Malware Analysis

Assembly Language and X86 Disassembly

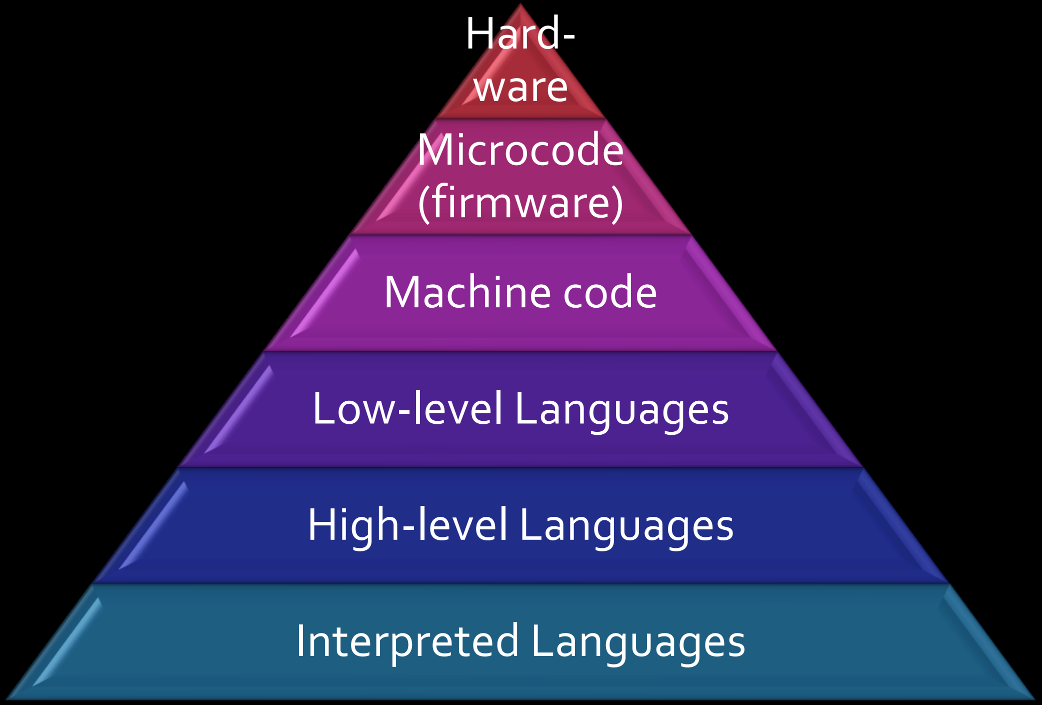
COMPILED BY: DHARMESH DAVE | ASST. PROF. | NATIONAL FORENSIC SCIENCES UNIVERSITY

Why?

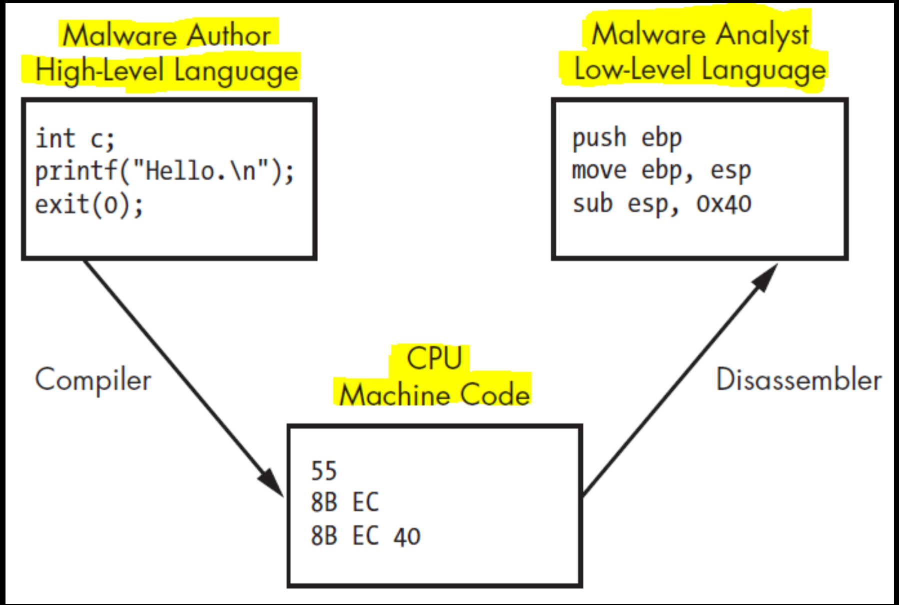
- There are certain limitations in Basic Dynamic Analysis:
- It can tell behviour of sample malware but can not reveal the logic written for that behaviour.
- Dynamic analysis may not run all command line malwares as related commands may not be found.
- It can tell you how your subject malware responds when it receives a specially designed packet, but you can learn the format of that packet only by digging deeper.
- That’s where disassembly comes in as a solution.

