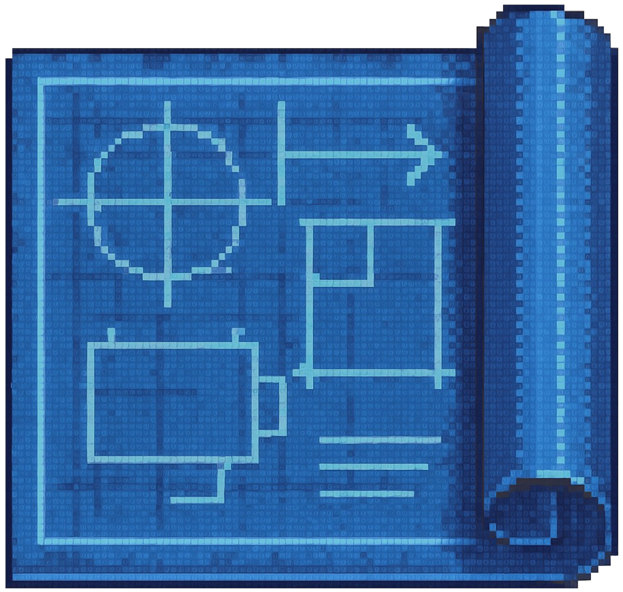
# HackerFrogs Afterschool
# Elf Reversing /w TryHackMe

Class:
Reverse Engineering

Workshop Number:
AS-REV-03

Document Version:
1.0

Special Requirements:
Registered account at tryhackme.com



## Welcome to HackerFrogs Afterschool!

HackerFrogs Afterschool is a cybersecurity program for learning beginner cybersecurity skills across a wide variety of subjects.

This workshop is the intro class to Reverse Engineering.

## What is Reverse Engineering?

According to Wikipedia, reverse engineering is a process or method through which one attempts to understand through deductive reasoning how a previously-made--

# What is Reverse Engineering?

device, process, system, or piece of software accomplishes a task with very little (if any) insight into exactly how it does so.

## What is Reverse Engineering?

In the case of cybersecurity, we'll be focusing on software reverse engineering, and in the realm of software reverse engineering, focusing on...

# What are x86 Elf Binary Executables?

x86 Linux ELF binaries

Executable and Linkable Format (ELF) is the standard format for executable binary files on Unix-like devices (such as Linux). But what about x86?

## x86 Chipset Instructions

x86 is the most-common CPU chipset used for PCs (desktop) and laptop computers, so we will learn the Assembly language associated with it to study reverse engineering

x86

## x86 Chipset Instructions

x86 is the most-common CPU chipset used for PCs (desktop) and laptop computers, so we will learn the Assembly language associated with it to study reverse engineering

## x86 Assembly Language

Assembly languages are the the lowest level programming languages, for any computing device. Low-level, in this context means it is the closest to raw computer instructions

## x86 Assembly Language

There is no one Assembly language, but there's a separate Assembly language for each type of computer processor

## Pico CTF

The CTF platform we will be using to learn basic digital forensics is called Pico CTF, which is one of the most well-known and well-respected CTF games, and is affiliated with Carnegie Mellon University.

# Pico CTF

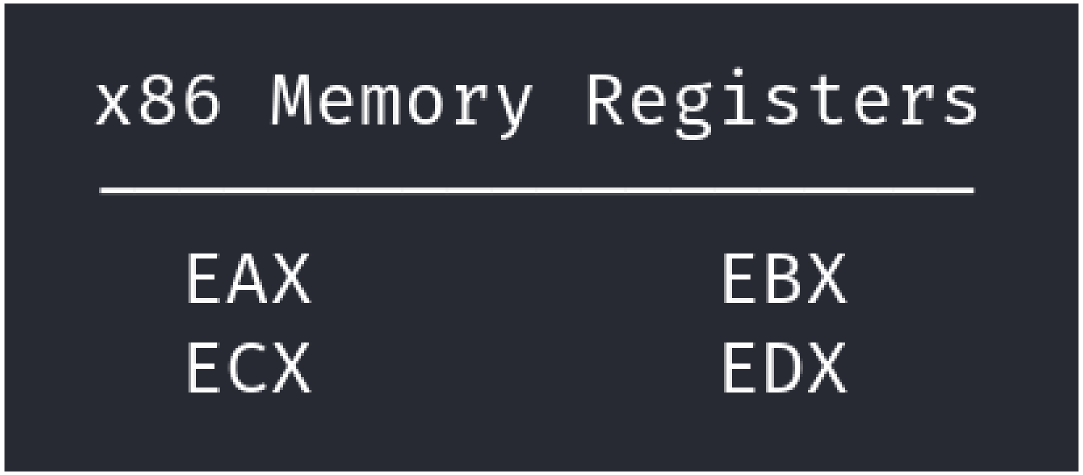
The Pico CTF platform has many challenges across different categories and difficulty levels. For this lesson, we'll be looking at Pico CTF challenges in the Reverse Engineering category.

# Pico CTF – Bit-O-Asm-1 Challenge

Let's learn more about x86 Assembly programming with a challenge from PicoCTF!

