HACKING THE METAL

An Introduction to Assembly Language Programming

A Workshop for Def Con 29

created by

eigentourist

WORKSHOP MATERIALS

To participate in this workshop, you are going to need these things:

- A laptop computer running Windows 10 (if you have Linux or MacOS, running Windows in a VM is totally fine)
- Visual Studio 2019 Community Edition installed – this edition is free, and you can download it from here.
- The Netwide Assembler, 64-bit edition for Windows – this is also free, and you can download the installer here.
- IDA Freeware Disassembler from Hex-Ray – yet another free resource, with installer available here.
- The GitHub – All code and artifacts that we are writing / studying / running will be posted at https://github.com/eigentourist/defcon29.
- Optional – Visual Studio Code for Windows – some folks prefer VS Code as an editor, and those of you who do probably have it already, but just in case, here you go.

WORKSHOP MATERIALS

A few details:

- Why Windows? – Let's face it, it's the operating system you'll encounter most often on personal computers. But finding material that teaches assembly language programming for Windows hasn't always been so easy. In recent years, some good material has appeared, both in print and on the web. We're hoping to add to that with this workshop.

- Why NASM? – the syntax is friendlier, and you can find this assembler on many platforms, so if you write for Linux, or for ARM, for example, NASM will be there.

- Why Assembly Language? – there are fewer reasons to write code in assembly language compared to some twenty or thirty years ago, but they still exist. What will serve you *very* well is the knowledge and intuition you will gain in learning to read and write code on this level.

INTRO TO THE INTRO

In the middle of a hands-on workshop, it's not always easy to stop and tell a story.

But there are a lot of stories relevant to the experience of low-level programming.

In the remainder of this document, we will cover a few highlights and a few concepts. The more of these things you know, the more you will understand how the world of computing machinery came to be what it is today, and why it behaves the way it does.

To begin with, the idea of a computing machine being programmable is a huge innovation, and the first computing machines were neither electronic, nor were they programmable. In other words, if you want it to do something else, you either take it apart and rebuild it, or you build a whole other machine.

INTRO TO THE INTRO

You could make a convincing case that computing machines have been around for thousands of years. Check out the Antikythera mechanism, for example. It's analog, not digital. It's a special-purpose machine, and there's no way to make it do anything else other than what it does.

The idea that a computing machine could have its behavior changed was a very sticky problem all the way into the 20th century. Before anyone could build one, the theory of how it might even be possible had to be worked out. Alan Turing is one of the pioneers who did that, and his ideas are probably the most well-known (if you are a fan of Lambda Calculus, feel free to loudly object!)

Soon afterward, a specific design created by Jon Von Neumann laid the groundwork for a real-world machine that could run one program to do a certain job, and then run a different program to do something totally different.

Perhaps the most powerful outcome of this innovation is the step that came soon afterward: *programs that write other programs*.

HACKING WHAT?

What really happens when you type out some code like this?

```
#include <stdio.h>

int main()
{
    printf("Hello, World!\n");
    return 0;
}
```

The computer on your desk, the phone in your pocket, and the server at your office don't speak this language. It's a language written for us, not for the machines. We can use it simply because there are programs that translate it into a form that the machines can work with. There are programs that write other programs.

THAT LOOKED FAMILIAR

Let's assume your computer runs Windows, which is a reasonable guess. Compare the code you just saw to this:

```
bits 64
default rel

segment .data
    msg db "Hello, World!", 0xd, 0xa, 0

segment .text
global main
extern ExitProcess
extern _CRT_INIT
extern printf
```

What on earth are we doing here? Hold on, we're not done yet…

WHAT MADNESS IS THIS?

What you just saw on the last slide is a prelude, a beginning to a program that does the same thing you saw before that. Now that we've set things up…

```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
    lea     rcx, [msg]
    call    printf
    xor     rax, rax
    call    ExitProcess
```

And we're done. Combined with the setup code on the last slide, this code will also print the famous "Hello, World!" greeting to a console window. Just like the very first program you saw, though, it needs translation.

But… not quite as much.