Introduction Levels of Abstraction

- In traditional computer architecture, a computer system can be represented as several levels of abstraction that create a way of hiding the implementation details. For example, you can run the Windows OS on many different types of hardware, because the underlying hardware is abstracted from the OS.



Reverse-Engineering



```
int c;
printf("Hello.\n");
exit(0);
```

```
55
8B EC
8B EC 40
```

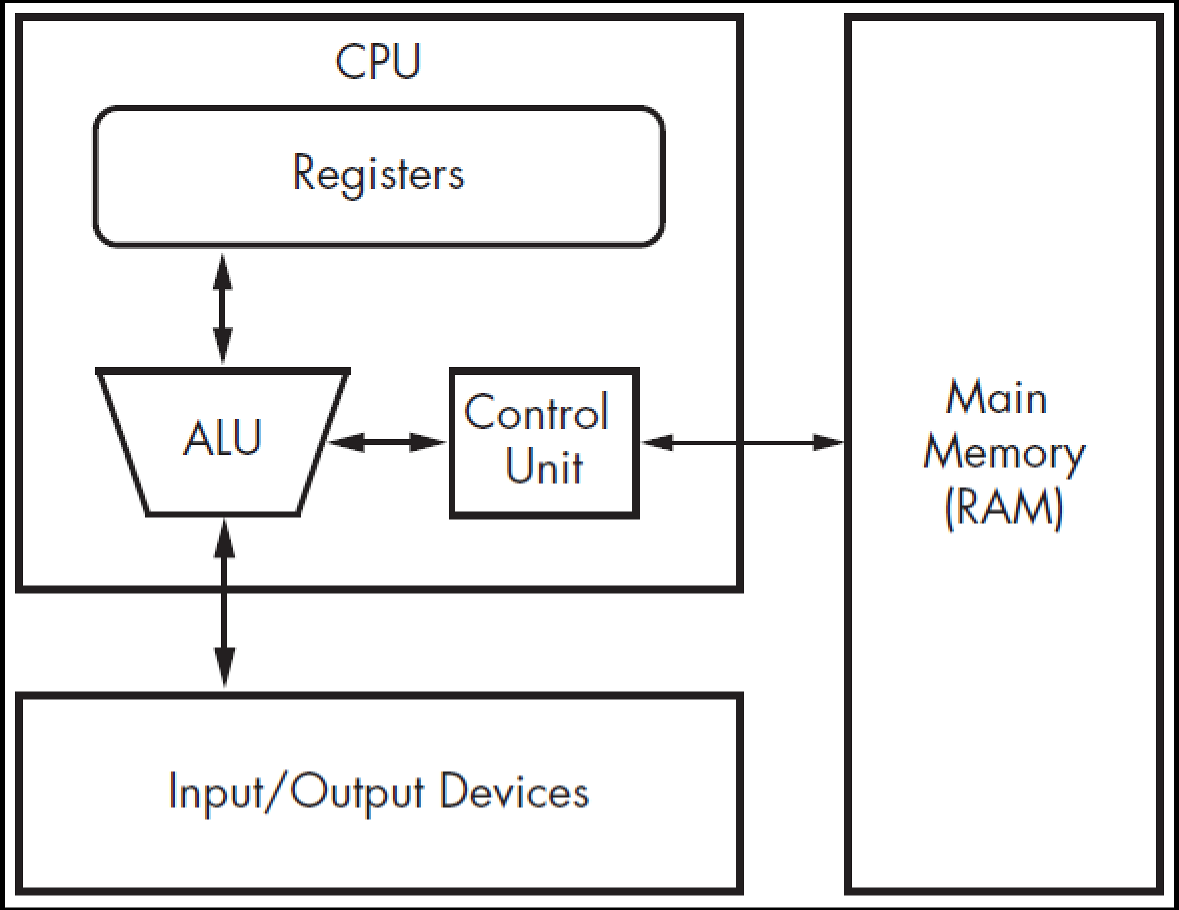
```
push ebp
move ebp, esp
sub esp, 0x40
```