https://play.picoctf.org/practice/challenge/391

# x86 Memory Registers

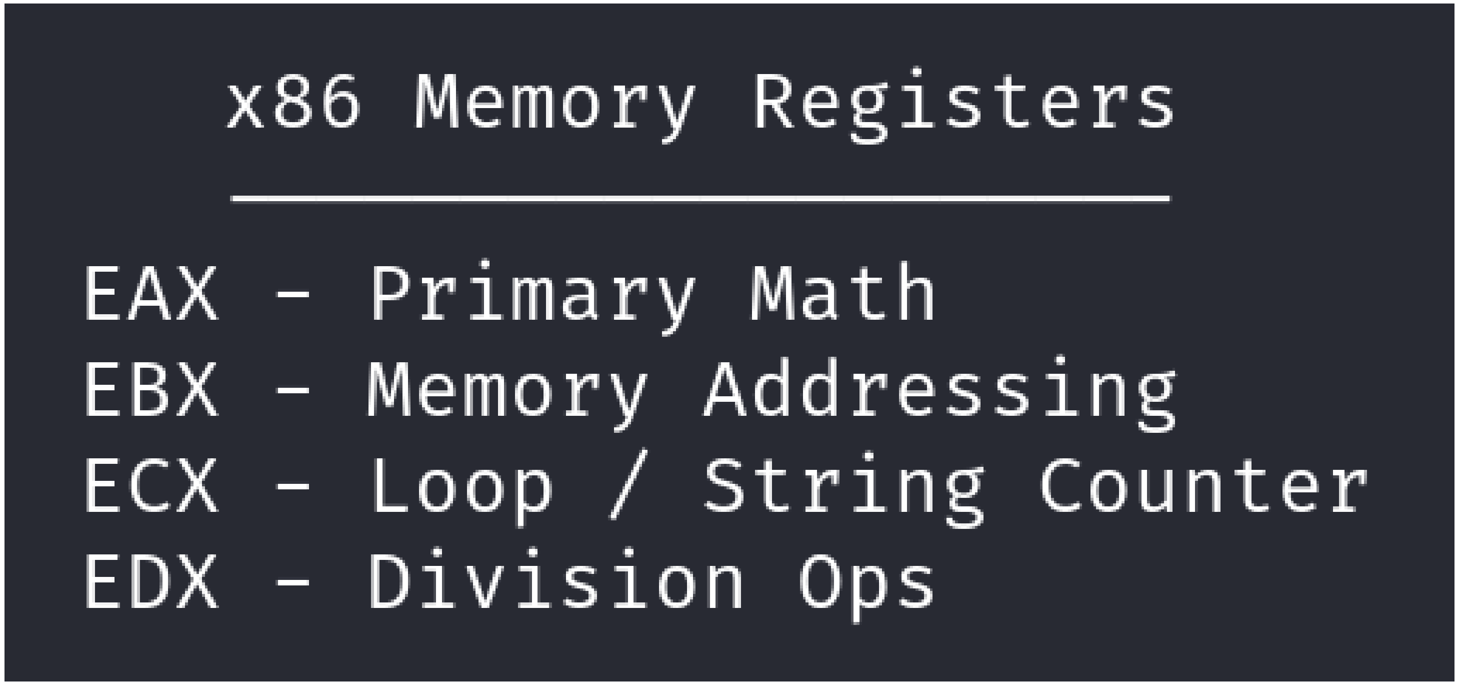


x86 Memory Registers

| | |
|---|---|
| EAX | EBX |
| ECX | EDX |

The challenge talks about the EAX register, which is a storage location in the a CPU's memory. There are many registers in x86 CPUs, but for now we'll just talk about these four commonly-used registers, EAX, EBX, ECX, and EDX

## x86 Memory Registers

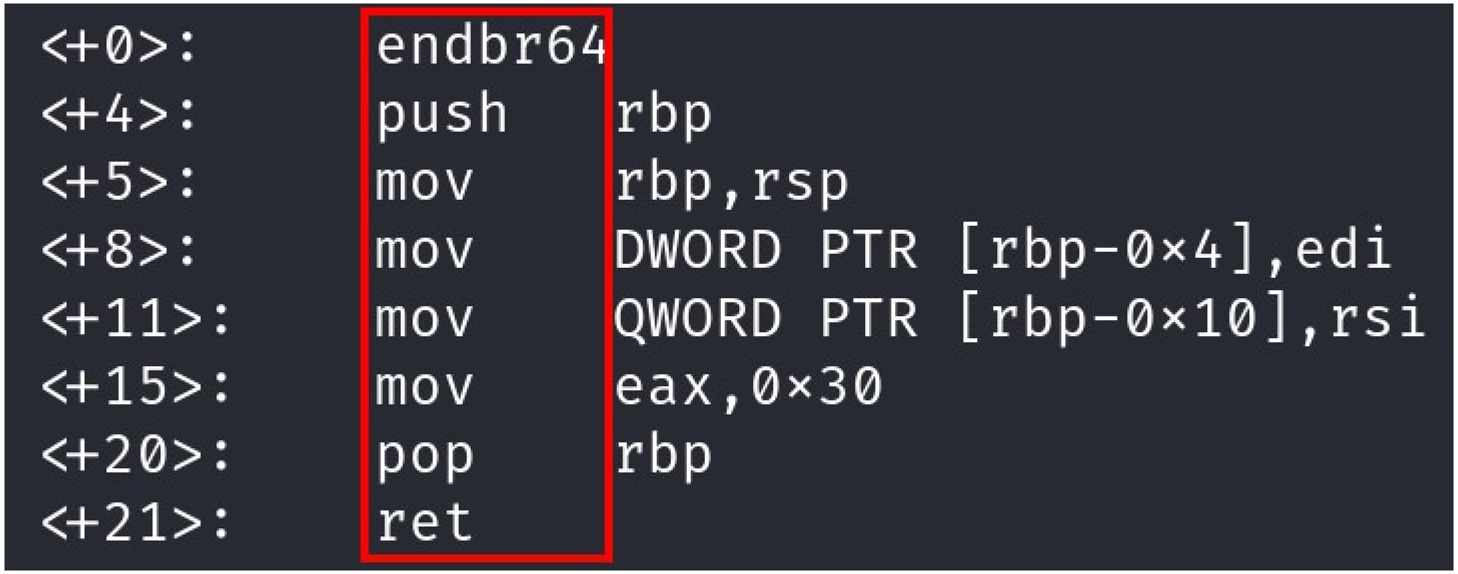


```
   x86 Memory Registers
   ____________________

EAX - Primary Math
EBX - Memory Addressing
ECX - Loop / String Counter
EDX - Division Ops
```

This summary shows what kind of data the registers were intended to hold, but any register can hold data for any purpose

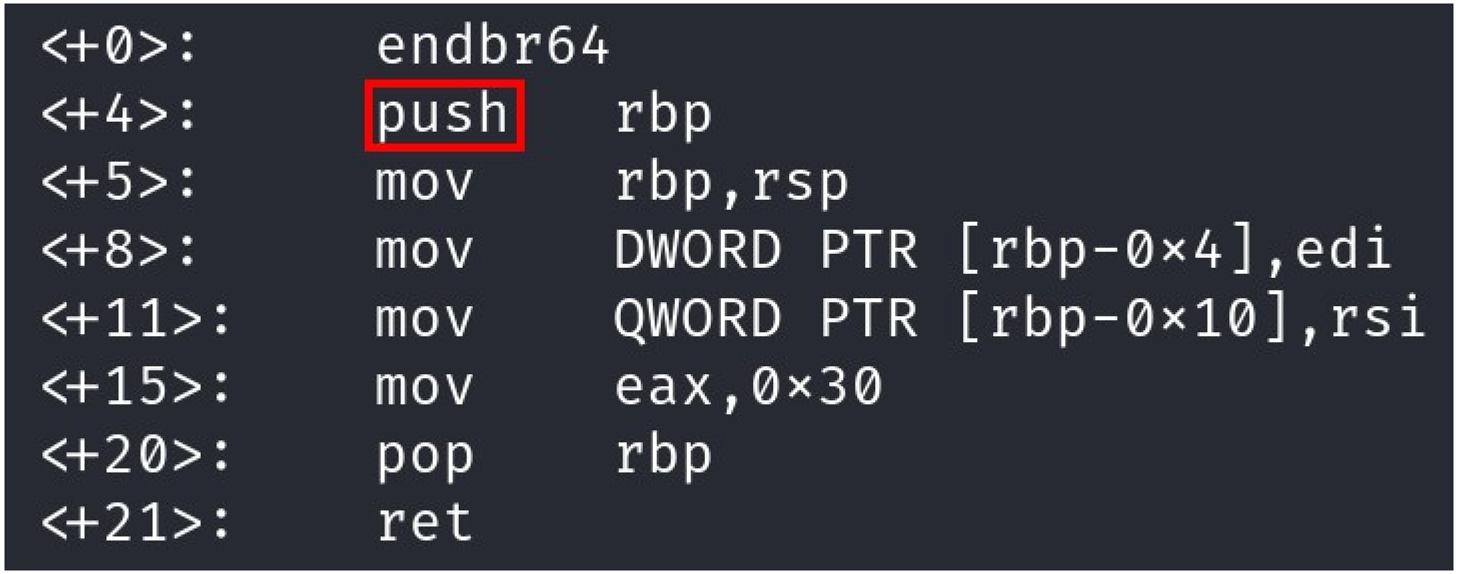
# x86 Instructions



```
<+0>:      endbr64
<+4>:      push   rbp
<+5>:      mov    rbp,rsp
<+8>:      mov    DWORD PTR [rbp-0×4],edi
<+11>:     mov    QWORD PTR [rbp-0×10],rsi
<+15>:     mov    eax,0×30
<+20>:     pop    rbp
<+21>:     ret
```

