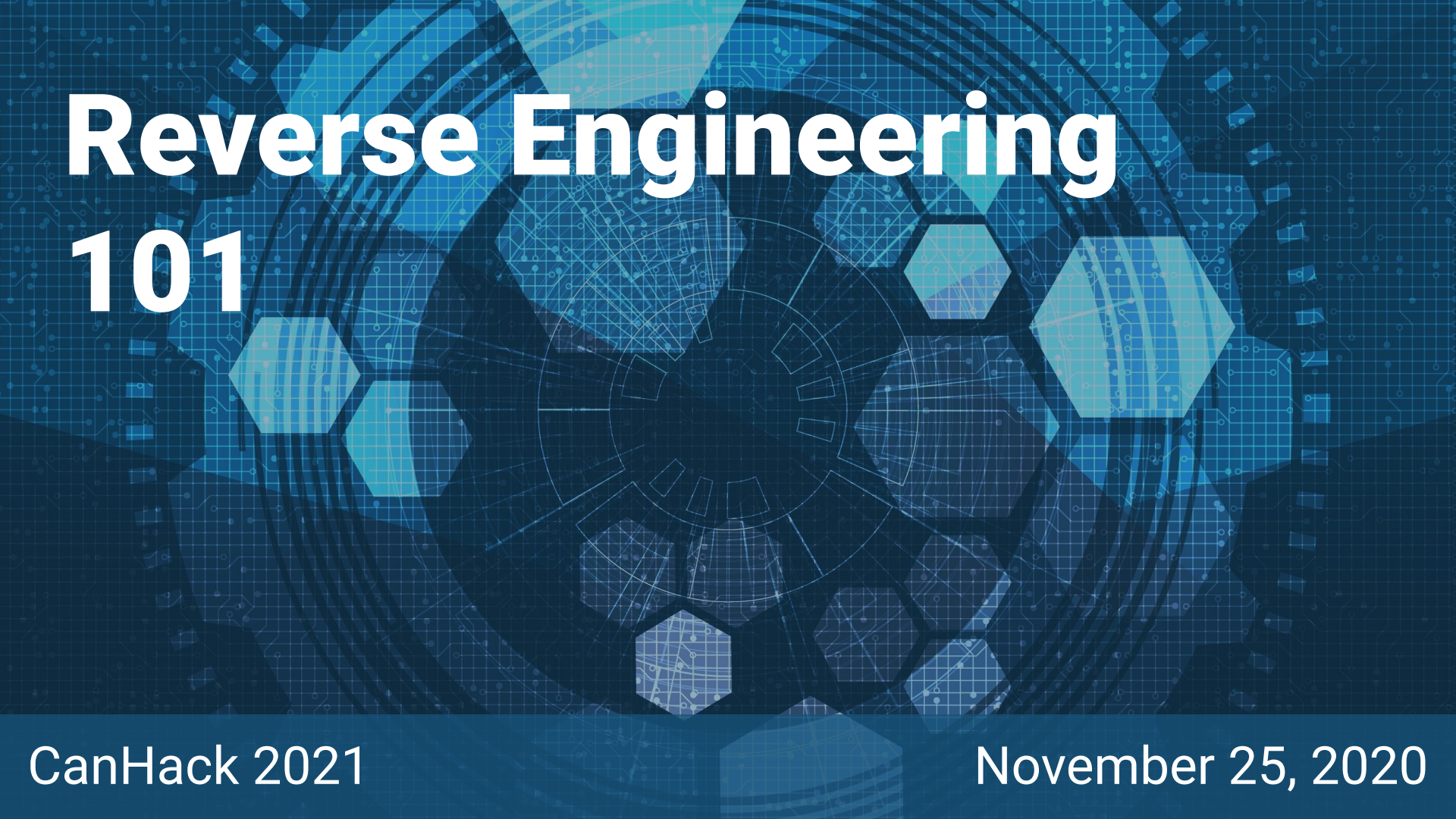
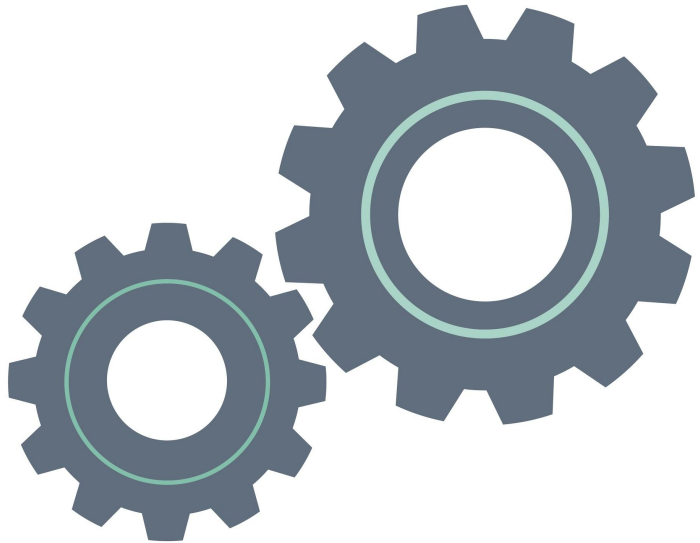

Reverse Engineering
101
CanHack 2021
November 25, 2020

Overview of Reverse Engineering

- What is Reverse Engineering
- How is Reverse Engineering used in CTFs
- Java
- Assembly Language
- What are registers and how do they work?
- Assembly Instructions
- Android Reverse Engineering



What is Reverse Engineering?

