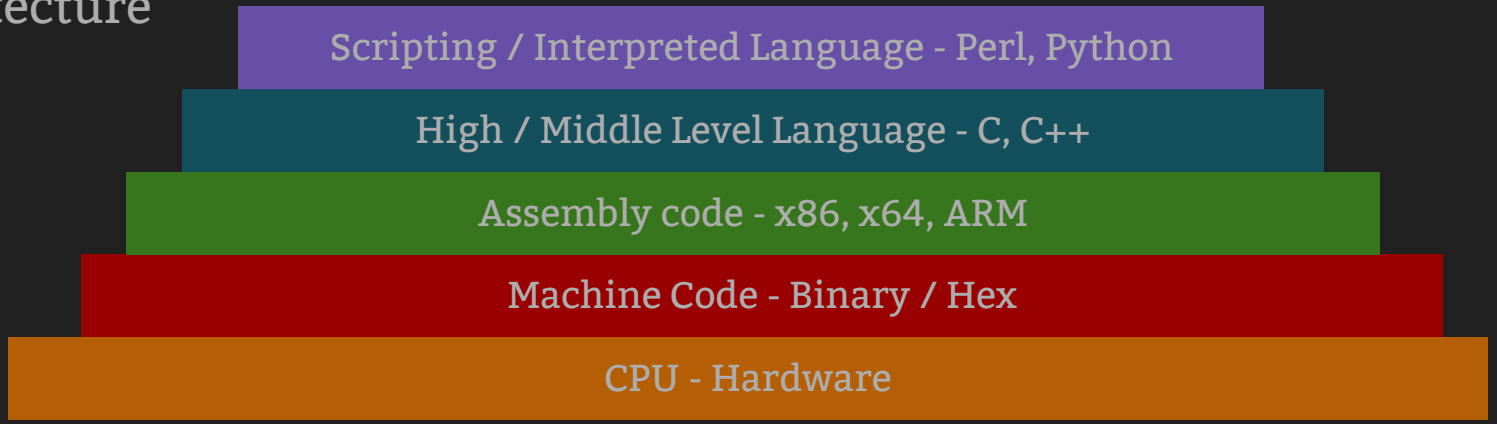
Malware analysis

Lecture 5
Advanced Static Analysis

Abstraction levels

- Usually assembly is the highest abstraction level that we can reliably recover
- Understanding assembly is essential for malware analysis
- x86 provides the highest attack surface -> we will focus on that architecture



Basics

- bit - 0 or 1
- nibble - 4 bits (one hexadecimal digit)
  $0010_2 = 4_{16}$
- byte - 8 bits
  $0010\ 1101_2 = 4D_{16}$
- word - 16 bits
  $0010\ 1101\ 0101\ 1010_2 = 4D\ 5A_{16}$
- dword - 32 bits
- qword - 64 bits

ASCII

- American Standard Code for Information Interchange
- Character encoding standard
- **0x00 - 0x1F** Control Characters (e.g. Null, Backspace, Line feed)
- **0x20 - 0x7E** Printable characters (letters, digits, punctuation marks, and a few miscellaneous symbols)
- **0x30 - 0x39** Digits
- **0x41 - 0x5A** Upper-case Letters
- **0x61 - 0x7A** Lower-case Letters
- Example:
  malware = 6d 61 6c 77 61 72 65

Unicode

- https://unicode.org/standard/standard.html
- The Unicode Standard is a character coding system designed to support the worldwide interchange, processing, and display of the written texts of the diverse languages and technical disciplines of the modern world. In addition, it supports classical and historical texts of many written languages.
- e.g. UTF-8, UTF-16, and UTF-32
- Example:
  malware = \u006d\u0061\u006c\u0077\u0061\u0072\u0065

Endianness

- order of bytes
- Big-endian - most significant byte first and least significant last
  e.g. Network traffic
- Little-endian - least significant byte first and most significant last
  e.g. x86 and x86-64
- Example: 0x12345678
  - big-endian: 0x12, 0x34, 0x56, 0x78
  - little-endian: 0x78, 0x56, 0x34, 0x12

