-bit constant
<con>           Any 8-, 16-, or 32-bit constant
```

Data Movement Instructions

mov — *Move (Opcodes: 88, 89, 8A, 8B, 8C, 8E, ...)*

The mov instruction copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory). While register-to-register moves are possible, direct memory-to-memory moves are not. In cases where memory transfers are desired, the source memory contents must first be loaded into a register, then can be stored to the destination memory address.

Syntax

```
mov <reg>,<reg>

mov <reg>,<mem>

mov <mem>,<reg>

mov <reg>,<const>

mov <mem>,<const>
```

Examples

```
mov eax, ebx – copy the value in ebx into eax

mov byte ptr [var], 5 – store the value 5 into the byte at location var
```

push — Push stack (Opcodes: FF, 89, 8A, 8B, 8C, 8E, ...)

The push instruction places its operand onto the top of the hardware supported stack in memory. Specifically, push first decrements ESP by 4, then places its operand into the contents of the 32-bit location at address [ESP]. ESP (the stack pointer) is decremented by push since the x86 stack grows down - i.e. the stack grows from high addresses to lower addresses.

Syntax

```
push <reg32>
push <mem>
push <con32>
```

Examples

```
push eax — push eax on the stack
push [var] — push the 4 bytes at address var onto the stack
```

pop — Pop stack

The pop instruction removes the 4-byte data element from the top of the hardware-supported stack into the specified operand (i.e. register or memory location). It first moves the 4 bytes located at memory location [SP] into the specified register or memory location, and then increments SP by 4.

Syntax

```
pop <reg32>
pop <mem>
```

Examples

```
pop edi — pop the top element of the stack into EDI.
pop [ebx] — pop the top element of the stack into memory at the four bytes starting at
location EBX.
```

lea — Load effective address

The lea instruction places the address specified by its second operand into the register specified by its first operand. Note, the contents of the memory location are not loaded, only the effective address is computed and placed into the register. This is useful for obtaining a pointer into a memory region.

Syntax

```
lea <reg32>,<mem>
```

Examples

```
lea edi, [ebx+4*esi] — the quantity EBX+4*ESI is placed in EDI.
lea eax, [var] — the value in var is placed in EAX.
lea eax, [val] — the value val is placed in EAX.
```

Arithmetic and Logic Instructions

add — Integer Addition

The add instruction adds together its two operands, storing the result in its first operand. Note, whereas both operands may be registers, at most one operand may be a memory location.

Syntax

```
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
```

Examples

```
add eax, 10 – EAX ← EAX + 10
add BYTE PTR [var], 10 – add 10 to the single byte stored at memory address var
```

sub — Integer Subtraction

The sub instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand. As with add

Syntax

```
sub <reg>,<reg>
sub <reg>,<mem>
sub <mem>,<reg>
sub <reg>,<con>
sub <mem>,<con>
```

Examples

```
sub al, ah – AL ← AL – AH
sub eax, 216 – subtract 216 from the value stored in EAX
```

inc, dec — Increment, Decrement

The inc instruction increments the contents of its operand by one. The dec instruction decrements the contents of its operand by one.

Syntax

```
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

Examples

```
dec eax – subtract one from the contents of EAX.

inc DWORD PTR [var] – add one to the 32-bit integer stored at location var
```

imul — Integer Multiplication

The imul instruction has two basic formats: two-operand (first two syntax listings above) and three-operand (last two syntax listings above).

The two-operand form multiplies its two operands together and stores the result in the first operand. The result (i.e., first) operand must be a register.

The three-operand form multiplies its second and third operands together and stores the result in its first operand. Again, the result operand must be a register. Furthermore, the third operand is restricted to being a constant value.

Syntax

```
imul <reg32>,<reg32>

imul <reg32>,<mem>

imul <reg32>,<reg32>,<con>

imul <reg32>,<mem>,<con>
```

Examples

```
imul eax, [var] – multiply the contents of EAX by the 32-bit contents of the
memory location var. Store the result in EAX.

imul esi, edi, 25 – ESI → EDI * 25
```

idiv — Integer Division

The idiv instruction divides the contents of the 64 bit integer EDX:EAX (constructed by viewing EDX as the most significant four bytes and EAX as the least significant four bytes) by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX.

Syntax

```
idiv <reg32>

idiv <mem>
```

Examples

```
idiv ebx – divide the contents of EDX:EAX by the contents of EBX. Place the quotient in
EAX and the remainder in EDX.

idiv DWORD PTR [var] – divide the contents of EDX:EAX by the 32-bit value stored at
memory location var. Place the quotient in EAX and the remainder in EDX.
```

and, or, xor — Bitwise logical and, or and exclusive or

These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location.

Syntax

```
and <reg>,<reg>
and <reg>,<mem>
and <mem>,<reg>
and <reg>,<con>
and <mem>,<con>
or <reg>,<reg>
or <reg>,<mem>
or <mem>,<reg>
or <reg>,<con>
or <mem>,<con>
xor <reg>,<reg>
xor <reg>,<mem>
xor <mem>,<reg>
xor <reg>,<con>
xor <mem>,<con>
```

Examples

```
and eax, 0FH – clear all but the last 4 bits of EAX.
xor edx, edx – set the contents of EDX to zero.
```

not — Bitwise Logical Not

Logically negates the operand contents (that is, flips all bit values in the operand).

Syntax

```
not <reg>
not <mem>
```

Example

```
not BYTE PTR [var] – negate all bits in the byte at the memory location var.
```

neg — Negate

Performs the two's complement negation of the operand contents.

Syntax

```
neg <reg>
neg <mem>
```

Example

```
neg eax – EAX → – EAX
```

shl, shr — Shift Left, Shift Right

These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros. The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL. In either case, shifts count of greater than 31 are performed modulo 32.

Syntax

```
shl <reg>,<con8>
shl <mem>,<con8>
shl <reg>,<cl>
shl <mem>,<cl>

shr <reg>,<con8>
shr <mem>,<con8>
shr <reg>,<cl>
shr <mem>,<cl>
```

Examples

```
shl eax, 1 – Multiply the value of EAX by 2 (if the most significant bit is 0)
shr ebx, cl – Store in EBX the floor of result of dividing the value of EBX by
2n wheren is the value in CL.
```

Control Flow Instructions

The x86 processor maintains an instruction pointer (IP) register that is a 32-bit value indicating the location in memory where the current instruction starts. Normally, it increments to point to the next instruction in memory begins after execution an instruction. The IP register cannot be manipulated directly but is updated implicitly by provided control flow instructions.

We use the notation <label> to refer to labelled locations in the program text. Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon. For example,

```
    mov esi, [ebp+8]
begin: xor ecx, ecx
    mov eax, [esi]
```

The second instruction in this code fragment is labeled begin. Elsewhere in the code, we can refer to the memory location that this instruction is located at in memory using the more convenient symbolic name begin. This label is just a convenient way of expressing the location instead of its 32-bit value.

jmp — Jump

Transfers program control flow to the instruction at the memory location indicated by the operand.

Syntax

```
jmp <label>
```

Example

```
jmp begin – Jump to the instruction labeled begin.
```

jcondition — Conditional Jump

These instructions are conditional jumps that are based on the status of a set of condition codes that are stored in a special register called the machine status word. The contents of the machine status word include information about the last arithmetic operation performed. For example, one bit of this word indicates if the last result was zero. Another indicates if the last result was negative. Based on these condition codes, several conditional jumps can be performed. For example, the jz instruction performs a jump to the specified operand label if the result of the last arithmetic operation was zero. Otherwise, control proceeds to the next instruction in sequence.

Several the conditional branches are given names that are intuitively based on the last operation performed being a special compare instruction, cmp (see below). For example, conditional branches such as jle and jne are based on first performing a cmp operation on the desired operands.

Syntax

```
je <label> (jump when equal)
jne <label> (jump when not equal)
jz <label> (jump when last result was zero)
jg <label> (jump when greater than)
jge <label> (jump when greater than or equal to)
jl <label> (jump when less than)
jle <label> (jump when less than or equal to)
```

Example

```
cmp eax, ebx
jle done
```

If the contents of EAX are less than or equal to the contents of EBX, jump to the label done. Otherwise, continue to the next instruction.

cmp — Compare

Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand.

Syntax

```
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

Example

```
cmp DWORD PTR [var], 10
jeq loop
```

If the 4 bytes stored at location var are equal to the 4-byte integer constant 10, jump to the location labeled loop.

call, ret — Subroutine call and return

These instructions implement a subroutine call and return. The call instruction first pushes the current code location onto the hardware supported stack in memory (see the push instruction for details), and then performs an unconditional jump to the code location indicated by the label operand. Unlike the simple jump instructions, the call instruction saves the location to return to when the subroutine completes.

The ret instruction implements a subroutine return mechanism. This instruction first pops a code location off the hardware supported in-memory stack (see the pop instruction for details). It then performs an unconditional jump to the retrieved code location.

Syntax

```
call <label>
ret
```

Instruction	Description	Aliases	Flags
jz	jump if zero	je	zf=1
jnz	jump if not zero	jne	zf=0
jl	jump if less	jnge	sf=1
jle	jump if less or equal	jng	zf=1 or sf=1
jg	jump if greater	jnle	zf=0 and sf=0
jge	jump if greater or equal	jnl	sf=0
jc	jump if carry	jb,jnae	cf=1
jnc	jump if not carry	jnb,jae	.

Question Set 1	

1	What is foo in the following example? How much space does it occupy in memory?
	
	`.data`
	`foo DW 1,1,2,3,5`

2	What is the value stored in EAX by the end of this code?
	
	`mov eax,0x2`
	`mov ebx,eax`
	`shl eax,0x2`
	`add eax,ebx`
	`and eax,0x8`

3	What should be the value of EAX at the beginning of the following code, such that by the end of it, EAX = 0? `xor eax, eax`

Calling Convention

Purposes of the Call Stack

1. Storing return addresses

The main reason for having a call stack is to keep track of the point to which each active subroutine should return when it finishes executing. If, for example, a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes. To accomplish this, the address following the call instruction - the return address - is pushed onto the call stack with each call.

2. Local-Data storage

A function frequently needs memory space for storing the values of local variables, the variables that are known only within the function and do not store values after it returns. It is convenient to allocate space for these variables by moving the top of the stack towards the lower addresses, to provide the necessary space.

3. Parameter passing

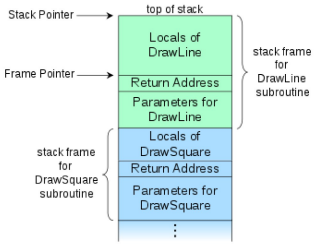
Subroutines often require that parameters values be supplied to them by the code which calls them. The call stack works well as a place for these parameters, especially since each call to a subroutine will be given separate space on the call stack for those values.

Call Stack Structure

A call stack is composed of stack frames - data structures containing subroutine state information. What is state information? Function parameters, local variables, return address and the frame's base address.

Each stack frame corresponds to a call to a subroutine which has not yet returned.

For example, if a subroutine named DrawLine is currently running (and was called by DrawSquare), the top part of the call stack might be laid out like in the picture (very rough, don't use as reference):



The stack frame at the top of the stack is for the currently executing routine. The stack frame usually includes at least the following items (in push order):

- the arguments (parameter values) passed to the routine (if any);
- the return address back to the routine's caller (e.g. in the DrawLine stack frame, an address into DrawSquare's code); and
- space for the local variables of the routine (if any).

Stack & Frame Pointers

The next two paragraphs might be very confusing. If necessary, read them again and again, use a pen and a paper, repeat until you feel comfortable with the presented information. It's important :)

Recall: ESP is the Stack Pointer - it always points to the top of the stack.

When values are pushed onto the stack - it decreases. When values are popped off the stack - it increases.

This register constantly changes.

During a function's execution, we need a way to access the function data - its parameters and local variables. Due to the dynamic nature of ESP - we cannot use it as an anchor. Instead, we take the value of ESP now when the function is called, and we put it in EBP - the Base Pointer. During the function execution, EBP will stay fixed (unlike ESP). Thus, EBP can be used as an anchor to access parameters and variables.

Note that when moving ESP's value into EBP, we override the value already stored in EBP. For this reason, we need to preserve EBP's previous value somehow - this is done by pushing EBP onto the stack right before it receives its new value.

Imagine DrawSquare calls DrawLine. DrawSquare's stack frame will appear first (in higher addresses) on the stack.

1. When DrawLine is called, the current EBP (Base Pointer of DrawSqaure) will be pushed onto the stack.

2. EBP will then be assigned the current value of ESP. From now on, EBP is used as DrawLine's base pointer to access its variables and parameters.

3. Throughout DrawLine, ESP might change many times, but EBP is fixed.

4. When DrawLine terminates, ESP will be be set to the value of EBP. This sets the pointer to the top of the stack to be equal to the base pointer, and practically frees the memory allocated on the stack for the local variables.

5. DrawSquare's EBP will be popped from the stack. From now on, EBP can be used again as DrawSquare's base pointer.

So, in fact, for some function func, func's stack frame contains the EBP value of func's caller, but the EBP register itself is func's base pointer.

Question Set 2	
3	Which register holds the address of the top of the stack?
4	Suppose the function multiply(x, y) creates 2 local variables - tmp1 and tmp2 - during its execution. What instruction do you expect seeing that allocates memory for these variables?
5	In what order will multiply's parameters, x and y, be pushed onto the stack before the call to multiply?
6	How can each of them be accessed during the function execution? Write the exact expressions.

2. Assembly to X86

2.1 Assignment Challenge

The following is a disassembled output of a simple C code snippet. Can you figure out what this code snippet does, and can you translate it back to a pseudocode (high-level language equivalent)? Use all of the concepts that you have learned so far to solve the challenge. The answer to the challenge will be covered in the next section, and we will also look at the original C code snippet after we solve this challenge:

```
mov dword ptr [ebp-4],1  ❶
mov eax,dword ptr [ebp-4]  ❷
mov dword ptr [ebp-8],eax  ❸
```

Theory Behind the Solution

The preceding program copies a value from one memory location to another. At ❶, the program copies a `dword` value `1` into a memory address (specified by `ebp-4`). At ❷, the same value is copied into the `eax` register, which is then copied into a different memory address, `ebp-8`, at ❸.

The disassembled code might be difficult to understand initially, so let me break it down to make it simple. We know that in a high-level language like C, a variable that you define (for example, `int val;`) is just a symbolic name for a memory address (as mentioned previously). Going by that logic, let's identify the memory address references and give them a symbolic name. In the disassembled program, we have two addresses (within square brackets): `ebp-4` and `ebp-8`. Let's label them and give them symbolic names; let's say, `ebp-4 = a` and `ebp-8 = b`. Now, the program should look like the one shown here:

```
mov dword ptr [a],1     ; treat it as mov [a],1
mov eax,dword ptr [a]   ; treat it as mov eax,[a]
mov dword ptr [b],eax   ; treat it as mov [b],eax
```

In a high-level language, when you assign a value to a variable, let's say `val = 1`, the value `1` is moved into the address represented by the `val` variable. In assembly, this can be represented as `mov [val], 1`. In other words, `val = 1` in a high-level language is the same as `mov [val],1` in assembly. Using this logic, the preceding program can be written into a high-level language equivalent:

```
a = 1
eax = a
b = eax ❹
```

Recall that, the registers are used by the CPU for temporary storage. So, let's replace all of the register names with their values on the right-hand side of the = sign (for example, replace `eax` with its value, `a`, at ❹). The resultant code is shown here:

```
a = 1
eax = a ❺
b = a
```

Solution

In the preceding program, the `eax` register is used to temporarily hold the value of `a`, so we can remove the entry at ❺ (that is remove the entry containing registers on the left side of the = sign). We are now left with the simplified code shown here:

```
a = 1
b = a
```

Data Type Determination

In high-level languages, variables have data types. Let's try to determine the data types of these variables: `a` and `b`. Sometimes, it is possible to determine the data type by understanding how the variables are accessed and used. From the disassembled code, we know that the `dword` value (4 bytes) `1` was moved into the variable `a`, which was then copied to `b`. Now that we know these variables are 4 bytes in size, it means that they could be of the type `int`, `float`, or `pointer`. To determine the exact data type, let's consider the following.

The variables `a` and `b` cannot be `float`, because, from the disassembled code, we know that `eax` was involved in the data transfer operation. If it was a floating point value, the floating point registers would have been used, instead of using a general purpose register such as `eax`.

The variables `a` and `b` cannot be a `pointer` in this case, because the value `1` is not a valid address. So, we can guess that `a` and `b` should be of the type `int`.

Based on these observations, we can now rewrite the program as follows:

```
int a;
int b;

a = 1;
b = a;
```

Now that we have solved the challenge, let's look at the original C code snippet of the disassembled output. The original C code snippet is shown as follows. Compare it with what we determined. Notice how it was possible to build a program similar to the original program (it is not always possible to get the exact C program back), and also, it's now much easier to determine the functionality of the program:

```
int x = 1;
int y;
y = x;
```

If you are disassembling a bigger program, it would be hard to label all of the memory addresses. Typically, you will use the features of the disassembler or debugger to rename memory addresses and to perform code analysis. You will learn the features offered by the disassembler and how to use it for code analysis in the next chapter. When you are dealing with bigger programs, it is a good idea to break the program into small blocks of code, translate it into some high-level language that you are familiar with, and then do the same thing for the rest of the blocks.

2.2 Arithmetic Challenge

Theory

You can perform addition, subtraction, multiplication, and division in assembly language. A addition and subtraction are performed using the `add` and `sub` instructions, respectively. These instructions take two operands: *destination* and *source*. The `add` instruction adds the source and destination and stores the result in the destination. The `sub` instruction subtracts the source from the destination operand, and the result is stored in the destination. These instructions set or clear flags in the `eflags` register, based on the operation. These flags can be used in the conditional statements. The `sub` instruction sets the zero flag, `(zf)`, if the result is zero, and the carry flag, `(cf)`, if the destination value is less than the source. The following outlines a few variations of these instructions:

```
add eax,42      ; same as eax = eax+42
add eax,ebx     ; same as eax = eax+ebx
add [ebx],42    ; adds 42 to the value in address specified by ebx
sub eax, 64h    ; subtracts hex value 0x64 from eax, same as eax = eax-0x64
```

There is a special increment (`inc`) and decrement (`dec`) instruction, which can be used to add `1` or subtract `1` from either a register or a memory location:

```
inc eax    ; same as eax = eax+1
dec ebx    ; same as ebx = ebx-1
```

Multiplication is done using the `mul` instruction. The `mul` instruction takes only one operand; that operand is multiplied by the content of either the `al`, `ax`, or `eax` register. The result of the multiplication is stored in either the `ax`, `dx and ax`, or `edx and eax` register.

If the operand of the `mul` instruction is *8 bits (1 byte),* then it is multiplied by the 8-bit `al` register, and the product is stored in the `ax` register. If the operand is *16 bits (2 bytes),* then it is multiplied with the `ax` register, and the product is stored in the `dx` and `ax` register. If the operand is a *32-bit (4 bytes),* then it is multiplied with the `eax` register, and the product is stored in the `edx` and `eax` register. The reason the product is stored in a register double the size is because when two values are multiplied, the output values can be much larger than the input values. The following outlines variations of `mul` instructions:

```
mul ebx  ;ebx is multiplied with eax and result is stored in EDX and EAX
mul bx   ;bx is multiplied with ax and the result is stored in DX and AX
```

Division is performed using the `div` instruction. The `div` takes only one operand, which can be either a register or a memory reference. To perform division, you place the dividend (number to divide) in the `edx and eax` register, with `edx` holding the most significant *dword*. After the `div` instruction is executed, the quotient is stored in `eax`, and the remainder is stored in the `edx` register:

```
div ebx   ; divides the value in EDX:EAX by EBX
```

Challenge

Let's take on another simple challenge. The following is a disassembled output of a simple C program. Can you figure out what this program does, and can you translate it back to a pseudocode?