Taking something apart and putting it back together again to understand how it works

Uses:

- Analyze malware and malicious programs to understand how they work to prevent against them
- Breaking down code to better understand the potential vulnerability of a software
- Understand how certain parts of the program work

What needs to be Reverse Engineered?

- Code
- Binary Files
- Assembly Instructions
- Malware
- Applications
- Programs written in Java, Python, C

Skills & Tools that are useful for reverse engineering

- Basic Programming knowledge to understand programs (Java, Python etc.)
- How Assembly Instructions, Registers work
- Memory Allocation

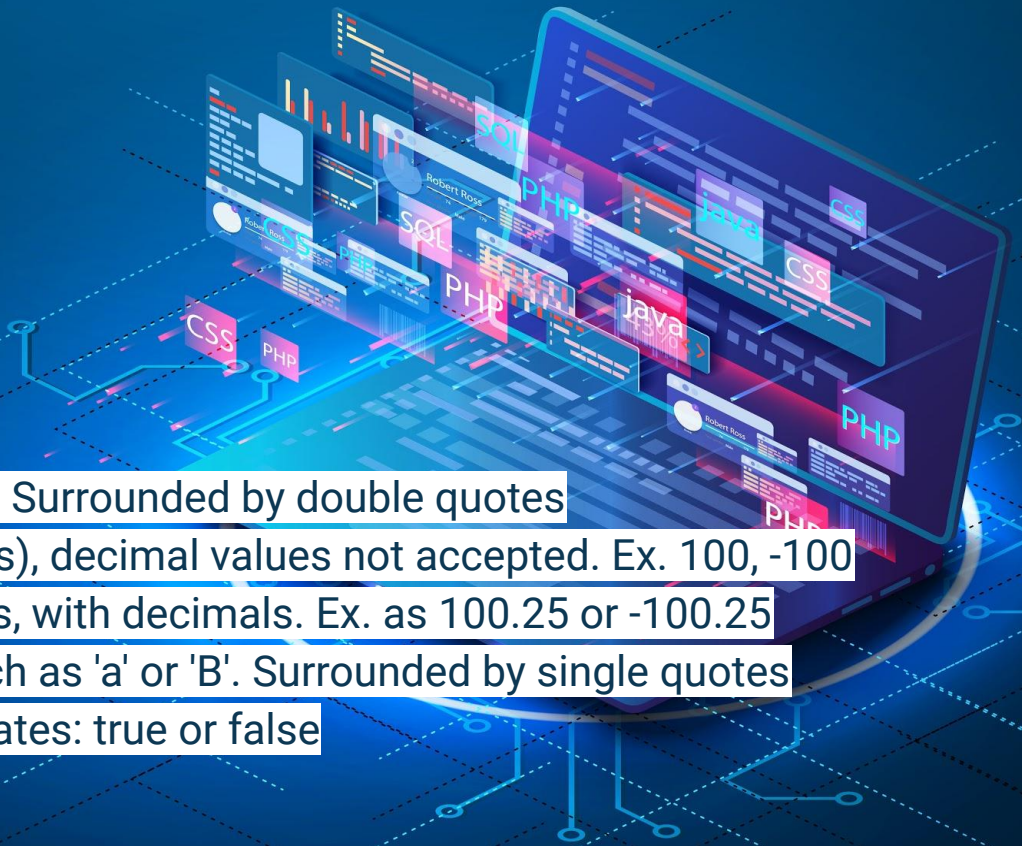
Tools:

- **Debuggers -** Allow you to step through another program one line at a time
- **Disassemblers -** Break down a compiled program into machine code
- **Decompilers -** Tool used to convert an executable program or low-level/machine language into a human readable format

Java

Java is a programming language

Used to build applications(desktop, mobile, web), games, and more



Variables

- string - stores text, such as "Hello". Surrounded by double quotes
- int - stores integers (whole numbers), decimal values not accepted. Ex. 100, -100
- float - stores floating point numbers, with decimals. Ex. as 100.25 or -100.25
- char - stores single characters, such as 'a' or 'B'. Surrounded by single quotes
- boolean - stores values with two states: true or false

Vault Door Training

```
import java.util.*;

class VaultDoorTraining {
    public static void main(String args[]) {
        VaultDoorTraining vaultDoor = new VaultDoorTraining();
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter vault password: ");
        String userInput = scanner.next();
                String input = userInput.substring("picoCTF{".length(),userInput.length()-1);
                if (vaultDoor.checkPassword(input)) {
                    System.out.println("Access granted.");
                } else {
                    System.out.println("Access denied!");
                }
    }

    // The password is below. Is it safe to put the password in the source code?
    // What if somebody stole our source code? Then they would know what our
    // password is. Hmm... I will think of some ways to improve the security
    // on the other doors.
    //
    // -Minion #9567
    public boolean checkPassword(String password) {
        return password.equals("w4rm1ng_Up_w1tH_jAv4_be8d9806f18");
    }
}
```

```
import java.util.*;

class VaultDoorTraining {
```

Every line of code that runs in Java must be inside a class.

The name of the java file must match the class name.

```
import java.util.*;

class VaultDoorTraining {
    public static void main(String args[]) {
```

The main() method is required and you will see it in every Java program

```
import java.util.*;

class VaultDoorTraining {
    public static void main(String args[]) {
        VaultDoorTraining vaultDoor = new VaultDoorTraining();
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter vault password: ");
        String userInput = scanner.next();
```

The Scanner class is used to get user input, and it is found in the java.util package. System.in tells the java compiler that system input will be provided through console(keyboard).