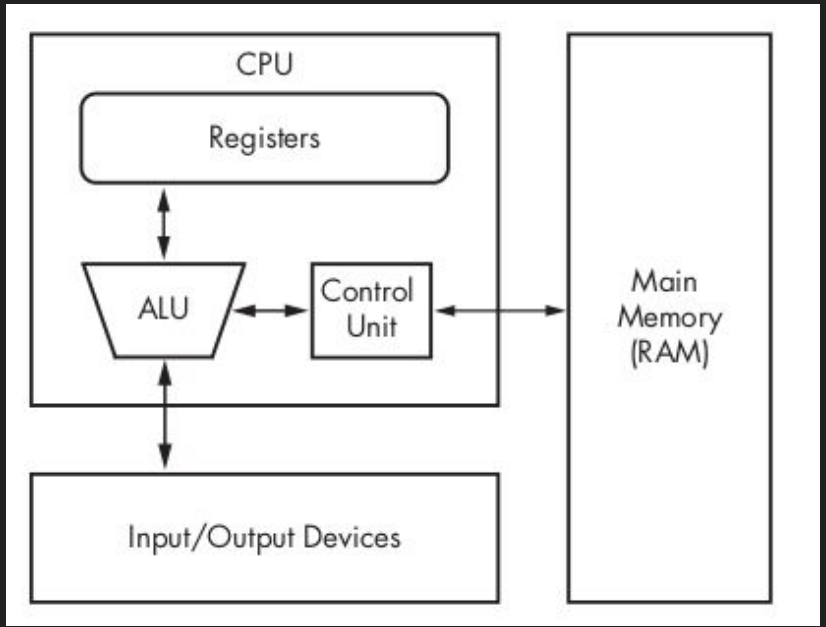
Data Interpretation

= 55 hexadecimal number
= 85 decimal number
= U ASCII character
= 01010101 binary

= 00401530 memory address
= 4199728 integer value

The x86 Architecture

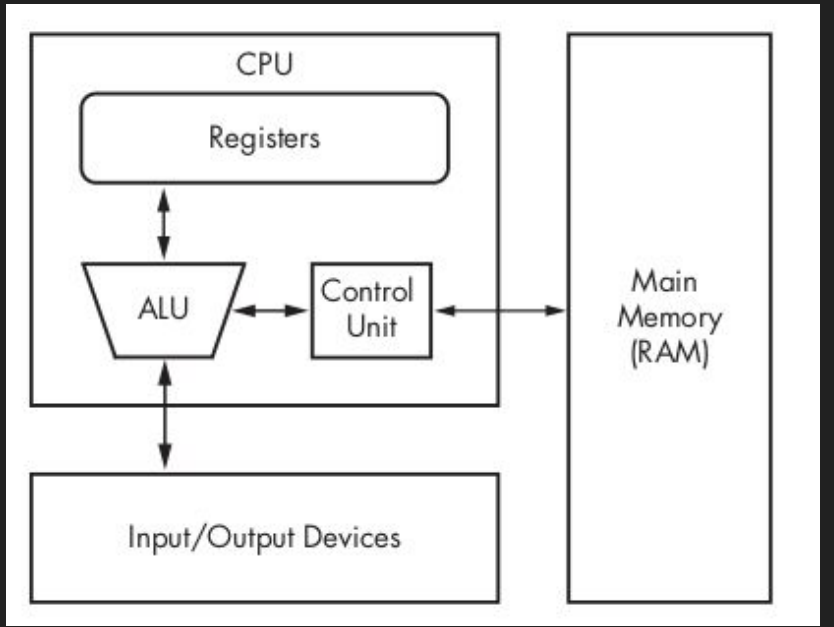
- The central processing unit (CPU) executes instructions.
- The main memory of the system (RAM) stores all data and code.
- An input/output system (I/O) interfaces with devices such as hard drives, keyboards, and monitors.