```
mov dword ptr [ebp-4], 16h
mov dword ptr [ebp-8], 5
mov eax, [ebp-4]
add eax, [ebp-8]
mov [ebp-0Ch], eax
mov ecx, [ebp-4]
sub ecx, [ebp-8]
mov [ebp-10h], ecx
```

Solution

You can read the code line by line and try to determine the program's logic, but it would be easier if you translate it back to some high-level language. To understand the preceding program, let's use the same logic that was covered previously. The preceding code contains four memory references. First, let's label these addresses - `ebp-4=a`, `ebp-8=b`, `ebp-0Ch=c`, and `ebp-10H=d`. After labeling the addresses, it translates to the following:

```
mov dword ptr [a], 16h
mov dword ptr [b], 5
mov eax, [a]
add eax, [b]
mov [c], eax
mov ecx, [a]
sub ecx, [b]
mov [d], ecx
```

Now, let's translate the preceding code into a pseudocode (high-level language equivalent). The code will as follows:

```
a = 16h    ; h represents hexadecmial, so 16h (0x16) is 22 in decimal
b = 5
eax = a
eax = eax + b  ❶
c = eax  ❶
ecx = a
ecx = ecx-b  ❶
d = ecx  ❶
```

Replacing all of the register names with their corresponding values on the right-hand side of the = operator (in other words, at ❶), we get the following code:

```
a = 22
b = 5
eax = a  ❷
eax = a+b  ❷
c = a+b
ecx = a  ❷
ecx = a-b  ❷
d = a-b
```

After removing all of the entries containing registers on the left-hand side of the = sign at ❷ (because registers are used for temporary calculations), we are left with the following code:

```
a = 22
b = 5
c = a+b
d = a-b
```

Now, we have reduced the eight lines of assembly code to four lines of pseudocode. At this point, you can tell that the code performs addition and subtraction operations and stores the results. You can determine the variable types based on the sizes and how they are used in the code (context), as mentioned earlier. The variables `a` and `b` are used in addition and subtraction, so these variables have to be of integer data types, and the variables `c` and `d` store the results of integer addition and subtraction, so it can be guessed that they are also integer types. Now, the preceding code can be written as follows:

```
int a,b,c,d;
a = 22;
b = 5;
c = a+b;
d = a-b;
```

If you are curious about how the original C program of the disassembled output looks, then the following is the original C program to satisfy your curiosity. Notice how we were able to write an assembly code back to its equivalent high-level language:

```
int num1 = 22;
int num2 = 5;
int diff;
int sum;
sum = num1 + num2;
diff = num1 - num2;
```

2.3 Conditional Jumps Challenge

Theory

In *conditional* jumps, the control is transferred to a memory address based on some condition. To use a conditional jump, you need instructions that can alter the flags (*set* or *clear*). These instructions can be performing an *arithmetic* operation or a *bitwise* operation. The x86 instruction provides the `cmp` instruction, which subtracts the *second operand (source operand)* from the *first operand (destination operation)* and alters the flags without storing the difference in the destination. In the following instruction, if the `eax` contained the value `5`, then `cmp eax,5` would set the zero flag (`zf=1`), because the result of this operation is zero:

```
cmp eax,5  ; subtracts eax from 5, sets the flags but result is not stored
```

Another instruction that alters the flags without storing the result is the `test` instruction. The `test` instruction performs a bitwise `and` operation and alters the flags without storing the result. In the following instruction, if the value of `eax` was zero, then the zero flag would be set (`zf=1`), because when you `and 0` with `0` you get `0`:

```
test eax,eax ; performs and operation, alters the flags but result in not
stored
```

Both `cmp` and `test` instructions are normally used along with the conditional `jump` instruction for decision making.

There are a few variations of conditional jump instructions; the general format is shown here:

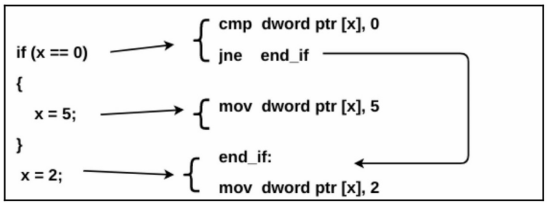
```
jcc <address>
```

The `cc` in the preceding format represents conditions. These conditions are evaluated based on the bits in the `eflags` register. The following table outlines the different conditional jump instructions, their aliases, and the bits used in the `eflags` register to evaluate the condition:

Instruction	Description	Aliases	Flags
`jz`	jump if zero	`je`	`zf=1`
`jnz`	jump if not zero	`jne`	`zf=0`
`jl`	jump if less	`jnge`	`sf=1`
`jle`	jump if less or equal	`jng`	`zf=1 or sf=1`
`jg`	jump if greater	`jnle`	`zf=0 and sf=0`
`jge`	jump if greater or equal	`jnl`	`sf=0`
`jc`	jump if carry	`jb,jnae`	`cf=1`
`jnc`	jump if not carry	`jnb,jae`	.

If-else statement

If (x == 0) {
  x = 5;
}
x = 2;
	cmp dword ptr [x], 0
	jne end_if
	mov dword ptr [x], 5
	end_if:
	mov dword ptr [x], 2



The following is the disassembled output of a program; let's translate the following code to its high-level equivalent. Use the techniques and the concepts that you learned previously to solve this challenge:

```
mov dword ptr [ebp-4], 1
cmp dword ptr [ebp-4], 0
jnz loc_40101C
mov eax, [ebp-4]
xor eax, 2
mov [ebp-4], eax
jmp loc_401025

loc_40101C:
 mov ecx, [ebp-4]
 xor ecx, 3
 mov [ebp-4], ecx

loc_401025:
```

Solution

Let's start by assigning the symbolic names to the address (`ebp-4`). After assigning the symbolic names to the memory address references, we get the following code:

```
mov dword ptr [x], 1
cmp dword ptr [x], 0  ❶
jnz loc_40101C  ❷
mov eax, [x]  ❹
xor eax, 2
mov [x], eax
```

```
    jmp loc_401025  ❸

    loc_40101C:
    mov ecx, [x]  ❺
    xor ecx, 3
    mov [x], ecx  ❻

    loc_401025:
```

In the preceding code, notice the `cmp` and `jnz` instructions at ❶ and ❷ (this is a conditional statement) and note that `jnz` is the same as `jne` (`jump if not equal to`). Now that we have identified the conditional statement, let's try to determine what type of conditional statement this is (`if`, or `if/else`, or `if/else if/else`, and so on); to do that, focus on the jumps. The conditional jump at ❷ is taken to `loc_401010C`, and before the `loc_40101C`, there is an unconditional jump at ❸ to `loc_401025`. From what we learned previously, this has the characteristics of an `if-else` statement. To be precise, the code from ❹ to ❸ is part of the `if` block and the code from ❺ to ❻ is part of the `else` block. Let's rename `loc_40101C` to `else` and `loc_401025` to `end` for better readability:

```
    mov dword ptr [x], 1  ❼
    cmp dword ptr [x], 0  ❶
    jnz else  ❷
    mov eax, [x]  ❹
    xor eax, 2
    mov [x], eax  ❽
    jmp end  ❸

    else:
    mov ecx, [x]  ❺
    xor ecx, 3
    mov [x], ecx  ❻
    end:
```

In the preceding assembly code, `x` is assigned a value of `1` at ❼; the value of `x` is compared with `0`, and if it is equal to `0` (❶ and ❷), the value of `x` is `xored` with `2`, and the result is stored in `x` (❹ to ❽). If `x` is not equal to `0`, then the value of `x` is `xored` with `3` (❺ to ❻).

Reading the assembly code is slightly tricky, so let's write the preceding code in a high-level language equivalent. We know that ❶ and ❷ is an `if` statement, and you can read it as `jump is taken to else, if x is not equal to 0` (remember `jnz` is an alias for `jne`).

Agenda

In this session, we will talk about:

- The Workshop
- What is RE and why do it
- x86 Overview - Going Deeper
- IDA Basics
- Exercises

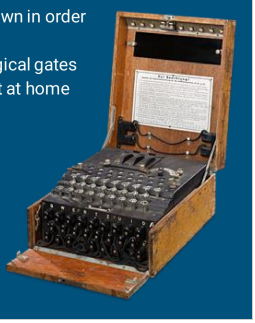
The Workshop

There are 3 goals to this workshop:

1. Introduce you to the world of RE
2. Prove you that it's possible
3. Show you that it's challenging

RE

- What is Reverse Engineering?
  - Reverse engineering means to take some product and break it down in order to understand how it was produced
  - In hardware: slicing electrical components and analyzing their logical gates
  - In a restaurant: tasting an amazing dish and trying to reproduce it at home



The Enigma was also RE'd ;)

Software RE

- In software reverse engineering:
  - The research object is **a program** - the machine code of an executable file
  - The goal is to **understand what the program does and how**

Why RE?

- because sometimes you can't run the program and you need to analyze it **statically**
- because 'basic' analysis does not provide the whole picture:
  - knowing that a program X uses a function F is one thing, but why is F used? What are its parameters? What does it return?
  - knowing that a program X sends a packet P does not tell us exactly what is in P, how it's parsed, etc.

Reverse Engineering is **Challenging**.

Why?

- Real-life software is not easy to analyze:
  - it might be obfuscated
  - it might be packed
  - it will probably be huge
- But one still has to start somewhere, right? :)

x86 Overview

Why Assembly?

Assembly is the most popular low-level language* (more precisely, a class of languages)

Low-level language: a human-readable version of a computer architecture's instruction set.

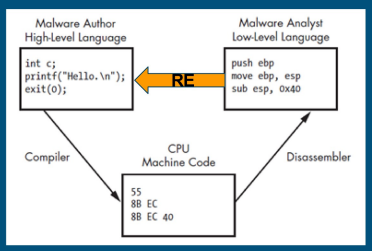
Compilation & Disassembly

The program author writes code in some high-level language, say C.

The C code is compiled into machine code - a series of bytes that the CPU understands.

The researcher usually has no access to the C source code, only to the bytes of machine code.

To make life easier, a disassembler translates these bytes into an easier-to-read textual representation.



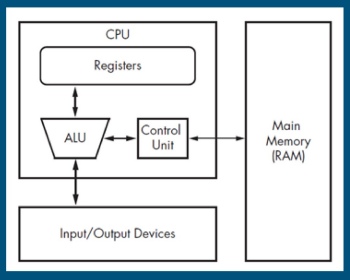
All images in these slides are from Michael Sikorsi & Andrew Honig's "Practical Malware Analysis"

x86 Architecture

As a program runs, the following loop is executed:

1. A CPU instruction is read from the main memory by the Control Unit
2. The instruction is processed and executed by the Arithmetic-Logic Unit, along with input from the user or the registers
3. The operation's output is stored in the CPU's registers or sent to an output device

Let's understand what an x86 instruction looks like.



Instructions

Assembly instruction = mnemonic + optional operand(s). For example:

```
mov eax, 0xFF ~ B8 FF 00 00 00
```

An operand can be:

- an immediate - *0x3*
- a register - *eax*
- a memory address - [0x400100 + 4]

Opcode: the bytes that correspond to the instruction and its operands

Popular Instructions

Data storage

- mov
- lea

Arithmetic

- add
- sub
- inc
- dec
- mul
- div

Logic

- or
- and
- xor
- shr
- shl

Stack

- push
- pop
- call
- ret

Control-Flow

- test
- cmp
- jmp
- jcc

Do you know all these? ;)

Quiz

What does each of the following instructions do? (answers are in the next slides)

- `mov eax, ebx`
- `mov eax, 0x42`
- `mov eax, [0x4037c4]`
- `mov eax, [ebx]`
- `mov eax, [ebx+esi*4]`

Quiz

What does each of the following instructions do?

- `mov eax, ebx` — **move what's in ebx to eax**
- `mov eax, 0x42`
- `mov eax, [0x4037c4]`
- `mov eax, [ebx]`
- `mov eax, [ebx+esi*4]`

Quiz

What does each of the following instructions do?

- `mov eax, ebx` — move what's in ebx to eax
- `mov eax, 0x42` — move 0x42 to eax
- `mov eax, [0x4037c4]`
- `mov eax, [ebx]`
- `mov eax, [ebx+esi*4]`

Quiz

What does each of the following instructions do?

- `mov eax, ebx` — move what's in ebx to eax
- `mov eax, 0x42` — move 0x42 to eax
- `mov eax, [0x4037c4]` — move what's in address 0x4037c4 to eax
- `mov eax, [ebx]`
- `mov eax, [ebx+esi*4]`

Quiz

What does each of the following instructions do?

- `mov eax, ebx` — move what's in ebx to eax
- `mov eax, 0x42` — move 0x42 to eax
- `mov eax, [0x4037c4]` — move what's in address 0x4037c4 to eax
- `mov eax, [ebx]` — move what's in the address in ebx to eax
- `mov eax, [ebx+esi*4]`

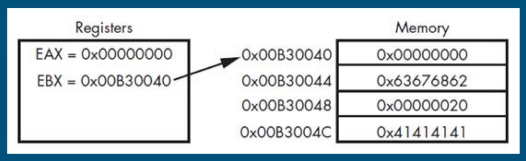
Quiz

What does each of the following instructions do?

- `mov eax, ebx` — move what's in ebx to eax
- `mov eax, 0x42` — move 0x42 to eax
- `mov eax, [0x4037c4]` — move what's in address 0x4037c4 to eax
- `mov eax, [ebx]` — move what's in the address in ebx to eax
- `mov eax, [ebx+esi*4]` — move what's in address ebx+esi*4 to

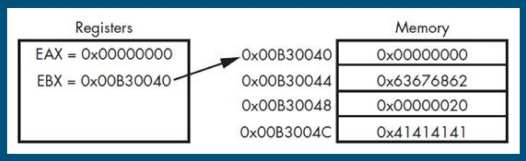
Quiz

- What's the difference? (answer is in the next slide)
  - `mov eax, [ebx + 8]`
  - `lea eax, [ebx + 8]`



Quiz

- What's the difference?
  - `mov eax, [ebx + 8]` `move what's in address ebx+8 (0x20) to eax`
  - `lea eax, [ebx + 8]` `move the value ebx+8 (0xB30048) to eax`



Registers

A register is the CPU's basic data storage unit, whose access time is the fastest.

General Registers
EAX (AX, AH, AL)
EBX (BX, BH, BL)
ECX (CX, CH, CL)
EDX (DX, DH, DL)
EBP (BP)
ESP (SP)
ESI (SI)
EDI (DI)

Segment Registers
CS [code section]
SS [stack section]
DS [data section]
ES, FS, GS [general]

These point to the base address of different memory sections

EFLAGS
a register with 32 bit-flags that provide information on previous operations (TBC)

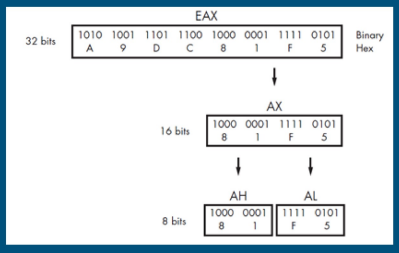
EIP
This register always holds the address of the next instruction to execute

Register Breakdown

EAX, EBX, ECX & EDX can be broken-down as follows (EAX is used as example):

- EAX - all 32 bits
- AX - 16 least-significant bits of EAX
- AH - 8 most-significant bits of AX
- AL - 8 least-significant bits of AX

Can you think how the 16 **most-significant** bits of EAX can be accessed? (answer is in the next slide)



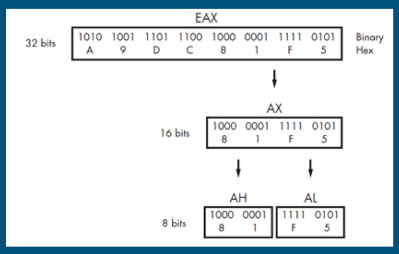
Register Breakdown

EAX, EBX, ECX & EDX can be broken-down as follows (EAX is used as example):

- EAX - all 32 bits
- AX - 16 least-significant bits of EAX
- AH - 8 most-significant bits of AX
- AL - 8 least-significant bits of AX

Can you think how the 16 **most-significant** bits of EAX can be accessed?

SHR EAX, 0x10 → Use AX



Register Conventions

- EAX - A function's return value
- ECX - Counters (loop variables)
- EAX:EDX - Quotient and remainder in multiplication and division

EFLAGS Register

32 bit-flags that give information on the result of previous computation. The most common flags are:

- **Zero Flag**: set if an operation result is 0; otherwise cleared.
- **Carry Flag**: set if an operation results is too large or too small for the destination operand; otherwise cleared
- **Sign Flag**: set if the MSB is set (namely, the result is negative); otherwise cleared

EFLAGS Quiz!

```
mov     eax, 0x1
mov     ebx, 0x0
sub     ebx, eax
```

What is the status of the flags (answers are in the next slides):

- zero-flag?
- carry-flag?
- sign-flag?

EFLAGS Quiz!