Asks the user for “Enter vault password:”

Get user input

```
import java.util.*;

class VaultDoorTraining {
    public static void main(String args[]) {
        VaultDoorTraining vaultDoor = new VaultDoorTraining();
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter vault password: ");
        String userInput = scanner.next();
            String input = userInput.substring("picoCTF{".length(),userInput.length()-1);
            if (vaultDoor.checkPassword(input)) {
                System.out.println("Access granted.");
            } else {
                System.out.println("Access denied!");
            }

}

    // The password is below. Is it safe to put the password in the source code?
    // What if somebody stole our source code? Then they would know what our
    // password is. Hmm... I will think of some ways to improve the security
    // on the other doors.
    //
    // -Minion #9567
    public boolean checkPassword(String password) {
        return password.equals("w4rm1ng_Up_w1tH_jAv4_be8d9806f18");
    }
}
```

Check the user.Input but the string “picoCTF{” and the length-1 which will be “}” this character is not checked

Compare if userinput = the string provided

If user input matches then print “Access granted” and if not then “Access denied”

Let’s take a look at some additional Java programs

Assembly Language

An assembly language is a low-level programming language designed for a specific type of processor.

Also called assembly or ASM

The instruction below tells an x86/IA-32 processor to move an immediate 8-bit value into a register:

```
10110000 01100001
```

This binary computer code can be made more human-readable by expressing it in hexadecimal:

```
B0 61
```

B0 means 'Move a copy of the following value into AL, and 61 is a hexadecimal representation of the value 01100001, which is 97 in decimal.

Assembly language for the 8086 family provides the mnemonic MOV (an abbreviation of move) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

```
MOV AL, 61h      ; Load AL with 97 decimal (61 hex)
```

Source: Wikipedia

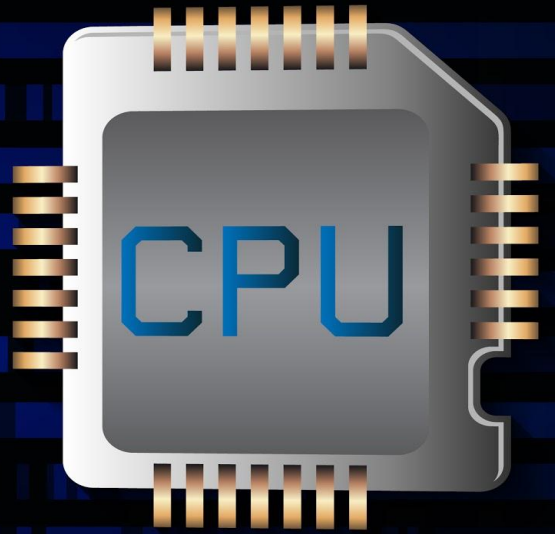
Central Processing Unit (CPU)

CPU comprises of:

- The Arithmetic Logic Unit (ALU)
  - The ALU consists of the Arithmetic Unit (responsible for mathematical functions) and Logic Unit (responsible for logical operations)

- The Control Unit (CU)
  - Controls and directs the main memory, (ALU), input and output devices, and is also responsible for the instructions that are sent to the CPU

- Registers
  - Small, extremely high-speed CPU storage locations where data can be efficiently read or manipulated, hold data temporarily



Registers

What do registers do?

- Holds temporary data that is is needed by the CPU to execute instructions
- Perform operations
- Store resulting data

Only hold a small amount of data, 64-bit architecture CPU’s hold 64 bits of data/register and 32-bit architecture CPU’s hold 32 bits of data/register

The CPU Architecture determines the design of the processor, instructions that are supported, size of registers and other factors.