- When malware is stored on a disk, it is typically in binary form at the **machine code level**. (machine code is the form of code that the computer can run quickly and efficiently)
- When we disassemble malware, we take the malware binary as input and generate assembly language code as output, usually with a disassembler.

Reverse-Engineering

- Assembly language is actually a class of languages. Each assembly dialect is typically used to program a single family of microprocessors, such as x86, x64, SPARC, PowerPC, MIPS, and ARM. x86 is by far the most popular architecture for PCs.
- Most 32-bit personal computers are x86, also known as Intel IA-32, and all modern 32-bit versions of Microsoft Windows are designed to run on the x86 architecture.
- The control unit gets instructions to execute from RAM using a register (the instruction pointer), which stores the address of the instruction to execute. The control unit gets instructions to execute from RAM using a register (the instruction pointer), which stores the address of the instruction to execute.

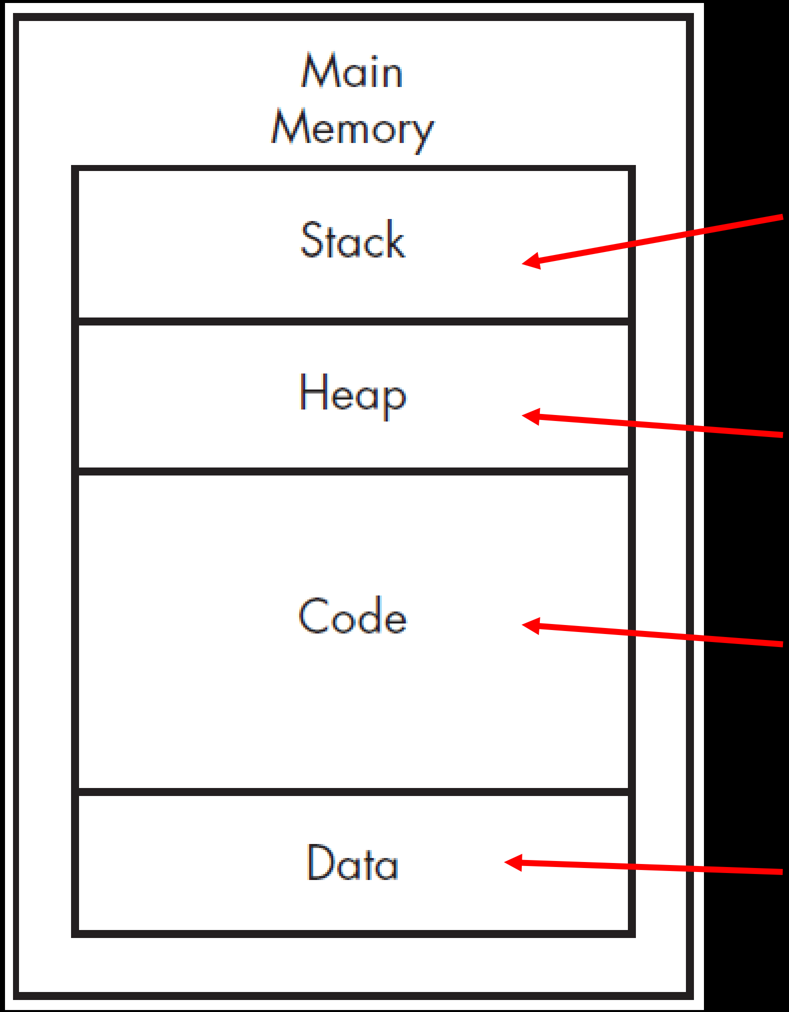
CPU Architecture



CPU Architecture

- The control unit <u>gets instructions to execute from RAM</u> using a register (the instruction pointer), which stores the address of the instruction to execute.
- Registers are the <u>CPU's basic data storage units</u> and are often used to save time so that the CPU doesn't need to access RAM.