In the Assembly program dump, we see that there five different types of instructions

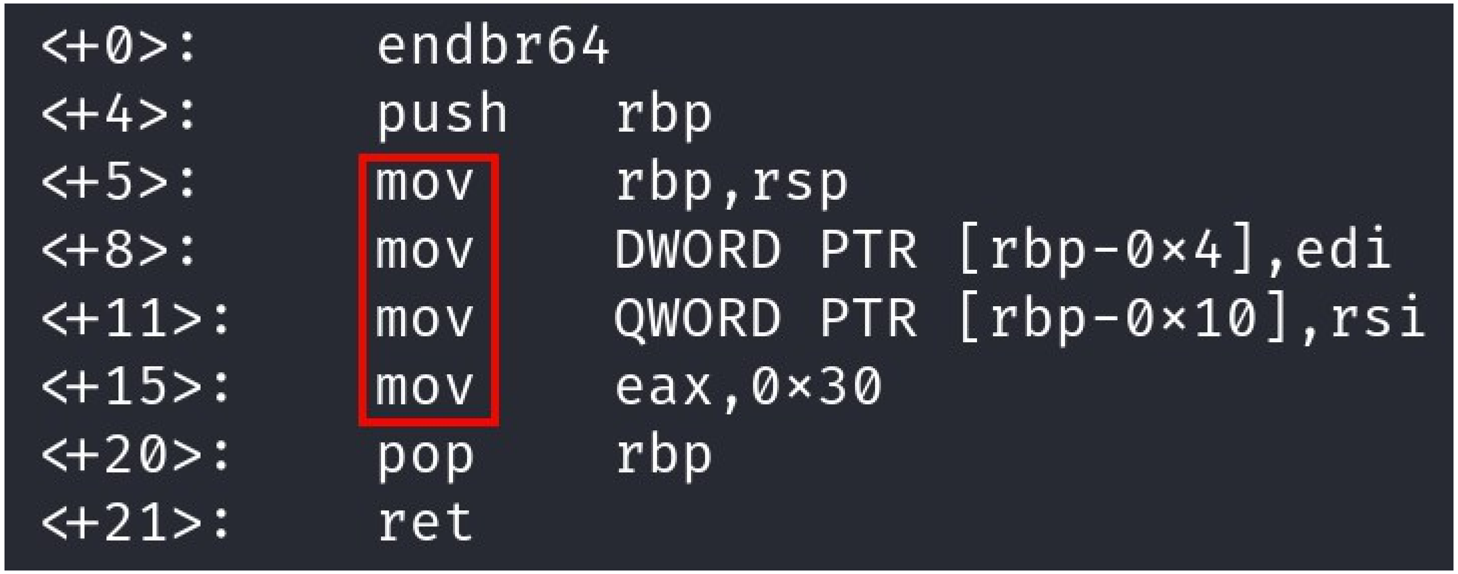
## Push Instruction



```
<+0>:     endbr64
<+4>:     push    rbp
<+5>:     mov     rbp,rsp
<+8>:     mov     DWORD PTR [rbp-0x4],edi
<+11>:    mov     QWORD PTR [rbp-0x10],rsi
<+15>:    mov     eax,0x30
<+20>:    pop     rbp
<+21>:    ret
```

The Push instruction is used to put data on top of the memory stack, which is a special storage location

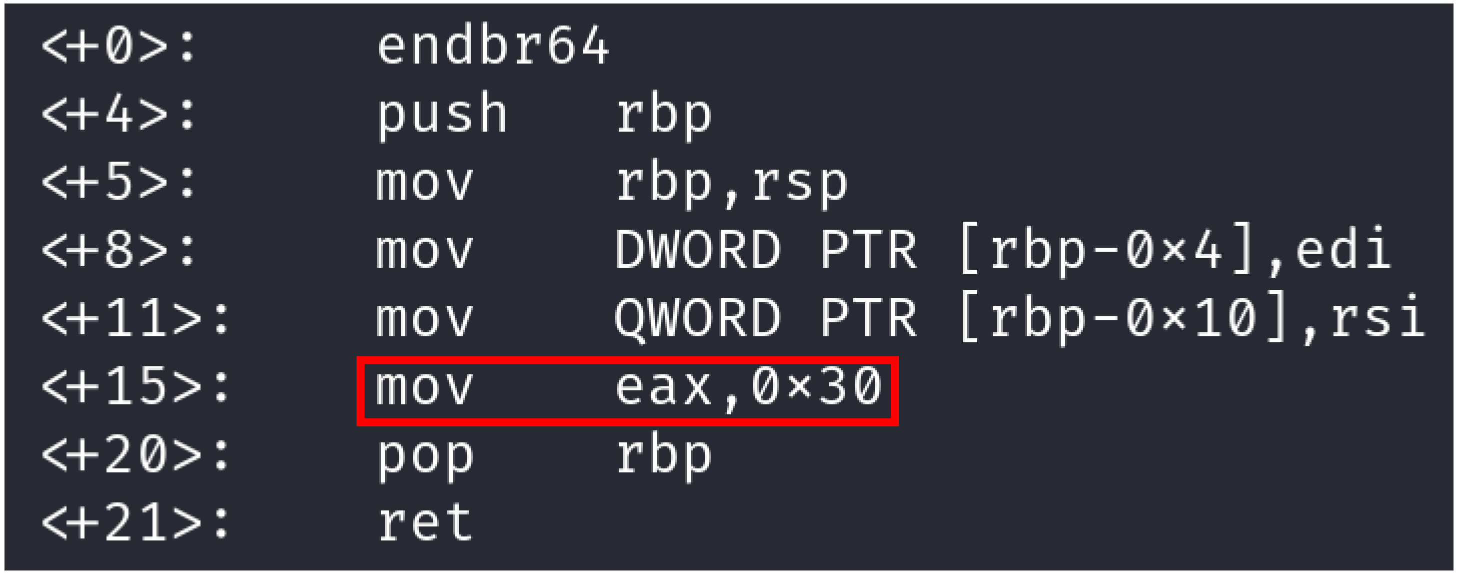
## Mov Instruction



```
<+0>:     endbr64
<+4>:     push   rbp
<+5>:     mov    rbp,rsp
<+8>:     mov    DWORD PTR [rbp-0x4],edi
<+11>:    mov    QWORD PTR [rbp-0x10],rsi
<+15>:    mov    eax,0x30
<+20>:    pop    rbp
<+21>:    ret
```