Common architecture for processors is x86 developed by Intel

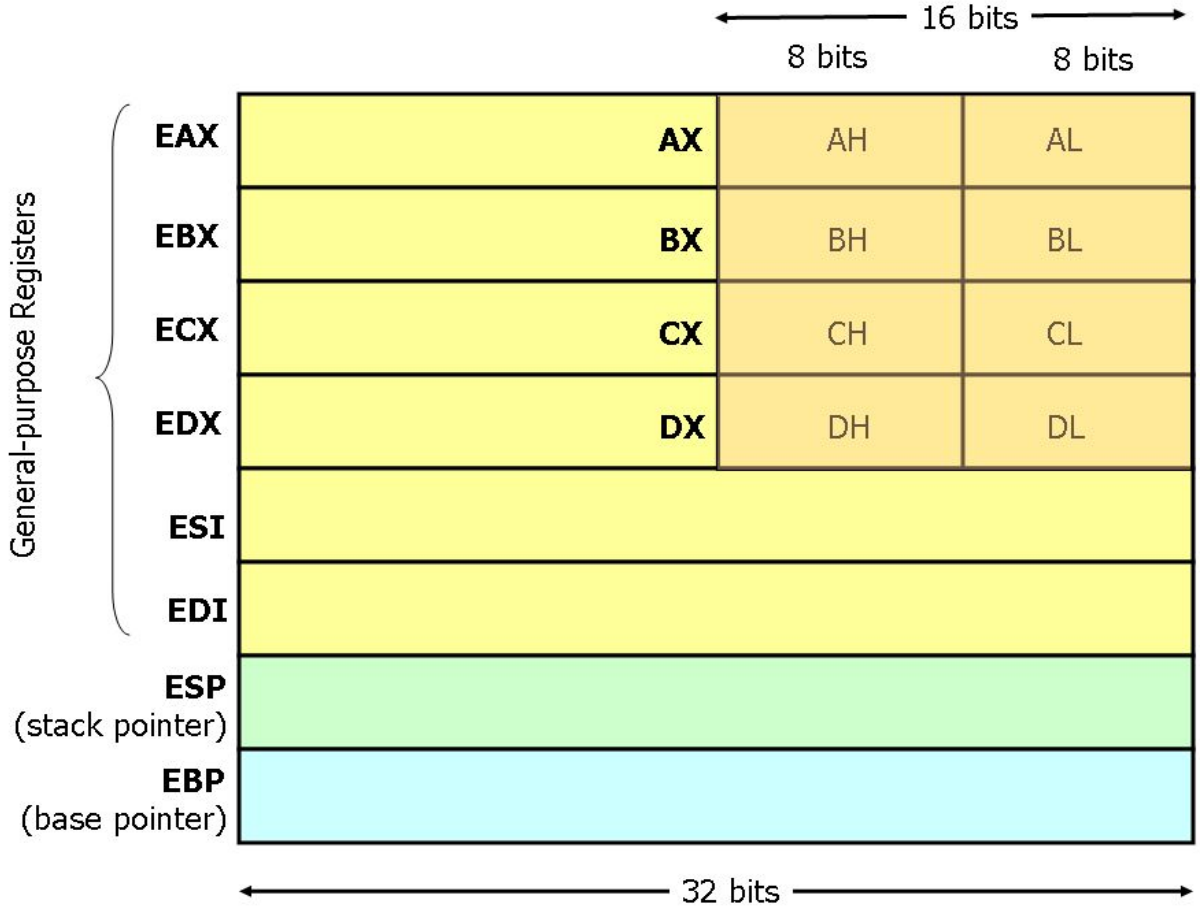


Image Source: University of Virginia Computer Science

EAX - Accumulator Register - used for storing operands and result data

EBX- Base register - Points to data

ECX - Counter Register - Loop operations

EDX- Data register. Input/output operations.

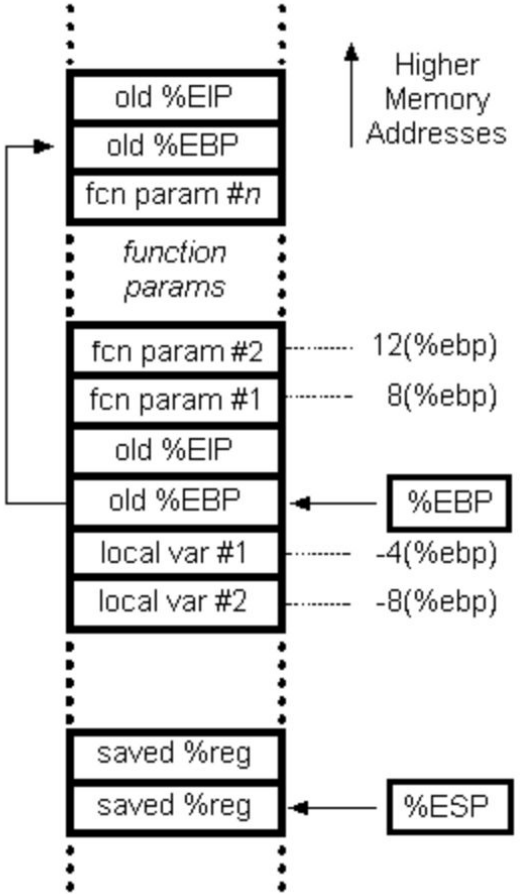
ESI/EDI- Source/Destination index for string operations.

ESP- Current position of data or address within the program stack, which changes automatically based on the operation

EBP- Frame pointer, contains the base address of the function's frame.

x86 architecture

- All x86 architectures use a stack as a temporary storage area in RAM that allows the processor to quickly store and retrieve data in memory
- Higher memory addresses are at the top of the stack
- LIFO (Last In First Out) Method is used, items that are “pushed” on top of the stack are “popped” first

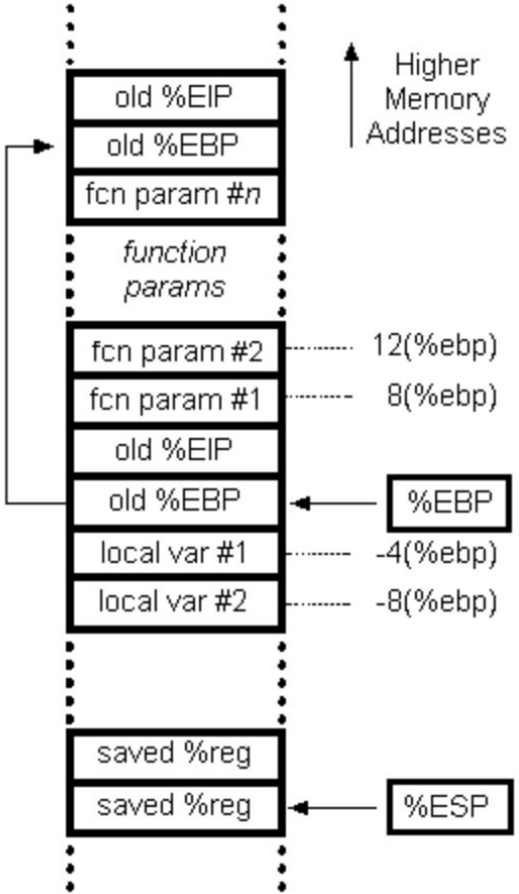


x86 architecture

- Data is stored using the Little Endian method
- 0x12345678 , it would be entered as 78, 56, 34, 12 into the stack
- In 32-bit registers, memory addresses of registers are 4 bytes apart
- By using a base pointer the return address will always be at ebp+4, the first parameter will always be at ebp+8, and the first local variable will always be at ebp-4

Note*

Two's complement is the standard way of representing negative integers in binary.



