

Reverse Engineering For Everyone!

— by @mytechnotalent

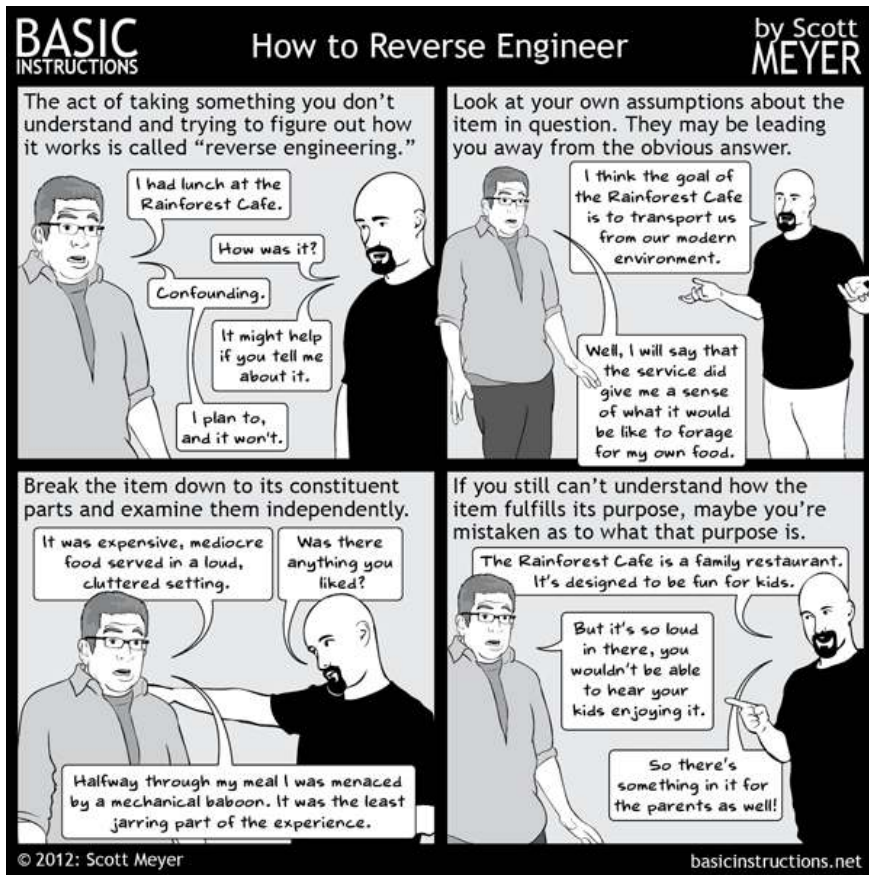


Wait, what's reverse engineering?

Wikipedia defines it as:

Reverse engineering, also called backwards engineering or back engineering, is the process by which an artificial object is deconstructed to reveal its designs, architecture, code, or to extract knowledge from the object. It is similar to scientific research, the only difference being that scientific research is conducted into a natural phenomenon.

Whew, that was quite a mouthful, wasn't it? Well, it is one of the main reasons why this tutorial set exists. To make reverse engineering *as simple as possible*.




This comprehensive set of reverse engineering tutorials covers x86, x64 as well as 32-bit ARM and 64-bit architectures. If you're a newbie looking to learn reversing, or just someone looking to revise on some concepts, you're at the right place. As a beginner, these tutorials will carry you from nothing upto the mid-basics of reverse engineering, a skill that everyone within the realm of cyber-

Part 1: Goals

security should possess. If you're here just to refresh some concepts, you can conveniently use the side bar to take a look at the sections that has been covered so far.

You can get the entire tutorial set in PDF or MOBI format. All these ebook versions will get updated automatically as new tutorials will be added.

Download here: [[PDF](#) | [MOBI](#)]

Gitbook crafted with  by [@0xInfection](#)

The x86 Architecture

Let's dive in rightaway!

Part 1: Goals

Essential to the discussion of basic reverse engineering is the concept of modern malware analysis. Malware analysis is the understanding and examination of information necessary to respond to a network intrusion.

This short tutorial will begin with the basic concepts of malware reverse engineering and graduate to an entry-level basic examination of Assembly Language.

The keys to the kingdom so to speak are rooted in the break-down of the respective suspected malware binary and how to find it on your network and ultimately to contain it.

Upon full identification of the files required for deeper analysis, it is critical to develop signatures to detect malware infections throughout your network whether it be a home based LAN or complex corporate WAN to which malware analysis is necessary to develop host-based and network signatures.

To begin with the concept of a host-based signature, we need to understand that these are utilized to find malicious code in a target machine. Host-based signatures are also referred to as indicators which can identify files created or edited by the infected code which can make hidden changes to a computers registry. This is quite in contrast with antivirus signatures because these concentrate on what the malware actually does rather than the make-up of the malware which makes them more effective in finding malware that can migrate or has been removed from the media.

In contrast, network signatures are used to find malicious code by examining network traffic. It is important to note such tools as WireShark and the like are often effective in such analysis.

Upon identification of these aforementioned signatures, the next step is to identify what the malware is actually doing.

In our next lesson we will discuss techniques of malware analysis.

Part 2: Techniques

There are two basic techniques that you can employ when analyzing malware. The first being static analysis and the other being dynamic analysis.

Static analysis uses software tools to examine the executable without running the actual decompiled instructions in Assembly. We will not focus on this type of analysis here as we are going to focus on actual disassembled binaries instead however in future courses we will.

Dynamic analysis uses disassemblers and debuggers to analyze malware binaries while actually running them. The most popular tool in the market today is called IDA which is a multi-platform, multi-processor disassembler and debugger. There are other disassembler/debugger tools as well on the market today such as Hopper Disassembler, OllyDbg and many more.

A disassembler will convert an executable binary written in Assembly, C, C++, etc into Assembly Language instructions that you can debug and manipulate.

Reverse engineering is much more than just malware analysis. At the end of our series, our capstone tutorial will utilize IDA as we will create a real-world scenario where you will be tasked by the CEO of ABC Biochemicals to secretly try to ethically hack his companies software that controls a bullet-proof door in a very sensitive Bio-Chemical lab in order to test how well the software works against real threats. The project will be very basic however it will ultimately showcase the power of Assembly Language and how one can use it to reverse engineer and ultimately provide solutions on how to better design the code to make it safer.

In our next lesson we will discuss various types of malware.

Part 3: Types Of Malware

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Malware falls into several categories of which I will touch briefly upon below.

A backdoor is malicious code that embeds itself into a computer to allow a remote attacker access with very little or sometimes no authority to execute various commands on any respective local computer.

A botnet allows an attacker access to a system however receive instructions not from one remote attacker but from a command-and-control server to which can control an unlimited amount of computers at the same time.

A downloader is nothing more than malicious code that has only one purpose which is to install other malicious software. Downloaders are frequently installed when a hacker gains access to a system initially. The downloader then installs additional software to control the system.

We find information access malware which gathers information from a computer and sends it directly to a host such as a keylogger or password grabber and usually used to obtain access to various online accounts that can be very sensitive.

There are malicious programs that launch other malicious programs which use non-standard options to get increased access or a greater cloaking/hiding technique when penetrating a system.

One of the most dangerous forms of malware is the rootkit which hides the existence of itself and additional malware from the user which makes it extremely hard to locate. A rootkit can manipulate processes such as hiding their IP in an IP scan so that a user may never know that they have a direct socket to a botnet or other remote computer.

Scareware is used to trick a user into purchasing additional software to falsely protect a user when there is no real threat whatsoever that exists. Once a user pays to have the tricked software removed from the computer it then can stay resident and later emerge in an altered form.

There are also various kinds of malware that send spam from a target machine which generates income for the attacker by allowing them to sell various services to other users.

The final form of malware is that of a traditional worm or virus which copies itself and goes after other computers.

This is the end the road for now regarding our discussion of malware because we first need to go back to the beginning and understand how a computer works at it's base level.

Part 1: Goals

In our next lesson we will begin our long journey into x86 Assembly Language. In order to truly understand the very basics of reverse engineering and malware we need to over the next several months take a deep dive into the core and build our way up.

Part 4: x86 Assembly Intro

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Ladies and Gentlemen, boys and girls, children of all ages! We are about to embark on a journey that will change your life forever!

There is vast material to cover to get a good understanding of Assembly Language and why it is important to understand the basics.

The first question we must answer is what is x86 Assembly Language to which the answer is a family of backward-compatible Assembly Languages which provide compatibility back to the Intel 8000 series of microprocessors. x86 Assembly Languages are used to produce object code for the aforementioned series of processors. It uses mnemonics to represent the instructions that the CPU can execute.

Assembly Language for the x86 microprocessor works in conjunction with various operating systems. We will focus on Linux Assembly Language utilizing the Intel syntax in addition to learning how to program in C to which we will disassemble the source code and analyze the respective Assembly.

x86 Assembly Language has two choices of syntax. The AT&T syntax was dominant in the Unix world since the OS was developed at AT&T Bell Labs. In contrast, the Intel syntax was originally used for the documentation of the x86 platform and was dominant in the MS-DOS and Windows environments.

For our purposes, when we are ultimately disassembling or debugging software, whether it be in a Linux or Windows environment, we will see the Intel syntax in large measure. This is essential whether we are examining a Windows binary in PE format or a Linux binary in ELF format. More on that later in this tutorial.

The main differences between the two is in the AT&T syntax, the source comes before the destination and in the Intel syntax, the destination comes before the source. We will discuss this in more detail later in the tutorial.

Before you run for the door and regret embarking on this journey, remember, some basic context helps to which we will develop throughout our quest. Many of these topics may be confusing at this point which is perfectly normal as we will develop them in time.

We will focus on Linux Assembly because Linux runs on a variety of hardware and is capable of running devices such as a cell phone, personal computer or a complex commercial server.

Linux is also open source and there are many versions. We will focus on Ubuntu in our demonstrations which can be freely obtained. In contrast, the Windows operating system is owned and controlled by one company, Microsoft, to which all updates, security patches and service patches come directly from them where Linux has millions of professionals providing the same absolutely free!

Part 1: Goals

We will also focus on a 32-bit architecture as ultimately most malware will be written for such in order to infect as many systems as possible. 32-bit applications/malware will work on 64-bit systems so we want to understand the basics of the 32-bit world.

In our next lesson we discuss the binary number system. Grab your cup of coffee you are going to need it!

Part 5: Binary Number System

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Binary numbers are what define the core of a computer. A bit within a computer is either on or off. A bit has either electricity turned on to it or it is absent of such. We will dive into this deeper in future tutorials.

Puzzled and confused, where do we go from here?

Have no fear! The binary number system is here! It is important to understand that in binary, each column has a value two times the column to its right and there are only two digits in the base which happen to be 0 and 1.