Reference: Michael Sikorski and Andrew Honig: **Practical Malware Analysis, The Hands-On Guide to Dissecting Malicious Software**. No Starch Press. ISBN: 978-1-593-27290-6

The x86 Architecture

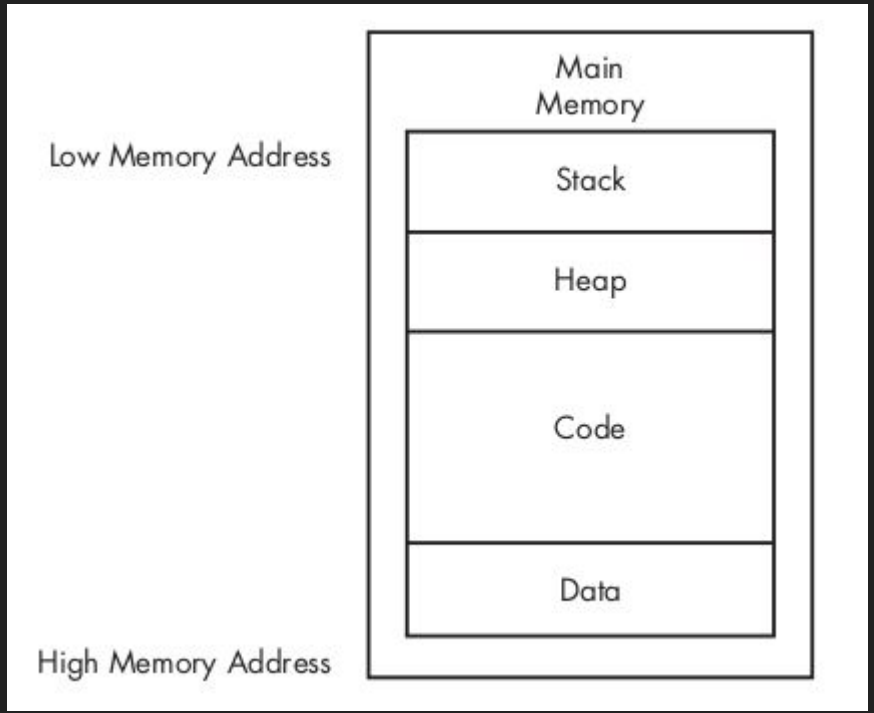
- The control unit gets instructions to execute from RAM using a register (the instruction pointer), which stores the address of the instruction to execute.
- Registers are the CPU's basic data storage units and are often used to
- save time so that the CPU doesn't need to access RAM.
- The arithmetic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



Reference: Michael Sikorski and Andrew Honig: **Practical Malware Analysis, The Hands-On Guide to Dissecting Malicious Software**. No Starch Press. ISBN: 978-1-593-27290-6