```
mov     eax, 0x1
mov     ebx, 0x0
sub     ebx, eax
```

What is the status of the flags:

- zero-flag? **0**
- carry-flag?
- sign-flag?

EFLAGS Quiz!

```
mov     eax, 0x1
mov     ebx, 0x0
sub     ebx, eax
```

What is the status of the flags:

- zero-flag? **0**
- carry-flag? **0**
- sign-flag?

EFLAGS Quiz!

```
mov     eax, 0x1
mov     ebx, 0x0
sub     ebx, eax
```

What is the status of the flags:

- zero-flag? **0**
- carry-flag? **0**
- sign-flag? **1**

Branching (Control Flow)

- Two types of jumps in x86:
  - Unconditional - just jump to where I tell you
    - jmp *0x401072*
  - Conditional - check the result of some computation, jump accordingly
    - cmp eax, 0x10 compare the value of eax with 0x10
    - jge *0x401072* jump to *0x401072* if eax is greater\equal than 0x10

Jumps

There are many types of conditional jumps. The set of conditional jump instructions is often referred to as *jcc* (where the *j* stands for *jump* and the *c* for *condition*).

Each jump instruction performs different checks on the **EFLAGS** register to determine whether the jump should be performed or not.

Instruction	Description
`jz loc`	Jump to specified location if ZF = 1.
`jnz loc`	Jump to specified location if ZF = 0.
`je loc`	Same as `jz`, but commonly used after a `cmp` instruction. Jump will occur if the destination operand equals the source operand.
`jne loc`	Same as `jnz`, but commonly used after a `cmp`. Jump will occur if the destination operand is not equal to the source operand.
`jg loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than the source operand.
`jge loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than or equal to the source operand.
`ja loc`	Same as `jg`, but an unsigned comparison is performed.
`jae loc`	Same as `jge`, but an unsigned comparison is performed.
`jl loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than the source operand.
`jle loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than or equal to the source operand.
`jb loc`	Same as `jl`, but an unsigned comparison is performed.
`jbe loc`	Same as `jle`, but an unsigned comparison is performed.
`jo loc`	Jump if the previous instruction set the overflow flag (OF = 1).
`js loc`	Jump if the sign flag is set (SF = 1).
`jecxz loc`	Jump to location if ECX = 0.

Conditionals

- Two operations are commonly used before conditional jumps:
  - ***test*** ~ perform a **logical AND**
  - ***cmp*** ~ perform **subtraction**
    Both instructions **do not** store the result, but change the flags in EFLAGS as needed

An Example

Here's a pair of instructions you will see **a lot** *(not necessarily with eax...)*

test eax, eax
jz 0x400100

Since *test* is practically a logical AND, and since X AND X == X is always true, the result of the first line is either zero (if EAX == 0) or some non-zero value (otherwise).

Therefore, this is an efficient way of checking if EAX equals zero or not.

How would you say in x86...?

- `if (a == b) goto 0x1000;`
- `if (a < b) goto 0x1000;`
- `if (a) goto 0x1000;`

How would you say in x86...?

- `if (a == b) goto 0x1000;` **`cmp a, b; jz 0x1000;`**
- `if (a < b) goto 0x1000;`
- `if (a) goto 0x1000;`

How would you say in x86...?

- `if (a == b) goto 0x1000;` **`cmp a, b; jz 0x1000;`**
- `if (a < b) goto 0x1000;` **`cmp a,b; jl 0x1000;`**
- `if (a) goto 0x1000;`

How would you say in x86…?

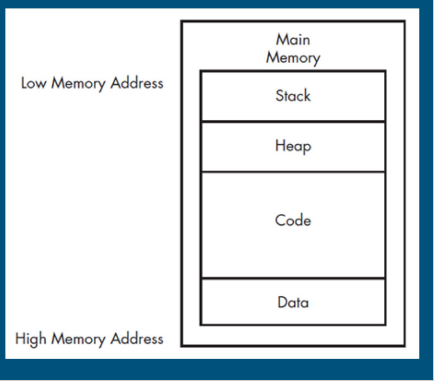
- `if (a == b) goto 0x1000;`
  `cmp a, b; jz 0x1000;`
- `if (a < b) goto 0x1000;`
  `cmp a,b; jl 0x1000;`
- `if (a) goto 0x1000;`
  `test a, a; jnz 0x1000;`

Main Memory

We have seen before how the CPU "talks" to the main memory ("RAM").

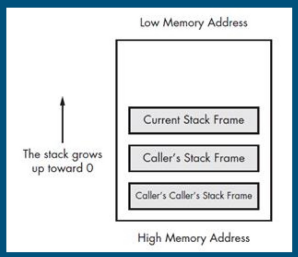
The following are (some of) the different sections of a process loaded into the RAM:

- **Data**: static / global variables, put in place when the program is loaded
- **Code**: the program's CPU instructions
- **Heap**: dynamic memory, allocated and freed during runtime
- **Stack**: variables and arguments local to the program's functions



The Stack

- A LIFO data structure with push & pop operations
- Stack-relevant registers:
  - **ESP** - **Stack Pointer**, always points to the top of the stack, therefore dynamic
  - **EBP** - **Base Pointer**, stays consistent within a function and is used to reference the function's local variables and parameters
- Stack-relevant instructions: *pop, push, call, ret* (and also *enter, leave…*)



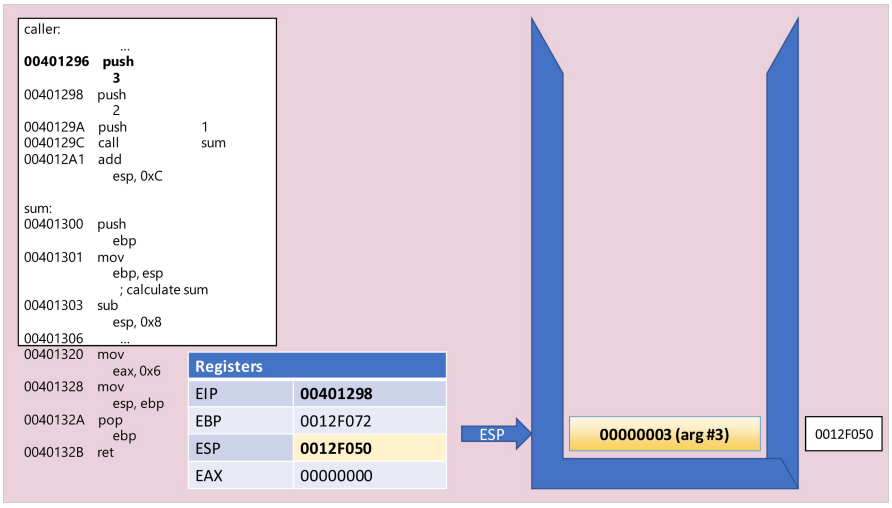
Function Calls - Demo

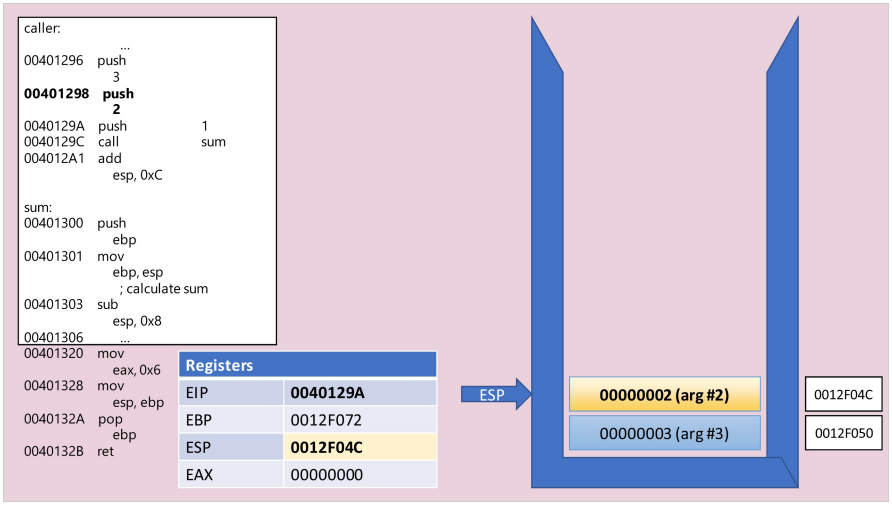
In the following slides, x86 code will be presented along with the stack in its current state.

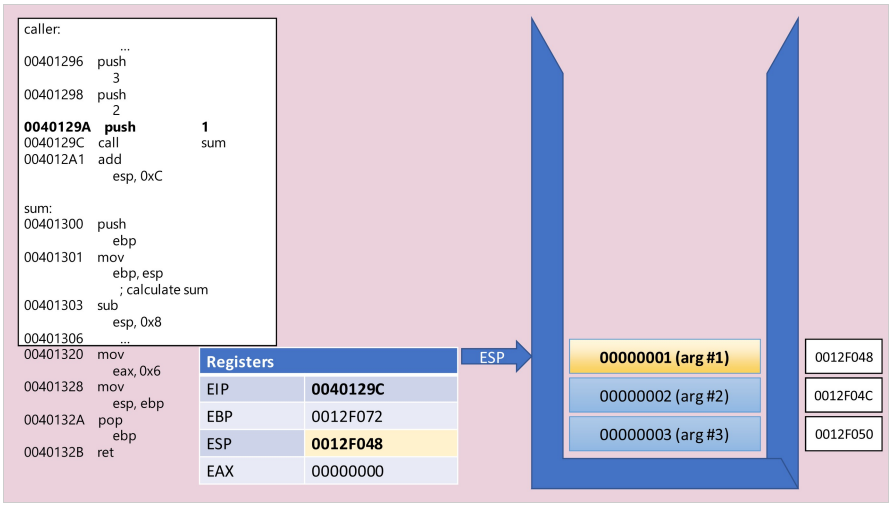
The code shows how a function named ***caller*** calls another function ***sum***.

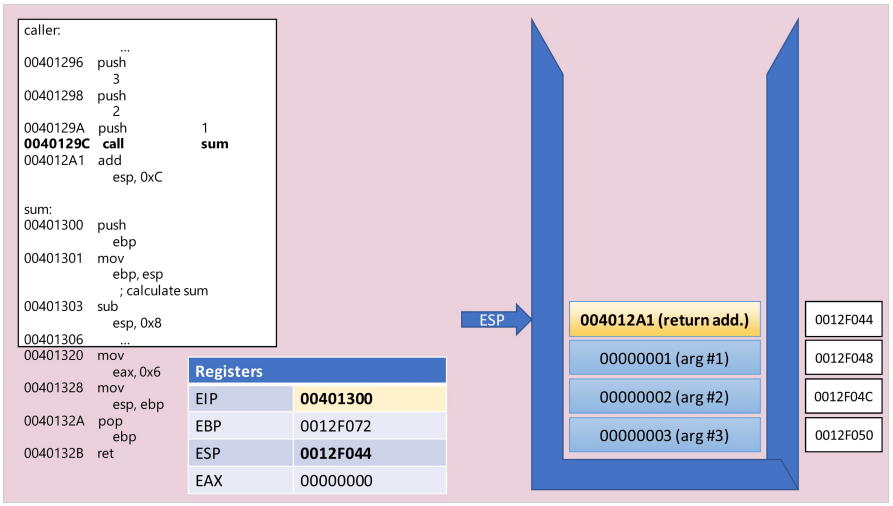
Each slide, look at the next instruction and try to predict the side-effects and changes to come.

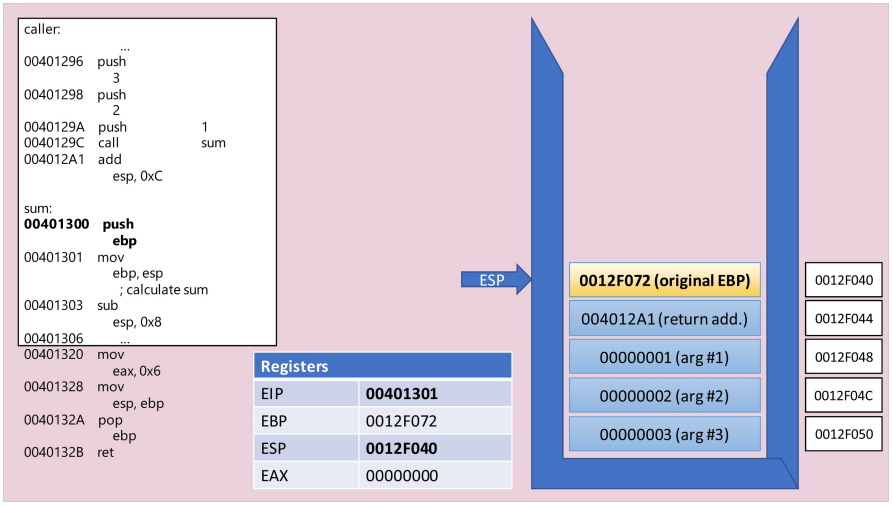
Note: instructions in **bold** have been just executed and their side-effects are already presented on the slide.

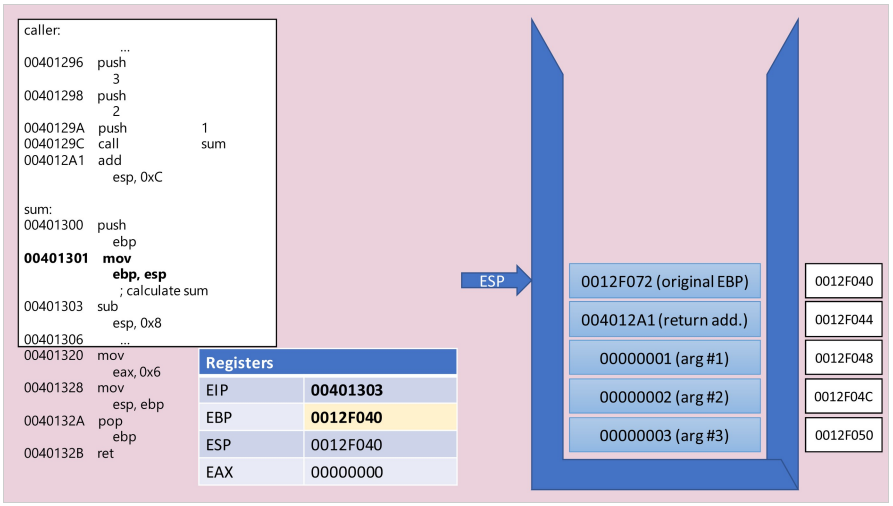


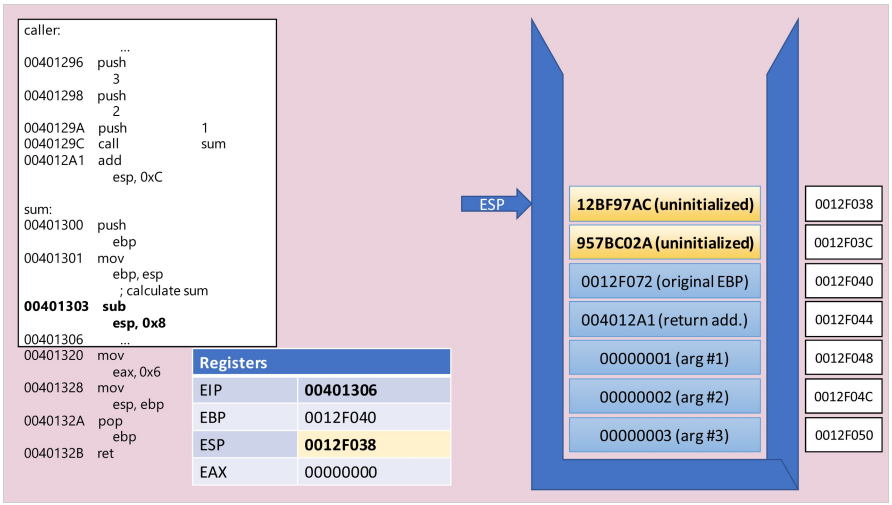


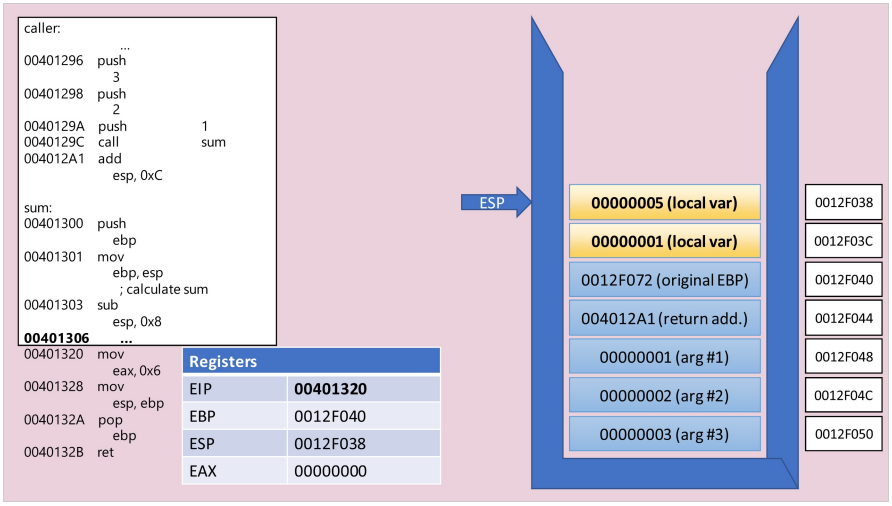


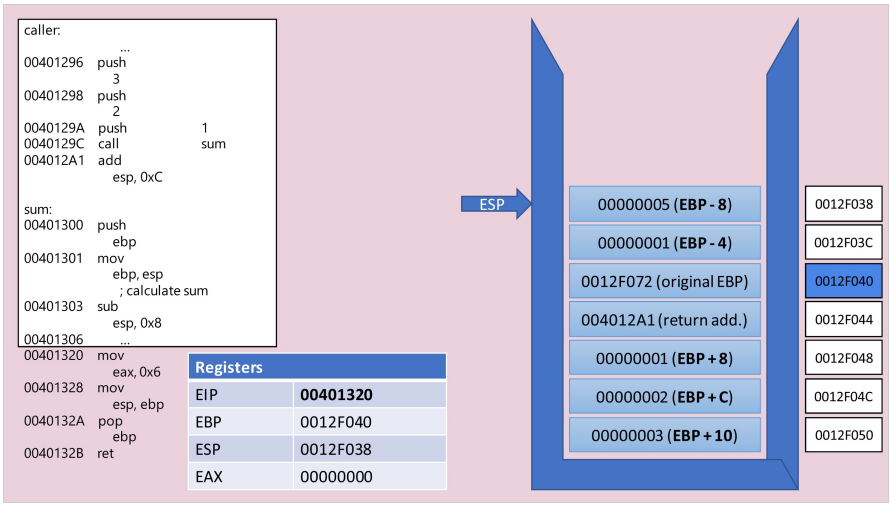

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401300
EBP 0012F072
ESP 0012F044
EAX 00000000
ESP
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F044
0012F048
0012F04C
0012F050

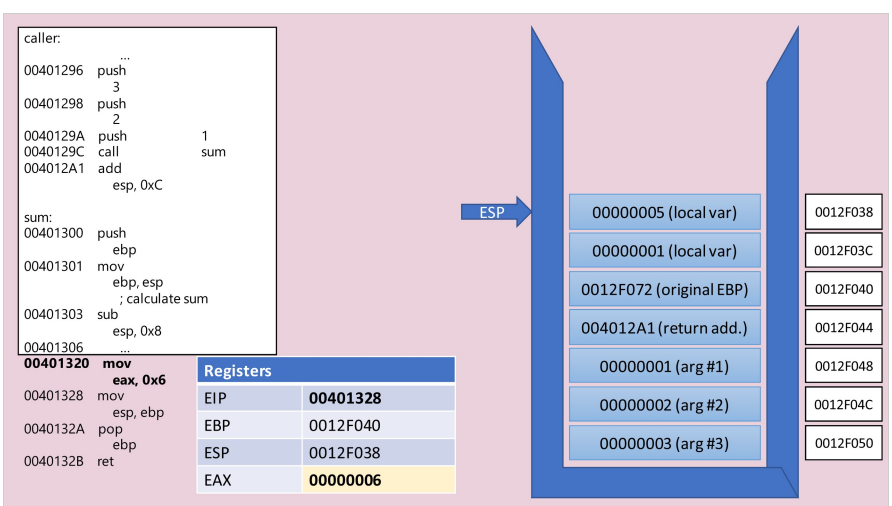

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401301
EBP 0012F072
ESP 0012F040
EAX 00000000
ESP
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F040
0012F044
0012F048
0012F04C
0012F050

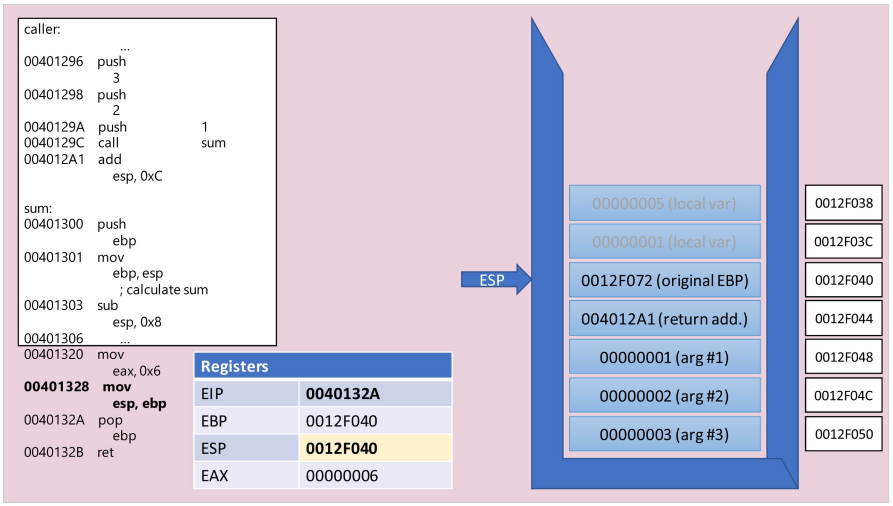

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401303
EBP 0012F040
ESP 0012F040
EAX 00000000
ESP
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F040
0012F044
0012F048
0012F04C
0012F050

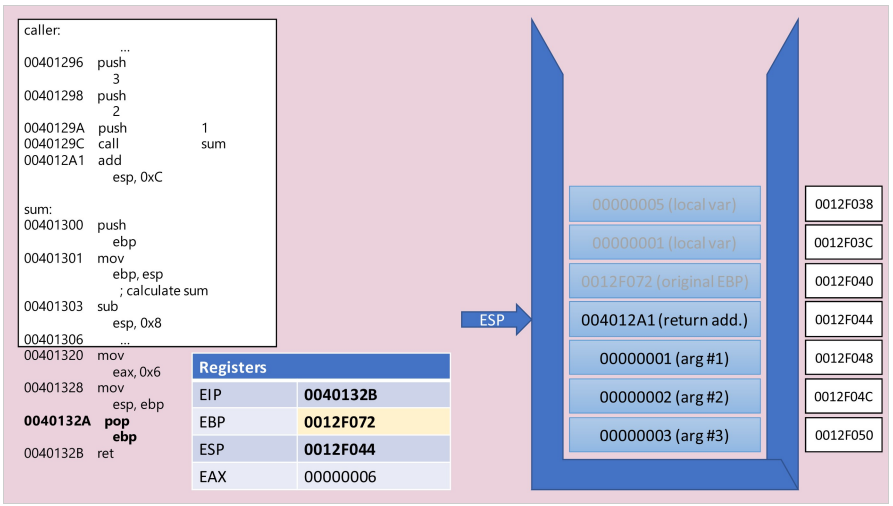

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401306
EBP 0012F040
ESP 0012F038
EAX 00000000
ESP
12BF97AC (uninitialized)
957BC02A (uninitialized)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050

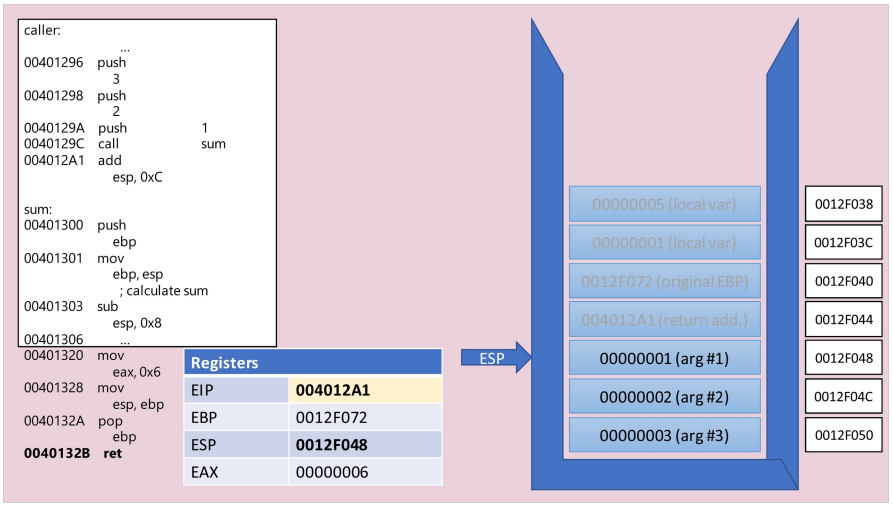

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401320
EBP 0012F040
ESP 0012F038
EAX 00000000
ESP
00000005 (local var)
00000001 (local var)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050

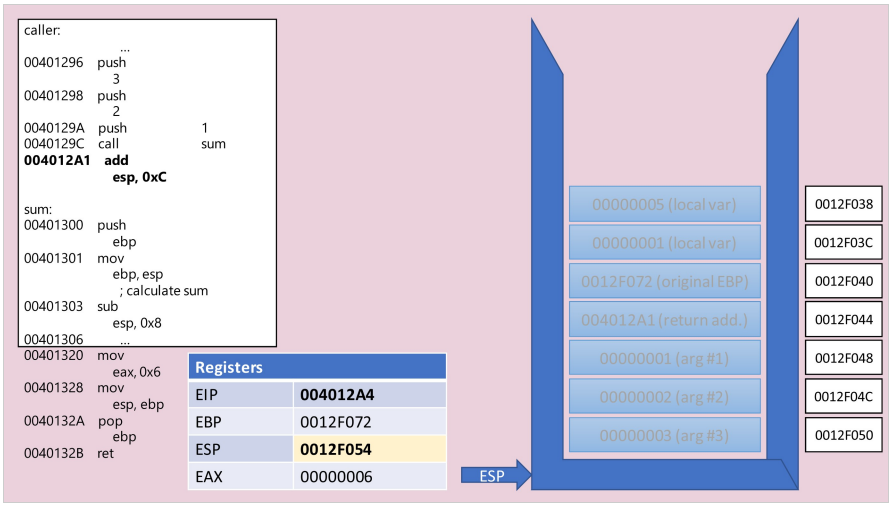

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401320
EBP 0012F040
ESP 0012F038
EAX 00000000
ESP
00000005 (EBP - 8)
00000001 (EBP - 4)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (EBP + 8)
00000002 (EBP + C)
00000003 (EBP + 10)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050


caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 00401328
EBP 0012F040
ESP 0012F038
EAX 00000006
ESP
00000005 (local var)
00000001 (local var)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050


caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 0040132A
EBP 0012F040
ESP 0012F040
EAX 00000006
ESP
00000005 (local var)
00000001 (local var)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050


caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 0040132B
EBP 0012F072
ESP 0012F044
EAX 00000006
ESP
00000005 (local var)
00000001 (local var)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050


caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 004012A1
EBP 0012F072
ESP 0012F048
EAX 00000006
ESP
00000005 (local var)
00000001 (local var)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050

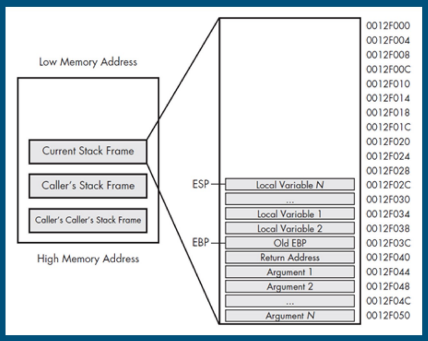

caller:
...
00401296 push 3
00401298 push 2
0040129A push 1
0040129C call sum
004012A1 add esp, 0xC
sum:
00401300 push ebp
00401301 mov ebp, esp
; calculate sum
00401303 sub esp, 0x8
00401306 ...
00401320 mov eax, 0x6
00401328 mov esp, ebp
0040132A pop ebp
0040132B ret
Registers
EIP 004012A4
EBP 0012F072
ESP 0012F054
EAX 00000006
ESP
00000005 (local var)
00000001 (local var)
0012F072 (original EBP)
004012A1 (return add.)
00000001 (arg #1)
00000002 (arg #2)
00000003 (arg #3)
0012F038
0012F03C
0012F040
0012F044
0012F048
0012F04C
0012F050

Function Calls - Zoom Out

In the previous slides we saw how a single stack frame is generated.

In the illustration on the right we can see multiple stack frames on top of each other, each has its own parameters (arguments), return address, its caller's EBP and local variables.

Now that we know the architecture pretty well, let's see how high-level language is compiled into 0x86 assembly.



C to x86

- A C program has two arguments for the main function:
  - `int main(int argc, char** argv)`

  For example, a program which is executed from the command line like that:
  `awesome.exe -r path_to_file.txt`

  will have the following *argc, argv* values:

- `argc = 3`
- `argv = [ address of "awesome.exe" string in memory,`
  `address of "-r" string in memory,`
  `address of "path_to_file.txt" string in memory ]`

Remember #1

- mov [ebp+*my_var*], *some_value*
  What does it mean?
  - *my_var* is an **offset** (-4, -8, -C...) from EBP to the stack address where *my_var* is stored (thanks IDA for helping us and renaming it!)
  - The meaning of this expression is *"assign the value some_value to my_var"*

Remember #2

- *function(a, b, c) → push c; push b; push a;*
  Function parameters are pushed in **reverse order**.

Remember #3

- *mov eax, [ebp+argv]* ; *argv* is again an offset from EBP to where *argv*'s address is stored
  *mov ecx, [eax+4*i]* ; This means "set *ecx to equal* ***argv[i]***"

C to x86

Let's look at the following C program.

What does it do?

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

1. It verifies that the number of arguments passed to the program is 3 (including the program's path).
2. It checks if the second argument is a flag "-r".
3. If that's the case, it deletes the file provided in the third argument.

C to x86

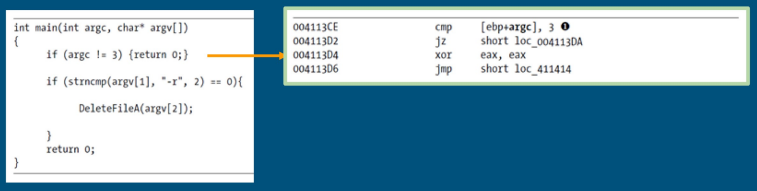
First, we compare *argc* to 3.
We learned that *cmp* is practically a *sub* - so 3 is subtracted from the value of *argc*. If the result is 0, then *argc* is indeed 3 and we jump to 0x004113DA.