In decimal, base 10, say we have the number 15 which means $(1 \times 10) + (5 \times 1) = 15$ therefore the 5 is the number times 1 and the 1 is that number times 10.

Binary works in a similar fashion however we are now referring to base 2. That same number in binary is 1111. To illustrate:

1	1	1	1
8s	4s	2s	1s

$$(8 \times 1) + (4 \times 1) + (2 \times 1) + (1 \times 1) \\ 8 + 4 + 2 + 1 = 15$$

Binary numbers are important because using them instead of the decimal system simplifies the design of computers and related technologies. The simplest definition of the binary number system is a system of numbering that uses only two digits, as we mentioned above, to represent numbers necessary for a computer architecture rather than using the digits 1 through 9 plus 0 to represent such.

In our next lesson we discuss the hexadecimal number system. It only gets more exciting from here!

Part 6: Hexadecimal Number System

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Now that we are binary masters, it's time to tackle the numbering system of numbering systems!

We learned in binary that each number represents a bit. If we combine 8 bits, we get a byte. A byte can be further subdivided into its top 4 bits and its low 4 bits. A combination of 4 bits is a nibble. Since 4 bits gives you the possible range from 0 - 15 a base 16 number system is easier to work with. Keep in mind when we say base 16 we start with 0 and therefore 0 - 15 is 16 different numbers.

This exciting number system is called hexadecimal. The reason why we use this number system is that in x86 Assembly it is much easier to express binary number representations in hexadecimal than it is in any other numbering system.

Hexadecimal is similar to every other number system except in hexadecimal, each column has a value of 16 times the value of the column to its right. The fun part about hexadecimal is that not only do we have 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 we have A, B, C, D, E and F and therefore 16 different symbols.

Lets look at a simple table to see how hexadecimal compares to decimal.

Decimal	Hexadecimal
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	A
11	B
12	C
13	D
14	E
15	F

Ok I see the smoke coming out of your ears but its ok! In decimal, everything is dealt with in the power of 10. Let's take the number 42 and examine it in decimal:

$$2 \times 10^0 = 2$$

$$4 \times 10^1 = 40$$

Remember 10 to the 0 power is 1 and 10 to the 1st power is 10, therefore, $2 + 40 = 42$.

Part 1: Goals

Grab your coffee, here comes the fun stuff!

If we understand that decimal is a base 10 number system, we can create a simple formula where b represents the base. In this case, $b = 10$.

$$(2 * b^0) + (4 * b^1)$$

$$(2 * 10^0) + (4 * 10^1) = 42$$

In binary, 42 decimal is 0010 1010 binary as follows:

$$0 * 2^0 = 0$$

$$1 * 2^1 = 2$$

$$0 * 2^2 = 0$$

$$1 * 2^3 = 8$$

$$0 * 2^4 = 0$$

$$1 * 2^5 = 32$$

$$0 * 2^6 = 0$$

$$0 * 2^7 = 0$$

$$0 + 2 + 0 + 8 + 0 + 32 + 0 + 0 = 42 \text{ decimal}$$

In hexadecimal, everything is dealt with in the power of 16. Therefore 42 in decimal is 2A in hexadecimal:

$$10 * 16^0 = 10$$

$$2 * 16^1 = 32$$

$$10 + 32 = 42 \text{ decimal} \Rightarrow 2A \text{ hexadecimal}$$

This is the same as saying:

$$10 * 1 = 10$$

$$2 * 16 = 32$$

$$10 + 32 = 42 \text{ decimal} \Rightarrow 2A \text{ hexadecimal}$$

Keep in mind 10 decimal is equal to A hexadecimal and 2 decimal is equal to 2 hexadecimal. In our formula above when we deal with A, B, C, D, E or F we need to convert them to their decimal equivalent.

Lets take another example of F5 hexadecimal. This would be as follows:

$$5 * 16^0 = 5$$

$$15 * 16^1 = 240$$

$$5 + 240 = 245 \text{ decimal} \Rightarrow F5 \text{ hexadecimal}$$

Lets look at a binary to hexadecimal table:

Binary	Hexadecimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

It is important to understand that every hexadecimal number is 4 bits long or called a nibble. This will become critical when we are reverse engineering our C programs into Assembly.

Lets look at this another way. Lets work with some more hexadecimal numbers and convert them to decimal:

Hexadecimal	Decimal
3A	$(3 \times 16) + (10 \times 1) = 58$
F1	$(15 \times 16) + (1 \times 1) = 241$
4AB	$(4 \times 256) + (10 \times 16) + (11 \times 1) = 1,195$
F1CD	$(15 \times 4096) + (1 \times 256) + (12 \times 16) + (13 \times 1) = 61,901$

To re-emphasize F1CD as a simple conversion:

$$D \text{ --- } 13 \times 1 = 13$$

$$C \text{ --- } 12 \times 16 = 192$$

$$1 \text{ --- } 1 \times 256 = 256$$

$$F \text{ --- } 15 \times 4096 = 61,440$$

$$13 + 192 + 256 + 61,440 = 61,901$$

Addition in hexadecimal works as follows. From this point forward all numbers in hexadecimal will have a 'h' next to the number:

Add

$$\begin{array}{r} 11 \\ F0BAh \\ + E9ADh \\ \hline 1DA67h \end{array}$$

$$\begin{aligned} A + D &= 10 + 13 \\ &= 23 \\ &= 16 + 7 \\ &= 17h \end{aligned}$$

$$\begin{aligned} 1 + B + A &= 1 + 11 + 10 \\ &= 22 \\ &= 16 + 6 \\ &= 16h \end{aligned}$$

$$\begin{aligned} 1 + 0 + 9 &= 1 + 0 + 9 \\ &= 10 \\ &= Ah \end{aligned}$$

$$\begin{aligned} F + E &= 15 + 14 \\ &= 29 \\ &= 16 + 13 \\ &= 10h \end{aligned}$$

Another example is as such:

Add

$$\begin{array}{r} 5C84h \\ + 329h \\ \hline 5FADh \end{array}$$

$$\begin{aligned} 9 + 4 &= 13 \\ &= Dh \end{aligned}$$

$$\begin{aligned} 8 + 2 &= 10 \\ &= Ah \end{aligned}$$

$$\begin{aligned} C + 3 &= 12 + 3 \\ &= 15 \\ &= Fh \end{aligned}$$

A final add example is as such:

Add

$$\begin{array}{r}
 1 \\
 2 \text{ A } 1 \text{ 8 h} \\
 + 4 \text{ 3 B h} \\
 \hline
 2 \text{ E } 5 \text{ 3 h}
 \end{array}$$

$$\begin{aligned}
 8 + B &= 13 \text{ [1 represents 1 group of 16 with 3 left over.]} \\
 1 + 1 + 3 &= 5 \\
 A + 4 &= E
 \end{aligned}$$

We will now focus on subtraction:

Subtract

$$\begin{array}{r}
 7 \text{ 13} \\
 \text{A } 8 \text{ 3 h} \\
 + 4 \text{ 3 B h} \\
 \hline
 6 \text{ 4 8 h}
 \end{array}$$

$$\begin{aligned}
 3 - B &= \text{undefined} \text{ [B represents 11 in decimal.]} \\
 &\text{[We can't sub 3 from 11.]} \\
 &\text{[We borrow 1 from 8 and make it a 7.]} \\
 &\text{[3 means we have 1 complete group of 16.]} \\
 &\text{[When added to 3 extra equals 19.]} \\
 &\text{[19 - B or 19 - 11 = 8]}
 \end{aligned}$$

$$\begin{aligned}
 7 - 3 &= 4 \\
 A - 4 &= 6
 \end{aligned}$$

You are probably asking yourself why is this guy spending so much time going over so many different ways of learning this! The answer is that each of us learn a little different from the next. I wanted to show several representations of hexadecimal compared to decimal and binary to help put together the whole picture.

It is fundamental that you understand what is going on here in order to proceed any further. If you have any questions, please comment below and I will be more than happy to help!

In our next lesson we discuss switches, transistors and memory.

Part 7: Transistors And Memory

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

In our last lesson, we took a very deep dive into the hexadecimal number system. I am going to keep this weeks lesson short so that you can re-read last weeks lesson. I can not emphasize how important it is to understand hexadecimal number conversions in addition to the ability to manually add and subtract them.

In the real world, we have calculators, in the real world we use the Windows operating system, in the real world professional reverse engineers use GUI debuggers like IDA Pro and others.

The question is, why am I not jumping right into the core of what real reverse engineers do? The answer is simple, one must have a deep respect and understanding of the machine in order to become great. We will never change the world without fully understanding it first. Patience and perseverance win the day.

I focus on on Linux and console-based programming because most professional servers utilize Linux and therefore is the greatest threat of malware. Understanding Linux Assembly allows you to very easily grasp the library-choking portable executable format of Windows Assembly in a much deeper way.

As I step off the soap box, lets get back to the basics of computers so here we go!

When we ask ourselves what is a computer one must go down to as about as basic as one can get.

Electronic computers are simply made out of transistor switches. Transistors are microscopic crystals of silicon that use electrical properties of silicon to act as switches. Modern computers have what are referred to as field-effect transistors.

Let's use an example of 3 pins. When an electrical voltage is applied to pin 1, current then flows between pins 2 and 3. When the voltage is removed from the first pin, current stops flowing between pins 2 and 3.

When we zoom out a bit we see that there are also diodes and capacitors when taken together with the transistor switches we now have a memory cell. A memory cell keeps a minimum current flow to which when you put a small voltage on its input pin and a similar voltage on its select pin, a voltage will appear and remain on its output pin. The output voltage remains in its set state until the voltage is removed from the input pin in conjunction with the select pin.

Why is this important you ask. Very simply, the presence of voltage indicates a binary 1 and the absence of voltage indicates a binary 0 therefore the memory cell holds one binary digit or bit which is either 1 or 0 meaning on or off.

In our next lesson we will discuss bytes and words.

Part 8 - Bytes, Words, Double Words, etc...

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Memory is measured in bytes. A byte is 8 bits. Two bytes are called a word and two words are called a double word which is four bytes (32-bit) and a quad word is eight bytes (64-bit).

A byte is 8 bits and is 2^8 power which is 256. The number of binary numbers 8 bits in size is one of 256 values starting at 0 and going to 255.