The Mov instruction is used to put data into a memory register

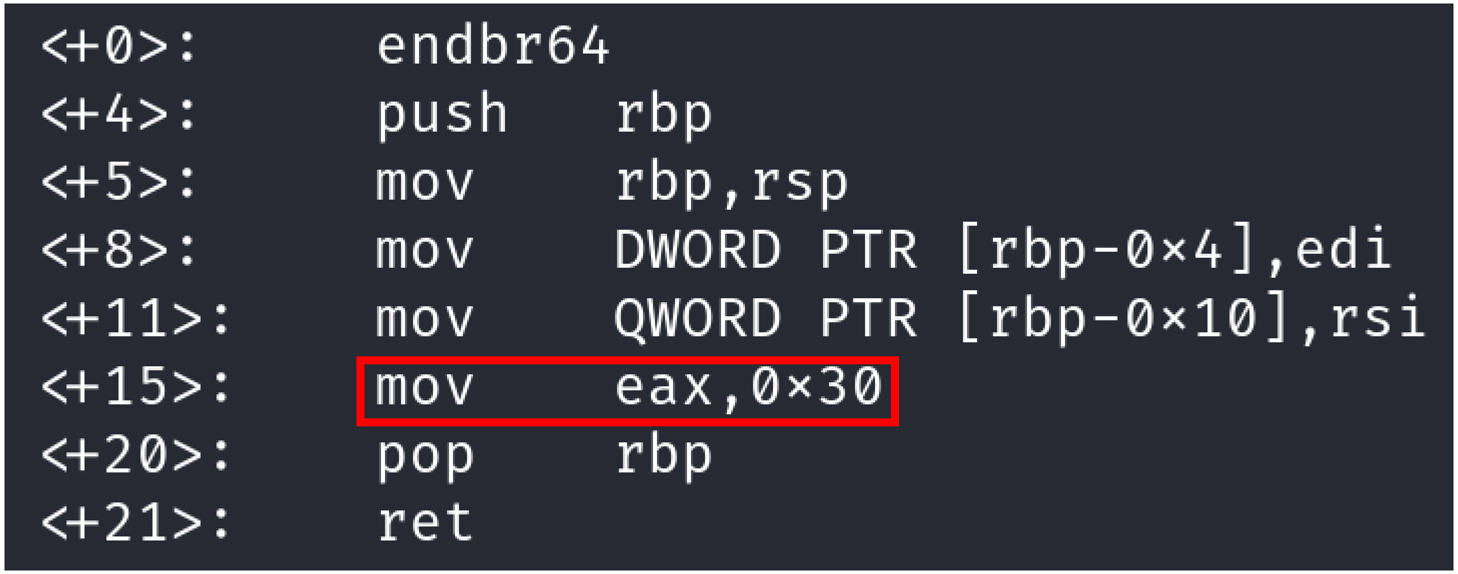
## Mov Instruction



```
<+0>:     endbr64
<+4>:     push   rbp
<+5>:     mov    rbp,rsp
<+8>:     mov    DWORD PTR [rbp-0x4],edi
<+11>:    mov    QWORD PTR [rbp-0x10],rsi
<+15>:    mov    eax,0x30
<+20>:    pop    rbp
<+21>:    ret
```

The first operand after the Mov instruction is the register to be affected, and the second operand is the data to be moved

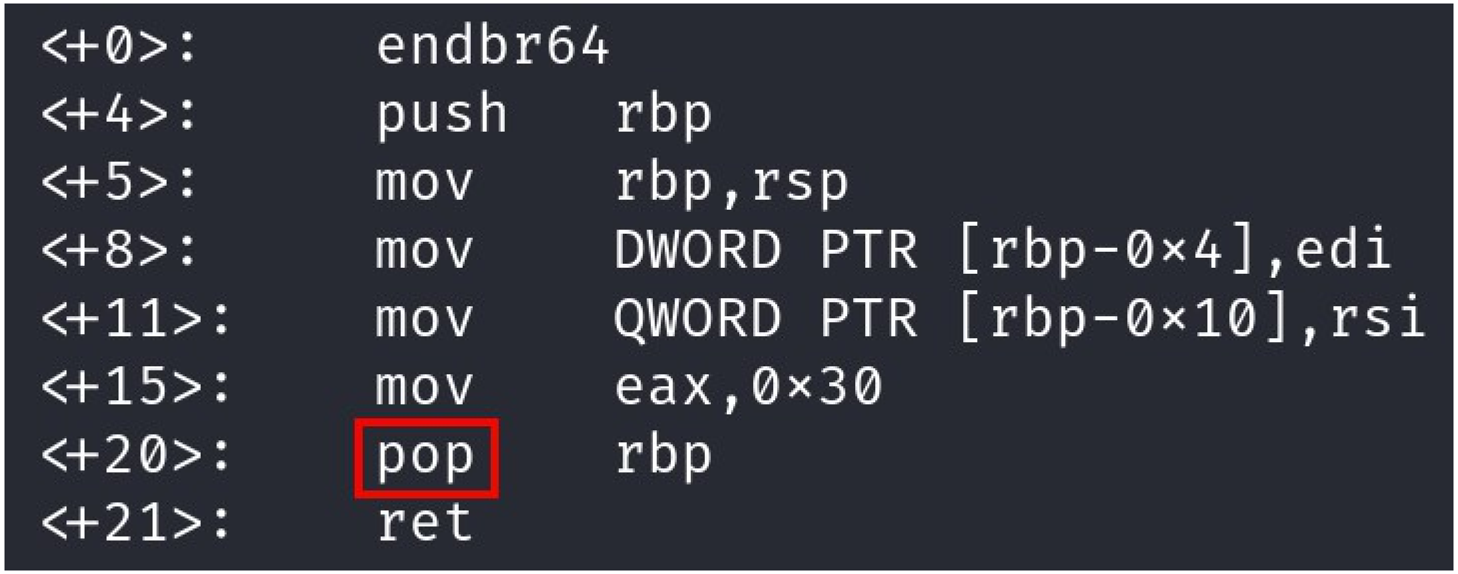
# Mov Instruction



```
<+0>:      endbr64
<+4>:      push   rbp
<+5>:      mov    rbp,rsp
<+8>:      mov    DWORD PTR [rbp-0x4],edi
<+11>:     mov    QWORD PTR [rbp-0x10],rsi
<+15>:     mov    eax,0x30
<+20>:     pop    rbp
<+21>:     ret
```

