Reversing

Time to get with the program

This guide is a brief introduction to C, Assembly Language, and Python that will be helpful for solving *Reversing* challenges.

Writing a C Program

C is one of the most important modern programming languages and finds use in almost all system programming applications. First thing first, here is a great tutorial on C and goes into great detail about programming in C. The C language is a must know for any modern programmer, so let's cover some important Computer Science concepts using C.

Let's start with the concept of a function. A function is essentially a chunk of code that takes an input and returns an output. It has a name, an arbitrary number of optional parameters or inputs, instructions, and an optional return value or output.

```c
int doubleNum(int a) {
   // Take a number and multiply it by 2
   int x = a * 2;
   return x;
}
```

This is a very basic program but the concepts in it are the same concepts used to write huge, complicated applications.

Let's dissect each part of the program to gain a deeper understanding. The block of code in general is called a *function* and is a set of instructions that usually complete a logical function like multiplying a number by 2. It usually has a name, here 'doubleNum'. Now, another important aspect of most programs is *datatype*, which quite literally is the kind of data that we are dealing with. It is usually associated with a *variable* which is just a structure that stores data. Here, the "int a" is a parameter for the function "doubleNum" and means that this function takes one number as input. The first "int" means that this function returns an integer as output.

x and a are variables that both store integers and are connected since $x = 2*a$.

The return statement tells the computer that this function should give *x* as an output. The last important part is the comments, denoted by the '//' and is essentially telling the computer to ignore the rest of the words on that line. You can write whatever you want there and is essentially just meant to explain the logic in your code.

Now that you know the essentials of writing a C function (or a function in any language really), we can talk more about how a computer even understanding what you are trying to get it to do. Whenever, I say that a particular keyword tells the computer to do something, I really mean that the programmer is telling the compiler to understand the rest of the stuff in a particular context. The compiler essentially converts a High level language into a low level language. It eventually gets converted into assembly language and finally into 1's and 0's, which finally the computer can understand and execute.

Compiling and Executing a C Program

There are multiple steps to converting a program from C to something a computer can understand. These steps include Compiling, Assembling and Linking. The compiler converts the code from C to assembly language (which we'll discuss later), and the assembler converts the code from assembly language to executable object. The main compiler in C is called GCC (GNU C Compiler) and has the following syntax:

```
$ gcc -Wall filename.c -o filename
```

The *filename.c* is a placeholder for the C file you want to compile. The *-Wall* and *-o* are flags that tell the compiler options that can change the way the compiler works. *-o* tells the compiler that the next token is the name of the executable object, *filename.o* and *-Wall* enables warning messages. Now, to run the file, you need to type

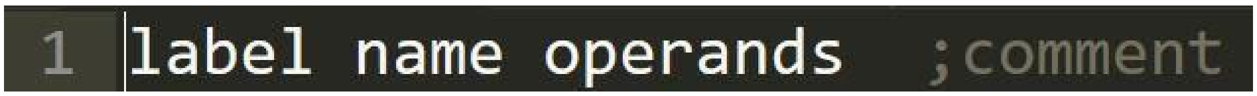
```
$ ./filename
```

which runs the file and takes any inputs and returns any outputs.

Assembly

We want to learn how to program in Assembly to cause changes without the added layer of abstraction provided by higher level languages and thus write code that runs faster and has more capabilities. Here is a great tutorial on everything you could want to know about assembly programming but as always, I will go through some of the more important concepts and provide a basic overview to help you get started.

Assembly programming is very different from normal programming in that we have to worry about things that high level language programmers don't have to worry about, like where something is stored and how it is stored in memory. However, we still have a program made of statements that the programmer writes that follows the syntax as such -



```
label name operands   ;comment
```