Every byte of memory in a computer has its own unique address. Let's review the disassembled instructions for a simple hello world application in Linux by setting a breakpoint on the main function. We will use the GDB debugger:

```
Starting program: /home/noroot/Desktop/Code/Example1/Example1
Breakpoint 1, main () at Example1.c:4
4      printf("hello world");
(gdb) disas main
Dump of assembler code for function main:
0x0804840b <+0>:   lea    ecx,[esp+0x4]
0x0804840f <+4>:   and    esp,0xffffffff
0x08048412 <+7>:   push  DWORD PTR [ecx-0x4]
0x08048415 <+10>:  push  ebp
0x08048416 <+11>:  mov    ebp,esp
0x08048418 <+13>:  push  ecx
0x08048419 <+14>:  sub    esp,0x4
=> 0x0804841c <+17>:  sub    esp,0xc
0x0804841f <+20>:  push  0x80484c0
0x08048424 <+25>:  call  0x80482e0 <printf@plt>
0x08048429 <+30>:  add    esp,0x10
0x0804842c <+33>:  mov    eax,0x0
0x08048431 <+38>:  mov    ecx,DWORD PTR [ebp-0x4]
0x08048434 <+41>:  leave
0x08048435 <+42>:  lea   esp,[ecx-0x4]
0x08048438 <+45>:  ret
End of assembler dump.
(gdb) █
```

Don't worry if this does not make sense yet. The point of utilizing this example is to give you a sneak peek into our first program that we will examine in addition to learning about memory in a computer.

Below is an examination of the ESP register. Again, it is not critical that you understand what a register is or what ESP does. We simply want to see what a memory location looks like:

```
(gdb) x/1xw $esp
0xffffd040:   0xf7fac3dc
```

We see the memory location of 0xffffd040 which of course is in hexadecimal. We also see the value inside the ESP register which is 0xf7fac3dc which is also in hexadecimal.

Part 1: Goals

It is important to understand that 0xffffd040 is 4 bytes and is a double word. As we learned in Part 6: Hexadecimal Number System, each hexadecimal digit is 4 bits long otherwise called a nibble. In 0xffffd040, lets look at the right most digit of 0. In this example, 0 (hexadecimal) is 4 bits long. If we look at 40 (in hexadecimal), we see that is a byte in length or 8 bits long. If we look at d040, we have two bytes or a word in length. Finally, fffd040 is a double word or 4 bytes in length which is 32-bits long. The 0x at the beginning of the address just designates that it is a hexadecimal value.

A computer program is nothing more than machine instructions stored in memory. A 32-bit CPU fetches a double word from a memory address. A double word is 4 bytes in a row which is read from memory and loaded into the CPU. As soon as it finishes executing, the CPU fetches the next machine instruction in memory from the instruction pointer.

Those of you new to assembly have now had your first look. Don't get discouraged or frustrated if you do not know what is going on here. We will take our time and go through dozens of examples to break down each step in future lessons. What is important is that you take your time and examine what each lesson is discussing. Please always feel free to comment below with any questions.

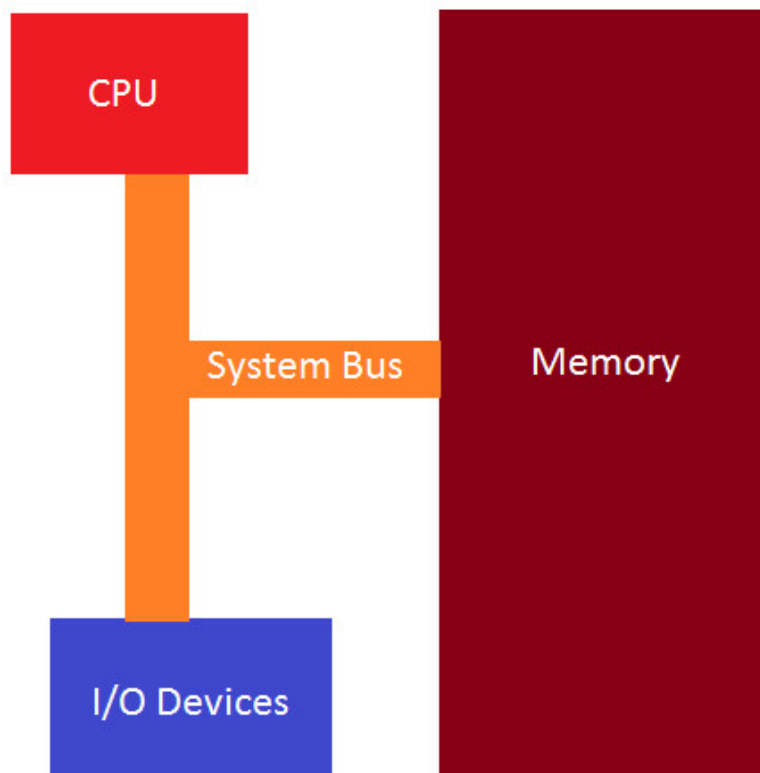
In our next tutorial we will discuss the basics of x86 Architecture.

Part 9: x86 Basic Architecture

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

A computer application is simply a table of machine instructions stored in memory to which the binary numbers which make up the program are unique only in the way the CPU deals with them.

The basic architecture is made up of a CPU, memory and I/O devices which are input/output devices which are all connected by a system bus as detailed below.



The CPU consists of 4 parts which are:

- 1)Control Unit - Retrieves and decodes instructions from the CPU and then storing and retrieving them to and from memory.
- 2)Execution Unit - Where the execution of fetching and retrieving instructions occurs.
- 3)Registers - Internal CPU memory locations used a temporary data storage.
- 4)Flags - Indicate events when execution occurs.



We will discuss 32-bit x86 so therefore a 32-bit CPU first fetches a double word (4 bytes or 32-bits in length) from a specific address in memory and is read from memory and loaded into the CPU. At this point the CPU looks at the binary pattern of bits within the double word and begins executing the procedure that the fetched machine instruction directs it to do.

Upon completion of executing an instruction, the CPU goes to memory and fetches the next machine instruction in sequence. The CPU has a register, which we will discuss registers in a future tutorial, called the EIP or instruction pointer that contains the address of the next instruction to be fetched from memory and then executed.

We can immediately see that if we controlled flow of EIP, we can alter the program to do things it was NOT intended to do. This is a popular technique upon which malware operates.

The entire fetch and execute process is tied to the system clock which is an oscillator that emits square-wave pulses at precise intervals.

In our next tutorial we will dive deeper into the IA-32 Architecture with a discussion of the General-purpose Registers.

Part 10: General-purpose Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

The general-purpose registers are used to temporarily store data as it is processed on the processor. The registers have evolved dramatically over time and continue to do so. We will focus on 32-bit x86 architecture for our purposes.

Each new version of general-purpose registers is created to be backward compatible with previous processors. This means that code utilizing 8-bit registers on the 8080 chips will still function on today's 64-bit chipset.

General-purpose registers can be used to hold any type of data to which some have acquired specific use which are used in programs. Lets review the 8 general-purpose registers in an IA-32 architecture.

EAX: Main register used in arithmetic calculations. Also known as accumulator, as it holds results of arithmetic operations and function return values.

EBX: The Base Register. Pointer to data in the DS segment. Used to store the base address of the program.

ECX: The Counter register is often used to hold a value representing the number of times a process is to be repeated. Used for loop and string operations.

EDX: A general purpose register. Additionally used for I/O operations. In addition will extend EAX to 64-bits.

ESI: Source Index register. Pointer to data in the segment pointed to by the DS register. Used as an offset address in string and array operations. It holds the address from where to read data.

EDI: Destination Index register. Pointer to data (or destination) in the segment pointed to by the ES register. Used as an offset address in string and array operations. It holds the implied write address of all string operations.

EBP: Base Pointer. Pointer to data on the stack (in the SS segment). It points to the bottom of the current stack frame. It is used to reference local variables.

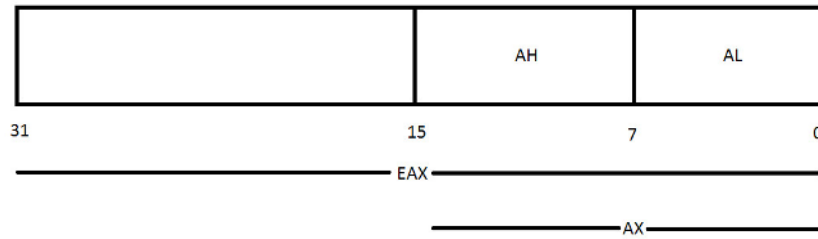
ESP: Stack Pointer (in the SS segment). It points to the top of the current stack frame. It is used to reference local variables.

Keep in mind each of the above registers are 32-bit in length or 4 bytes in length. Each of the lower 2 bytes of the EAX, EBX, ECX, and EDX registers can be referenced by AX and then subdivided by the names AH, BH, CH and DH for high bytes and AL, BL, CL and DL for the low bytes which are 1 byte each.

In addition, the ESI, EDI, EBP and ESP can be referenced by their 16-bit equivalent which is SI, DI, BP, SP.

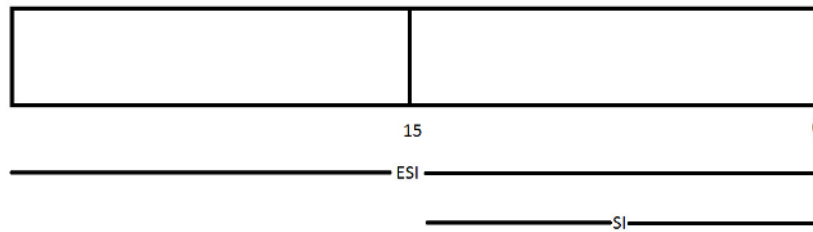
This can be a bit confusing to someone who has not studied computer engineering however let me illustrate in the table below:

Part 1: Goals



EAX would have AX as its 16-bit segment and then you can further subdivide AX into AL for the low 8 bits and AH for the high 8 bits. The same holds true for EBX, ECX and EDX as well. EBX would have BX as its 16-bit segment and then you can further subdivide BX into BL for the low 8 bits and BH for the high 8 bits. ECX would have CX as its 16-bit segment and then you can further subdivide CX into CL for the low 8 bits and CH for the high 8 bits. EDX would have DX as its 16-bit segment and then you can further subdivide DX into DL for the low 8 bits and DH for the high 8 bits.

ESI, EDI, EBP and ESP can be broken down into its 16-bit segments as follows:



ESI would have SI as its 16-bit segment, EDI would have DI as its 16-bit segment, EBP would have BP as its 16-bit segment and ESP would have SP as its 16-bit segment.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the Segment Registers.

Part 11: Segment Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The segment registers are used specifically for referencing memory locations. There are three different methods of accessing system memory of which we will focus on the flat memory model which is relevant for our purposes.

There are six segment registers which are as follows:

CS: Code segment register stores the base location of the code section (.text section) which is used for data access.

DS: Data segment register stores the default location for variables (.data section) which is used for data access.

ES: Extra segment register which is used during string operations.

SS: Stack segment register stores the base location of the stack segment and is used when implicitly using the stack pointer or when explicitly using the base pointer.

FS: Extra segment register.

GS: Extra segment register.