In this example, the hex 30 value is moved into the eax register

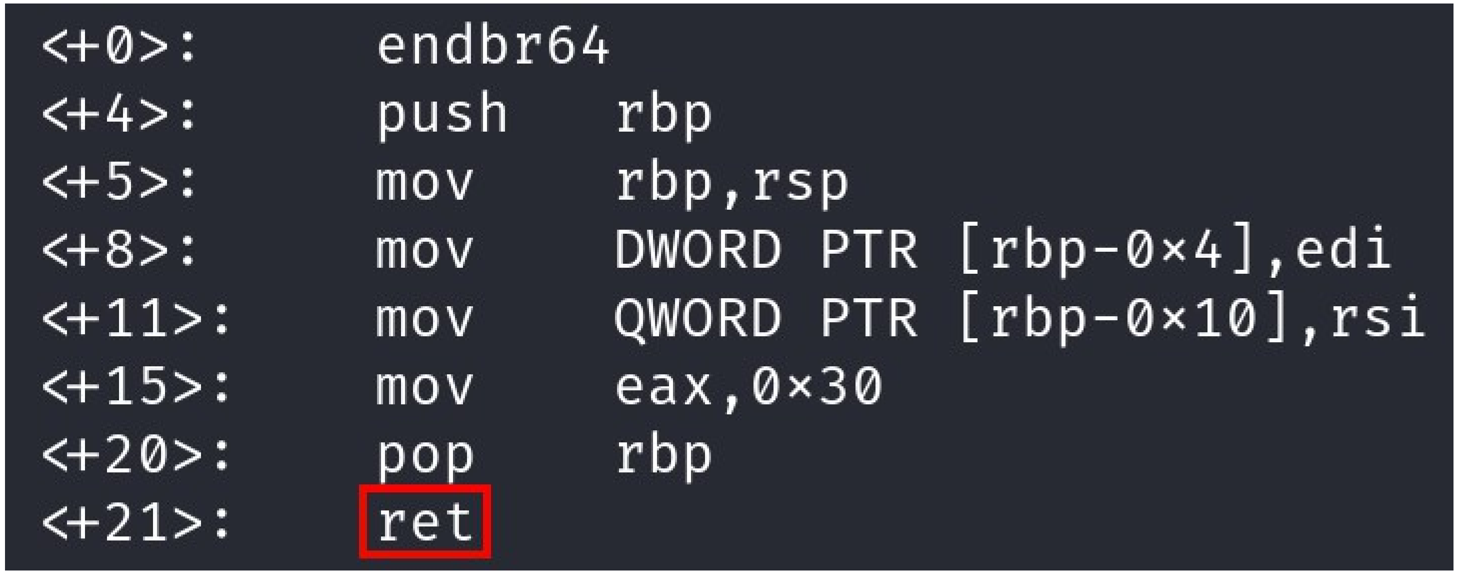
## Pop Instruction



```
<+0>:     endbr64
<+4>:     push   rbp
<+5>:     mov    rbp,rsp
<+8>:     mov    DWORD PTR [rbp-0×4],edi
<+11>:    mov    QWORD PTR [rbp-0×10],rsi
<+15>:    mov    eax,0×30
<+20>:    pop    rbp
<+21>:    ret
```

The Pop instruction is used to remove the first value off of the memory stack and move it to memory register

## Ret Instruction



```
<+0>:     endbr64
<+4>:     push   rbp
<+5>:     mov    rbp,rsp
<+8>:     mov    DWORD PTR [rbp-0x4],edi
<+11>:    mov    QWORD PTR [rbp-0x10],rsi
<+15>:    mov    eax,0x30
<+20>:    pop    rbp
<+21>:    ret
```

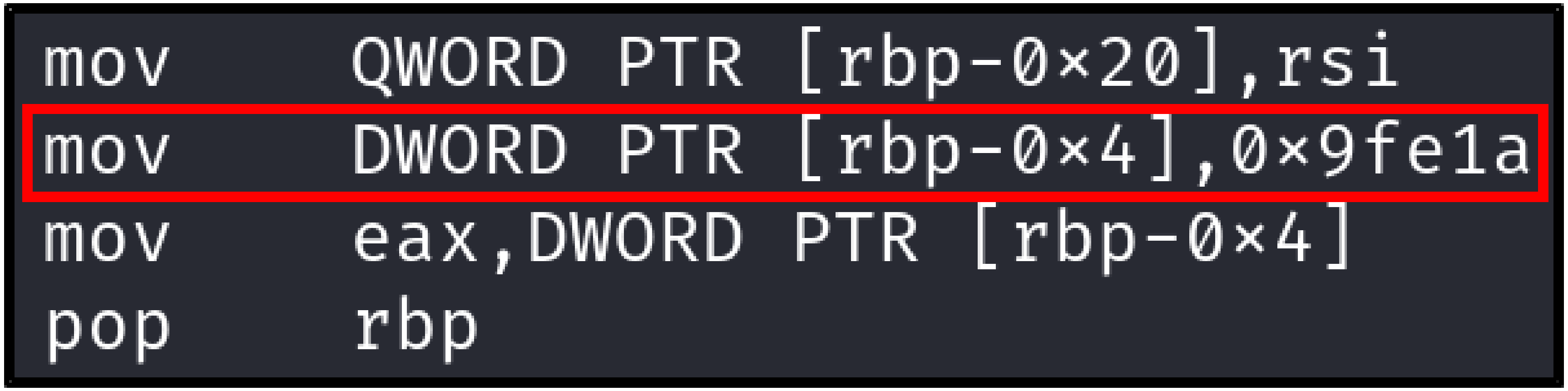
The Ret instruction is used to finish execution of a programming function and return to execution of the main function

# Pico CTF – Bit-O-Asm-2 Challenge

Let's learn more about x86 Assembly programming with a challenge from PicoCTF!

https://play.picoctf.org/practice/challenge/392

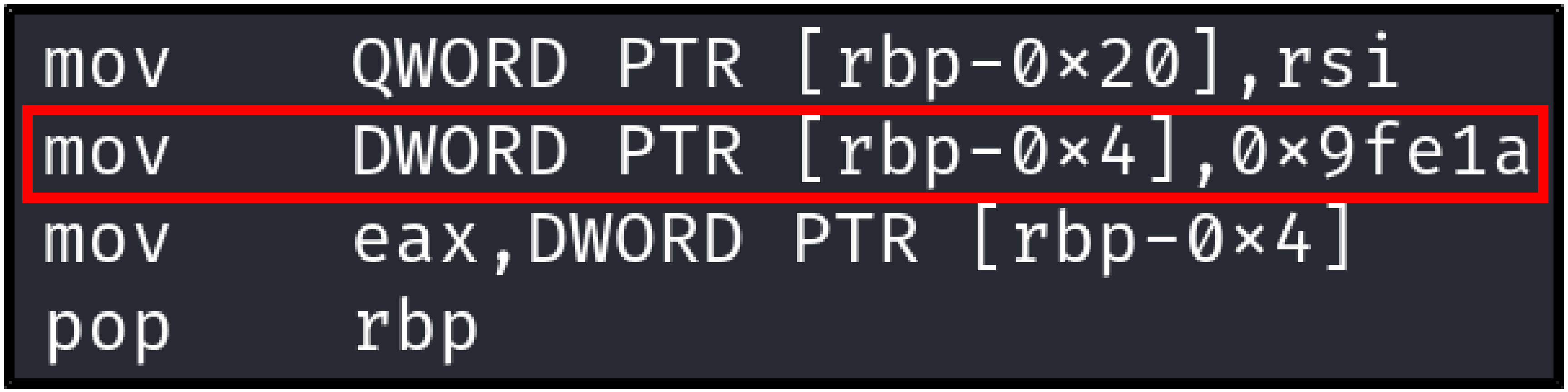
## Variable Memory Locations



```
mov     QWORD PTR [rbp-0x20],rsi
mov     DWORD PTR [rbp-0x4],0x9fe1a
mov     eax,DWORD PTR [rbp-0x4]
pop     rbp
```