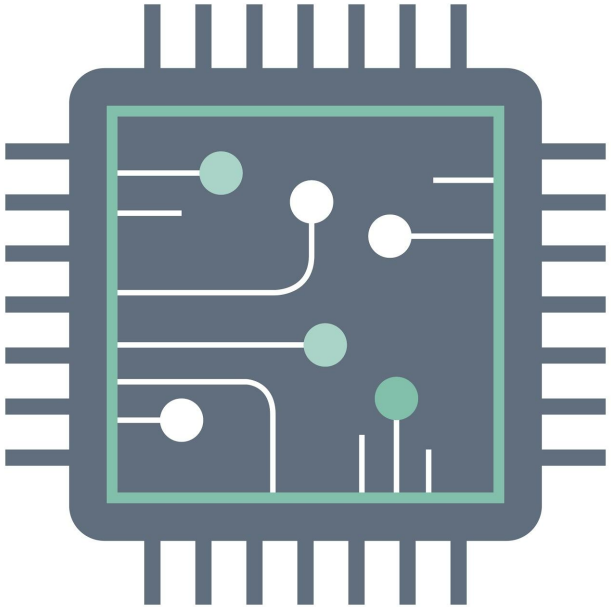
x86 assembly instructions

Assembly instructions represent a single operation for the CPU to perform.

Examples of Assembly instructions:

mov D, S	Move source to destination
add S, D	Add source to destination
je/jne	Jump when equal / Jump when not equal
jg	Jump when greater than

BYTE	00	1 byte/8 bits
WORD	00 00	2 bytes/ 16 bits
DWORD	00 00 00 00	4 bytes/ 32 bits



Additional Instructions

Prologue for x

The function prologue prepares the stack and registers for use within the function

A function prologue typically looks like this:

- Push current base pointer onto the stack, so it can be restored later.
- Assigns the value of base pointer to the address of stack pointer (which is pointed to the top of the stack) so that the base pointer will pointed to the top of the stack.
- Moves the stack pointer further by decreasing its value to make room for function's local variables.

```
push    ebp
mov     ebp, esp
sub     esp, N
```

Data 2 from Function	[ebp + 0xc]
Data 1 from Function	[ebp + 0x8]
Return address [ebp + 0x4]	
Old ebp	
Local variable 1	[ebp - 0x4]
Local variable 2	[ebp - 0x8]
	[ebp - 0xc]
	[ebp - 0x10]

Lets try reading some assembly instructions

```
asm1:
        <+0>:push   ebp
        <+1>:mov    ebp,esp
        <+3>:cmp    DWORD PTR [ebp+0x8],0x3fb
        <+10>:      jg     0x512 <asm1+37>
        <+12>:      cmp    DWORD PTR [ebp+0x8],0x280
        <+19>:      jne    0x50a <asm1+29>
        <+21>:      mov    eax,DWORD PTR [ebp+0x8]
        <+24>:      add    eax,0xa
        <+27>:      jmp    0x529 <asm1+60>
        <+29>:      mov    eax,DWORD PTR [ebp+0x8]
        <+32>:      sub    eax,0xa
        <+35>:      jmp    0x529 <asm1+60>
        <+37>:      cmp    DWORD PTR [ebp+0x8],0x559
        <+44>:      jne    0x523 <asm1+54>
        <+46>:      mov    eax,DWORD PTR [ebp+0x8]
        <+49>:      sub    eax,0xa
        <+52>:      jmp    0x529 <asm1+60>
        <+54>:      mov    eax,DWORD PTR [ebp+0x8]
        <+57>:      add    eax,0xa
        <+60>:      pop    ebp
        <+61>:      ret
```

2e0

Compare 2e0 with 3fb
2e0 is smaller than 3fb so don't jump

Compare 2e0 with 280
Jump if not equal to +29

Move 2e0 into eax → **eax = 2e0**

2e0 - 0xa= *2D6*

2e0	[ebp + 0x8]
Return address	[ebp + 0x4]
Old ebp	

ASM2

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31>
<+20>:   add    DWORD PTR [ebp-0x4],0x1
<+24>:   add    DWORD PTR [ebp-0x8],0xd1
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1
<+38>:   jle    0x501 <asm2+20>
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

What does asm2(0x4,0x2d) return?

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31>
<+20>:   add    DWORD PTR [ebp-0x4],0x1
<+24>:   add    DWORD PTR [ebp-0x8],0xd1
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1
<+38>:   jle    0x501 <asm2+20>
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

What does asm2(0x4,0x2d) return?

0x2d [ebp + 0xc]
0x4 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10      reserve 16 bytes on the stack
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31>
<+20>:   add    DWORD PTR [ebp-0x4],0x1
<+24>:   add    DWORD PTR [ebp-0x8],0xd1
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1
<+38>:   jle    0x501 <asm2+20>
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

0x2d [ebp + 0xc]
0x4 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp
[ebp - 0x4]
[ebp - 0x8]
[ebp - 0xc]
[ebp - 0x10]

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc] eax= 0x2d
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8] eax = 0x4
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31> jump to +31
<+20>:   add    DWORD PTR [ebp-0x4],0x1
<+24>:   add    DWORD PTR [ebp-0x8],0xd1
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1
<+38>:   jle    0x501 <asm2+20>
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

~~0x2d~~ [ebp + 0xc]
~~0x4~~ [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp
0x2d [ebp - 0x4]
0x4 [ebp - 0x8]
[ebp - 0xc]
[ebp - 0x10]

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31> jump to +31
<+20>:   add    DWORD PTR [ebp-0x4],0x1
<+24>:   add    DWORD PTR [ebp-0x8],0xd1
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1 0x4 < 0x5fa1
<+38>:   jle    0x501 <asm2+20> jump if lower than to +20
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