Each segment register is 16-bits and contains the pointer to the start of the memory-specific segment. The CS register contains the pointer to the code segment in memory. The code segment is where the instruction codes are stored in memory. The processor retrieves instruction codes from memory based on the CS register value and an offset value contained in the instruction pointer (EIP) register. Keep in mind no program can explicitly load or change the CS register. The processor assigns its values as the program is assigned a memory space.

The DS, ES, FS and GS segment registers are all used to point to data segments. Each of the four separate data segments help the program separate data elements to ensure that they do no overlap. The program loads the data segment registers with the appropriate pointer value for the segments and then reference individual memory locations using an offset value.

The stack segment register (SS) is used to point to the stack segment. The stack contains data values passed to functions and procedures within the program.

Segment registers are considered part of the operating system and can neither read nor be changed directly in almost all cases. When working in the protected mode flat model (x86 architecture which is 32-bit), your program runs and receives a 4GB address space to which any 32-bit register can potentially address any of the four billion memory locations except for those protected areas defined by the operating system. Physical memory may be larger than 4GB however a 32-bit register can only express 4,294,967,296 different locations. If you have more than 4GB of memory in your computer, the OS must arrange a

Part 1: Goals

4GB region within memory and your programs are limited to that new region. This task is completed by the segment registers and the OS keeps close control of this.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the Instruction Pointer Register.

Part 12: Instruction Pointer Register

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The instruction pointer register called the EIP register is simply the most important register you will deal with in any reverse engineering. The EIP keeps track of the next instruction code to execute. EIP points to the next instruction to execute. If you were to alter that pointer to jump to another area in the code you have complete control over that program.

Lets jump ahead and dive into some code. Here is an example of a simple hello world application in C that we will go into more detail much later in our tutorial series. For our purposes today, we will see the raw POWER of assembly language and particularly that of the EIP register and what we can do to completely hack program control.

```
#include <stdio.h>
#include <stdlib.h>

void unreachableFunction(void) {
    printf("I'm hacked! I am a hidden function!\n");
    exit(0);
}

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

Don't worry if you do not understand what it does or its functionality. What to take note of here is the fact we have a function called `unreachableFunction` that is never called by the `main` function. As you will see if we can control the EIP register we can hack this program to execute that code!

```
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o eipExample eipExample.c
noroot@noroot-VirtualBox:~/Desktop$ nano eipExample.c
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o eipExample eipExample.c
noroot@noroot-VirtualBox:~/Desktop$ ./eipExample
Hello World!
```

We have simply compiled the code to work with the IA32 instruction set and ran it. As you can see there is no call to the `unreachableFunction` of any kind as it is unreachable under normal conditions as you can see the 'Hello World!' printed when executed.

```
(gdb) set disassembly-flavor intel
(gdb) b main
Breakpoint 1 at 0x804846c: file eipExample.c, line 10.
(gdb) r
Starting program: /home/noroot/Desktop/eipExample

Breakpoint 1, main () at eipExample.c:10
10      printf("Hello World!\n");
(gdb) disas
Dump of assembler code for function main:
   0x0804845b <+0>:   lea    ecx,[esp+0x4]
   0x0804845f <+4>:   and    esp,0xffffffff
   0x08048462 <+7>:   push  DWORD PTR [ecx-0x4]
   0x08048465 <+10>:  push  ebp
   0x08048466 <+11>:  mov    ebp,esp
   0x08048468 <+13>:  push  ecx
   0x08048469 <+14>:  sub    esp,0x4
=> 0x0804846c <+17>:  sub    esp,0xc
   0x0804846f <+20>:  push  0x8048535
   0x08048474 <+25>:  call  0x8048300 <puts@plt>
   0x08048479 <+30>:  add    esp,0x10
   0x0804847c <+33>:  mov    eax,0x0
   0x08048481 <+38>:  mov    ecx,DWORD PTR [ebp-0x4]
   0x08048484 <+41>:  leave
   0x08048485 <+42>:  lea   esp,[ecx-0x4]
   0x08048488 <+45>:  ret
End of assembler dump.
```

We have disassembled the program using the GDB Debugger. We have set a breakpoint on the main function and ran the program. The => shows where EIP is pointing to when we step to the next instruction. If we follow normal program flow, 'Hello World!' will print to the console and exit.

```
(gdb) c
Continuing.
Hello World!
[Inferior 1 (process 3034) exited normally]
```

If we run the program again and do an examination of where EIP is pointing to we will see:

```
(gdb) r
Starting program: /home/noroot/Desktop/eipExample

Breakpoint 1, main () at eipExample.c:10
10      printf("Hello World!\n");
(gdb) x/1xb $eip
0x804846c <main+17>:  0x83
(gdb) x/1xw $eip
0x804846c <main+17>:  0x680cec83
```

We can see EIP is pointing to main+17 or the address of 0x680cec83.

Lets examine the **unreachableFunction** and see where it starts in memory and write down that address.

```
(gdb) disas unreachableFunction
Dump of assembler code for function unreachableFunction:
   0x0804843b <+0>:    push   ebp
   0x0804843c <+1>:    mov    ebp,esp
   0x0804843e <+3>:    sub    esp,0x8
   0x08048441 <+6>:    sub    esp,0xc
   0x08048444 <+9>:    push  0x8048510
   0x08048449 <+14>:   call  0x8048300 <puts@plt>
   0x0804844e <+19>:   add    esp,0x10
   0x08048451 <+22>:   sub    esp,0xc
   0x08048454 <+25>:   push  0x0
   0x08048456 <+27>:   call  0x8048310 <exit@plt>
End of assembler dump.
```

The next step is to set EIP to address 0x0804843b so that we hijack program flow to run the unreachableFunction.

```
(gdb) set $eip = 0x0804843b
(gdb) x/1xw $eip
0x804843b <unreachableFunction>:    0x83e58955
```

Now that we have hacked control of EIP, lets continue and watch how we have hijacked the operation of a running program to our advantage!

```
(gdb) c
Continuing.
I'm hacked!  I am a hidden function!
[Inferior 1 (process 3048) exited normally]
```

Tada! We have hacked the program!

So the question in your mind is why did you show me this when I have no idea of what any of this is? It is important to understand that when we are doing a lengthy tutorial such as this we should sometimes look forward to see why we are taking so many steps to learn the basics before we dive in. It is important however to show you that if you stay with the tutorial your hard work will pay off as we will learn how to hijack any running program to make it do whatever we want in addition to proactively breaking down a malicious program so that we can not only disable it but trace it back to a potential IP of where the hack originated.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the Control Registers.

Part 13: Control Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

There are five control registers which are used to determine the operating mode of the CPU and the characteristics of the current executing task. Each control register is as follows:

CR0: System flag that control the operating mode and various states of the processor.

CR1: (Not Currently Implemented)

CR2: Memory page fault information.

CR3: Memory page directory information.

CR4: Flags that enable processor features and indicate feature capabilities of the processor.

The values in each of the control registers can't be directly accessed however the data in the control register can be moved to one of the general-purpose registers and once the data is in a GP register, a program can examine the bit flags in the register to determine the operating status of the processor in conjunction with the current running task.

If a change is required to a control register flag value, the change can be made to the data in the GP register and the register moved to the CR. Low-level System Programmers usually modify the values in control registers. Normal application programs do not usually modify control register entries however they might query flag values to determine the capabilities of the host processor chip on which the program is currently running.

In our next tutorial we will continue our discussion of the IA-32 Architecture with the topic of Flags.

Part 14: Flags

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The topic of flags are one of the most extremely complex and complicated concepts of assembly language and program flow control when reverse engineering. This information below will become much clearer as we enter into the final phase of our training when we reverse engineer C applications into assembly language.

What is important here is to take away the fact that flags help control, check and verify program execution and are a mechanism to determine whether each operation that is performed by the processor is successful or not.

Flags are critical to assembly language applications as they are a check to verify each programs functions successful execution.

We are dealing with 32-bit assembly to which a single 32-bit register which contains a group of status, control and system flags exist. This register is called the EFLAGS register as it contains 32 bits of information that are mapped to represent specific flags of information.

There are three kinds of flags which are status flags, control flags and system flags.

Status flags are as follows:

CF: Carry Flag

PF: Parity Flag

AF: Adjust Flag

ZF: Zero Flag

SF: Sign Flag

OF: Overflow Flag

The carry flag is set when a math operation on an unsigned integer value generates a carry or borrow for the most significant bit. This is an overflow condition for the register involved in the math operation. When this occurs, the remaining data in the register is not the correct answer to the math operation.

The parity flag is used to indicate corrupt data as a result of a math operation in a register. When checked, the parity flag is set if the total number of 1 bits in the result is even and is cleared if the total number of 1 bits in the result is odd. When the parity flag is checked, an application can determine whether the register has been corrupted since the operation.

The adjust flag is used in Binary Coded Decimal math operations and is set if a carry or borrow operation occurs from bit 3 of the register used for the calculation.

The zero flag is set if the result of an operation is zero.

The sign flag is set to the most significant bit of the result which is the sign bit and indicates whether the result is positive or negative.

The overflow flag is used in signed integer arithmetic when a positive value is too big or a negative value is too small to be represented in the register.

Control flags are utilized to control specific behavior in the processor. The DF flag which is the direction flag is used to control the way strings are handled by the processor. When set, string instructions automatically decrement memory addresses to get the next byte in the string. When cleared, string instructions automatically increment memory addresses to get the next byte in the string.

System flags are used to control OS level operations which should NEVER be modified by any respective program or application.

TF: Trap Flag

IF: Interrupt Enable Flag

IOPL: I/O Privilege Level Flag

NT: Nested Task Flag

RF: Resume Flag

VM: Virtual-8086 Mode Flag

AC: Alignment Check Flag

VIF: Virtual Interrupt Flag

VIP: Virtual Interrupt Pending Flag

ID: Identification Flag

The trap flag is set to enable single-step mode and when in this mode the processor performs only one instruction code at a time, waiting for a signal to perform the next instruction. This is essential when debugging.

The interrupt enable flag controls how the processor responds to signals received from external sources.

The I/O privilege field indicates the input-output privilege level of the currently running task and defines access levels for the input-output address space which must be less than or equal to the access level required to access the respective address space. In the case where it is not less than or equal to the access level required, any request to access the address space will be denied.

The nested task flag controls whether the currently running task is linked to the previously executed task and is used for chaining interrupted and called tasks.

The resume flag controls how the processor responds to exceptions when in debugging mode.

The VM flag indicates that the processor is operating in virtual-8086 mode instead of protected or real mode.

Part 1: Goals

The alignment check flag is used in conjunction with the AM bit in the CR0 control register to enable alignment checking of memory references.

The virtual interrupt flag replicates the IF flag when the processor is operating in virtual mode.

The virtual interrupt pending flag is used when the processor is operating in virtual mode to indicate that an interrupt is pending.

The ID flag indicates whether the processor supports the CPUID instruction.

In our next tutorial we will discuss the stack.

Part 15: Stack

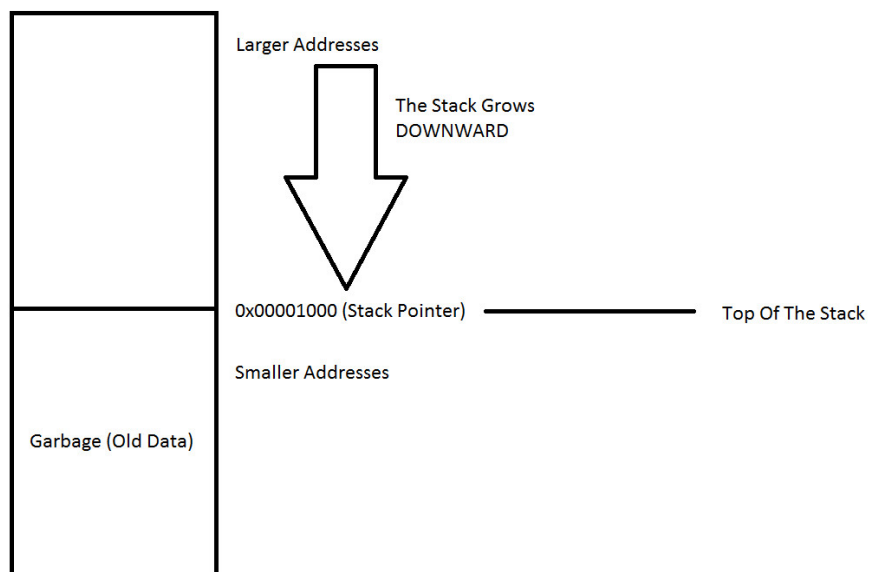
For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Functions are the most fundamental feature in software development. A function allows you to organize code in a logical way to execute a specified task. It is not critical that you understand how functions work at this stage it is only important that you understand that when we start learning to develop, we want to minimize duplication by using functions that can be called multiple times rather than duplicate code taking up excessive memory.

When a program starts to execute a certain contiguous section of memory is set aside for the program called the stack.

The stack pointer is a register that contains the top of the stack. The stack pointer contains the smallest address, lets say for example 0x00001000, such that any address smaller than 0x00001000 is considered garbage and any address greater than 0x00001000 is considered valid.

The above address is random and is not an absolute where you will find the stack pointer from program to program as it will vary. Lets look at what the stack looks like from an abstract perspective:



The above diagram is what I want you to keep clear in your mind as that is what is actually happening in memory. The next series of diagrams will show the opposite of what is shown above.

You will see the stack growing upward in the below diagrams however in reality it is growing downward from higher memory to lower memory.

In the addMe example below, the stack pointer (ESP), when examined in memory on a breakpoint on the main function, lists 0xffffd050. When the program calls the addMe function from main, ESP is now 0xffffd030 which is LOWER in memory.

Therefore the stack grows DOWNWARD despite the diagram showing it pointing upward. Just keep in mind when the arrows below are pointing upward they are actually pointing to lower memory addresses.

The stack bottom is the largest valid address of the stack and is located in the larger address area or top of the memory model. This can be confusing as the stack bottom is higher in memory. The stack grows downward in memory and it is critical that you understand that now as we go forward.

The stack limit is the smallest valid address of the stack. If the stack pointer gets smaller than this, there is a stack overflow which can corrupt a program to allow an attacker to take control of a system. Malware attempts to take advantage of stack overflows. As of recent, there are protections build into modern OS that attempt to prevent this from happening.

There are two operations on the stack which are push and pop. You can push one or more registers by setting the stack pointer to a smaller value. This is usually done by subtracting four times the number of registers to be pushed onto the stack and copying the registers to the stack.

You can pop one or more registers by copying the data from the stack to the registers, then to add a value to the stack pointer. This is usually done by adding four times the number of registers to be popped on the stack.

Let us look at how the stack is used to implement functions. For each function call there is a section of the stack reserved for the function. This is called the stack frame.

Let's look at the C program we created in tutorial 12 and examine what the main function looks like:

```
#include <stdio.h>
#include <stdlib.h>

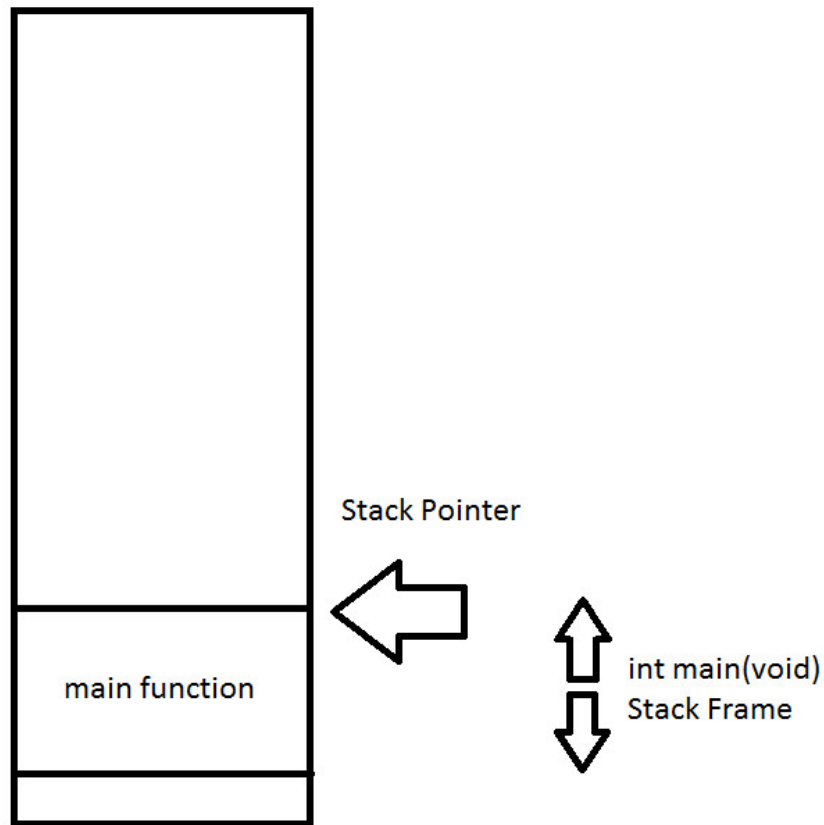
void unreachableFunction(void) {
    printf("I'm hacked! I am a hidden function!\n");
    exit(0);
}

int main(void) {
    printf("Hello World!\n");

    return 0;
}
```

We see two functions here. The first one is the unreachableFunction to which will never execute under normal circumstances and we also see the main function that will always be the first function to be called onto the stack.

When we run this program, the stack will look like this:



We can see the stack frame for `int main(void)` above. It is also referred to as the activation record. A stack frame exists whenever a function has started but yet to complete. For example, inside of the body of the `int main(void)` there is a call to `int addMe(int a, int b)` which takes two arguments `a` and `b`. There needs to be assembly language code in `int main(void)` to push the arguments for `int addMe(int a, int b)` onto the stack. Lets examine some code.

```
#include <stdio.h>

int addMe(int a, int b);

int main(void){
    int result = addMe(2, 3);

    printf("The result of the addMe function is %d!\n", result);