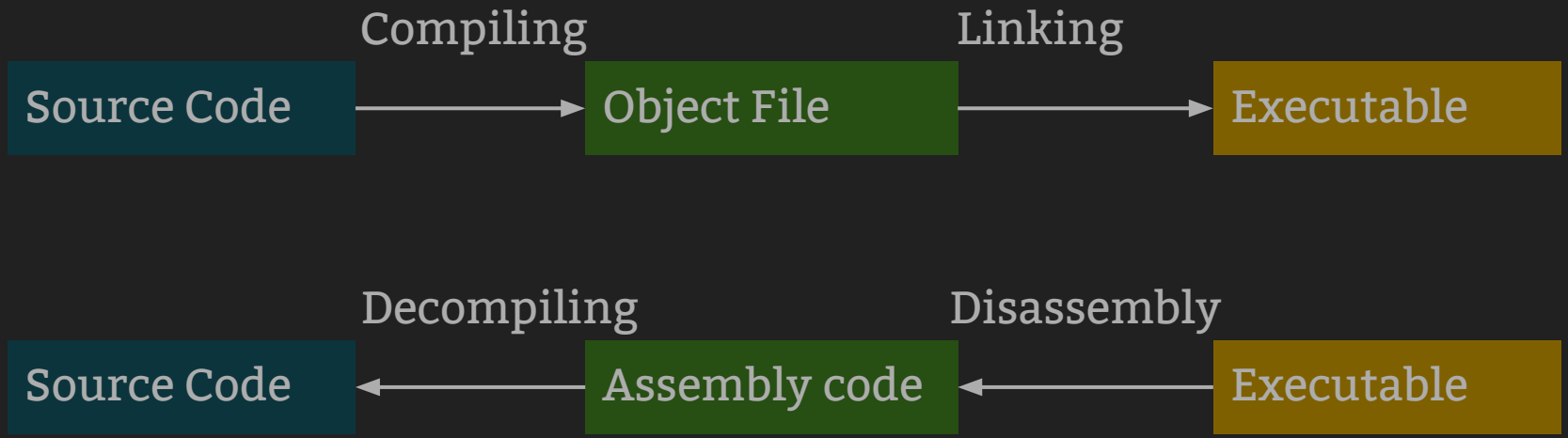
Memory

- **Data** - data required by the program
- **Code** - instructions executed by the CPU
- **Heap** - dynamic memory
- **Stack** - used for local variables and parameters for functions and to help control program flow.



Reference: Michael Sikorski and Andrew Honig: **Practical Malware Analysis, The Hands-On Guide to Dissecting Malicious Software**. No Starch Press. ISBN: 978-1-593-27290-6

Program Compilation



https://godbolt.org/ - assembly output of compiled code with a given compiler and settings

HelloWorld

- helloworld.c

```c
#include <stdio.h>
int main()
{
   printf("Hello, World!");
   return 0;
}
```

Source Code

- helloworld.o: i686-w64-mingw32-gcc -c -o helloworld.o helloword.c

```
hanna@hannapc01:~/stuff/University$ size helloworld.o
   text    data     bss     dec     hex filename
     84       0       0      84      54 helloworld.o
```

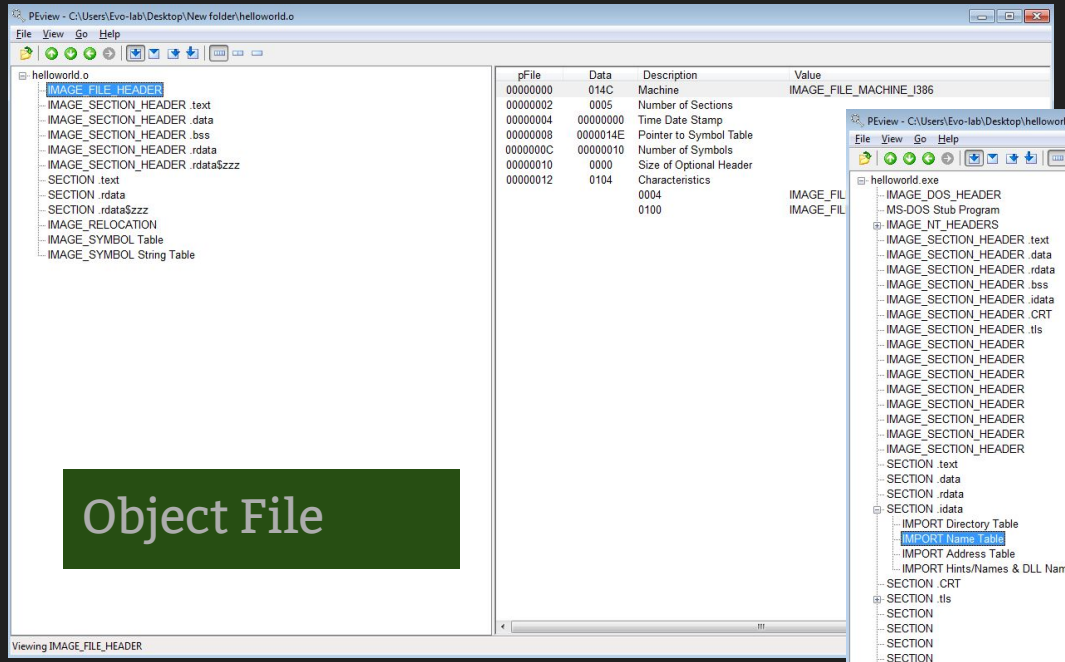
Object File

- helloworld.exe: i686-w64-mingw32-gcc -o helloworld.exe helloword.c

```
hanna@hannapc01:~/stuff/University$ size helloworld.exe
   text    data     bss     dec     hex filename
   7232    1512    1008    9752    2618 helloworld.exe
```