```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113CE    cmp     [ebp+argc], 3
004113D2    jz      short loc_004113DA
004113D4    xor     eax, eax
004113D6    jmp     short loc_411414
```

C to x86

If the result is non zero, then *argc* is not equal to 3. We zeroize EAX by XORing it with itself and then jump to an address where the function returns (*trust me with that*).



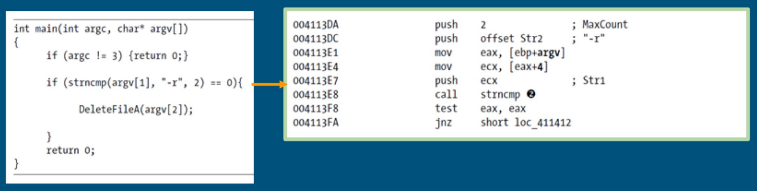
```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113CE    cmp     [ebp+argc], 3 ❶
004113D2    jz      short loc_004113DA
004113D4    xor     eax, eax
004113D6    jmp     short loc_411414
```

C to x86

Next, we can see a call to *strncmp*. The following parameters are pushed to it:

1. 2, the number of characters to compare
2. a literal string "-r", to which some other string should be compared
3. ECX - what does it hold?

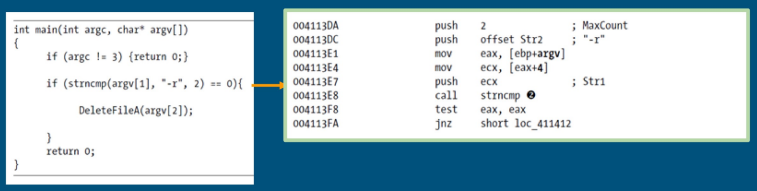


```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

```
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
```

C to x86

ECX stores whatever is in address *eax+4*.
EAX, in turn, stores whatever is in address *ebp+argv*, which is the address to the *argv* array.
Therefore, ECX = *(*argv_address* + 4) = *argv*[1]



```
int main(int argc, char* argv[])
{
      if (argc != 3) {return 0;}

      if (strncmp(argv[1], "-r", 2) == 0){

            DeleteFileA(argv[2]);

      }
      return 0;
}
```

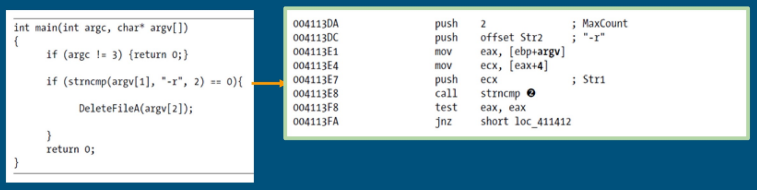
```
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
```

C to x86

Given that parameters are pushed in reverse, the actual call is:

`strncmp(argv[1], "-r", 2)`

which is exactly what we see in our C code :)

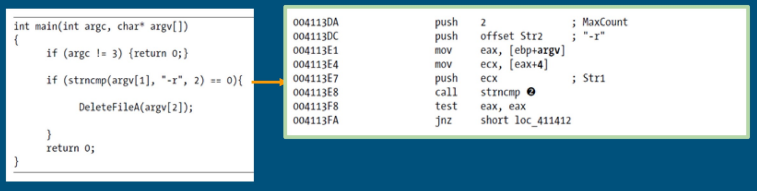


```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

```
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
```

C to x86

If the comparison holds (namely, the strings are equal, and the user indeed passed *"-r"* as the second argument) - EAX will be equal to zero and there will not be a jump. Otherwise - if the strings are not equal - we jump and return zero (*trust me once again...*)



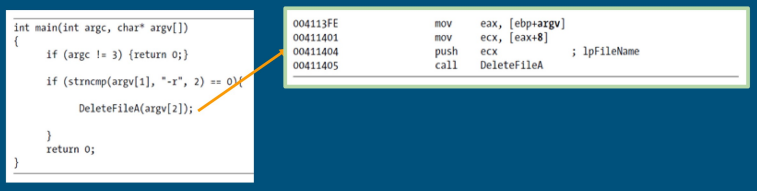
```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

```
004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412
```

C to x86

For the case in which the string **are** equal, we reach the following code. We push ECX, which (similarly to the previous block) hold **(argv + 8) = argv[2]*.