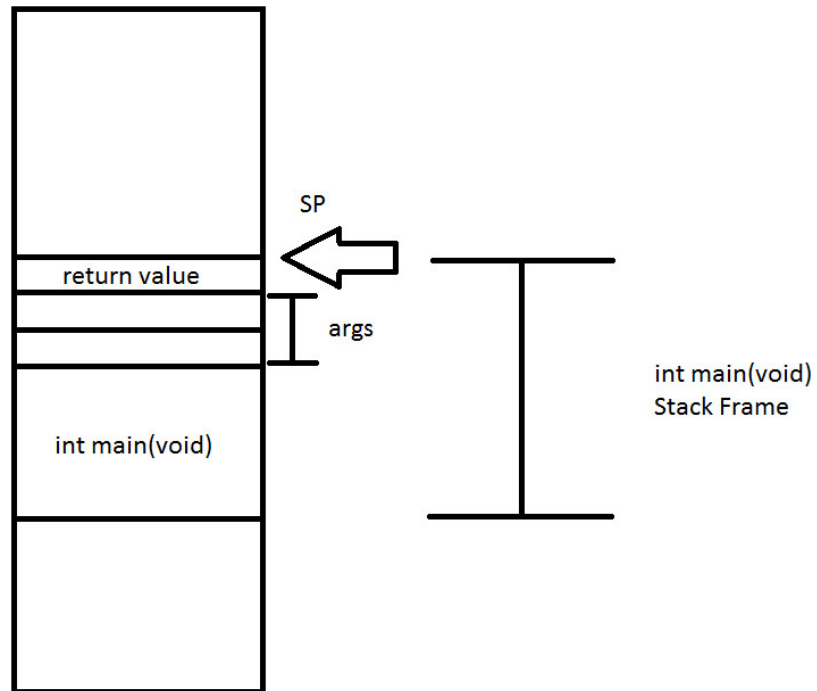
    return 0;
}

int addMe(int a, int b){
    return a + b;
}
```

When we compile and run this program we will see the value of 5 to be print out like this:

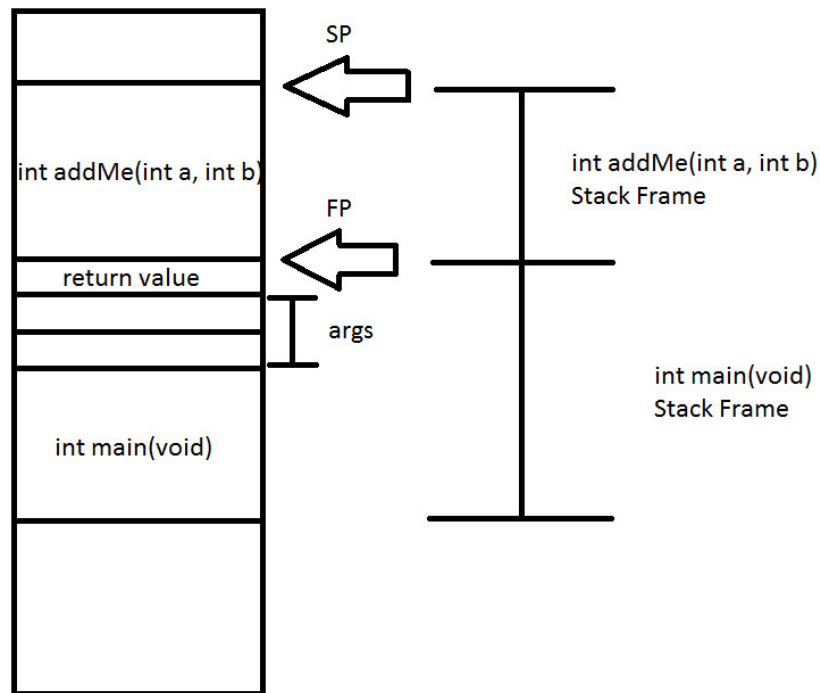
```
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o addMe addMe.c
noroot@noroot-VirtualBox:~/Desktop$ ./addMe
The result of the addMe function is 5!
```

Very simply, `int main(void)` calls `int addMe(int a, int b)` first and will get put on the stack like this:



You can see that by placing the arguments on the stack, the stack frame for `int main(void)` has increased in size. We also reserved space for the return value which is computed by `int addMe(int a, int b)` and when the function returns, the return value in `int main(void)` gets restored and execution continues in `int main(void)` until it finishes.

Once we get the instructions for `int addMe(int a, int b)`, the function may need local variables so the function needs to push some space on the stack which would look like:



`int addMe(int a, int b)` can access the arguments passed to it from `int main(void)` because the code in `int main(void)` places the arguments just as `int addMe(int a, int b)` expects it.

FP is the frame pointer and points to the location where the stack pointer was just before `int addMe(int a, int b)` moved the stack pointer or SP for `int addMe(int a, int b)`'s own local variables.

The use of a frame pointer is essential when a function is likely to move the stack pointer several times throughout the course of running the function. The idea is to keep the frame pointer fixed for the duration of `int addMe(int a, int b)`'s stack frame. In the meantime, the stack pointer can change values.

We can use the frame pointer to compute the locations in memory for both arguments as well as local variables. Since it does not move, the computations for those locations should be some fixed offset from the frame pointer.

Once it is time to exit `int addMe(int a, int b)`, the stack pointer is set to where the frame pointer is which pops off the `int addMe(int a, int b)` stack frame.

In sum, the stack is a special region of memory that stores temporary variables created by each function including main. The stack is a LIFO which is last in, first out data structure which is managed and optimized by the CPU closely. Every time a function declares a new variable it is pushed onto the stack. Every time a function exists, all of the variables pushed onto the stack by that function are freed or deleted. Once a stack variable is freed, that region of memory becomes available for other stack variables.

The advantage of the stack to store variables is that memory is managed for you. You do not have to allocate memory manually or free it manually. The CPU manages and organizes stack memory very efficiently and is very fast.

Part 1: Goals