- The arithmetic logic unit (ALU) <u>executes an instruction fetched from RAM and places the results in registers or memory.</u>

Main Memory



Used for local variables and parameters for functions, and to help control program flow.

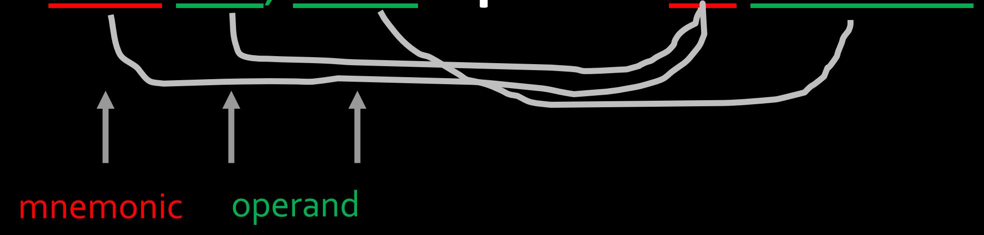
Used for dynamic values to create (allocate) new values and eliminate (free) values.

Instructions fetched by the CPU to execute the program's tasks. Controls the program.

Values are put when program is initially loaded, the values are static and global in nature.

Instructions

- Instructions are the building blocks of assembly programs. In x86 assembly, an instruction is made of a mnemonic and zero or more operands.
- **Exmple:** mov ecx, 0x42 **Opcode:** B9 42 00 00



- Each instruction corresponds to opcodes (operation codes) that tell the CPU which operation the program wants to perform.
- Disassemblers translate opcodes into human-readable instructions.
- x86 architecture uses the little-endian format.

Instructions

- Types of Endian: Big and Little.
- Changing between endianness is something malware must do during network communication.
- Example: the IP address 127.0.0.1 will be represented as 0x7F000001 in bigendian format (over the network) and 0x0100007F in little-endian format (locally in memory).

Operands

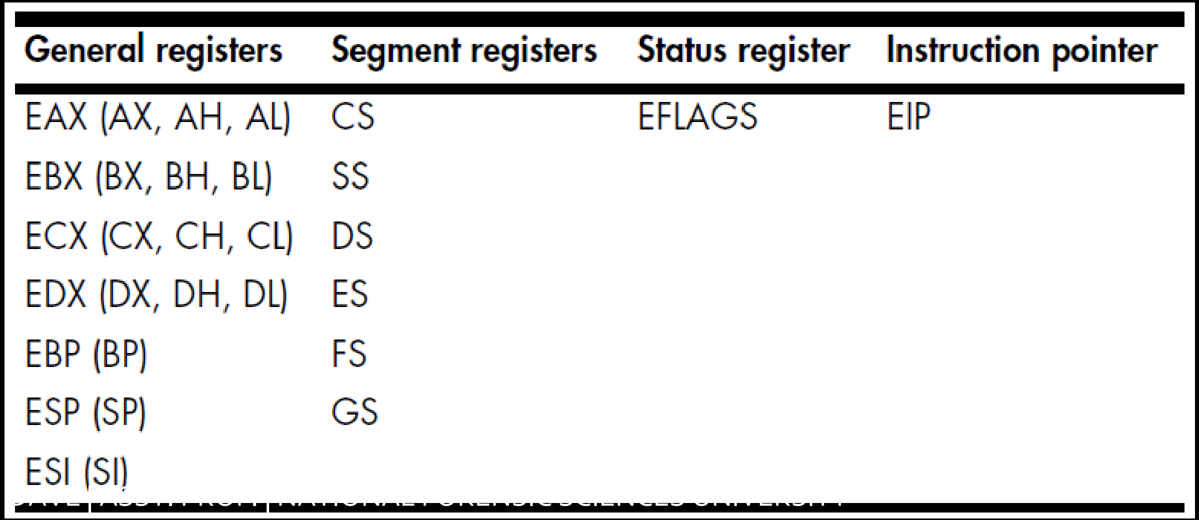
- Operands are used to identify the data used by an instruction. Three types of operands can be used:

1. Immediate: fixed values i.e. 0x42
2. Register: refer to registers i.e. ecx
3. Memory address: refer to a memory address that contains the value of interest, typically denoted by a value, register, or equation between brackets i.e. [eax]

Registers

- A register is a small amount of data storage available to the CPU. x86 processors have a collection of registers available for use as temporary storage or workspace.
- x86 Registers falls into following four categories:

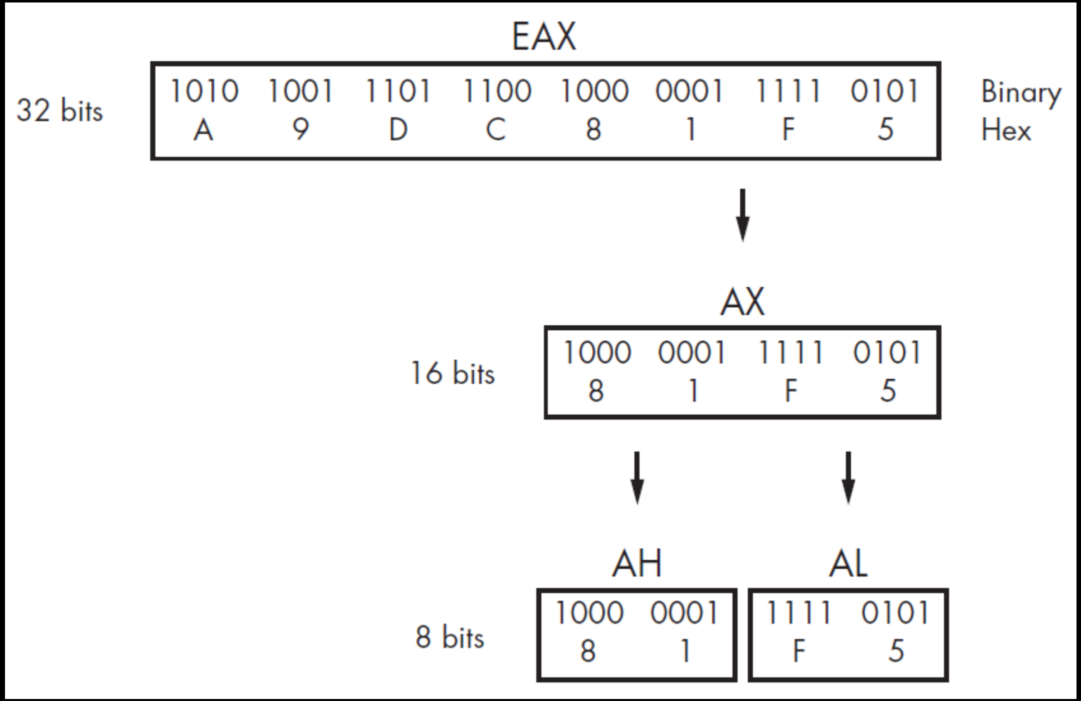
1. General Register: used by the CPU during execution.
2. Segment Register: used to track sections of memory.
3. Status Flag: used to make decisions.
4. Instruction Pointer: used to keep track of the next instruction to execute.



General registers	Segment registers	Status register	Instruction pointer
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Registers

- All general registers are 32 bits in size and can be referenced as either 32 or 16 bits in assembly code.
- Four registers (EAX, EBX, ECX, and EDX) can also be referenced as 8-bit values using the lowest 8 bits or the second set of 8 bits.