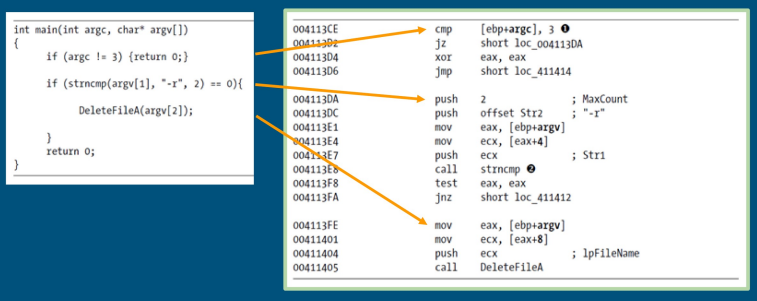
Then we call *DeleteFileA* to delete the file whose path was just pushed.



```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

```
004113FE    mov     eax, [ebp+argv]
00411401    mov     ecx, [eax+8]
00411404    push    ecx             ; lpFileName
00411405    call    DeleteFileA
```

C to x86 - All Code



```
int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){

        DeleteFileA(argv[2]);

    }
    return 0;
}
```

```
004113CE    cmp     [ebp+argc], 3 ❶
004113D2    jz      short loc_004113DA
004113D4    xor     eax, eax
004113D6    jmp     short loc_411414

004113DA    push    2               ; MaxCount
004113DC    push    offset Str2     ; "-r"
004113E1    mov     eax, [ebp+argv]
004113E4    mov     ecx, [eax+4]
004113E7    push    ecx             ; Str1
004113E8    call    strncmp ❷
004113F8    test    eax, eax
004113FA    jnz     short loc_411412