It is critical that you understand that when a function exits, all of its variables are popped off the stack and lost forever. The stack variables are local. The stack grows and shrinks as functions push and pop local variables.

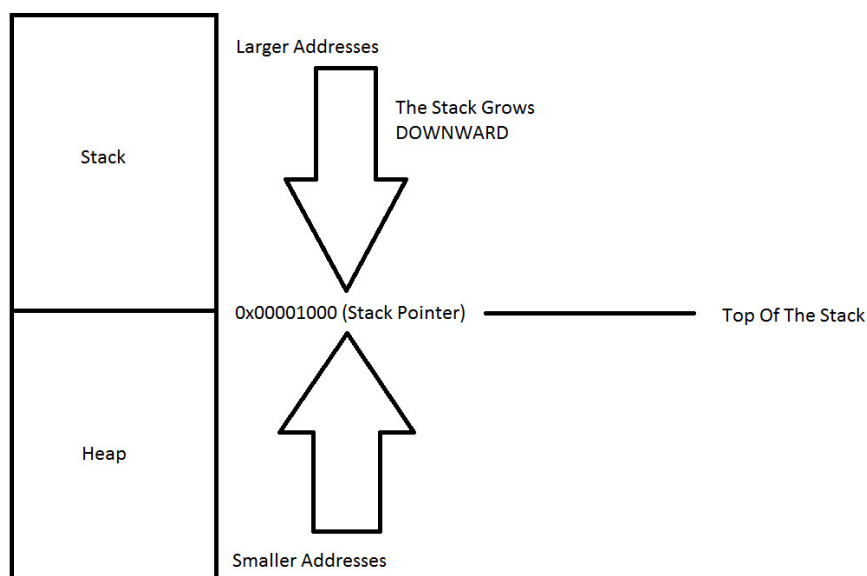
I can see your head spinning around and around. Keep in mind, these topics are complicated and will continue to develop in future tutorials. We have been dealing with a lot of confusing topics such as registers, memory and now the stack and it can be overwhelming. If you ever have questions, please comment below and I will help you to better understand this framework.

In our next tutorial we will discuss the heap.

Part 16: Heap

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Our next step in the Basic Malware Reverse Engineering section focuses on the heap. Keep in mind, the stack grows downward and the heap grows upward. It is very, very important that you understand this concept as we progress forward in our future tutorials.



The heap is the region of your computer's memory that is not managed automatically for you, and is not as tightly managed by the CPU. It is free-floating region of memory and is larger than the stack allocation of memory.

To allocate memory on the heap, you must use **malloc()** or **calloc()**, which are built-in C functions. Once you have allocated memory on the heap, you are responsible for freeing it by using **free()** to de-allocate that memory once you don't need it any more.

If you don't do this step, your program will have what is known as a memory leak. That is, memory on the heap will still be set aside and won't be available to other processes that need it.

Unlike the stack, the heap does not have size restrictions on variable size. The only thing that would limit the heap is the physical limitations of your computer. Heap memory is slightly slower to be read from and written to, because you have to use pointers to access memory on the heap. When we dive into our C tutorial series we will demonstrate this.

Unlike the stack, variables created on the heap are accessible by any function, anywhere in your program. Heap variables are essentially global in scope.

Part 1: Goals

If you need to allocate a large block of memory for something like a struct or a large array and you need to keep that variable around for a good duration of the program to which must be accessed globally, then you should choose the heap for this purpose. If you need variables like arrays and structs that can change size dynamically such as arrays that can grow or shrink as needed, then you will likely need to allocate them on the heap, and use dynamic memory allocation functions like **malloc()**, **calloc()**, **realloc()** and **free()** to manage that memory manually.