For this challenge, it's important to understand how variables are saved in x86 Assembly

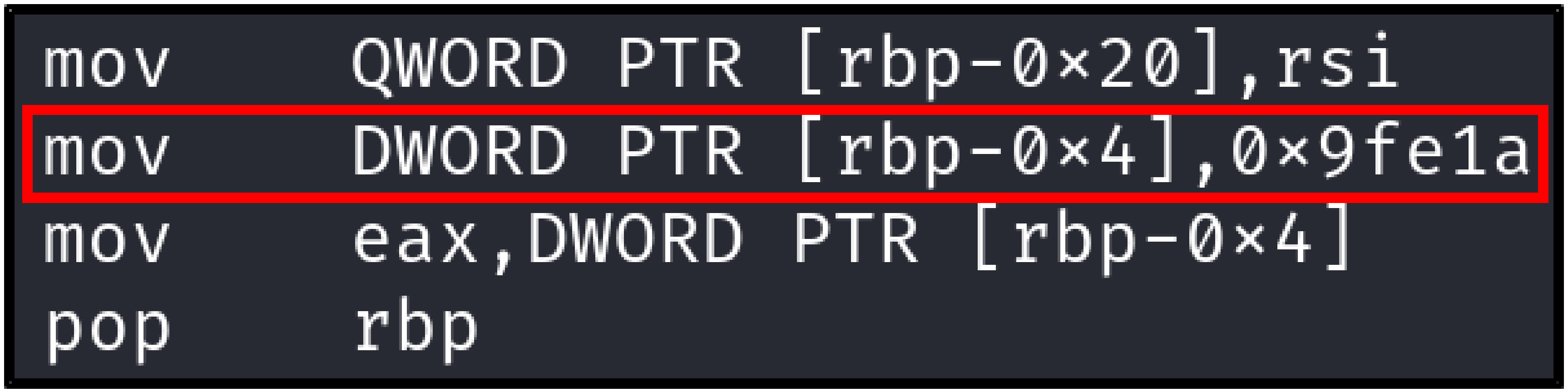
## Variable Memory Locations



```
mov     QWORD PTR [rbp-0x20],rsi
mov     DWORD PTR [rbp-0x4],0x9fe1a
mov     eax,DWORD PTR [rbp-0x4]
pop     rbp
```

When the program starts, memory locations will be allocated for variables, and we can move data through those variable locations just like any other memory registers

## Variable Memory Locations



```
mov     QWORD PTR [rbp-0x20],rsi
mov     DWORD PTR [rbp-0x4],0x9fe1a
mov     eax,DWORD PTR [rbp-0x4]
pop     rbp
```

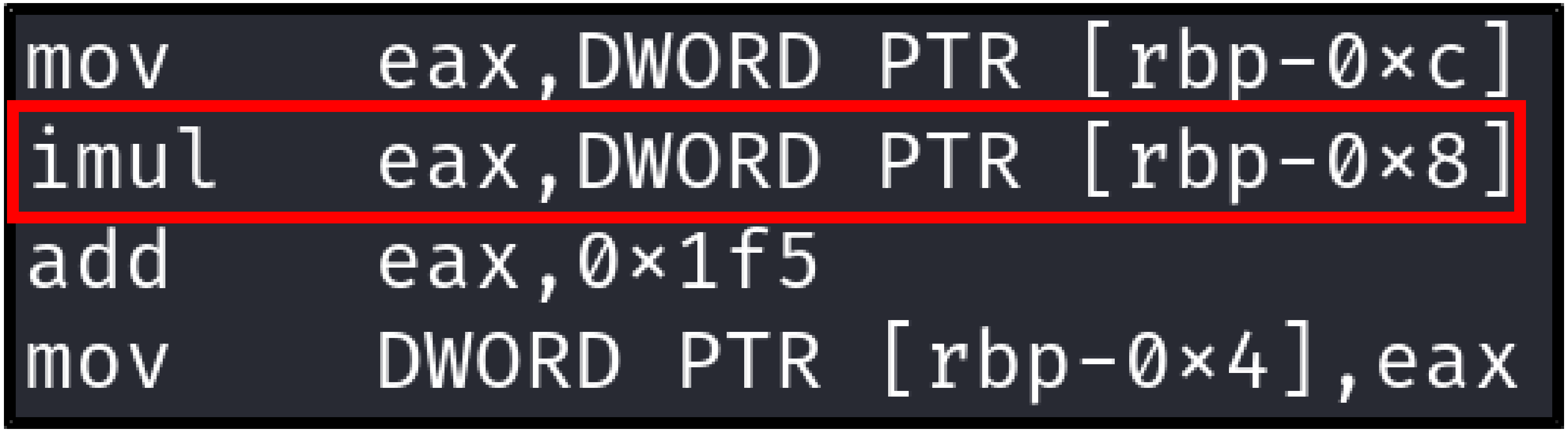
So the name of the variable memory location is **DWORD PTR [rbp-0x4]**, and hex value **9fe1a** is moved into that location

## Pico CTF – Bit-O-Asm-3 Challenge

Let's learn more about x86 Assembly programming with a challenge from PicoCTF!

https://play.picoctf.org/practice/challenge/393

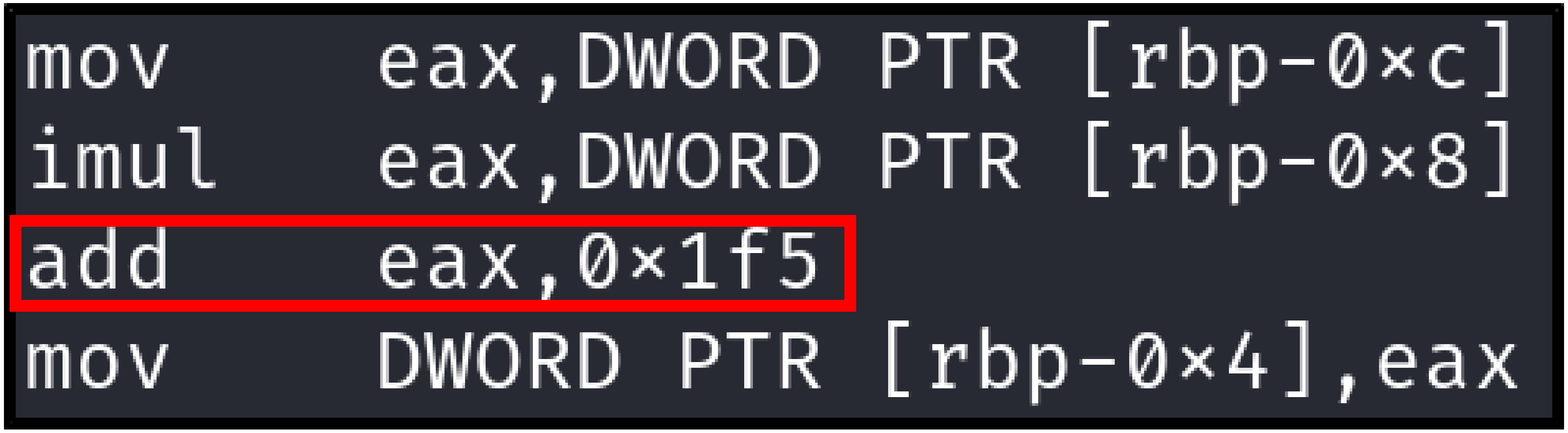
## Imul Instruction



```
mov     eax,DWORD PTR [rbp-0xc]
imul    eax,DWORD PTR [rbp-0x8]
add     eax,0x1f5
mov     DWORD PTR [rbp-0x4],eax
```