004113FE    mov     eax, [ebp+argv]
00411401    mov     ecx, [eax+8]
00411404    push    ecx             ; lpFileName
00411405    call    DeleteFileA
```

1. Initial Compromise (using any exploit)
2. Two Possibilities: 1. Persistence 2. Privelage Escalation

Persistence:

- Registry : Rinkeys, Run Once
- Services
- Scheduled tasks
- Alternate Datastreams
- Memory Injection
- Startup folder
- Component Object Model (COM Hijacking)
- DLL Hijacking
- Fileless persistance (wmi based, Add a powershell's oneliner in registry)

what are the techniques used in malware to achieve persistence??

Malware persistence is the ability of a malware to remain active on a system even after a reboot or a system shutdown. There are several techniques that malware can use to achieve persistence:
Scheduled Tasks: Malware can create a scheduled task to automatically run the malware at regular intervals, using the Windows Task Scheduler API.
Registry keys: Malware can create a new registry key or modify an existing one to automatically run the malware at system startup.
Startup folders: Malware can create a new shortcut in the startup folder or modify an existing one to automatically run the malware at system startup.
Services: Malware can create a new service or modify an existing one to automatically run the malware at system startup.
DLL Injection: Malware can inject itself into a legitimate process, making it difficult to detect and remove.
Rootkits: Malware can use rootkit techniques to hide its presence on the system, making it difficult to detect and remove.
Alternate Data Streams: Malware can use alternate data streams to hide files on the system, making it difficult to detect and remove.
Filter Drivers: Malware can use filter drivers to intercept and modify system calls, making it difficult to detect and remove.
Bootkits: Malware can infect the boot sector of a hard drive, making it difficult to detect and remove.
Fileless Malware: Malware can reside only in memory and not on the hard drive which makes it difficult to detect and remove.
Protocol abuse: malware can use a protocol in a way that is not intended to achieve persistence.
Self-replication: Malware can replicate itself across different systems and network shares, achieving persistence through multiple copies.
Kernel-mode rootkits: Malware can infect the kernel of the operating system, making it more difficult to detect and remove.
Cloud-based persistence: malware can use cloud services to achieve persistence by syncing its files and settings across multiple devices.
Code injection: malware can inject its code into legitimate processes, making it difficult to detect and remove.
File-less malware: malware can exist only in memory and registry, making it difficult to detect and remove.
Hijacking legitimate processes: malware can use legitimate process such as explorer.exe to execute its code, making it difficult to detect and remove.
Hooking: malware can use hooking techniques to intercept and modify system calls, making it difficult to detect and remove.
Stealing credentials: malware can steal credentials such as passwords or tokens, to achieve persistence by logging in as an authorized user.
Script-based persistence: malware can use scripts such as powershell or javascript to achieve persistence and evade detection.
Social engineering: malware can use social engineering techniques to trick users into installing and running the malware, allowing it to achieve persistence.

It's worth noting that these are just a few examples of the many techniques that malware can use to achieve persistence, and new techniques are constantly being developed. Additionally, malware can use a combination of these techniques to make it even harder to detect and remove.

Scheduled Tasks:

Here is the list of APIs that malware can use to achieve persistence using scheduled tasks, in the correct order:
CoInitializeEx(): This function initializes the COM library, and must be called before any other task scheduler APIs can be used.
CoCreateInstance(): This function creates an instance of the Task Scheduler, and returns an interface pointer that can be used to interact with the Task Scheduler.
ITaskFolder::CreateTask(): This function creates a new scheduled task, and returns an interface pointer that can be used to configure the task.
ITaskDefinition::get_Actions(): This function retrieves the task's action collection, which can be used to add the path to the malware executable to the task.
ITaskFolder::RegisterTaskDefinition(): This function saves the task and makes it available for execution.
CoUninitialize(): This function un-initializes the COM library, and should be called after the task has been created to release the resources associated with the task scheduler.
It's worth noting that malware can use different methods to create scheduled tasks, including using the command line utility "schtasks.exe" or the graphical user interface of the Task Scheduler, and it can use different triggers such as time-based, event-based or login-based to achieve persistence. Also, malware can use different methods to hide the scheduled tasks it creates, making it harder to detect and remove

how are registry keys used to achieve persistence??

Registry keys can be used by malware to achieve persistence by automatically running the malware at system startup or login. The malware can create a new registry key or modify an existing one, to run the malware at specific events.
Here's an overview of how malware can use registry keys to achieve persistence:
The malware creates a new registry key or modifies an existing one, using the Windows Registry API.

The new registry key is placed in a specific location, such as the Run or RunOnce key, that is executed at system startup or login.
The registry key is configured to run the malware at system startup or login, by specifying the path to the malware executable.
The registry key is saved and activated, so that the malware is automatically run at the next system startup or login.
The malware continues to run in the background, achieving persistence on the system.
It's worth noting that malware can use different methods to create or modify registry keys, including using the command line utility "reg.exe" or the graphical user interface of the Registry Editor, and it can use different keys and values to achieve persistence. Also, malware can use different methods to hide the registry keys it creates, making it harder to detect and remove.

"Run" and "RunOnce" are two specific registry keys located in the Windows Registry that are used to automatically run programs at system startup or login. These keys are located in the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run and HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RunOnce respectively.
The "Run" key is used to specify programs that should be automatically started every time a user logs on to the system. Programs specified in this key will be launched automatically in the background, and will continue to run until the user logs off or the system is shut down.
The "RunOnce" key is used to specify programs that should be automatically started once and then removed from the registry. Programs specified in this key will be launched automatically in the background and will run only once, after which the key and its value will be deleted from the registry.
Both keys can be used by malware to achieve persistence by adding the path to the malware executable in the key value, so the malware will be launched automatically at system startup or login.

Here is the list of APIs that malware can use to achieve persistence using registry keys, in the correct order:
RegCreateKeyEx(): This function creates a new registry key or opens an existing one, and returns a handle to the key that can be used to access and modify its values.
RegSetValueEx(): This function sets the value of a specified registry key, and can be used to specify the path to the malware executable, which will be run at system startup or login.
RegCloseKey(): This function closes a handle to a registry key, and should be called after a key has been modified or created to release the resources associated with the key.
RegQueryValueEx(): This function retrieves the value of a specified registry key, and can be used by the malware to check whether a specific key and value have been created or modified.
RegDeleteKeyEx(): This function deletes a registry key and its subkeys, and can be used by malware to remove any traces of itself from the registry.
It's worth noting that malware can use different methods to create or modify registry keys, including using the command line utility "reg.exe" or the graphical user interface of the Registry Editor, and it can use different keys and values to achieve persistence. Also, malware can use different methods to hide the registry keys it creates, making it harder to detect and remove.

Startup folders:
Startup folders can be used by malware to achieve persistence by automatically running the malware at system startup. The malware can create a new shortcut in the startup folder or modify an existing one, to run the malware at system startup.
Here's an overview of how malware can use Startup Folders to achieve persistence:
The malware creates a new shortcut or modifies an existing one, using the Windows Shell API.
The shortcut is placed in one of the system startup folders, such as the All Users Startup folder or the Current User Startup folder.
The shortcut is configured to run the malware at system startup, by specifying the path to the malware executable.
The system startup folder is saved and activated, so that the malware is automatically run at the next system startup.
The malware continues to run in the background, achieving persistence on the system.
It's worth noting that malware can use different methods to create or modify the shortcut, such as using the command line utility "mklink" or the graphical user interface of the Windows File Explorer, and it can use different startup folders to achieve persistence. Also, malware can use different methods to hide the shortcut it creates, making it harder to detect and remove.

Here is the list of APIs that malware can use to achieve persistence using Startup Folders, in the correct order:
SHGetSpecialFolderPath(): This function retrieves the path of a special folder, such as the All Users Startup folder or the Current User Startup folder.
CreateFile(): This function creates or opens a file, and can be used to create a new shortcut file in the Startup folder.
WriteFile(): This function writes data to a file, and can be used to write the contents of the shortcut file, which includes the path to the malware executable.
CloseHandle(): This function closes a handle to a file, and should be called after the shortcut file has been created or modified to release the resources associated with the file.
ShellExecute(): This function starts an application and can be used to run the malware executable at system startup.
It's worth noting that malware can use different methods to create or modify the shortcut, such as using the command line utility "mklink" or the graphical user interface of the Windows File Explorer, and it can use different startup folders to achieve persistence. Also, malware can use different methods to hide the shortcut it creates, making it harder to detect and remove.

Services:
Services can be used by malware to achieve persistence by automatically running the malware in the background, even if the user is not logged in. Malware can create a new service or modify an existing one, to run the malware at system startup.
Here's an overview of how malware can use services to achieve persistence:
The malware creates a new service or modifies an existing one, using the Windows Service Control Manager (SCM) API.
The service is configured to run the malware executable at system startup.
The service is started, so that the malware is automatically run at the next system startup.
The malware continues to run in the background, achieving persistence on the system.
It's worth noting that malware can use different methods to create or modify services, such as using the command line utility "sc.exe" or the graphical user interface of the Services Manager, and it can use different methods to hide the service it creates, making it harder to detect and remove.
The malware can use the following APIs in the correct order to achieve persistence using services:
OpenSCManager(): This function opens the service control manager and returns a handle to it.
CreateService(): This function creates a new service, and returns a handle to the service.
StartService(): This function starts a service, which launches the malware executable.
CloseServiceHandle(): This function closes a handle to a service or service control manager, and should be called after a service has been created or modified to release the resources associated with the service.
DeleteService(): This function deletes a service, and can be used by malware to remove any traces of itself from the system.

DLL injection:

DLL injection is a technique that can be used by malware to achieve persistence by injecting a malicious DLL into a legitimate process. The malware can use the injected DLL to run code in the context of the legitimate process, which can help it evade detection and achieve persistence on the system.
Here's an overview of how malware can use DLL injection to achieve persistence:
The malware identifies a running process that it can inject the DLL into. This can be done using the Windows API EnumProcesses() and EnumProcessModules() functions.
The malware allocates memory in the target process using the VirtualAllocEx() function.
The malware copies the malicious DLL into the memory allocated in step 2, using the WriteProcessMemory() function.
The malware creates a remote thread in the target process, using the CreateRemoteThread() function. The remote thread is used to run the malicious DLL code.
The malware can use the WaitForSingleObject() function to wait for the thread to finish and use the GetExitCodeThread() function to check the return value of the thread.
The malware continues to run in the background, achieving persistence on the system.
It's worth noting that malware can use different methods to inject DLL, such as using the functions LoadLibrary() or LoadLibraryEx() to inject DLL directly into the process address space, or using the function CreateRemoteThread() to create a thread that calls the LoadLibrary() function, in order to achieve persistence. Also, malware can use different methods to hide the DLL it injects, making it harder to detect and remove.

Registry APIs
RegCreateKeyEx
RegSetValueEx
RegGetValue

File I/O APIs:
CreateFile
ReadFile
WriteFile

How to create a process in memory itself? (this is a malware with file, using Download + Execute (multi stage malware))
CreateProcess/A/W

How to do file less persistence ?
Fileless persistence is a technique used by malware to achieve persistence on a compromised system without writing any files to disk. Instead, the malware stores its payload, configuration, and other data in memory, making it more difficult for security software to detect and remove.
Here are a few methods that malware can use to achieve fileless persistence:
Registry keys: The malware can create or modify registry keys to execute its payload every time the system starts. For example, it can create a new Run or RunOnce key in the Windows registry to execute its payload when the system boots up.
Scheduled tasks: The malware can create a scheduled task that runs at a specific time or interval, or when certain events occur, such as when a user logs in. This can be used to execute the malware's payload and to ensure that it continues to run even if the initial infection is removed.
WMI events: The malware can use the Windows Management Instrumentation (WMI) service to create event filters and consumers that can be used to execute its payload when certain events occur on the system, such as when a user logs in or when a specific process is started.
Alternate Data Streams: The malware can use the NTFS file system's alternate data streams feature to store its payload and configuration in a hidden stream that is attached to a legitimate file. This allows the malware to persist on the system without creating any new files.
Memory Injection: The malware can use techniques such as process injection or DLL injection to load its payload into the memory of a legitimate process. Once the payload is running in memory, the malware can use various methods to maintain persistence, such as creating a new thread or modifying the memory of the process to redirect execution to the malware's payload.
It's important to note that these are just a few examples of how malware can achieve fileless persistence

Normal method with file
DLL injection
APIs:
VItualAlloc
RtlMoveMemory
VirtualProtect
CreateRemoteThread
LoadLibrary

Fileless persistance:

- Reflective DLL injection
  APIs:
  VItualAlloc
  RtlMoveMemory
  VirtualProtect
  CreateRemoteThread
  VirtualProtect(many calls would be visible)

Some Malwares use Shell:
Reverse/foreard shell. Reverse is used to bypass firewall

- Shell Malware:
  You would find the following APIs in almost same order
  CreateProcess()
  Network I/O
  CreateProcess()
  Network I/O

- Event/Keyboard Capture Malwares:
  APIs:
- Keyboard
  GetAsyncKeyState (but this is very common in many malware so blocked by AV mostly). This is genuinely used for applications which use HOT keys (hidden apps which opens when you press certain keys)
  Instead you may use:
  SetWIndowsHook
  Or
  DLL Injection(very old and common)

  Key logger malware is a type of malware that is designed to record keystrokes on a compromised system. There are several Windows API functions that can be used by key logger malware to achieve this:
  GetAsyncKeyState(): This function can be used to determine the status of a specific key, whether it is pressed or released. This can be used by malware to record all keystrokes on the system.
  GetKeyboardState(): This function can be used to retrieve the state of all keys on the keyboard, including the state of the modifier keys (Ctrl, Alt, Shift, etc.). This can be used by malware to record all keystrokes and the state of the modifier keys.
  SetWindowsHookEx(): This function can be used to install a hook procedure, which can be used to monitor keystrokes and other events in the system. Malware can use this function to install a hook procedure that records all keystrokes on the system.
  GetForegroundWindow(): This function can be used to retrieve the handle of the foreground window, which is the window that currently has focus. This can be used by malware to determine the active window and record keystrokes only when a specific window is active.
  MapVirtualKey(): This function can be used to translate the virtual key code of a keystroke into its corresponding character or ASCII value. This can be used by malware to record the character or ASCII value of a keystroke, rather than the virtual key code.
  It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions, such as exfiltrating the recorded keystrokes to a command and control server or using the recorded information to perform other malicious activities.

- Mouse
  SetWindowsHookEx()

- Clipboard

- Screen capture:

- Webcam Capture:

- Audio Capture:

- File Modification Capture:
  DLL injection (easily detected)

  File modification tracking malware is a type of malware that is designed to monitor and track changes to files on a compromised system. Here are a few Windows API functions that can be used by file modification tracking malware to achieve this:
  CreateFile(): This function can be used to open or create a file, and it can be used by malware to open files for monitoring or to create new files.
  ReadFile(): This function can be used to read data from a file, and it can be used by malware to read the contents of a file before and after it is modified.
  WriteFile(): This function can be used to write data to a file, and it can be used by malware to

write new data to a file.
SetFilePointer(): This function can be used to move the file pointer to a specific location in a file, and it can be used by malware to move to specific location in a file before and after it is modified.
FindFirstChangeNotification() and FindNextChangeNotification(): These functions can be used to monitor changes to a specified directory and its subdirectories, and it can be used by malware to track changes to files within a specific folder.
GetFileAttributes(): This function can be used to retrieve file attributes, and it can be used by malware to check if a file has been modified.

It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions, such as exfiltrating the recorded information to a command and control server or using the recorded information to perform other malicious activities.

May Also use:
Event Tracing for windows(ETW)

- Process Monitoring Malware:

Covering Tracks:

75. File Hiding
- Alternated Data stream
- Rootkit (turla malware)
- Filter Driver

76. Hide Network activity
- Https (TLS/TCP)
- Protocol abuse (to identify protocol abuse, some indicators are: traffic volume high, IP repetition)

77. Anti reversing
78. Anti debugging
79. Process Spoof

Process spoofing is a technique used by malware to hide its presence on a compromised system by disguising its process as a legitimate process. Here are a few techniques that malware can use to achieve process spoofing:

Process hollowing: Malware can create a new process with the same name and properties as a legitimate process, then replace the legitimate process's code with its own malicious code.
Process injection: Malware can inject its code into a legitimate process, allowing it to execute alongside the legitimate process and hide its presence.
DLL injection: Malware can inject a malicious DLL into a legitimate process, allowing it to execute its code alongside the legitimate process and hide its presence.
Process replacement: Malware can terminate a legitimate process and replace it with its own malicious process, disguising its presence as the legitimate process.
Process doppelganging: Malware can create a new process that closely mimics the behavior and characteristics of a legitimate process.
Thread injection: Malware can inject its code into a legitimate process by creating a new thread within the process and running the malicious code on that thread.

It's important to note that these are just a few examples of process spoofing techniques that malware can use, and malware authors are continually developing new methods to evade detection. Also, it's important to keep in mind that the use of process spoofing by malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions, such as disguising its malicious activities and making it difficult for security software to detect and remove the malware.

- Legit name with malicious process
- Process hollowing (genuine case is used in debugger)
  - Create process (in suspended mode)
  - Unmap memory
  - Map memory
  - Write into process memory
  - Resume thread

- Anti Reversing/ anti debugging (APTs)
  - APTs require to be persistent for a very long time. So use anti-reverse and anti-debugging
  - State sponsored malware
  - Pegasus (implemented turing machine in image viewer)
  - API used: time read and time compare (basically a precalculated time is set and if executed in debugger with breaks then the time taken is more, so this time check may be implmented to anti-debug)
  - Isdebuggerpresent()
  - Process enumaration
  - NtQueryInformationProcess
  - Self Debuggjng (if a process is already in debugger then you cant debug it in another debugger simultaneously)
    - Uses windows Debug apis
    - Create 02 process
    - Parent process is debugger
    - Attach with child process (debugging process)
    - https://learn.microsoft.com/en-us/windows/win32/debug/debugging-functions
  - Compress code
    - pack with UPX
    - Process hollowing
  - Encrypt Code:
    - encrypt
    - Process hollowing – createproces, map, unmap, write, resume
  - Themida/ Vmprotect
    - It basically runs the processes in a virtual machine
    - This hence doesn't let you debug and analyse(because you don't get control since it is in VM)
  - Break point ??? Virtual protect (check for RX & RWX ???????)
  - Hypervisor assisted malware + CPU/ISP emulation [used in pegasus(ios)]
  - Breakpoint:
    - Check every instruction for first byte 0xCC(means brakpoint is set)
    - Now after finding brakpoints
    - Change memory permissions (virtualprotect API)
    - Give Read, write & Execution (do not do it in obvious order to make it suspicious)
    - Remove break points
    - Now that malware has removed the break point, how will you analsye. Set Hardware breakpont !!!
    - Now, how to bypass that
    - Scan SCH register
      Some possibilities (out of many):
      1. The program may call SetThreadContext, or ZwContinue which sets all registers using a new CONTEXT. If the DRx registers are reset then your hardware breakpoints are removed.
      2. The program calls ZwSetInformationThread with the 0x11 (ThreadHideFromDebugger) parameter. If the thread is hidden from the debugger then breakpoint exceptions will not be caught by a debugger but will be passed to the program, if unhandled - it will crash.

  - Anti-patching:
    - Same as software breakpoint ????

HOw does DLL get injected in a process??
createRemoteThread
VirtualAlloc (reserves memory in heap)
CreateRemoteThread -> LoadLibrary->this calls the DLL main

Reflective DLL
VirtualAlloc (reserves memory in heap)
CreateRemoteThread
WriteProcessmemory-> then load the DLL in Heap

Why there is Zw & Nt & Rtl prefix on some windows APIs? What is the difference between Zw and Nt and Rtl.

In the Windows API, there are three main prefixes used in function names: "Rtl", "Zw", and "Nt". These prefixes indicate the level of access and functionality provided by the function.

- "Rtl" stands for "Run Time Library" and is a set of functions that provide core operating system services for user-mode programs. These functions typically have a higher level of abstraction than "Nt" functions, and provide a more convenient and safer way to perform common tasks, such as memory allocation and string manipulation.
- "Zw" stands for "Windows NT executive" and is a set of functions that provide low-level access to the Windows NT operating system. These functions provide direct access to the kernel and other system-level components, and are typically used by device drivers and other system-level programs. Zw functions are less abstract than Rtl functions, and can be more powerful, but also more risky to use.
- "Nt" stands for "New Technology" and is a set of functions that provide an interface to the Windows NT operating system. These functions provide the same level of access and functionality as "Zw" functions, but are typically more stable and better suited for use by user-mode programs.

In general, Rtl functions should be used by user-mode programs, Zw functions are intended for kernel-mode (system-level) use and Nt functions can be used in both user-mode and kernel-mode. Some of the differences between these functions are that Rtl functions are more abstract and safer, Zw functions are less abstract and more powerful but also more risky, and Nt functions are a balance between Rtl and Zw functions in terms of abstraction and risk.

Most Used APIs:

DialogBoxParamA

The DialogBoxParam function is used to create a modal dialog box. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in four parameters:

hInstance: Handle to the instance of the module that contains the dialog box template.
lpTemplate: Pointer to a null-terminated string that specifies the dialog box template or the resource identifier of the dialog box template.
hWndParent: Handle to the window that owns the dialog box.