General Registers

- **AX:** is the primary accumulator; it is used in input/output and most arithmetic instructions. For example, in multiplication operation, one operand is stored in EAX or AX or AL register according to the size of the operand.
- **BX:** is known as the base register as it could be used in indexed addressing.
- **CX:** is known as the count register as the ECX, CX registers store the loop count in iterative operations.
- **DX:** is known as the data register. It is also used in input/output operations.

General Registers

- **Instruction Pointer (EIP)** - the 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- **Stack Pointer (ESP)** - the 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- **Base Pointer (EBP)** - the 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with **DI** and **SI** as base register for special addressing.

General Registers

- The 32-bit index registers **ESI** and **EDI** and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers:
- Source Index (SI) - it is used as source index for string operations
- Destination Index (DI) - it is used as destination index for string operations.

Status Flag Registers

- The EFLAGS register is a status register. In the x86 architecture, it is 32 bits in size, and each bit is a flag. During execution, each flag is either set (1) or cleared (0) to control CPU operations or indicate the results of a CPU operation.
- The 32-bit instruction pointer register and 32-bit flags register combined are considered as the control registers.
- Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

Status Flag Registers

- Overflow Flag (OF): indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
- Direction Flag (DF): determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.
- Interrupt Flag (IF): determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
- Trap Flag (TF): allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

Status Flag Registers

- Sign Flag (SF): shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
- Zero Flag (ZF): indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.
- Auxiliary Carry Flag (AF): contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
- Parity Flag (PF): indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.
- Carry Flag (CF): contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

Segment Registers

- Segments are specific areas defined in a program for containing data, code and stack. There are three main segments:
- **Code Segment:** it contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
- **Data Segment:** it contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
- **Stack Segment:** it contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

Assembly Language

- Assemblers Type:
  - NASM (Netwide Assembler)
  - MASM (Microsoft Assembler)
  - TASM (Borland Turbo Assembler)
  - GAS (GNU Assembler)
- An assemble program can be divide into 3 sections:
  - The data section: .data
  - The bss section: .bss
  - The text section: .text

Assembly Language

- <u>Data Section</u>: The data section is used for declaring initialized data or constants. This data does not change at runtime. You can declare various constant values, file names or buffer size, etc., in this section.
- *– Syntax: section .data*
- <u>Bss section</u>: The bss section is used for declaring variables.
- *– Syntax: section .bss*
- <u>Text Section</u>: The text section is used for keeping the actual code. This section must begin with the declaration global _start, which tells the kernel where the program execution begins.
- *– Syntax: section .text*

```
            global _start
        _start:
```

Assembly Language

```
section .text
    global _start ;must be declared for linker (ld)
_start:          ;tells linker entry point
    mov edx,len ;message length
    mov ecx,msg ;message to write
    mov ebx,1 ;file descriptor (stdout)
    mov eax,4 ;system call number (sys_write)
    int 0x80 ;call kernel
    mov eax,1 ;system call number (sys_exit)
    int 0x80 ;call kernel
section .data
    msg db 'Hello, world!', 0xa ;our dear string
    len equ $ - msg ;length of our dear string
```

Assembly Instructions (Simple)

- mov - move data from one location to another
  - mov eax, ebx - Copies the contents of EBX into the EAX register
  - mov eax, [ebx] - Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
- lea - used to put a memory address into the destination
  - lea eax, [ebx+8] - will put EBX+8 into EAX.

Assembly Instructions (Arithmatic)

- sub eax, 0x10 - Subtracts 0x10 from EAX
- add eax, ebx - Adds EBX to EAX and stores the result in EAX
- inc edx - Increments EDX by 1
- dec ecx - Decrements ECX by 1
- mul 0x50 - Multiplies EAX by 0x50 and stores the result in EDX:EAX
- div 0x75 - Divides EDX:EAX by 0x75 and stores the result in EAX and the remainder in EDX

Assembly Instructions (Logical and Shifting)

- xor eax, eax - quick way to set the EAX register to zero
    - This is done for optimization, because this instruction requires only 2 bytes, whereas mov eax, 0 requires 5 bytes
- mov eax, 0xA
- shl eax, 2 - Shifts the EAX register to the left 2 bits; these two instructions result in EAX = 0x28, because 1010 (0xA in binary) shifted 2 bits left is 101000 (0x28)
- mov bl, 0xA
- ror bl, 2 - Rotates the BL register to the right 2 bits; these two instructions result in BL = 10000010, because 1010 rotated 2 bits right is 10000010

Assembly Instructions (NOP)

- NOP (no operation)
- The opcode for this instruction is 0x90. It is commonly used in a NOP sled for buffer overflow attacks, when attackers don’t have perfect control of their exploitation.
- Logical Gate operations are generally used for Encryption.