[ebp + 0xc]
[ebp + 0x8]
Return address [ebp + 0x4]
Old ebp
0x2d [ebp - 0x4]
0x4 [ebp - 0x8]
[ebp - 0xc]
[ebp - 0x10]

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31>
<+20>:   add    DWORD PTR [ebp-0x4],0x1 0x2d + 0x1 = 2e
<+24>:   add    DWORD PTR [ebp-0x8],0xd1    0x4 + 0xd1 = d5
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1 d5 < value
<+38>:   jle    0x501 <asm2+20>  Jump again to 20
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

This process will keep taking place until the value at [ebp-0x8] > 0x5fa1

Easier to calculate how many times 0xd1 needs to be added into [ebp - 0x8] so that we can jump

[ebp + 0xc]	
[ebp + 0x8]	
Return address [ebp + 0x4]	
Old ebp	
0x2e	[ebp - 0x4]
d5	[ebp - 0x8]
	[ebp - 0xc]
	[ebp - 0x10]

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31>
<+20>:   add    DWORD PTR [ebp-0x4],0x1 0x2d + 0x1 = 2e
<+24>:   add    DWORD PTR [ebp-0x8],0xd1    0x4 + 0xd1 = d5
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1 d5 < value
<+38>:   jle    0x501 <asm2+20>  Jump again to 20
```

[ebp + 0xc]	
[ebp + 0x8]	
Return address [ebp + 0x4]	
Old ebp	
0x2d	[ebp - 0x4]
0x4	[ebp - 0x8]
	[ebp - 0xc]
	[ebp - 0x10]

0xd1(z) + 0x4 = 0x5fa1
0xd1(z) = 5f9d
z = 75 remainder 18

z needs to be greater than 75 so it will have to added 76 times

asm2:

```
<+0>:    push   ebp
<+1>:    mov    ebp,esp
<+3>:    sub    esp,0x10
<+6>:    mov    eax,DWORD PTR [ebp+0xc]
<+9>:    mov    DWORD PTR [ebp-0x4],eax
<+12>:   mov    eax,DWORD PTR [ebp+0x8]
<+15>:   mov    DWORD PTR [ebp-0x8],eax
<+18>:   jmp    0x50c <asm2+31>
<+20>:   add    DWORD PTR [ebp-0x4],0x1 0x2d + 0x1 = 2e
<+24>:   add    DWORD PTR [ebp-0x8],0xd1    0x4 + 0xd1 = d5
<+31>:   cmp    DWORD PTR [ebp-0x8],0x5fa1 d5 < value
<+38>:   jle    0x501 <asm2+20>  Jump again to 20
<+40>:   mov    eax,DWORD PTR [ebp-0x4]
<+43>:   leave
<+44>:   ret
```

0xd1(76) + 0x4 = 0x605a > 0x5fa1

What is the value at [epb -0x4] 0x1(76) + 0x2d =

<u>0xa3</u>

[ebp + 0xc]	
[ebp + 0x8]	
Return address [ebp + 0x4]	
Old ebp	
0xa3	[ebp - 0x4]
0x605a	[ebp - 0x8]
	[ebp - 0xc]
	[ebp - 0x10]

ASM3

```
asm3:
        <+0>:     push   ebp
        <+1>:     mov    ebp,esp
        <+3>:     xor    eax,eax
        <+5>:     mov    ah,BYTE PTR [ebp+0xa]
        <+8>:     shl    ax,0x10
        <+12>:    sub    al,BYTE PTR [ebp+0xc]
        <+15>:    add    ah,BYTE PTR [ebp+0xd]
        <+18>:    xor    ax,WORD PTR [ebp+0x10]
        <+22>:    nop
        <+23>:    pop    ebp
        <+24>:    ret
```

What does asm3(0xd73346ed,0xd48672ae,0xd3c8b139) return?

```
asm3:
        <+0>:     push   ebp
        <+1>:     mov    ebp,esp
        <+3>:     xor    eax,eax
        <+5>:     mov    ah,BYTE PTR [ebp+0xa]
        <+8>:     shl    ax,0x10
        <+12>:    sub    al,BYTE PTR [ebp+0xc]
        <+15>:    add    ah,BYTE PTR [ebp+0xd]
        <+18>:    xor    ax,WORD PTR [ebp+0x10]
        <+22>:    nop
        <+23>:    pop    ebp
        <+24>:    ret
```

What does asm3(0xd73346ed,0xd48672ae,0xd3c8b139) return?

39 b1 c8 d3 [ebp + 0x10]
ae 72 86 d4 [ebp + 0xc]
ed 46 33 d7 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp

```
asm3:
    <+0>:    push   ebp
    <+1>:    mov    ebp,esp
    <+3>:    xor    eax,eax        sets eax to 0
    <+5>:    mov    ah,BYTE PTR [ebp+0xa]
    <+8>:    shl    ax,0x10        clear the register
    <+12>:   sub    al,BYTE PTR [ebp+0xc]
    <+15>:   add    ah,BYTE PTR [ebp+0xd]
    <+18>:   xor    ax,WORD PTR [ebp+0x10]
    <+22>:   nop
    <+23>:   pop    ebp
    <+24>:   ret
```

What does asm3(0xd73346ed,0xd48672ae,0xd3c8b139) return?

39 b1 c8 d3 [ebp + 0x10]
ae 72 86 d4 [ebp + 0xc]
ed 46 33 d7 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp

			AH	AL
EAX	00	00	33	00

After shifting by 16, the register has been cleared.