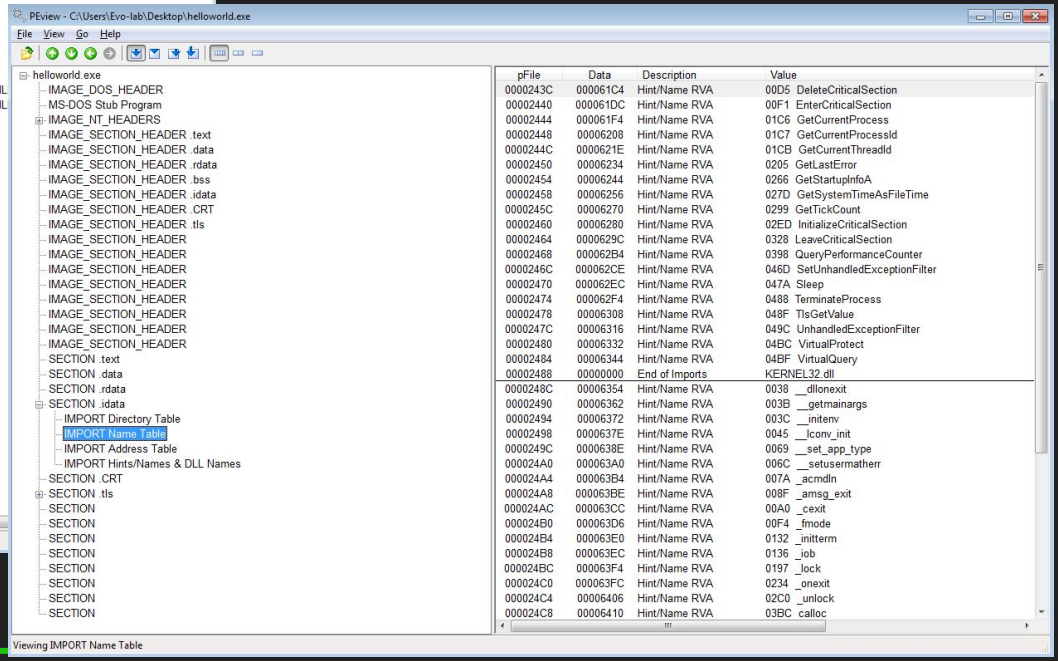
Executable

HelloWorld



Object File

Executable



HelloWorld

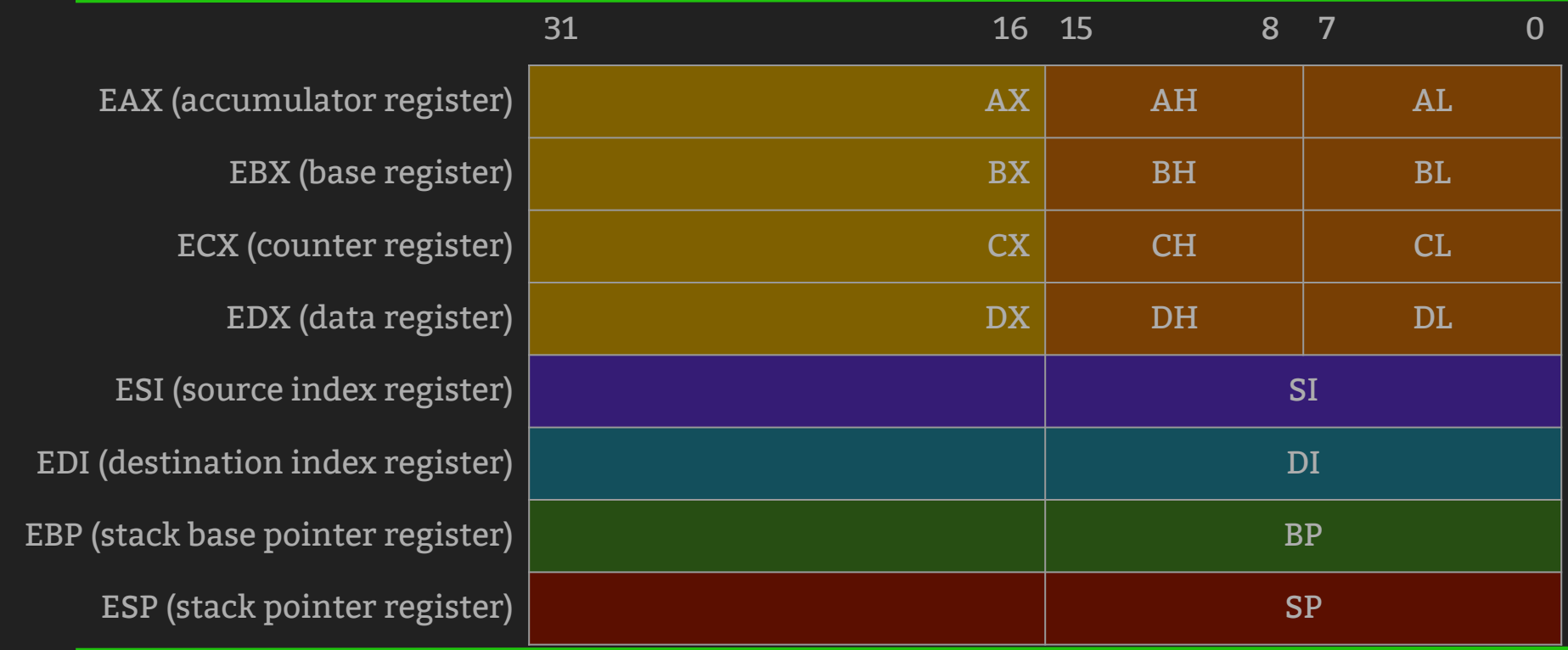
Assembly code

```
.text:00401530                          ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401530                                  public _main
.text:00401530                          _main   proc near               ; CODE XREF: ___tmainCRTStartup+26E↑p
.text:00401530
.text:00401530                          argc    = dword ptr  8
.text:00401530                          argv    = dword ptr  0Ch
.text:00401530                          envp    = dword ptr  10h
.text:00401530
.text:00401530 55                               push    ebp
.text:00401531 89 E5                            mov     ebp, esp
.text:00401533 83 E4 F0                         and     esp, 0FFFFFFF0h
.text:00401536 83 EC 10                         sub     esp, 10h
.text:00401539 E8 82 01 00 00                   call    ___main
.text:0040153E C7 04 24 00 40 40 00             mov     dword ptr [esp], offset aHelloWorld ; "Hello, World!"
.text:00401545 E8 DA 10 00 00                   call    _printf
.text:0040154A B8 00 00 00 00                   mov     eax, 0
.text:0040154F C9                               leave
.text:00401550 C3                               retn
.text:00401550                          _main   endp
```

Registers

- Small data storage available to the CPU