STACK

- Memory for functions, local variables, and flow control is stored in a **stack**, which is a data structure characterized by pushing and popping. You push items onto the stack, and then pop those items off. A stack is a last in, first out (LIFO) structure.
- The register support stack includes the ESP and EBP registers.
- **ESP** is the stack pointer and typically contains a memory address that points to the top of stack.
- **EBP** is the base pointer that stays consistent within a given function, so that the program can use it as a placeholder to keep track of the location of local variables and parameters.
- The stack is used for short-term storage only. It frequently stores *local variables, parameters, and the return address*. Its primary usage is for the management of data exchanged between function calls.

Assembly Instructions (STACK)

- *instructions: push, pop, call, leave, enter, and ret.*
- *Function calls - prologue and epilogue*
- *pusha: pushes the 16-bit registers on the stack in the following order: AX, CX, DX, BX, SP, BP, SI, DI.*
- *pushad: pushes the 32-bit registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.*
- *call memory_location: A function is called using*

Function Calls (STACK)

- The main code calls and temporarily transfers execution to **functions** before returning to the main code.
- Many functions contain a prologue—a few lines of code at the start of the function. The prologue prepares the stack and registers for use within the function.
- In the same vein, an epilogue at the end of a function restores the stack and registers to their state before the function was called.
- The following list summarizes the flow of the most common implementation for function calls.

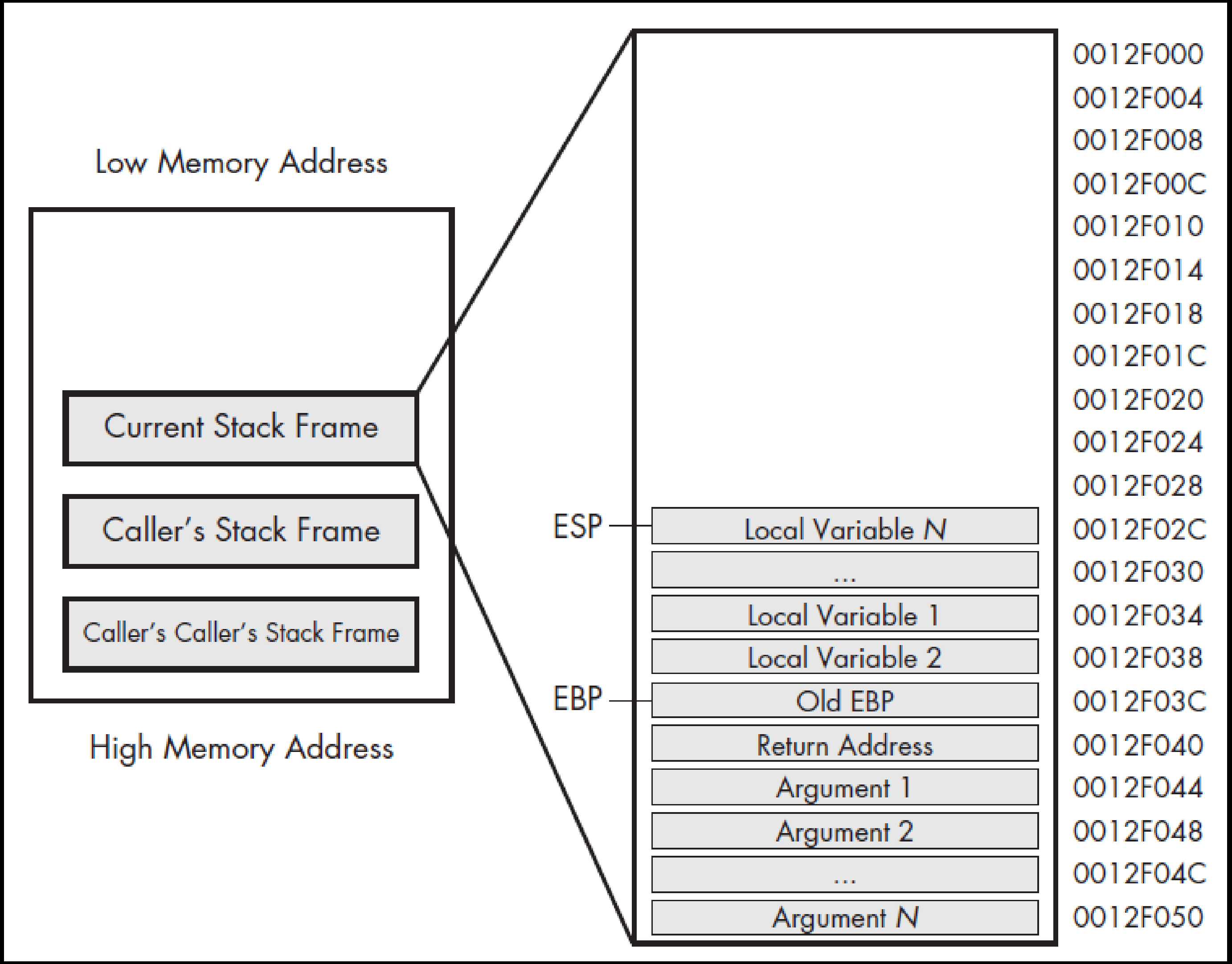
Function Calls (STACK)