```
asm3:
    <+0>:     push   ebp
    <+1>:     mov    ebp,esp
    <+3>:     xor    eax,eax
    <+5>:     mov    ah,BYTE PTR [ebp+0xa]
    <+8>:     shl    ax,0x10
    <+12>:    sub    al,BYTE PTR [ebp+0xc]
    <+15>:    add    ah,BYTE PTR [ebp+0xd]
    <+18>:    xor    ax,WORD PTR [ebp+0x10]
    <+22>:    nop
    <+23>:    pop    ebp
    <+24>:    ret
```

What does asm3(0xd73346ed,0xd48672ae,0xd3c8b139) return?

00 - ae = -ae

For negative hex numbers we have to get the TWO's complement which is 52

39 b1 c8 d3 [ebp + 0x10]
ae 72 86 d4 [ebp + 0xc]
ed 46 33 d7 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp

			AH	AL
EAX	00	00	00	52

```
asm3:
    <+0>:     push   ebp
    <+1>:     mov    ebp,esp
    <+3>:     xor    eax,eax
    <+5>:     mov    ah,BYTE PTR [ebp+0xa]
    <+8>:     shl    ax,0x10
    <+12>:    sub    al,BYTE PTR [ebp+0xc]
    <+15>:    add    ah,BYTE PTR [ebp+0xd]
    <+18>:    xor    ax,WORD PTR [ebp+0x10]
    <+22>:    nop
    <+23>:    pop    ebp
    <+24>:    ret
```

00 + 72 = 72

What does asm3(0xd73346ed,0xd48672ae,0xd3c8b139) return?

39 b1 c8 d3 [ebp + 0x10]
ae 72 86 d4 [ebp + 0xc]
ed 46 33 d7 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp

			AH	AL
EAX	00	00	72	52