lpDialogProc: Pointer to the dialog box procedure. It returns an integer value, the value of the nResult parameter specified in the call to the EndDialog function used to terminate the dialog box.

GetDlgItem

The GetDlgItem function retrieves a handle to a control in a dialog box. The function takes in two parameters:

hDlg: Handle to the dialog box that contains the control.
nIDDlgItem: Identifier of the control to be retrieved. It returns a handle to the specified control if successful, otherwise it returns NULL. The handle can be used to manipulate the control, for example, by changing its text or position.

GetDlgItemInt

The GetDlgItemInt function retrieves the integer value of a specified control in a dialog box. The function takes in two parameters:

hDlg: Handle to the dialog box that contains the control.
nIDDlgItem: Identifier of the control to be retrieved. It returns the integer value of the control if successful, otherwise it returns zero. The control must be an edit control, a list box, or a combo box containing an edit control. The function retrieves the text of the control and converts it to an integer using the function atoi(). If the function fails, it sets the error code to ERROR_INVALID_PARAMETER.

It's important to note that, this function do not check if the value is in range or not, it just converts the string to an integer.

GetDlgTextA

The GetDlgItemTextA function retrieves the text of a specified control in a dialog box. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in three parameters:

hDlg: Handle to the dialog box that contains the control.
nIDDlgItem: Identifier of the control to be retrieved.
lpString: Pointer to a buffer that will receive the text of the control.
cchMax: Maximum number of characters to be copied to the buffer. It returns the number of characters copied to the buffer if successful, otherwise it returns zero. The function retrieves the text of the control and copies it to the specified buffer. If the text is longer than the maximum number of characters specified, it is truncated. It's important to note that the function does not include the null terminator in the count of characters copied to the buffer.

GetWindowTextA

The GetWindowTextA function retrieves the text of the specified window's title bar (if it has one). The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in three parameters:

hWnd: Handle to the window or control whose text is to be retrieved.
lpString: Pointer to a buffer that will receive the text of the window.
nMaxCount: Maximum number of characters to be copied to the buffer. It returns the number of characters copied to the buffer if successful, otherwise it returns zero.

The function retrieves the text of the specified window's title bar and copies it to the specified buffer. If the text is longer than the maximum number of characters specified, it is truncated. It's important to note that the function does not include the null terminator in the count of characters copied to the buffer.

It can be used on any window that has a title bar and also used on controls that have a text, like buttons, edit boxes, etc.

GetwindowWord

The GetWindowWord function retrieves the specified 16-bit value at the specified offset into the extra window memory of a specified window. The function takes in two parameters:

hWnd: Handle to the window whose 16-bit value is to be retrieved.
nIndex: Specifies the zero-based offset of the value to be retrieved. It returns the requested 16-bit value if successful, otherwise it returns zero.

This function is used to retrieve the value of a specific 16-bit value stored in the extra window memory of a window. The value is retrieved from the specified offset of the extra window memory block. The nIndex parameter specifies the offset of the 16-bit value to be retrieved. It's important to note that, Extra Window memory is a memory block associated with a window, which can be used to store additional data associated with the window.

LoadStringA

The LoadStringA function retrieves a string from the specified string table resource in a specified module. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in four parameters:

hInstance: Handle to the instance of the module whose executable file contains the string table resource.
uID: Identifier of the string to be retrieved.
lpBuffer: Pointer to the buffer that will receive the string.
cchBufferMax: Maximum number of characters to be copied to the buffer. It returns the number of characters copied to the buffer if successful, otherwise it returns zero.

This function is used to load a string from a resource-only module or from the module that contains the current executable file. The string is identified by its unique identifier in the string table resource. The function copies the string to the specified buffer, truncating it if necessary to fit the maximum number of characters specified. It's important to note that the function does not include the null terminator in the count of characters copied to the buffer.

lstrcmpA

The lstrcmpA function compares two null-terminated strings. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in two parameters:

lpString1: Pointer to the first null-terminated string to be compared.
lpString2: Pointer to the second null-terminated string to be compared.

It returns an integer value indicating the lexicographic relation between the strings:

- A value less than zero indicates that the first string is less than the second.
- A value of zero indicates that the two strings are equal.
- A value greater than zero indicates that the first string is greater than the second.

This function compares two null-terminated strings and returns an integer indicating the lexicographic relation between the two strings. The function uses the ASCII character set for the comparison. It can be used to compare two string for sorting, searching or any other purpose.
It's important to note that, the function is case-sensitive, so it will differentiate uppercase and lowercase characters. If you want a case-insensitive comparison use lstrcmpiA.

wsprintfA or wsprintfW (A is ASCII & W is Wide Character Set)

The wsprintfA function formats and stores a series of characters and values in a buffer. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in two parameters:

lpOut: Pointer to the buffer that will receive the formatted output.
lpFmt: Pointer to a null-terminated format string.
... : A variable number of additional arguments, the number and type of which depend on the contents of the format string.

It returns the number of characters stored in the output buffer if successful, otherwise it returns a negative value.
This function is similar to the printf function in C, it formats a series of characters and values and stores the result in a buffer. The format string contains placeholders for the values, which are replaced by the corresponding arguments. The function uses the ASCII character set for the output. It's important to note that, this function is considered to be insecure and it's not recommended to use it in new code, instead use the safe version of this function such as snprintf_s.

The _snwprintf_s function is a "safe" version of the wsprintfW function. It formats and stores a series of characters and values in a buffer, ensuring that the operation does not result in a buffer overflow. The function takes in four parameters:

buffer: Pointer to the buffer that will receive the formatted output.
sizeOfBuffer: Size of the buffer, in characters.
format: Pointer to a null-terminated format string.
... : A variable number of additional arguments, the number and type of which depend on the contents of the format string. It returns the number of characters stored in the output buffer if successful, otherwise it returns a negative value.

This function is similar to the wsprintfW function, it formats a series of characters and values and stores the result in a buffer. The format string contains placeholders for the values, which are replaced by the corresponding arguments. The function uses the wide character set (Unicode) for the output.
The _snwprintf_s function guarantees that the output will not exceed the size of the buffer, preventing buffer overflow. If the formatted output would exceed the size of the buffer, the function will return -1, set errno to EINVAL and the buffer will be set to an empty string. It's a secure version of the wsprintfW function and it is recommended to use it in new code.

MessageBeep

The MessageBeep function plays a waveform sound. The function takes in one parameter:

uType: Specifies the sound to be played. It can be one of the following values:
  - MB_OK: Plays the OK sound
  - MB_ICONASTERISK: Plays the asterisk sound
  - MB_ICONEXCLAMATION: Plays the exclamation sound
  - MB_ICONHAND: Plays the hand sound
  - MB_ICONQUESTION: Plays the question sound
  - MB_ICONWARNING: Plays the warning sound
  - MB_ICONERROR: Plays the error sound
  - MB_ICONINFORMATION: Plays the information sound
  - MB_ICONSTOP: Plays the stop sound
  - MB_OKCANCEL: Plays the OK sound
  - MB_RETRYCANCEL: Plays the retry sound
  - MB_YESNO: Plays the question sound
  - MB_YESNOCANCEL: Plays the question sound
  - MB_ABORTRETRYIGNORE: Plays the hand sound

It returns a nonzero value if successful, otherwise it returns zero.
The MessageBeep function plays a waveform sound specified by the uType parameter. The function can be used to play a sound to indicate the completion of an operation or to provide a warning or error message. It's important to note that, the sound played by the MessageBeep function is determined by the current sound scheme.

MessageBoxA

The MessageBoxA function creates, displays, and operates a message box. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in four parameters:

hWnd: Handle to the owner window of the message box to be created.
lpText: Pointer to a null-terminated string that contains the message to be displayed.
lpCaption: Pointer to a null-terminated string that contains the message box title.
uType: Specifies the contents and behavior of the message box. It can be a combination of flags from the following list:
  - MB_ABORTRETRYIGNORE
  - MB_CANCELTRYCONTINUE
  - MB_HELP
  - MB_OK
  - MB_OKCANCEL
  - MB_RETRYCANCEL
  - MB_YESNO
  - MB_YESNOCANCEL

It returns an integer value indicating the user's action:

- IDABORT, IDCANCEL, IDIGNORE, IDNO, IDOK, IDRETRY, IDYES.

The MessageBoxA function creates, displays, and operates a message box. The message box contains a message and an icon, and it can have one or more buttons. The function uses the ASCII character set for the message and title strings. The message box is modal, meaning that it blocks input to other windows in the same thread while it is displayed.

MessageBoxExA

The MessageBoxExA function creates, displays, and operates a message box. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in five parameters:

hWnd: Handle to the owner window of the message box to be created.
lpText: Pointer to a null-terminated string that contains the message to be displayed.
lpCaption: Pointer to a null-terminated string that contains the message box title.
uType: Specifies the contents and behavior of the message box. It can be a combination of flags from the following list:
  - MB_ABORTRETRYIGNORE
  - MB_CANCELTRYCONTINUE
  - MB_HELP
  - MB_OK
  - MB_OKCANCEL
  - MB_RETRYCANCEL
  - MB_YESNO
  - MB_YESNOCANCEL

wLanguageId: Specifies the language identifier to be used for the message box.

It returns an integer value indicating the user's action:

- IDABORT, IDCANCEL, IDIGNORE, IDNO, IDOK, IDRETRY, IDYES.

The MessageBoxExA function is similar to the MessageBoxA function, but with the additional parameter

wLanguageId that allows to specify the language identifier to be used for the message box. This allows to display message box in a specific language, regardless of the language of the operating system or the current user locale.

SendMessageA

The SendMessageA function sends the specified message to a window or windows. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in four parameters:

hWnd: Handle to the window whose window procedure will receive the message.
Msg: Specifies the message to be sent.
wParam: Specifies additional message-specific information.
lParam: Specifies additional message-specific information.

It returns a LRESULT value that depends on the message sent.
The SendMessageA function sends the specified message to the window procedure of the specified window. The message is typically processed by the window procedure, which may or may not return a value. The function uses the ASCII character set for any text-related messages. It's important to note that, This function is synchronous and it will not return until the window procedure has processed the message. This can cause a deadlock if the process that calls SendMessageA also owns the window being sent the message.

SendDlgItemMessageA

The SendDlgItemMessageA function sends a message to the specified control in a dialog box. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in five parameters:

hDlg: Handle to the dialog box that contains the control.
nIDDlgItem: Specifies the identifier of the control to receive the message.
Msg: Specifies the message to be sent.
wParam: Specifies additional message-specific information.
lParam: Specifies additional message-specific information.

It returns a LRESULT value that depends on the message sent.
The SendDlgItemMessageA function sends the specified message to the specified control in the specified dialog box. This function is similar to the SendMessageA function, but it allows you to send a message to a control in a dialog box directly, rather than sending the message to the dialog box and then having the dialog box forward the message to the control. It's important to note that, this function is synchronous and it will not return until the control has processed the message.

ReadFile

The ReadFile function reads data from a file or input/output (I/O) device. The function takes in the following parameters:

hFile: A handle to the file or I/O device to be read.
lpBuffer: A pointer to the buffer that will receive the read data.
nNumberOfBytesToRead: The number of bytes to be read from the file or I/O device.
lpNumberOfBytesRead: A pointer to a variable that receives the number of bytes read.
lpOverlapped: A pointer to an OVERLAPPED structure, which is used for asynchronous operations.

It returns a nonzero value if successful, otherwise it returns zero and the error code can be retrieved by calling GetLastError.
The ReadFile function reads data from a file or I/O device, and stores it in the buffer pointed to by lpBuffer. The function can read data synchronously or asynchronously, depending on whether the file handle was created with the FILE_FLAG_OVERLAPPED flag. It's important to note that, if the file handle was not created with the FILE_FLAG_OVERLAPPED flag, the lpOverlapped parameter must be set to NULL and the read operation will be synchronous. If the file handle was created with the FILE_FLAG_OVERLAPPED flag, the read operation will be asynchronous and lpOverlapped must point to a valid OVERLAPPED structure.

WriteFile

The WriteFile function writes data to a file or output/input (O/I) device. The function takes in the following parameters:

hFile: A handle to the file or O/I device to be written to.
lpBuffer: A pointer to the buffer that contains the data to be written.
nNumberOfBytesToWrite: The number of bytes to be written to the file or O/I device.
lpNumberOfBytesWritten: A pointer to a variable that receives the number of bytes written.
lpOverlapped: A pointer to an OVERLAPPED structure, which is used for asynchronous operations.

It returns a nonzero value if successful, otherwise it returns zero and the error code can be retrieved by calling GetLastError.
The WriteFile function writes data from a buffer to a file or O/I device. The function can write data synchronously or asynchronously, depending on whether the file handle was created with the FILE_FLAG_OVERLAPPED flag. It's important to note that, if the file handle was not created with the FILE_FLAG_OVERLAPPED flag, the lpOverlapped parameter must be set to NULL and the write operation will be synchronous. If the file handle was created with the FILE_FLAG_OVERLAPPED flag, the write operation will be asynchronous and lpOverlapped must point to a valid OVERLAPPED structure. Additionally, it's important to check the return value of the function to ensure the successful completion of the write operation.

CreateFileA

The CreateFileA function creates or opens a file or I/O device. The "A" at the end of the function name indicates that the function uses the ASCII character set. The function takes in the following parameters:

lpFileName: A pointer to a null-terminated string that specifies the name of the file or I/O device to be created or opened.
dwDesiredAccess: Specifies the requested access to the file or I/O device, which can be one or a combination of the following values:
  - GENERIC_READ: Enables read access to the file or I/O device.
  - GENERIC_WRITE: Enables write access to the file or I/O device.

dwShareMode: Specifies the sharing mode of the file or I/O device, which can be one or a combination of the following values:
  - FILE_SHARE_READ: Enables subsequent open operations on the file to request read access.
  - FILE_SHARE_WRITE: Enables subsequent open operations on the file to request write access.

lpSecurityAttributes: A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes.
dwCreationDisposition: Specifies the action to take if the file already exists or if the file does not exist, it can be one of the following values:
  - CREATE_NEW: Creates a new file, fails if the specified file already exists.
  - CREATE_ALWAYS: Creates a new file, overwrites the specified file if it already exists.
  - OPEN_EXISTING: Opens the specified file, fails if the specified file does not exist.
  - OPEN_ALWAYS: Opens the specified file, creates a new file if the specified file does not exist.

dwFlagsAndAttributes: Specifies the file attributes and flags for the file, it can be one or a combination of the following values:
  - FILE_ATTRIBUTE_ARCHIVE: The file should be archived.

hTemplateFile:

GetPrivateProfileIntA

GetPrivateProfileIntA is a function in the Windows API that retrieves an integer value from a specified section and key in an initialization file (INI file).
The function takes three parameters:

- lpAppName: A pointer to a null-terminated string that specifies the name of the section in the INI file.
- lpKeyName: A pointer to a null-terminated string that specifies the name of the key whose value is to be retrieved.
- nDefault: The default value to be returned if the specified key is not found in the INI file or if the value of the key is not a valid integer.

The function returns the value of the specified key as an integer. If the key is not found or if the value of the key is not a valid integer, the function returns the default value specified in the nDefault parameter. The A at the end of the function name indicates that the function uses ANSI character set. The similar function GetPrivateProfileIntW uses the wide-character (Unicode) set.

WritePrivatesProfileStringA

WritePrivateProfileStringA is a function in the Windows API that writes a string to a specified section and key in an initialization file (INI file).
The function takes four parameters:

- lpAppName: A pointer to a null-terminated string that specifies the name of the section in the INI file.
- lpKeyName: A pointer to a null-terminated string that specifies the name of the key whose value is to be set.
- lpString: A pointer to a null-terminated string that contains the value to be written.
- lpFileName: A pointer to a null-terminated string that specifies the name of the INI file.

The function returns a nonzero value if successful, or zero if an error occurred.
This function can be used to add, modify or delete a key-value pair in an INI file. If the key does not exist and lpString is not NULL, the key is created and the value is set. If the key does exist and lpString is not NULL, the value is set. If lpString is NULL, the key is deleted.
The A at the end of the function name indicates that the function uses ANSI character set. The similar function WritePrivateProfileStringW uses the wide-character (Unicode) set.

GetPrivateProfileStringA

GetPrivateProfileStringA is a function in the Windows API that retrieves a string value from a specified section and key in an initialization file (INI file).
The function takes four parameters:

- lpAppName: A pointer to a null-terminated string that specifies the name of the section in the INI file.
- lpKeyName: A pointer to a null-terminated string that specifies the name of the key whose value is to be retrieved.
- lpDefault: A pointer to a null-terminated string that specifies the default value to be returned if the specified key is not found in the INI file.
- lpReturnedString: A pointer to a buffer that receives the retrieved string.
- nSize : Specifies the size of the buffer pointed to by the lpReturnedString parameter, in TCHARs.

The function returns the number of characters copied to the buffer, not including the terminating null character.
The A at the end of the function name indicates that the function uses ANSI character set. The similar function GetPrivateProfileStringW uses the wide-character (Unicode) set.

Xxx___XXXX important

GetDlgItemTextA
GetWindowTextA
lstrcmpA
GetPrivateProfileStringA
GetPrivateProfileIntA
RegQueryValueExa
WritePrivateProfileStringA
GetPrivateProfileIntA

Malware APIs

CreateProcessInternal() Eg: Remcos RAT

In malware, CreateProcessInternal() can be used to create new processes that can be used to perform malicious actions. For example, malware may use this function to create a new process that runs in the background and continues to execute malicious code even if the original malware is detected and removed. Additionally, malware can use this function to create new processes that impersonate legitimate processes, making it more difficult for security software to detect the malware. Moreover, malware can use this function to create new processes with elevated privileges, allowing it to perform actions that would otherwise be restricted. It's important to note that the use of CreateProcessInternal() in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

VirtualAlloc()

VirtualAlloc() is a Windows API function that can be used to reserve or commit memory in a process's virtual address space. In malware, this function can be used to allocate memory for malicious code, data, or other purposes. For example, malware may use this function to allocate memory for a payload that will be injected into another process, or to create a new memory space where it can execute code without being detected.
Malware can use VirtualAlloc() to allocate memory in a way that makes it difficult for security software to detect the malware. For example, it can use this function to allocate memory in a non-executable region, and then use a technique called "memory patching" to change the memory's permissions so that it can execute code. Additionally, malware can use VirtualAlloc() to allocate memory in a way that makes it difficult for security software to analyze the malware's behavior. For example, it can use this function to create multiple memory regions with different permissions, making it more difficult for security software to identify which memory regions contain the malware's code or data.
It's important to note that the use of VirtualAlloc() in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

An example of how malware might use the VirtualAlloc() function is as follows:

The malware uses VirtualAlloc() to allocate a block of memory in the target process's address space. This memory will be used to store the malware's payload, which contains the malicious code that the malware will execute.
The malware uses WriteProcessMemory() to write the payload into the allocated memory block.
The malware uses CreateRemoteThread() to create a new thread in the target process that starts execution at the beginning of the payload. This causes the payload to execute within the context of the target process, allowing the malware to perform its malicious actions.
The malware can use VirtualAlloc() to allocate more memory for storing data, config, communication with C&C server or as a place to unpack payloads.
The malware can use VirtualProtect() to change the memory protection of the allocated memory block, allowing it to execute code from the allocated memory, or making it harder for security software to detect the payload.

It's important to note that this is just one example of how malware might use VirtualAlloc(), and different malware may use this function in different ways. Additionally, malware can use other memory allocation functions to achieve similar goals.

VirtualFree()

VirtualAllocEx()

VirtualAllocEx() is a variation of the VirtualAlloc() function that allows you to allocate memory in the address space of a specific process, rather than in the address space of the calling process.
In malware, VirtualAllocEx() can be used to allocate memory in the address space of a target process, which can then be used to inject malware into that process. For example, malware might use VirtualAllocEx() to allocate memory in the address space of a system process, such as "explorer.exe", and then use WriteProcessMemory() to copy the malware's payload into that memory. After that, it can use CreateRemoteThread() to create a new thread in the target process that starts execution at the beginning of the payload, thus executing the malware within the context of the target process.
Additionally, malware can use VirtualAllocEx() to allocate memory in the address space of a specific process as a place to hide, for example, its config, data, or other information.
It's important to note that this is just one example of how malware might use VirtualAllocEx(), and different malware may use this function in different ways. Additionally, malware can use other memory allocation functions to achieve similar goals.

VirtualAllocEx() | ZwProtectVirtualMemory()

VirtualAllocEx() is a Windows API function that allows you to allocate memory in the address space of a specific process. It can be used to reserve or commit memory in the virtual address space of the specified process, and it can also be used to change the memory protection of the allocated memory.
ZwProtectVirtualMemory() is a Windows API function that allows you to change the memory protection of a region of memory in the virtual address space of a process. It can be used to make a region of memory read-only, read-write, or execute-only, and it can also be used to mark a region of memory as non-executable.
Both functions can be used by malware to hide their code and data from security software. For example, malware can use VirtualAllocEx() to allocate memory in the address space of a target process and use ZwProtectVirtualMemory() to change the memory protection of the allocated memory so that it is non-executable. This makes it harder for security software to detect the malware's code and data, as it is stored in a non-executable memory region.
Additionally, malware can use VirtualAllocEx() to allocate memory in the address space of a specific process and use ZwProtectVirtualMemory() to change the memory protection of the allocated memory to execute-only, this way malware code can execute from that memory region.
It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

WriteProcessMemory() | NtWriteProcessMemory()

WriteProcessMemory() and NtWriteProcessMemory() are both Windows API functions that allow you to write data to the memory of a specified process.
WriteProcessMemory() is a user-mode function that is provided by the Win32 API. It can be used by user-mode programs to write data to the memory of another process. It is relatively safe to use and typically used by user-mode programs such as malware.