THE LANGUAGE OF THE MACHINE

That last version needs less translation because it's a closer representation of what goes on inside the computing machinery. It's not quite the real thing, but it's close.

You might be ready to offer that the real thing is a long series of ones and zeros. That's a good, reasonable guess. But beneath those binary digits, there's a boundary – and crossing that boundary lands you in the bizarre world we live in, where things aren't digital. They're analog.

We covered a bit of that history already. And we left off at the discovery of a ground-breaking concept: programs that write other programs.

There are roughly three kinds of programs that can do that: interpreters, compilers, and assemblers.

THE LANGUAGE OF THE MACHINE

An interpreter is a program that will read a set of directions in some form that the machine has no way to handle, and it will command the machine to do certain things based on those directions.

A compiler will read that set of directions and write a new program in the numeric language of the machine. That new program will execute the same logic described in those directions, but it exists independently, and it can be copied and executed any number of times – the original directions are no longer needed, unless and until the behavior of the program needs to be changed.

An assembler is a very simple form of compiler. It reads directions written in a language that is modeled closely after the structure of the machine. That language has a vocabulary that maps to specific machine operations. Different machines will have different architectures, and each one will have an assembly language that matches their unique features.

THE LANGUAGE OF THE MACHINE… CONT'D

The assembly language "Hello, World" program you saw earlier uses a vocabulary that manipulates specific parts of the machine and causes them to do simple, fundamental things: move a piece of data from one place to another, or modify the data stored in some location, for example.

When we look at how the machine operates as it executes a program, we start to understand how these tiny operations become so powerful: there is a numerical code for each of these operations, and these codes are stored, one after another, in the machine's memory, just like any other kind of data you might think of.

One of the processor's on-board storage spaces – known as registers – contains a number that tells it what part of memory to load those numeric codes from. This register, known as the instruction pointer, changes its value after the code is loaded, and the operation it represents is complete. The code located at the new address held in the instruction pointer is loaded, its operation is executed, and the value in the instruction pointer ticks forward to the next memory location…

THE LANGUAGE OF THE MACHINE… CONT'D

...and in many cases, the code that the processor fetches and executes will *modify the contents of the instruction pointer*, causing the processor to jump to a whole other area of memory, and start reading its contents as code, executing the operation that corresponds to each number, one after the other.

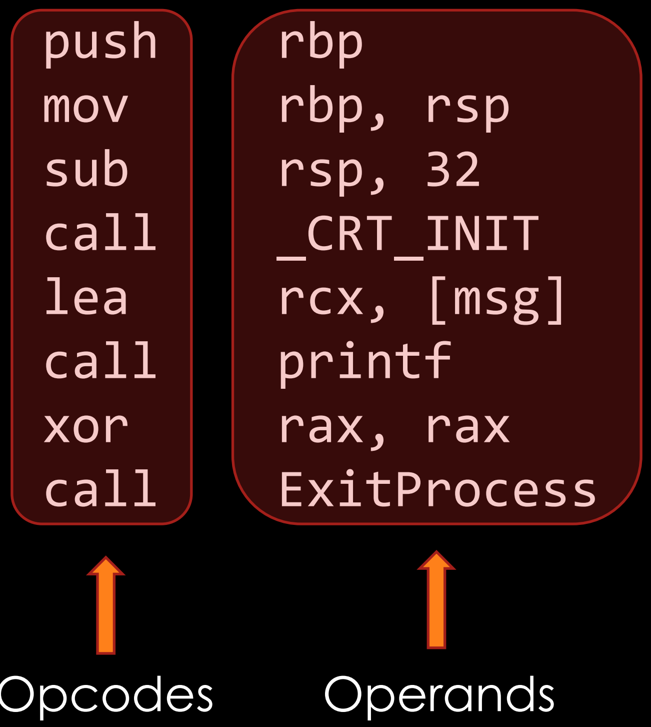
At some point, one of these machine instructions will map to an operation that changes the instruction pointer's value again. When that happens, the processor may return to executing code from the original area of memory it was reading from.