```
asm3:
    <+0>:     push   ebp
    <+1>:     mov    ebp,esp
    <+3>:     xor    eax,eax
    <+5>:     mov    ah,BYTE PTR [ebp+0xa]
    <+8>:     shl    ax,0x10
    <+12>:    sub    al,BYTE PTR [ebp+0xc]
    <+15>:    add    ah,BYTE PTR [ebp+0xd]
    <+18>:    xor    ax,WORD PTR [ebp+0x10]
    <+22>:    nop
    <+23>:    pop    ebp
    <+24>:    ret
```

7252 xor b139

0xc36b

What does asm3(0xd73346ed,0xd48672ae,0xd3c8b139) return?

39 b1 c8 d3 [ebp + 0x10]
ae 72 86 d4 [ebp + 0xc]
ed 46 33 d7 [ebp + 0x8]
Return address [ebp + 0x4]
Old ebp

			AH	AL
EAX	00	00	72	52

Android Reverse Engineering

Reverse Engineering APK (Android Package Kit) files

An APK file is an app created for Android, Google's mobile operating system.

It’s helpful to view these files using an Android Emulator

Reverse Engineering APK files purpose:

- To better understand how an app or the features work
- Determine if there is anything malicious taking place on the app
- Libraries they are using

Tools for Android Reverse Engineering

Android Emulators- simulates Android devices on your computer

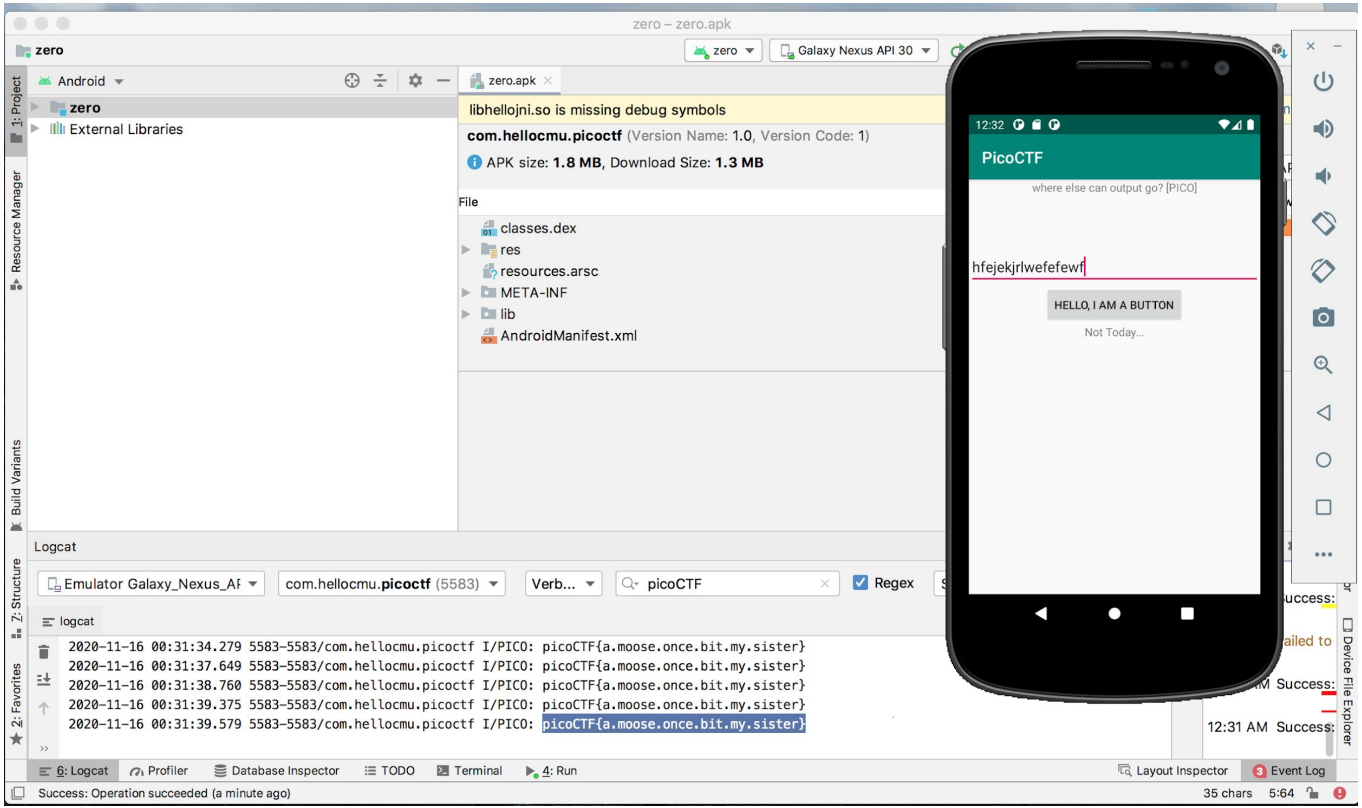
Android Studio

APK decompilers

- JADX
- Apktool

Droids0

Analyze App log output using Android Developer Studio

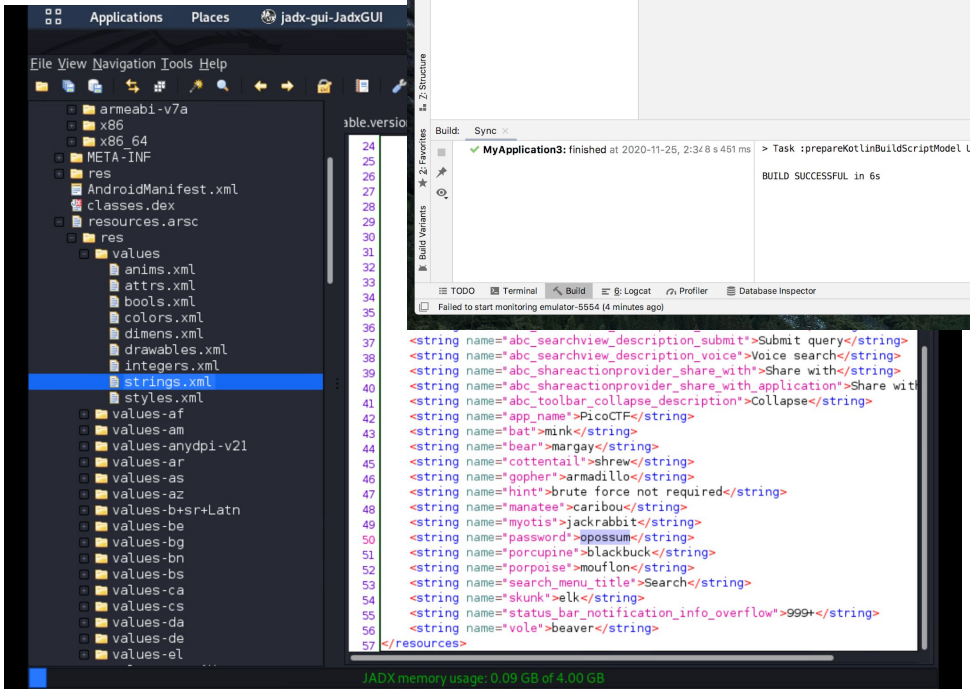


Droids1

Analyze APK file contents using JADX

Find password string

Use an emulator to input the password to get the flag



Resources

Java:

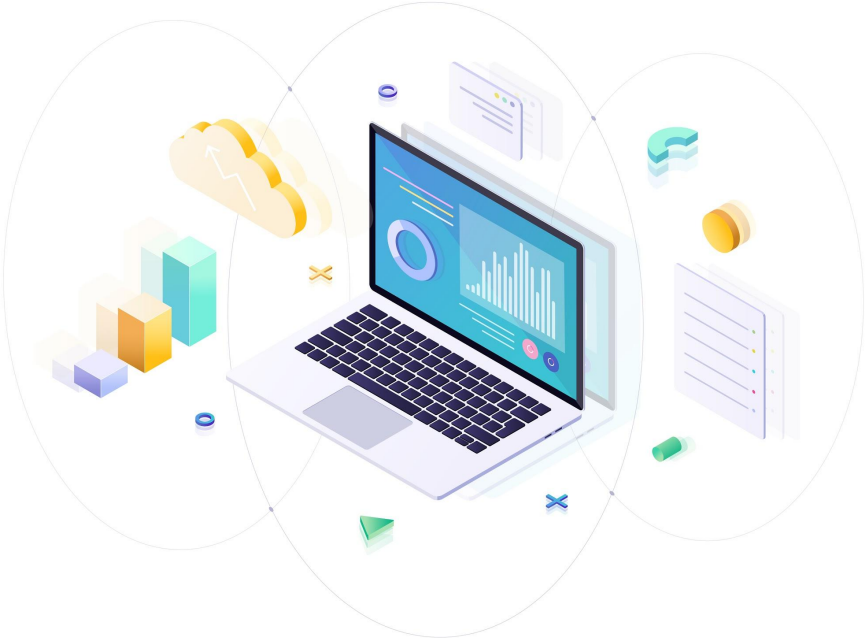
https://www.w3schools.com/java/default.asp

Assembly Language:

https://www.secjuice.com/guide-to-x86-assembly/
https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
http://unixwiz.net/techtips/win32-callconv-asm.html
Hex Calculator for mathematical operations
Two complements calculator for Hex

Android:

https://developer.android.com/studio
JADX
Apktool



Thank you!

Questions?

See you next week for Binary Exploitation 101!