- It is faster for the CPU to access data in the register than in the memory
- Types:
  - General registers - used by the CPU during execution
  - Segment registers - used to track sections of memory
  - Status flags - used to make decisions
  - Instruction pointer (EIP) - used to keep track of the next instruction to execute

Registers


31
16
15
8
7
0
EAX (accumulator register)
AX
AH
AL
EBX (base register)
BX
BH
BL
ECX (counter register)
CX
CH
CL
EDX (data register)
DX
DH
DL
ESI (source index register)
SI
EDI (destination index register)
DI
EBP (stack base pointer register)
BP
ESP (stack pointer register)
SP

Registers

- Segment registers: CS, SS, DS, ES, FS, GS
- EFLAGS register: 32-bit status register, each bit is a flag (1 or 0).
  - ZF - The zero flag is set when the result of an operation is equal to zero; otherwise, it is cleared.
  - CF - The carry flag is set when the result of an operation is too large or too small for the destination operand; otherwise, it is cleared.
  - SF - The sign flag is set when the result of an operation is negative or cleared when the result is positive. This flag is also set when the most significant bit is set after an arithmetic operation.
  - TF - The trap flag is used for debugging. The x86 processor will execute only one instruction at a time if this flag is set.
- EIP - instruction pointer. Contains the address of the next instruction to execute.