The Imul instruction multiplies whatever value is in the first operand by the value in the second operand

## Add Instruction



```
mov     eax,DWORD PTR [rbp-0x c]
imul    eax,DWORD PTR [rbp-0x8]
add     eax,0x1f5
mov     DWORD PTR [rbp-0x4],eax
```

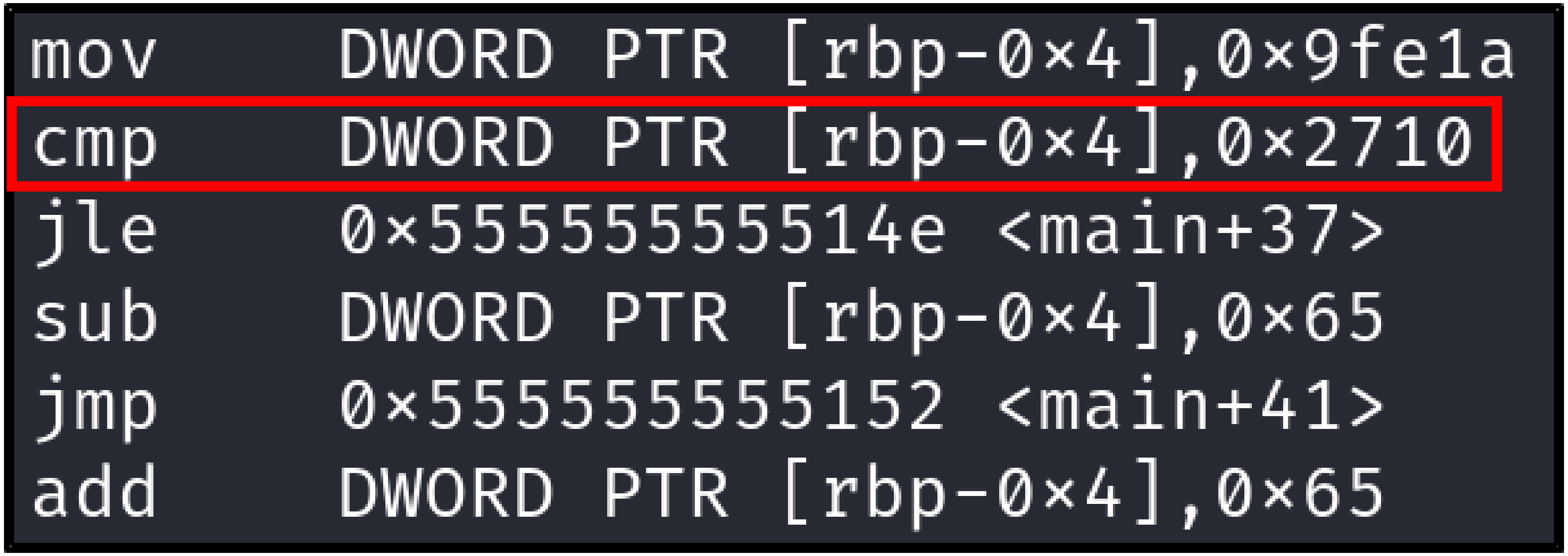
The Add instruction adds whatever value is in the first operand by the value in the second operand

# Pico CTF – Bit-O-Asm-4 Challenge

Let's learn more about x86 Assembly programming with a challenge from PicoCTF!

https://play.picoctf.org/practice/challenge/394

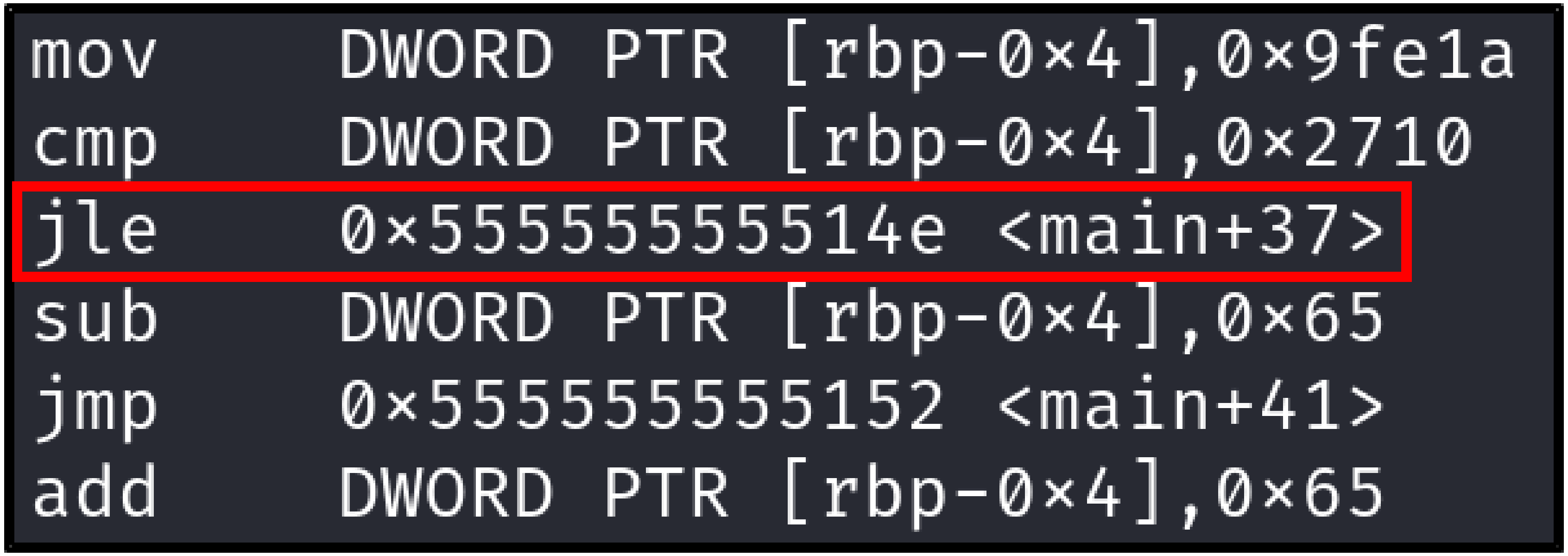
## Cmp Instruction



```
mov     DWORD PTR [rbp-0x4],0x9fe1a
cmp     DWORD PTR [rbp-0x4],0x2710
jle     0x55555555514e <main+37>
sub     DWORD PTR [rbp-0x4],0x65
jmp     0x555555555152 <main+41>
add     DWORD PTR [rbp-0x4],0x65
```