Now, let's dissect this -

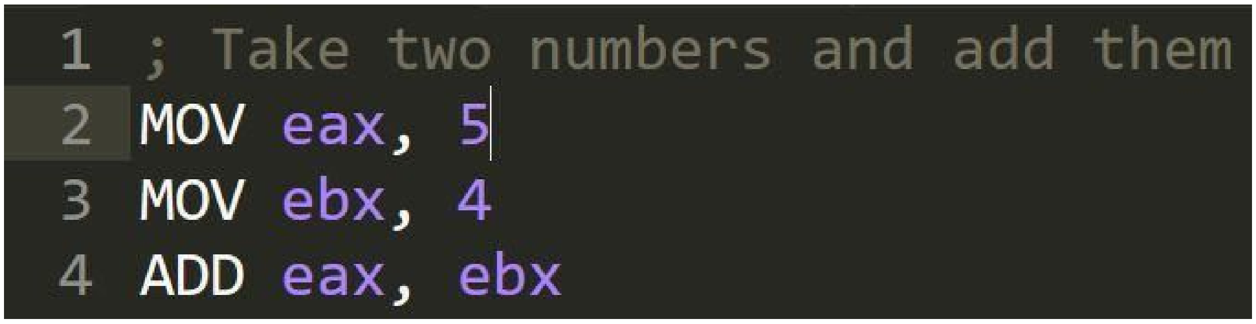
1. *label* - This is a optional personal name we give to a statement in our code
2. *name* - This is the name of the operation we want to perform (ADD, MOV etc.)
3. *operands* - This is the optional list of parameters to the function
4. *comment* - This is denoted by a ; and functions as a comment in normal languages

Registers

Before we can move to talking about different instructions in assembly language, we need to talk about a very important concept - *registers*. In general, we want to be able to store variables and we cannot store them directly into main memory as that would make access to the data very slow. Therefore, most computers have 32 special locations in their CPU where we can store 32 bits of information. There special locations are called registers and help us store and recall variables quickly and easily.

Some Common Instructions

Here is a list of common instructions in the X86-64 instruction set. However, let's discuss some important instructions to get an idea of what is going on. Here is a sample program that performs a simple exercise of adding two numbers.



```
; Take two numbers and add them
MOV eax, 5
MOV ebx, 4
ADD eax, ebx
```

We start this off with a comment explaining what this code snippet is doing. *MOV* is the instruction that takes a register and a register or data point and puts it into the other register. It has the syntax as follows

MOV Dest, Source

Here, the destination is eax which is a register and 5 which is a data value. Therefore, we put the integer values 5 and 4 into registers *rax* and *rbx* respectively. Then, we perform the *ADD* instruction that adds the two data points it is given.

ADD Dest, Source

This instruction takes the data in *Source* and adds it to the data in *Dest* and stores it back in dest. Therefore, it adds the value 4 into the *rax* register and store the result, 9 back into *rax* register.

This is only a very simple example of assembly programming. However, you can understand what some code is trying to do by following the similar concept. Pull up the descriptions of the instructions online and start converting the snippet, line by line, and piece together what the code is trying to do. Note, we did not use labels.

However, using just labels and a JMP statement, we can implement loops, functions and recursion!

Big Endian vs. Little Endian

These are ways to store numbers or data in memory addresses. Let's use a 16-bit word as example[1], $(0xFEED)_{16}$ in this case. Let's also assume we are storing this word starting at address 0x4000. We store these words in terms of bytes and not bits, so we need some conversion. Recall that 16 bits is 2 bytes (since 1 byte is 8 bits). The word is stored in pairs to make up the required 1 byte per memory location so our two parts will be 'FE' and 'ED'.

1. *Big Endian*: This refers to big end first, which means that we store the most significant byte at the smallest memory location and the rest follow normally. Therefore, to store our word in this case, memory location 0x4000 will have the byte *FE* and memory location 0x4001 will store the byte *ED*.
2. *Little Endian*: This refers to little end first and opposed to Big Endian, we store the least significant bit at the smallest memory address and the rest follow normally with the most significant bit at the last memory address. So, 0x4000 will have the byte *ED* and 0x4001 will have *FE*.

Big Endian is commonly used in Networking application, while Little Endian is most commonly used in processors

Python

Python is an amazing High Level language that is both easy to learn and allows the programmer to do a lot of different tasks. That is probably why it is used so frequently! Here is a link to Carnegie Mellon's introductory CS course using Python and you can pretty much find whatever you could want to know about the basics of Python and CS on this website.

[1] In this case, 'word' means any number in Base-16 or Hexadecimal. So, a 16-bit word is a hexadecimal number with 16 bits or 4 values.