Reference: Michael Sikorski and Andrew Honig: **Practical Malware Analysis, The Hands-On Guide to Dissecting Malicious Software**. No Starch Press. ISBN: 978-1-593-27290-6

Data Transfer Instructions

- `mov` - moves data between locations
- `mov destination,source`
- `mov eax, 10h` - copies 0x10 into the EAX register
- `mov eax,ebx` - copies the value in EBX to EAX
- `mov eax,[0x4053AB]` - copies four bytes located 0x4053AB in the memory into the EAX register
- `mov eax,[ebx]` - copies the 4 bytes at the memory location specified by the EBX register into the EAX register
- `mov eax, [ebx+esi*4]` - copies the 4 bytes at the memory location specified by the result of the equation ebx+esi*4 into the EAX register

Data Transfer Instructions

- `mov [0x4053AB],eax` - copies the 4-byte value in the EAX register to the 0x4053AB memory location
- `mov [ebx],eax` - copies the 4-byte value in the EAX register to the memory location specified by the EBX register

Data Transfer Instructions

- `lea` - load effective address
- useful to calculate values
- `lea destination,source`
- `lea eax,[ebx+8]` - copy EBX+8 to the EAX register
- `mov eax,[ebx+8]` - copy the value at memory address specified by EBX+8 to the EAX register

Arithmetic Operations

- add destination,source
- sub destination,source
- sub instruction : zero flag (ZF) is set if the result is zero
  carry flag (CF) is set if destination is less than source
- inc - increment a register or a memory location by one
- dec - decrement a register or a memory location by one
- `add eax,10h`
- `add eax,ebx`
- `sub eax,10h`
- `inc eax`
- `dec edx`

Arithmetic Operations

- `mul` - multiplies the EAX register with the given value, the result is stored in EDX and EAX (EDX stores the most significant 32 bits)
- `div` - divides the 64-bit value stored in EDX and EAX by the given value, the quotient is stored in EAX, the remainder in EDX
- `mul value`
- `div value`
- `mul 10h`
- `mul ebx`
- `mul bx`
- `div ebx`
- `div 75h`

Bitwise Operations

- `not value`
- `xor destination,source`
- `or destination,source`
- `and destination,source`
- `xor eax,eax` - clears the EAX register
- `shr, shl` - shift registers, CF flag contains the last bit shifted out
- `shr destination,count`
- `shl destination, count`
- `ror, rol` - rotate registers
- `ror destination,count`
- `rol destination, count`

Bitwise Operations

- `not value`
- `xor destination,source`
- `or destination,source`
- `and destination,source`
- `xor eax,eax` - clears the EAX register
- `shr, shl` - shift registers, CF flag contains the last bit shifted out
- `shr destination,count`
- `shl destination, count`
- `ror, rol` - rotate registers
- `ror destination,count`
- `rol destination, count`

Conditionals

- `test` - same as `and`, but only sets the flags, doesn't store the result
- `test eax,eax` - ZF set to 1 if the result is 0
- `cmp` - same as sub, but only sets the flags, doesn't store the result
- `cmp destination, source`
- dest = src -> ZF=1, CF=0
- dest < src -> ZF=0, CF=1
- dest > src -> ZF=0, CF=0

Branching

- A branch is a sequence of code that is conditionally executed depending on the flow of the program
- `jmp loc` - unconditional jump
- conditional jumps
  - jz loc / je loc- jump if zero, ZF = 1
  - jnz loc / jne loc - jump if not zero, ZF = 0
  - jg loc / jnle loc - jump if greater, ZF = 0 and SF = 0
  - jge loc / jnl - jump if greater or equal, SF = 0
  - jl loc / jnge loc - jump if less, SF = 1
  - jle loc / jng - jump if less or equal, SF = 1 or ZF = 1
  - etc.