NtWriteProcessMemory() is a kernel-mode function that is provided by the Windows NT API. It can be used by kernel-mode programs to write data to the memory of another process. It provides a lower level of abstraction and more direct access to the memory of the target process. It is typically used by device drivers and other system-level programs.
Both functions can be used by malware to inject malicious code or data into the memory of a target process, and both functions are used by malware to perform process hollowing.
It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

ResumeThread() | NtResumeThread()

ResumeThread() and NtResumeThread() are both Windows API functions that allow you to resume execution of a thread that has been previously suspended.
ResumeThread() is a user-mode function that is provided by the Win32 API. It can be used by user-mode programs to resume execution of a thread that has been previously suspended. It is relatively safe to use and typically used by user-mode programs.
NtResumeThread() is a kernel-mode function that is provided by the Windows NT API. It can be used by kernel-mode programs to resume execution of a thread that has been previously suspended. It provides a lower level of abstraction and more direct access to the target thread. It is typically used by device drivers and other system-level programs.
Both functions can be used by malware to resume execution of a thread that has been previously suspended by the malware, for example to resume execution of a thread that has been created by the malware using CreateRemoteThread() or NtCreateThreadEx() in order to execute malicious code within the context of the target process.
It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

CryptDecrypt() | RtlDecompressBuffer()

CryptDecrypt() and RtlDecompressBuffer() are both Windows API functions, but they serve different purposes.
CryptDecrypt() is a function that belongs to the Microsoft Cryptographic API and can be used to decrypt data that has been previously encrypted using the CryptEncrypt() function or another encryption algorithm. This function can be used to decrypt sensitive data such as passwords, credit card numbers, or confidential files. It can also be used by malware to decrypt its own encrypted payload or configuration data.
RtlDecompressBuffer() is a function that belongs to the Run-Time Library (RTL) of the Windows NT operating system and can be used to decompress data that has been previously compressed using the RtlCompressBuffer() function or another compression algorithm. This function can be used to decompress data to reduce its size for storage or transmission. Malware can use this function to decompress payloads or configuration data that it has previously compressed in order to evade detection or analysis by security software.
It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

NtCreateSection() + MapViewOfSection() | ZwMapViewOfSection()

NtCreateSection() and MapViewOfSection() are both Windows API functions that allow you to create and map a section object, which can be used to create a memory-mapped file or to create a shared memory region between multiple processes.
NtCreateSection() is a kernel-mode function that is provided by the Windows NT API. It can be used to create a section object, which can be used to create a memory-mapped file or a shared memory region between multiple processes. This function is typically used by device drivers and other system-level programs.
MapViewOfSection() is a user-mode function that is provided by the Win32 API. It can be used to map a view of a section object into the address space of the calling process. This function is relatively safe to use and typically used by user-mode programs such as malware.
ZwMapViewOfSection() is another kernel-mode function that is provided by the Windows NT API, it is similar to MapViewOfSection() but it's intended for kernel-mode use, it can be used to map a view of a section object into the address space of the specified process.
Both functions can be used by malware to create a shared memory region between multiple processes, or to create a memory-mapped file that can be used to store and share data between multiple processes. Additionally, malware can use these functions to map a section object into the address space of a target process and use it to inject code or data into the target process.
It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

UnmapViewOfSection() | ZwUnmapViewOfSection()

UnmapViewOfSection() and ZwUnmapViewOfSection() are both Windows API functions that allow you to unmap a view of a section object from the address space of a process.
UnmapViewOfSection() is a user-mode function that is provided by the Win32 API. It can be used by user-mode programs to unmap a view of a section object from the address space of the calling process. This function is relatively safe to use and typically used by user-mode programs such as malware.
ZwUnmapViewOfSection() is a kernel-mode function that is provided by the Windows NT API. It can be used by kernel-mode programs to unmap a view of a section object from the address space of a specified process. It provides a lower level of abstraction and more direct access to the memory of the target process. It is typically used by device drivers and other system-level programs.
Both functions can be used by malware to unmap a view of a section object that it has previously mapped into the address space of a target process. This can be used by malware to remove the injected code or data from the target process, making it harder for security software to detect the malware.
It's important to note that the use of these functions in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

NtWriteVirtualMemory()

NtWriteVirtualMemory() is a Windows API function that allows a caller to write data to the virtual memory of a specified process. It is a kernel-mode function and provides a lower level of abstraction and more direct access to the memory of the target process than the user-mode WriteProcessMemory()

function.
In malware, NtWriteVirtualMemory() can be used to write malicious code or data into the memory of a target process, for example to perform process injection or to modify the memory of a running process in order to evade detection or analysis by security software.
Additionally, malware can use NtWriteVirtualMemory() to write to a specific memory location, for example, to patch or modify a specific instruction in the memory of a target process. This technique is often used in process hollowing, where malware creates a new process, maps its own malicious code into the process's memory, and then modifies the process's memory to redirect execution to the malware's code.
It's important to note that the use of NtWriteVirtualMemory() in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

NtReadVirtualMemory()

NtReadVirtualMemory() is a Windows API function that allows a caller to read data from the virtual memory of a specified process. It is a kernel-mode function and provides a lower level of abstraction and more direct access to the memory of the target process than the user-mode ReadProcessMemory() function.
In malware, NtReadVirtualMemory() can be used to read data from the memory of a target process, for example, to gather sensitive information such as passwords, credit card numbers, or confidential files. This can be done by reading the memory of specific processes that are known to handle or store sensitive data, such as web browsers, email clients, or financial applications.
Additionally, malware can use NtReadVirtualMemory() to read specific memory locations, for example, to analyze the memory of a running process to detect the presence of security software or to extract information about the system or other running processes.
It's important to note that the use of NtReadVirtualMemory() in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

setWindowsHookExa

SetWindowsHookEx() is a Windows API function that allows a program to install a hook procedure, which can be used to monitor various events or messages in the system. Hook procedures are typically used to extend or modify the behavior of the system or other programs.
In malware, SetWindowsHookEx() can be used to install a hook procedure that can be used to monitor keystrokes, mouse events, system messages, or other events in the system. This can be used by malware to steal sensitive information such as passwords, credit card numbers, or confidential files.
Additionally, malware can use SetWindowsHookEx() to install a hook procedure that can be used to modify the behavior of other programs, for example, to redirect network traffic, to disable security software, or to perform other malicious actions.
It's important to note that the use of SetWindowsHookEx() in malware is not necessarily malicious by itself, but it can be used in malicious ways if combined with other malicious actions.

Basic Anti Debugging APIs:

IsDebuggerPresent()

IsDebuggerPresent is a function in the Windows API that is used to determine whether a debugger is currently attached to the calling process.
The function takes no parameter and return a non-zero value if a debugger is present or zero if no debugger is present.
It is commonly used by malware or software to check if it is being run in a controlled environment, in order to hide their malicious activity. This can be used by software vendors for security measures, to determine if the software is being run in a development environment or in a production environment. This can also be used by attackers to detect if the target system is being monitored or not, in order to evade detection or take different actions.

CheckRemoteDebuggerPresent()

CheckRemoteDebuggerPresent is a function in the Windows API that is used to determine whether a debugger is currently attached to the calling process from a remote location. This function is similar to IsDebuggerPresent but it also checks for remote debugging.
This function takes two parameters:

- hProcess : Handle to the process to be checked.
- pbDebuggerPresent : Pointer to a variable that receives a value that indicates whether a debugger is present.

The function returns non-zero if it's successful and 0 if it fails. The pbDebuggerPresent returns non-zero if a debugger is present, zero if no debugger is present.
This function is also commonly used by malware or software to check if it is being run in a controlled environment, in order to hide their malicious activity. This can also be used by attackers to detect if the target system is being monitored or not, in order to evade detection or take different actions.

OutputDebugString()

OutputDebugString is a function in the Windows API that sends a null-terminated string to the debugger for display. This function is useful for debugging purposes, as it allows the developer to output text to a debugger or a debugging console.
The function takes one parameter:

- lpOutputString : A pointer to the null-terminated string to be displayed.

This function does not return any value. The output will be sent to the debugger that is currently attached to the process, if any. If no debugger is attached, the output will be ignored.
The output can be viewed using a debugging tool such as the Microsoft Visual Studio Debugger, or by using the OutputDebugString viewer (DebugView) from Sysinternals. This function can also be used to write messages to a log file. The log file can be read using the same tool.

FindWindow()

FindWindow is a function in the Windows API that is used to search for a window with a specific class name and window name.
The function takes two parameters:

- lpClassName : A pointer to a null-terminated string that specifies the class name of the window to search for. This parameter can be set to NULL if the class name is not known.
- lpWindowName : A pointer to a null-terminated string that specifies the window name (the window's title) of the window to search for.

The function returns the handle to the first window it finds with the specified class name and window name. If no such window is found, the function returns NULL.
This function is useful for interacting with windows that are created by other applications. For example, it can be used to find the handle of a specific window and then send messages to it using the SendMessage function.

Advance API Anti-Debugging APIs:

NtQueryInformationProcess

NtQueryInformationProcess is a function in the Windows NT Native API that is used to retrieve information about a process. The function is part of the Windows NT Native API, which is a set of low-level system functions that provide direct access to the operating system.
The function takes three parameters:

- ProcessHandle : Handle to the process of which information is to be retrieved.
- ProcessInformationClass : Specifies the type of information to be retrieved.
- ProcessInformation : Pointer to a buffer that receives the requested information.

The ProcessInformationClass parameter can take several values, such as:

- ProcessBasicInformation : Retrieves basic information about the process, such as its unique identifier and the process's handle to its token.
- ProcessDebugPort : Retrieves the handle to the port that the process is being debugged on.
- ProcessWow64Information : Retrieves information about whether the process is running under WOW64.

This function is intended for use by system-level components, such as device drivers or the operating system itself. It is not recommended for use by regular applications as it requires a deep understanding of the internal workings of the operating system.

NtSetInformationThread

NtSetInformationThread is a function in the Windows NT Native API that is used to set information about a thread. The function is part of the Windows NT Native API, which is a set of low-level system functions that provide direct access to the operating system.
The function takes four parameters:

- ThreadHandle : Handle to the thread of which information is to be set.
- ThreadInformationClass : Specifies the type of information to be set.
- ThreadInformation : Pointer to a buffer that contains the information to be set.
- ThreadInformationLength: Specifies the size, in bytes, of the buffer pointed to by the ThreadInformation parameter.

The ThreadInformationClass parameter can take several values, such as:

- ThreadHideFromDebugger : Indicates whether the thread should be hidden from the debugger.
- ThreadPriority : Sets the priority level of the thread.
- ThreadImpersonationToken : Sets the impersonation token for the thread.

This function is intended for use by system-level components, such as device drivers or the operating system itself. It is not recommended for use by regular applications as it requires a deep understanding of the internal workings of the operating system and can cause system instability if used incorrectly.

Timing APIs:

RDPMC/RDTSC

RDPMC is a machine-level instruction in the x86 and x86-64 instruction set architectures that is used to read the performance counters of a processor. The instruction stands for "Read Performance-Monitoring Counter".
The instruction is executed by the processor, which reads the specified performance counter and returns the value in a general-purpose register. The specific counter that is read depends on the value in the ECX register when the instruction is executed.
The performance counters can be used to measure various aspects of processor performance, such as the number of instructions executed, cache misses, branch mispredictions, etc. These counters can provide valuable information for performance analysis and optimization.
RDPMC instruction has some restriction to use it in the user-mode, it can only be used in kernel mode and in certain situations in user mode. It requires certain privilege level and certain platform support in order to use it.

RDTSC (Read Time-Stamp Counter) is a machine-level instruction in the x86 and x86-64 instruction set architectures that is used to read the time-stamp counter of a processor. The time-stamp counter is a register in the processor that increments at a constant rate, usually at the processor's clock frequency. When the instruction is executed, the current value of the time-stamp counter is returned in the EDX:EAX register pair. The high-order 32 bits are returned in EDX and the low-order 32 bits are returned in EAX.
RDTSC instruction is commonly used to measure the performance of code by measuring the number of clock cycles between two points in the code. This can be used to measure the performance of different algorithms or code paths, and to identify performance bottlenecks.
It is important to note that, RDTSC instruction provides a value that corresponds to the number of clock cycles that have occurred since the last reset, power-up or clock change event. It does not provide the wall-clock time. The value can be affected by power management, clock throttling and other factors, so it should be used with caution.

GetLocalTime()

GetLocalTime is a function in the Windows API that retrieves the current local date and time.
The function takes one parameter:

- lpLocalTime : A pointer to a SYSTEMTIME structure that receives the current local date and time.

The SYSTEMTIME structure contains the following information:

- wYear : The current year
- wMonth : The current month (1-12)
- wDayOfWeek : The current day of the week (0-6)
- wDay : The current day of the month (1-31)
- wHour : The current hour (0-23)
- wMinute : The current minute (0-59)
- wSecond : The current second (0-59)
- wMilliseconds : The current milliseconds (0-999)

This function retrieves the local time from the system clock and does not take into account any time zones. To retrieve the time in a different time zone, GetSystemTime and GetTimeZoneInformation can be used.
It is important to note that, the time retrieved by this function is the current time as of when the function is called, it does not keep updating.

GetSystemTime()

GetSystemTime is a function in the Windows API that retrieves the current system date and time.
The function takes one parameter:

- lpSystemTime : A pointer to a SYSTEMTIME structure that receives the current system date and time.

The SYSTEMTIME structure contains the following information:

- wYear : The current year
- wMonth : The current month (1-12)
- wDayOfWeek : The current day of the week (0-6)
- wDay : The current day of the month (1-31)
- wHour : The current hour (0-23)
- wMinute : The current minute (0-59)
- wSecond : The current second (0-59)
- wMilliseconds : The current milliseconds (0-999)

This function retrieves the Coordinated Universal Time (UTC) from the system clock, and not the local time. To get the local time, GetLocalTime can be used.
It is important to note that, the time retrieved by this function is the current time as of when the function is called, it does not keep updating.

GetSystemTime retrieves the current Coordinated Universal Time (UTC) from the system clock. UTC is a time standard that is widely used in the world as the reference time, it does not take into account any time zones or daylight saving time.
GetLocalTime retrieves the current local date and time from the system clock. It takes into account the time zone and daylight saving time configured on the system, so the time returned may be different from the UTC time.
In summary, GetSystemTime returns the current time in UTC format, while GetLocalTime returns the current time in the local time zone of the system.

GetTickCount()

GetTickCount is a function in the Windows API that retrieves the number of milliseconds that have elapsed since the system was started.
The function takes no parameters and returns a DWORD value representing the number of milliseconds that have elapsed since the system was started.
This function is useful for measuring time intervals and for timing operations. However, it should be

used with caution because the time value wraps around to zero if the system runs continuously for 49.7 days.
It's important to note that this function is not a high-resolution timer, the resolution of the timer is limited to the resolution of the system clock, which is typically in the range of 10 milliseconds to 16 milliseconds. For high-resolution timing, QueryPerformanceCounter and QueryPerformanceFrequency can be used.

ZwGetTickCount()/ KiGetTickCount()

ZwGetTickCount is a Windows system call that retrieves the current tick count of the system. The tick count is the number of milliseconds that have elapsed since the system was started. It can be used to measure the time elapsed between two events or to schedule time-sensitive operations. It is part of the Windows NT operating system and can be called using the ntdll.dll library.

QueryPerformanceCounter()

QueryPerformanceCounter() is a Windows API function that retrieves the current value of the performance counter, which is a high-resolution (typically microsecond-level) time stamp that can be used for performance measurement and other timing-related tasks. The function returns the current performance counter value as a 64-bit unsigned integer, which can be used in conjunction with QueryPerformanceFrequency() to determine the number of seconds or other units of time that have elapsed between two performance counter values.

timeGetTime()

timeGetTime() is a Windows API function that retrieves the current value of the multimedia timer, which is a high-resolution (typically millisecond-level) time stamp that can be used for animation and other multimedia-related tasks. The function returns the current timer value as a 32-bit unsigned integer, which can be used in conjunction with other multimedia timer functions to determine the number of milliseconds that have elapsed between two timer values. This function is part of multimedia timer which is one of the ways to measure time in windows and is less precise than QueryPerformanceCounter()

LINUX APIs

Fork()
Clone()
Execve()
Fuse()
Ptrace() - many uses
Inotify()
Fanotify()
Ebpf()

Instructor:
Shesh Sarangdhar (6000464780)
Worked with NTRO for the last 8 years

Sciroit Technologies LLP ???

Malware Detection Evasion:

1. OS dependent detection evasion
2. OS independent detection evasion
3. Organic growth model using genetic algorithm [Langturn, von neumann, etc]

OS based detection Evasion:

1st Gen Malware Hooking

IDT (interruct descriptor tab;e)
IAT/EAT (import export table)
SSDT (Sytem services descriptor table)
DLL injection
Transient Services (explorer.exe, svchost.exe) - System/kernel level services used by OS

DLL Injection:

APIs:
OpenProces
VirtualAlloc
WriteProcessMemory
CreateRemoteThread() ->

Modes in OS: Protected Mode and SMM(System Management Mode) and _____

Protected Mode:

SMM: can cause Hardware damage (overclocking and bursting CPUs)

How to use SSDT to detect DLL injection??
By finding the Diff between the old SSDT and new SSDT would do it.
All IDT, IAT/EAT etc are static so you can diff it and detect the injections
So, how do you make it completely undetectable?
EPOC data structure used in Task Manager

To keep a track a process, Task manager connects the header pointer & tail pointer of all the process as a linked list

Method: Direct Kernel Object Manipulation/Management (DKOM)

In this you remove the link between p2 with p1 and p3, therefore making p2 invisible to tracker (Task manager/ Anti-Virus)

How to detect DKOM ?
Kernel cant see the process but it still exists in the Virtual memory

So by Virtual memory checks you might be able to find a memory block with no link to any process. There that is mostly the malware. Using Volatility.

Dump the memory using volatility and map it with SSDt, EAT, IAt etc and find the un linked block of memory.

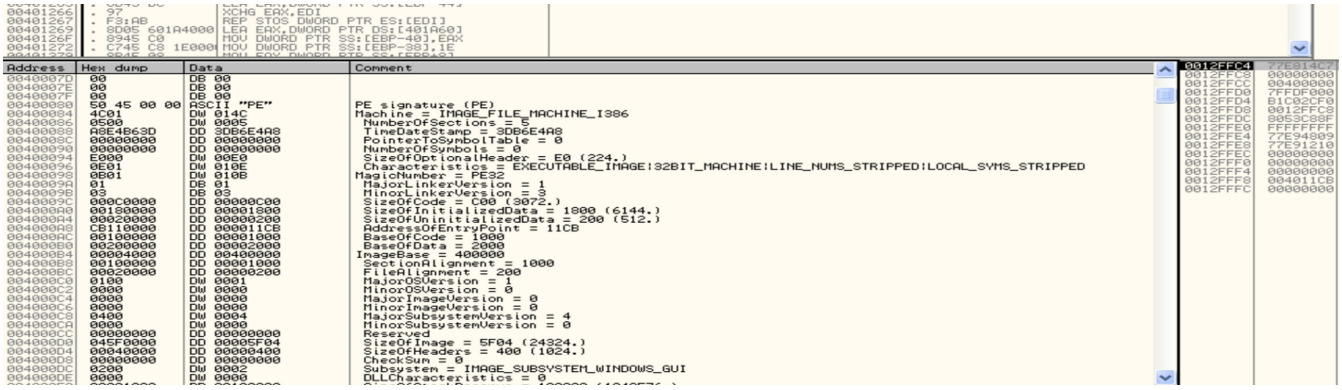
How to bypass Volatility detection ?

Virtual Memory Subversion [Read about MAS structure]
By changing a bit in EBP and ESP, so you get completely lost when you look at the stack.
How do you achieve it ?
There are two views [One the OS show you/volatility i.e the virtual memory and the 2nd one physical memory which the OS actually uses to run the program] Therefore code executes but shows an altered virtual memory block to Volatility. Therefore to bypass this also do manual analysis of Virtual Memory rather than using Volatality.

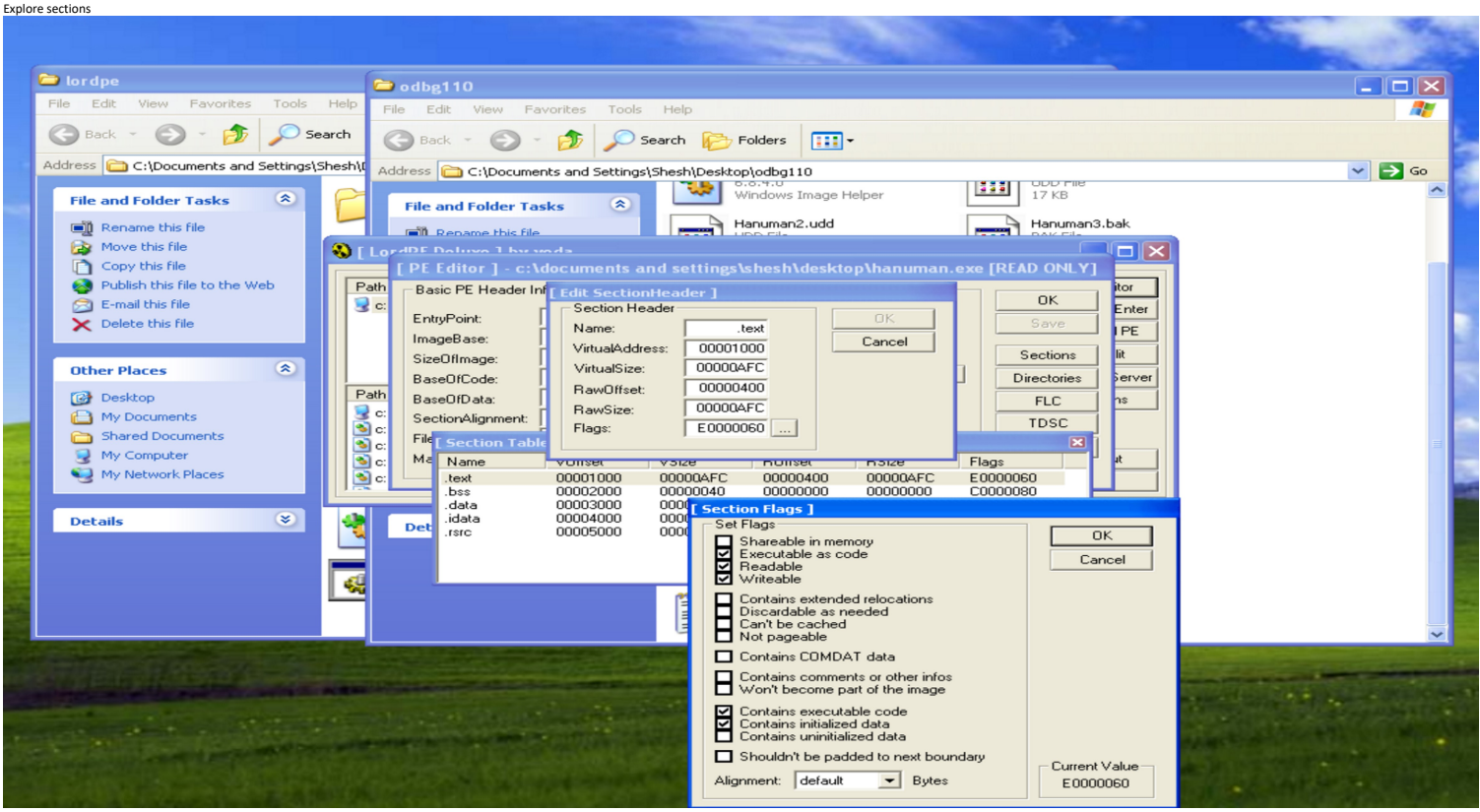
Practical:
Using Windows XP
Why windows XP? It doesn't have ASLR, PE etc.

Example: Hanuman.exe
Tool: OllyDbg

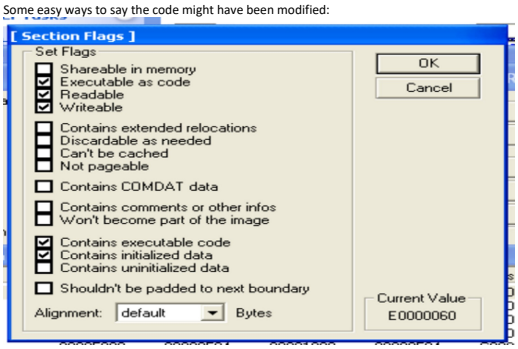
Run Hanuman in Ollydbg
Run
Go to memory[press "M"]
Locate PE header and Right Click -> Dump in CPU [this is done to look at how PE header looks like]
Right click -> special->click on PE header



Go to LordPE
Select PE editor and select Hanuman
Explore sections



Some easy ways to say the code might have been modified:



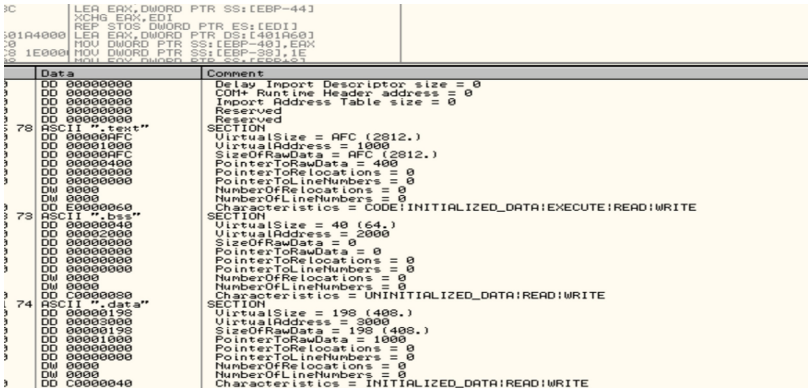
Basic way to modify the binary: [add encoder]
Basically insert an encoder and change the entry point of code from actual entry point and change the entry point to Encoder. Therefore, every time the code is encoded. This is done by adding some bytes to

RSRC setion. But the rsrc is not code section therefore I doesn't have Permission to execute. Now change the permissions to make the insterted encoder code in rsrc executable.

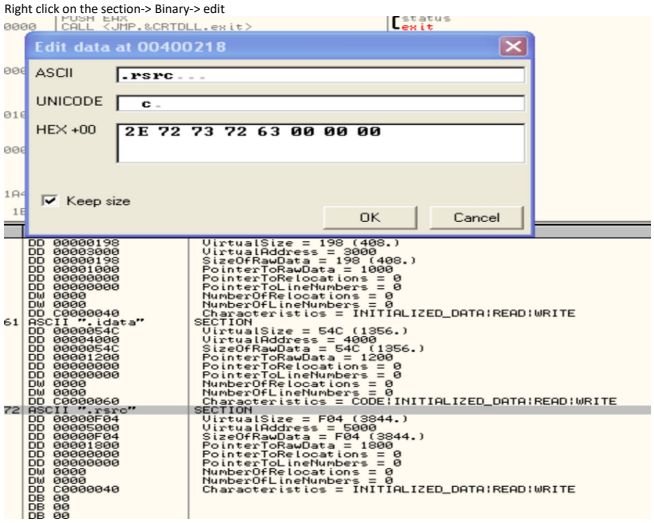
Now when the code is executed it wont run because the doe is gibberish.
Therefore make sure you add a decoder the 2nd time onwards in the entry point
1st time running - entry point at Encoder
2nd time onwards running - entry point is set at decoder
There when the code lies stored in memory the AV cant detect it
It is decoded only in RUNTIME !!!!

In order to achieve this you would have to modify permissions for .text and .rsrc
Make them writable and initialized data

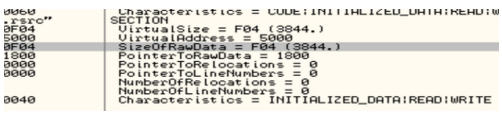
You can do this in either using LordPE or Olly:



Right click on the section-> Binary-> edit



Right click-> copy to executable file -> right -> save file [SAVING FILE]



Virtual size = 5000 [this means the starting point of rsrc]
Rawdata = F04 [size of rsrc]
Now you would want to add your encoder code after 5F04 [5000+F04]
And change entry point that was in .text to 5F04

Now since you are planning to add code in rsrc you have to extend its size by 100 bytes (lets say 100 bytes)
So, change value of Raw Data [inorder to accommodate 100 more bytes]
And in Virtual
Save file

Now if you try to run the exe file it would throw a memory error.
So open the exe in HxD and fill the added 100 bytes with Zeros. [note: if the exe is already open then close it to be able to be editable in HxD]
Now when this exe is run in debugger, it would say that the entry point is outside the text section.

Therefore, you could add your encoder in text section only (wherever you find enough space to insert code in text. You can also insert in many places and jump from one address to other inside text only)
Minimum 10to15bytes of consecutive memory addresses is required in text - Add minimum 3 nop before actual instructions, just in case if CPU optimization is used and one or two bytes uper niche ho sakta hei, just to be safe that CPU doesn't hit our instruction.

Now, in Encoder Code: you would create use MOV EAX Hanuman.00F45 (this is the address from which the encoder should start encoding)
And MOV ECX Hanuman.00F67 (till this point it has to be encoded)
Logic:
ADD
XOR

Decoder:
Logic:
XOR
SUB

UCAL Berkley Project by Mr. Henderson Researches on AI combined with Organic growth model using genetic algorithm [Langturn, von neumann, etc]

Is DKOM Virtual memory evasion ?

Tool: System Virginity Verifier ??????

What is the highest privilege?
Anti Authority System

Create another user that is higher that Admin and AAS. Therefore Admin can't do shit to prevent the Malware.