The instructions of a machine's processor are known as its instruction set. In the "Hello, World" program earlier, we can see that they have a certain structure. First, we see an operation code, or opcode, which tells the processor what to do. Then comes an operand, which is the target of the operation.

Lots of instructions have more than one operand. For example, the MOV instruction copies the data in one place (the source) to another place (the destination.)

THE LANGUAGE OF THE MACHINE… CONT'D

Let's take a closer look..



```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
    lea     rcx, [msg]
    call    printf
    xor     rax, rax
    call    ExitProcess
```

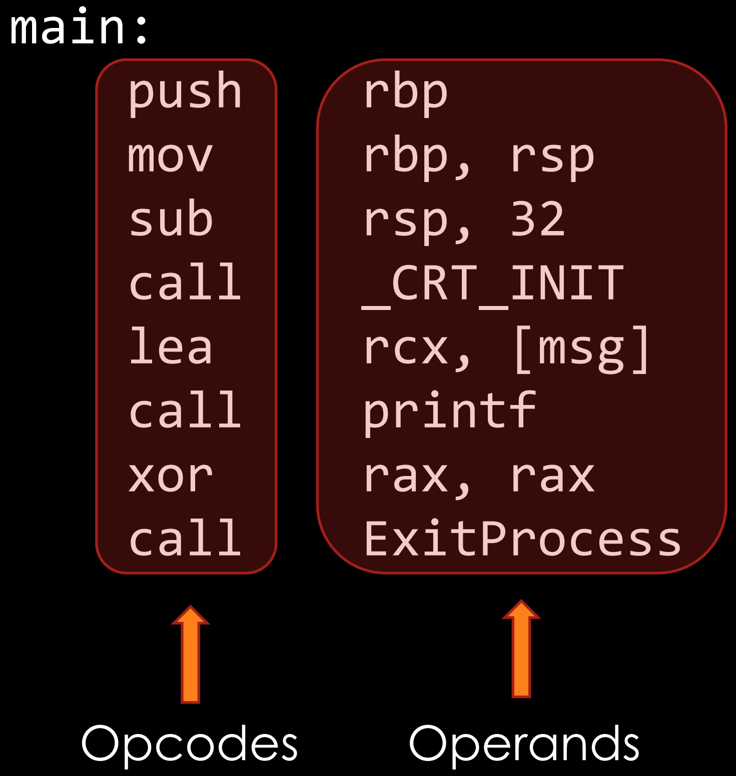
Instructions will always have opcodes first, and then operands. The order of the operands is important -- for example, the MOV instruction expects to have the destination listed first, followed by the source.

So, the second instruction (mov rbp, rsp) is copying the contents of the register rsp into the register rbp.

Modern microprocessors have many on-board registers, and there are rules that programmers need to follow regarding many of them. Some of them are for general usage, and others (like the instruction pointer that we've talked about) are for special purposes.

THE LANGUAGE OF THE MACHINE… CONT'D

Let's take a closer look. Opcodes come first, followed by operands.



```
main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, 32
    call    _CRT_INIT
    lea     rcx, [msg]
    call    printf
    xor     rax, rax
    call    ExitProcess
```

Opcodes

Operands

The rules that govern how the different registers of the processor are used come from two sources: the manufacturer of the processor, and the rules of the operating system running on the machine.

In this workshop, we will learn the rules about 64-bit assembly language in Windows. These rules are a bit different from the 32-bit Windows world, and there aren't as many sources that teach them.

Once we have a few ground rules under our belt, we will move on to writing and running some interesting code, building each new lesson on what we have already learned.

TUNE INTO THE SIGNAL

Understanding how to write (and maybe more importantly, how to read and understand) code at the assembly language level will deepen your understanding of how code behaves at its fundamental level, no matter what language it was originally written in.

Whether you're coming to the conference in person, or viewing the events remotely, it's good to have you back this year. The theme chosen for this year's conference (Can't Stop The Signal) resonates with me by way of symbolizing the thread of knowledge that we all share and benefit from. This workshop is my contribution to The Signal – an introduction to a world of powerful knowledge that is harder to find and learn about in contemporary times, but never fails to reward those who study it.

See you there!

-eigentourist