Stack

- Every thread has its stack
- Memory area for function parameters, local variables, return addresses
- LIFO structure - last in, first out
- Grows from higher addresses to lower addresses
- `push source` - copy value from source on top of the stack
- `pop dectionation` - pop value from top of the stack to destination
- EBP - base pointer
- ESP - stack pointer, points to the top of the stack

Functions

- Functions are block of codes for a specific task

`call function_location`

- function_location is copied to EIP and the address of the next instruction is pushed to the stack

`ret` - return instruction

- Pops the top of the stack to the EIP register

Functions

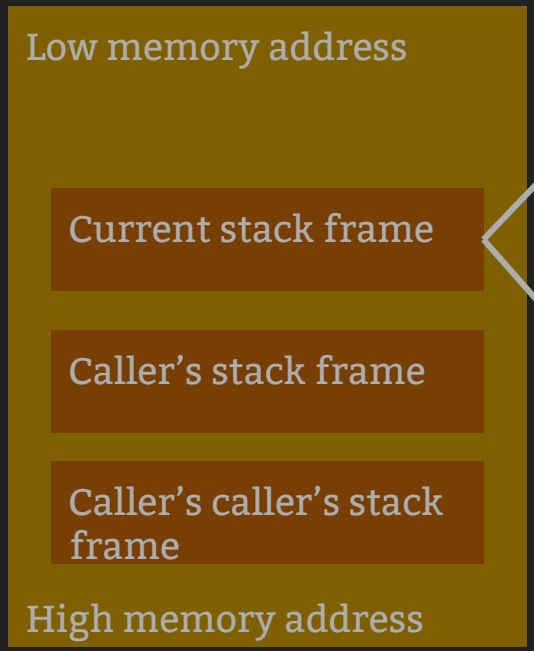
Prologue

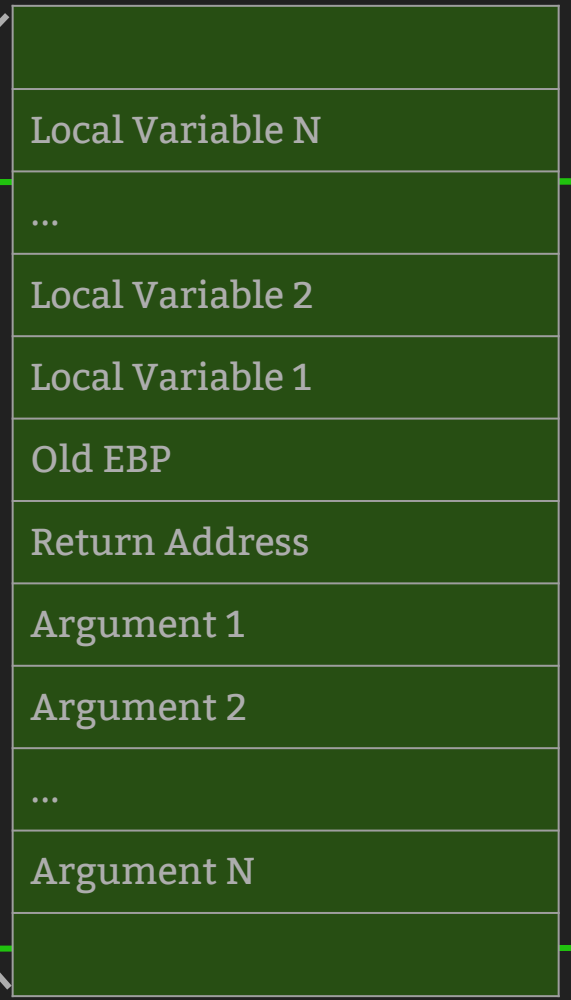
```
push ebp        saves ebp on the stack (can be restored after return)
mov ebp,esp     esp and ebp points to the top of the stack
sub esp,8       allocate space for local variables
```

Epilogue

```
mov esp,ebp         restore stack
pop ebp
ret
```

Functions


Low memory address
Current stack frame
Caller's stack frame
Caller's caller's stack frame
High memory address


Local Variable N
...
Local Variable 2
Local Variable 1
Old EBP
Return Address
Argument 1
Argument 2
...
Argument N
ESP
EBP

Function call

1. arguments pushed to the stack
2. call function_location (EIP = function_location, return address pushed to the stack)
3. EBP pushed to the stack, space is allocated for local variables
4. function execution
5. stack is restored - local variables are freed and EBP is restored
6. return - EIP = return address, execution continues after the call instruction

IDA

- Interactive DisAssembler
- IDA is a cross-platform, multi-processor disassembler and debugger developed by Hex-Rays

LET'S PLAY