The Cmp instruction compares the first operand against the second operand,and returns either equal, less-than, or greater-than

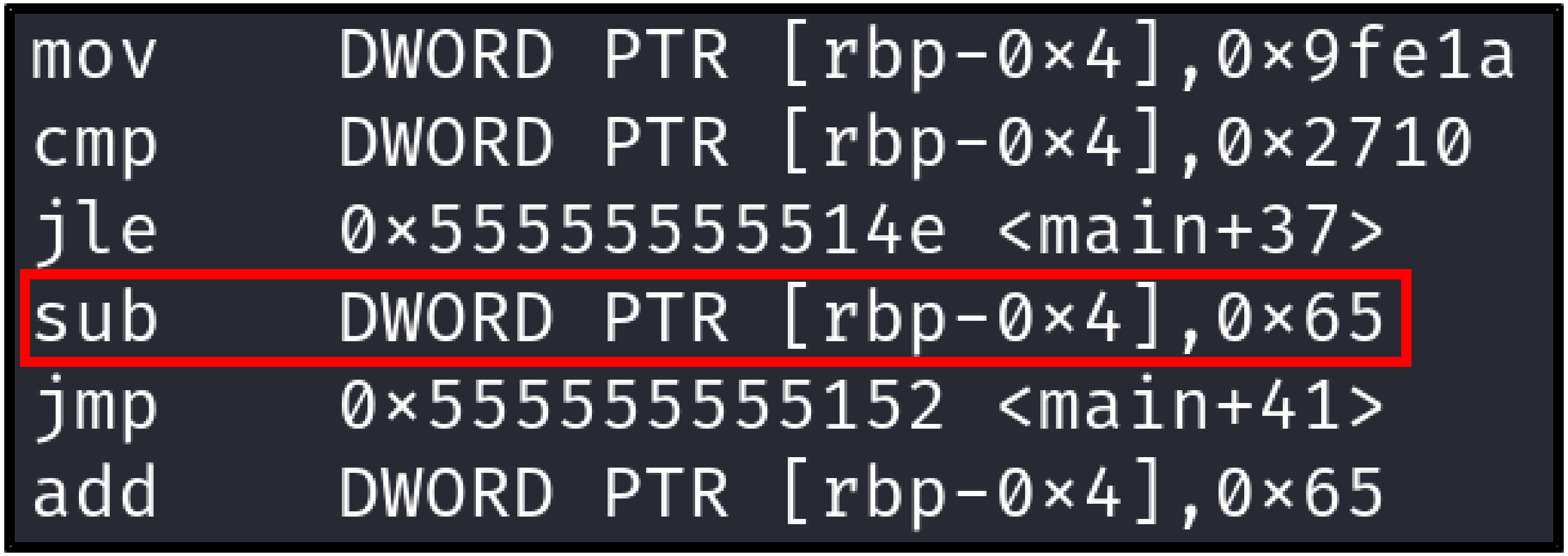
# Jle Instruction



```
mov     DWORD PTR [rbp-0x4],0x9fe1a
cmp     DWORD PTR [rbp-0x4],0x2710
jle     0x55555555514e <main+37>
sub     DWORD PTR [rbp-0x4],0x65
jmp     0x555555555152 <main+41>
add     DWORD PTR [rbp-0x4],0x65
```

The Jle instruction moves program execution to the indicated memory address if the previous Cmp instruction returned less than or equal

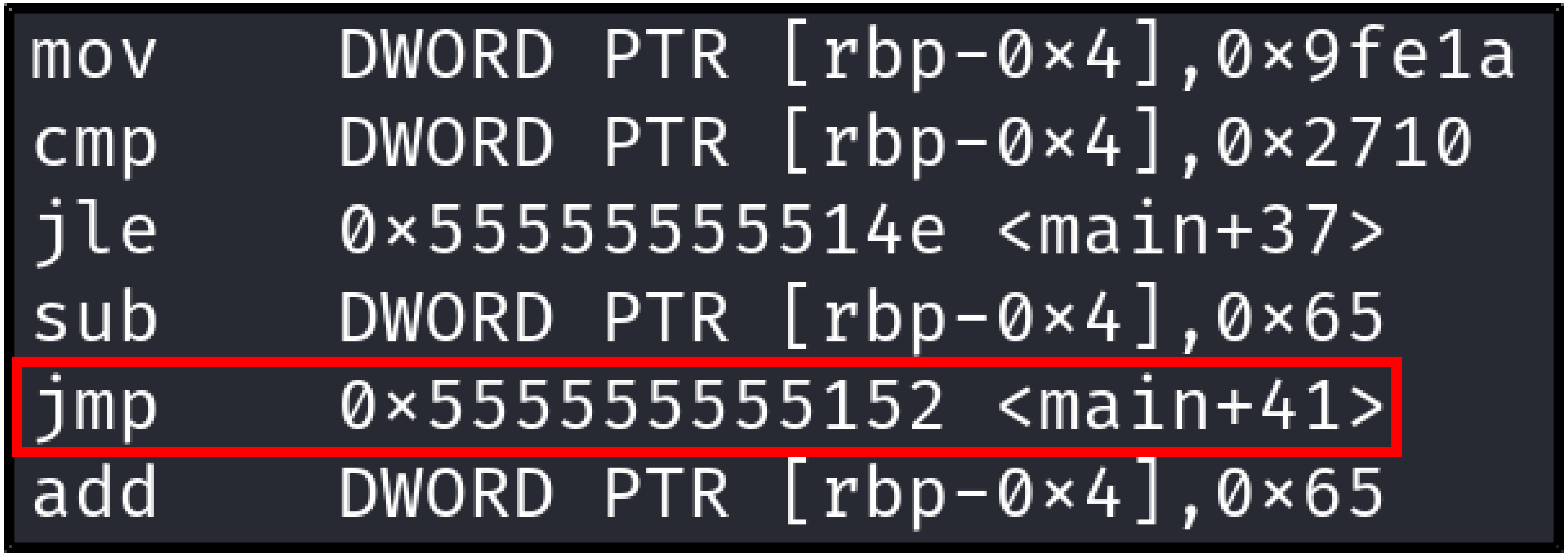
## Sub Instruction



```
mov     DWORD PTR [rbp-0x4],0x9fe1a
cmp     DWORD PTR [rbp-0x4],0x2710
jle     0x55555555514e <main+37>
sub     DWORD PTR [rbp-0x4],0x65
jmp     0x555555555152 <main+41>
add     DWORD PTR [rbp-0x4],0x65
```

The Sub instruction subtracts the number in the second operand from the first operand

## Jmp Instruction



```
mov     DWORD PTR [rbp-0x4],0x9fe1a
cmp     DWORD PTR [rbp-0x4],0x2710
jle     0x55555555514e <main+37>
sub     DWORD PTR [rbp-0x4],0x65
jmp     0x555555555152 <main+41>
add     DWORD PTR [rbp-0x4],0x65
```

The Jmp instruction moves program execution to the indicated memory location. This is also called an unconditional jump

## Summary

Let's review the concepts we learned in today's workshop:

## Until Next Time, HackerFrogs!