- 1. Arguments are placed on the stack using push instructions.
- 2. A function is called using *call memory_location*. This causes the current instruction address (that is, the contents of the EIP register) to be pushed onto the stack. This address will be used to return to the main code when the function is finished. When the *function begins*, EIP is set to memory_location (the start of the function).
- 3. Through the use of a function prologue, *space is allocated on the stack* for local variables and EBP (the base pointer) is pushed onto the stack. This is done to save EBP for the calling function.
- 4. The function performs its work.

Function Calls (STACK)

- 5. Through the *use of a function epilogue*, the stack is restored. ESP is adjusted to free the local variables, and EBP is restored so that the calling function can address its variables properly. The leave instruction can be used as an epilogue because it sets ESP to equal EBP and pops EBP off the stack.
- 6. The function returns by calling the ret instruction. This pops the return address off the stack and into EIP, so that the program *will continue executing* from where the original call was made.
- 7. The stack is adjusted to remove the arguments that were sent, unless they'll be used again later.

Function Calls (STACK)



Assembly Instructions (Conditionals)

- **Conditionals** are instructions that perform the comparison.
- The two most popular conditional instructions are ***test*** and ***cmp***. *(the operands involved are not modified by the instruction. These instructions only set the flags.)*
- The test instruction is identical to the **and** instruction; The *zero flag (ZF)* is typically the flag of interest after the test instruction. *i.e. test eax, eax used to check Null.*
- The cmp instruction is identical to the **sub** instruction; The *zero flag (ZF) and carry flag (CF)* may be changed as a result of the cmp instruction.

cmp dst, src	ZF	CF
dst = src	1	0
dst < src	0	1
dst > src	0	0

Branching: Assembly Instructions (JUMP)

- A branch is a sequence of code that is conditionally executed depending on the flow of the program.
- Two types:
  - Conditional (jnz,jz) and unconditional (jmp location)
- Conditional jump works along with flags.
- jz loc - Jump to specified location if ZF = 1.
- jnz loc - Jump to specified location if ZF = 0.
- 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.

Assembly Instructions (JUMP)

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.

Assembly Instructions (REP)

- Rep instructions are a set of instructions for manipulating data buffers. They are usually in the form of an array of bytes, but they can also be single or double words.
- rep - Repeat until ECX = 0
- repe, repz - Repeat until ECX = 0 or ZF = 0

References

- Practical Malware Analysis by Michael Sikorski