The next step is to dive into programming C in the Linux environment where we step-by-step disassemble each C program so in effect you will be learning both C programming and Assembly so that you can progress your skills in Malware Analysis and Reverse Engineering.

I look forward to seeing you all next week when we take a comprehensive step-by-step tutorial on how to install Linux on your current computer using the FREE Virtual Box software tool.

Part 17 – How To Install Linux

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

If you do not have Linux installed on a computer within your household, I would suggest installing Virtual Box which is an open-source free virtual environment which you can install on your existing computer to have a version of Linux you can program with. Below is a link to download and install Virtual Box as there are versions for both Windows and Mac.

<https://www.virtualbox.org/wiki/Downloads>



In addition, you will need a copy of Linux to which I will be working with Ubuntu. Below is a link to download the .iso file to which you will install once you have Virtual Box installed.

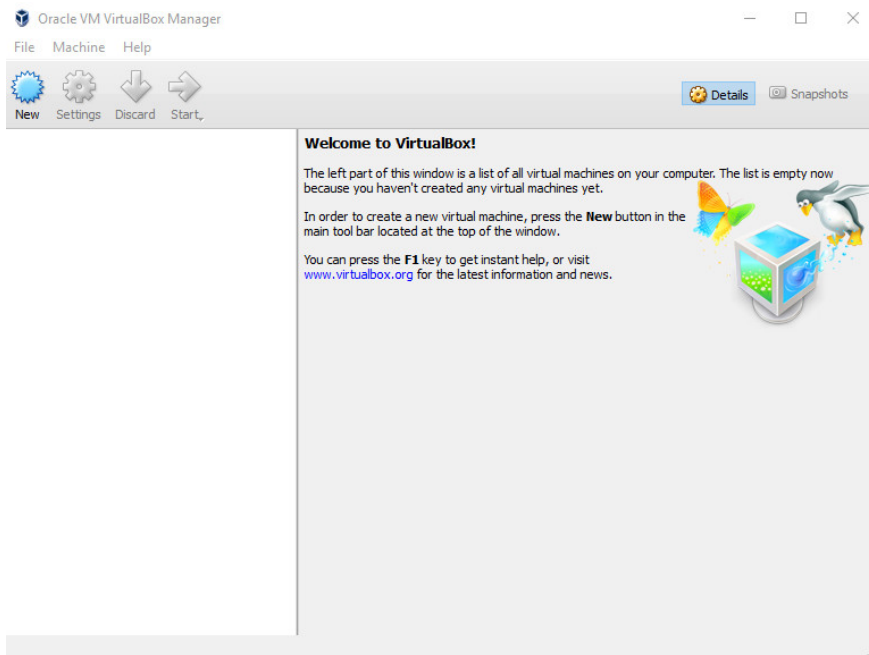
<http://www.ubuntu.com/download/desktop>



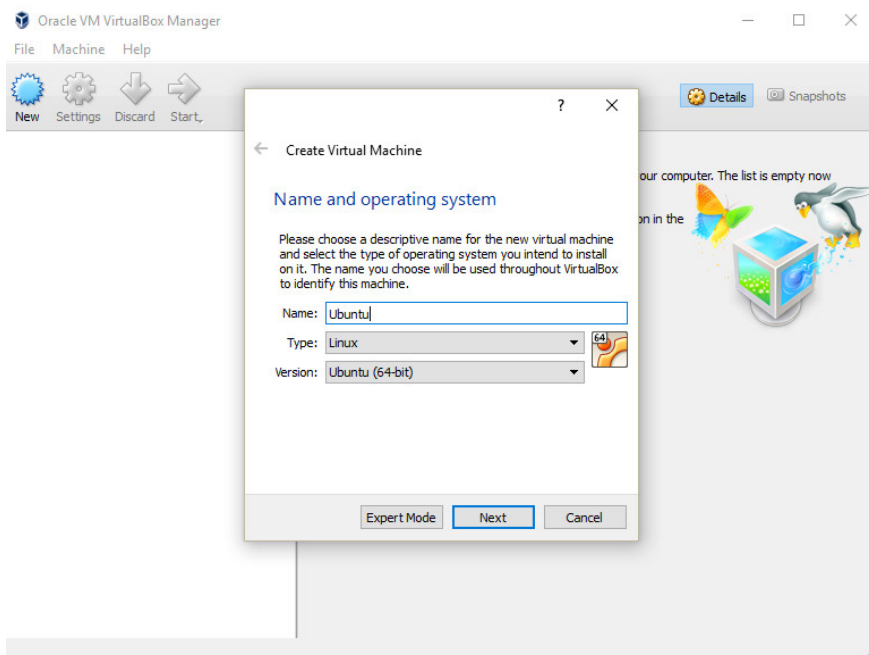
After you download the above .iso, go to your **Download** directory and first execute and run the **VirtualBox-5.0.24-108355-Win.exe** or whatever version of VirtualBox that is currently available. If you are running a Mac, you will download the **.dmg** file. Simply double-click on the file to execute and run it.

Part 1: Goals

After you install **VirtualBox-5.0.24-108355-Win.exe** or the Mac **.dmg** file and you will see this screen:

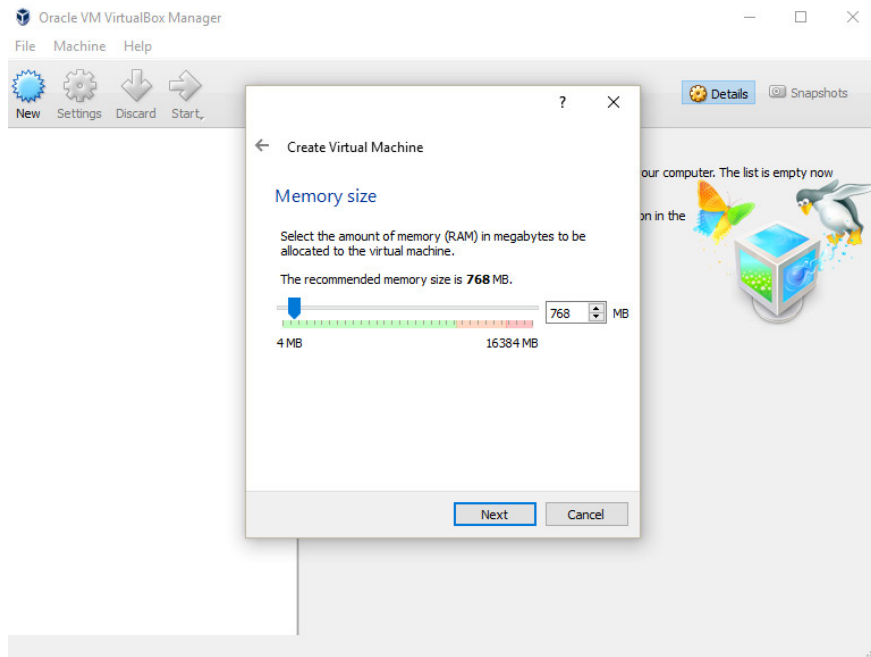


Click on the **New** button above which is located in the top-left corner of the screen as it is a big blue cog-looking circle.

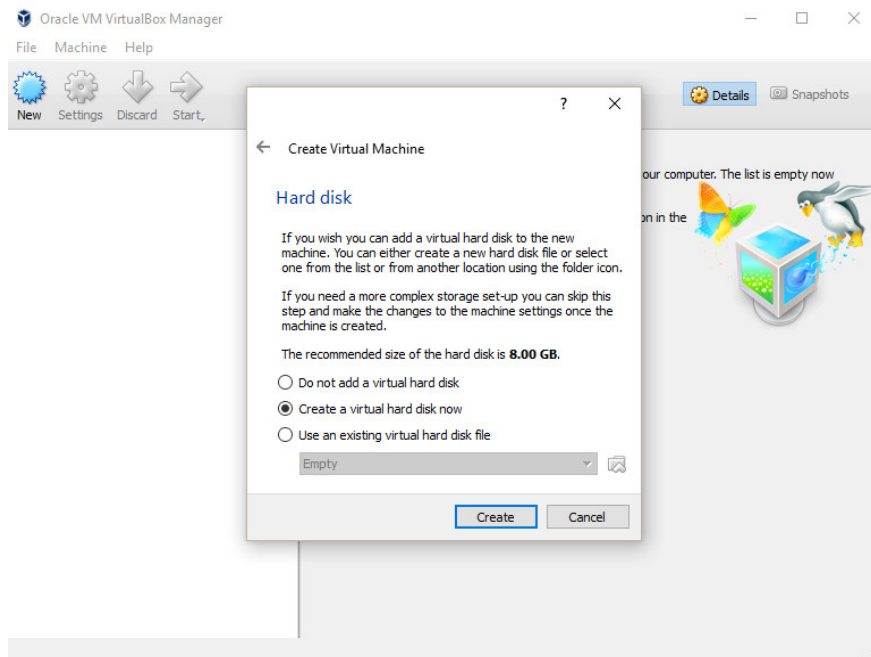


In the name field above, type **Ubuntu** and click the next button.

Part 1: Goals

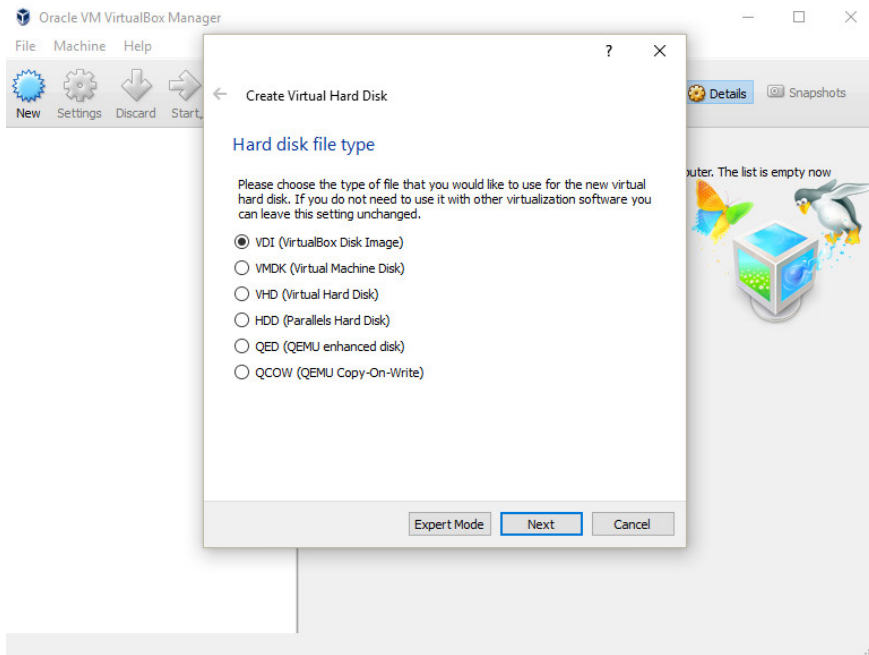


It is important to click on the blue slider bar above and select an amount of ram that points to an area in green so that it does not overwhelm your computer resources. After moving the slider, click next.

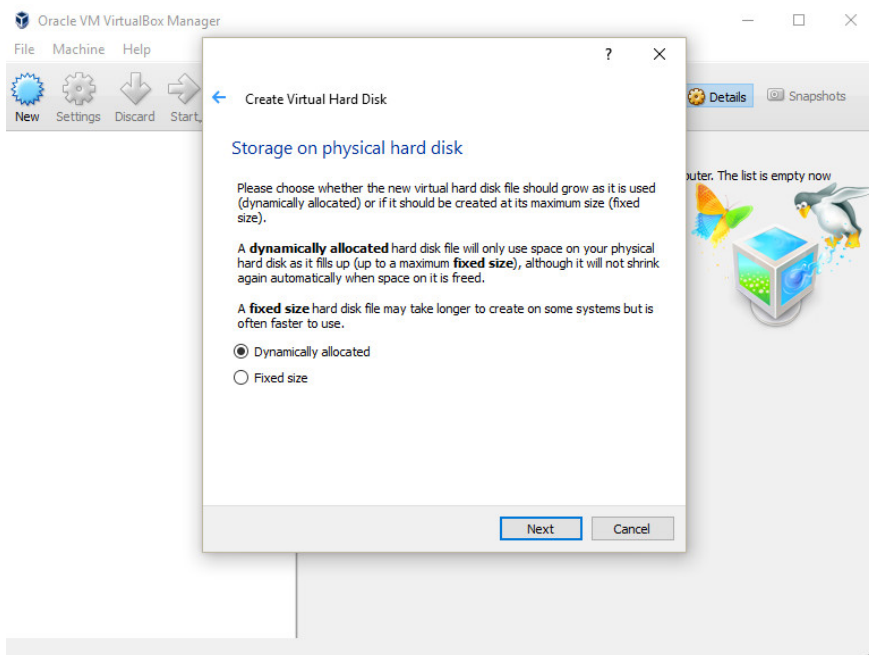


Then click create.

Part 1: Goals

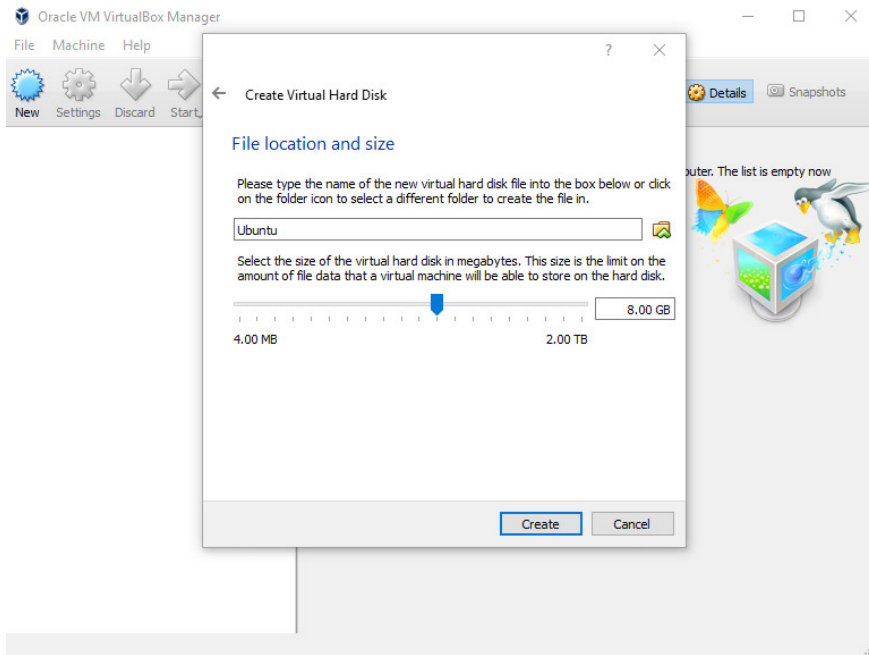


Then click next.

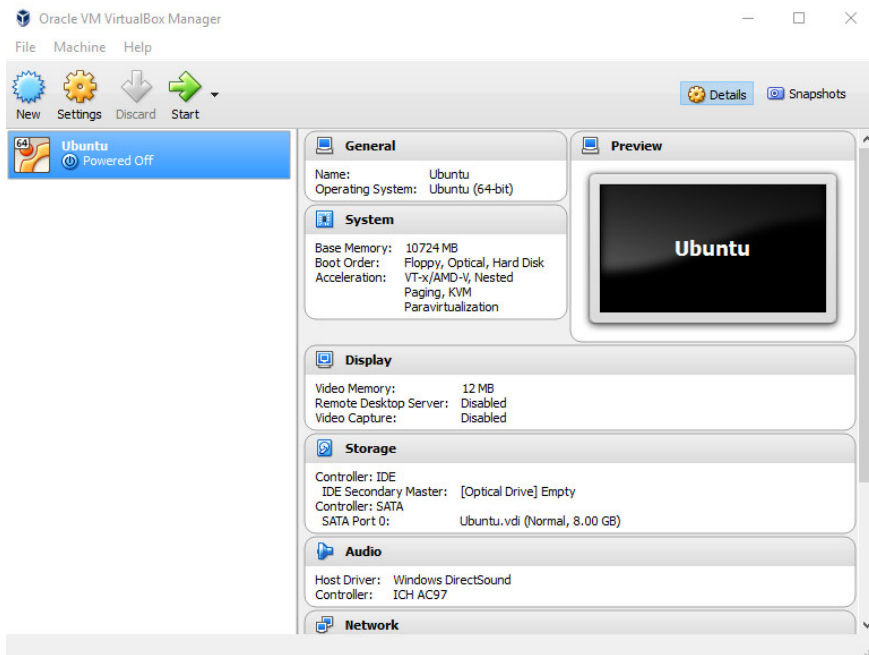


Then click next.

Part 1: Goals

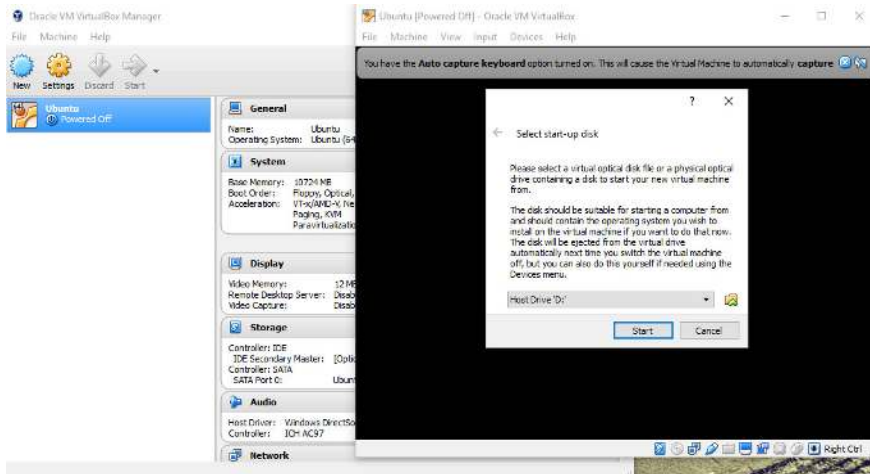


Please move the dial up to 16.00 GB rather than 8.00 GB shown above then click create.

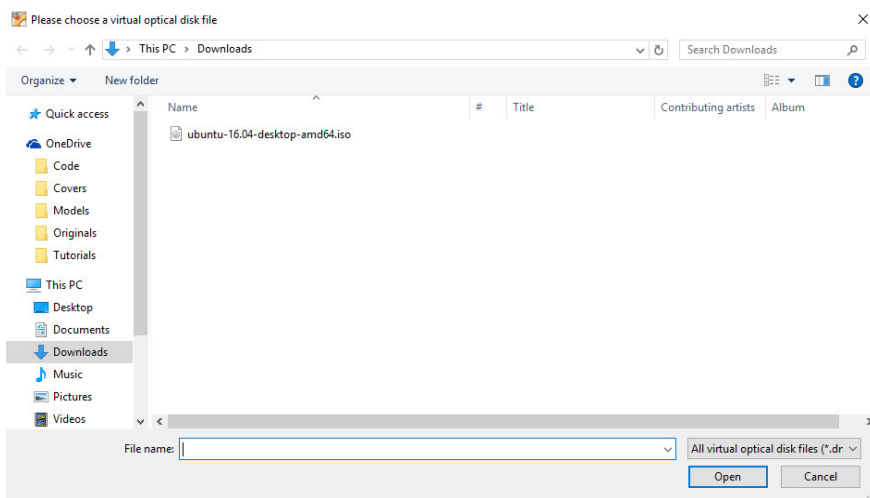


The next step is to click on the green start button.

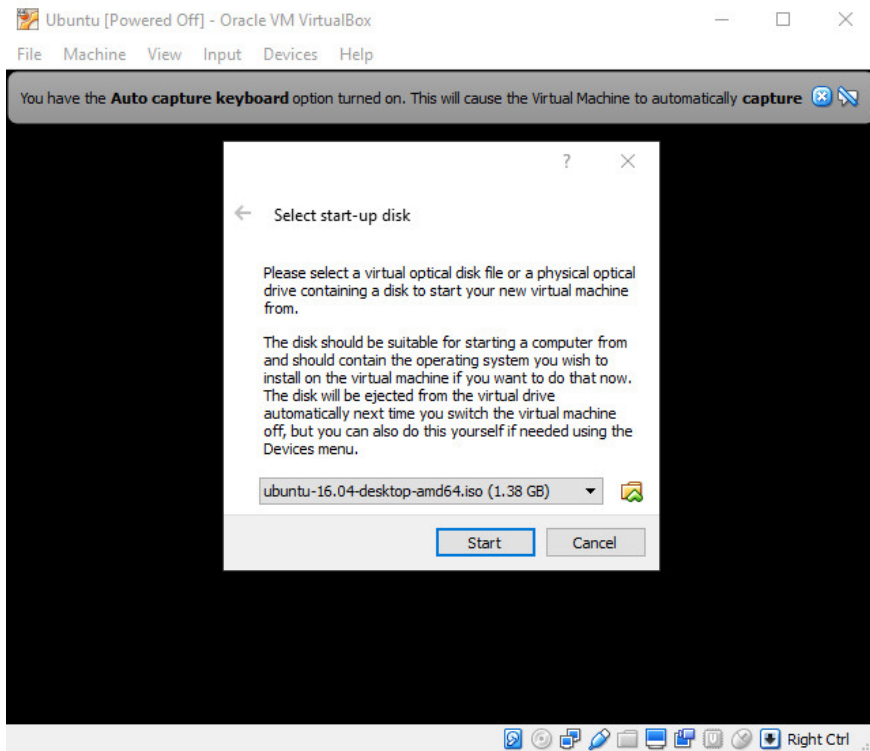
Part 1: Goals



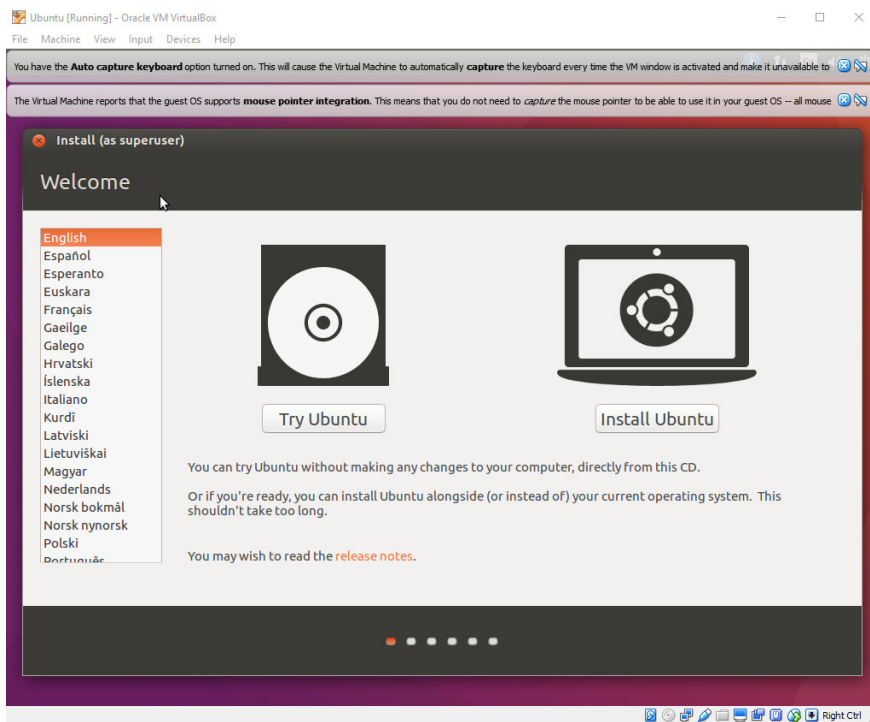
The next step is to click on the yellow folder just above the cancel button.



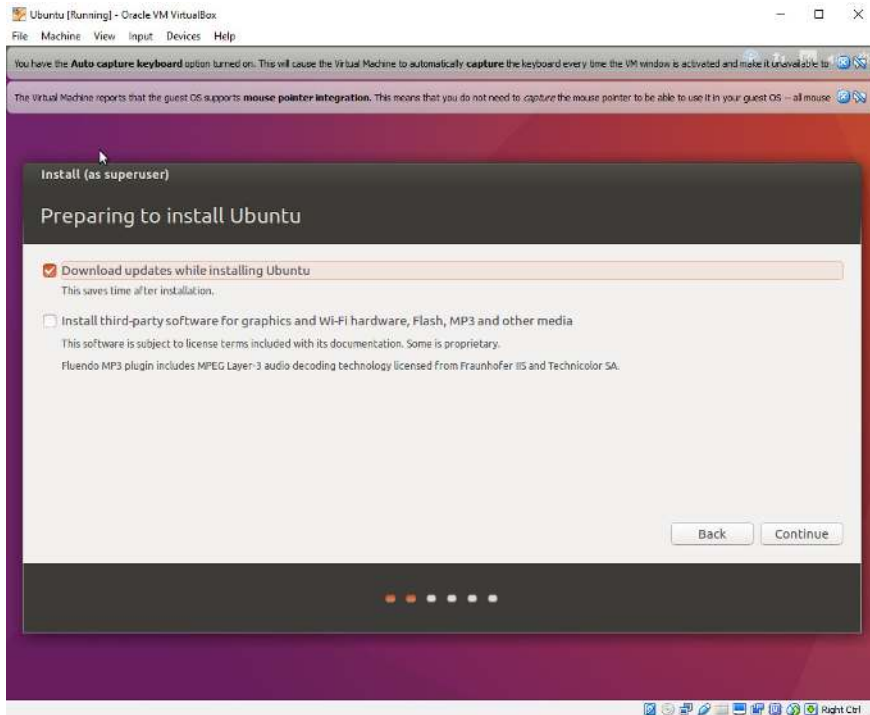
The next step is to click on the .iso file that should be in your Download directory and click open.



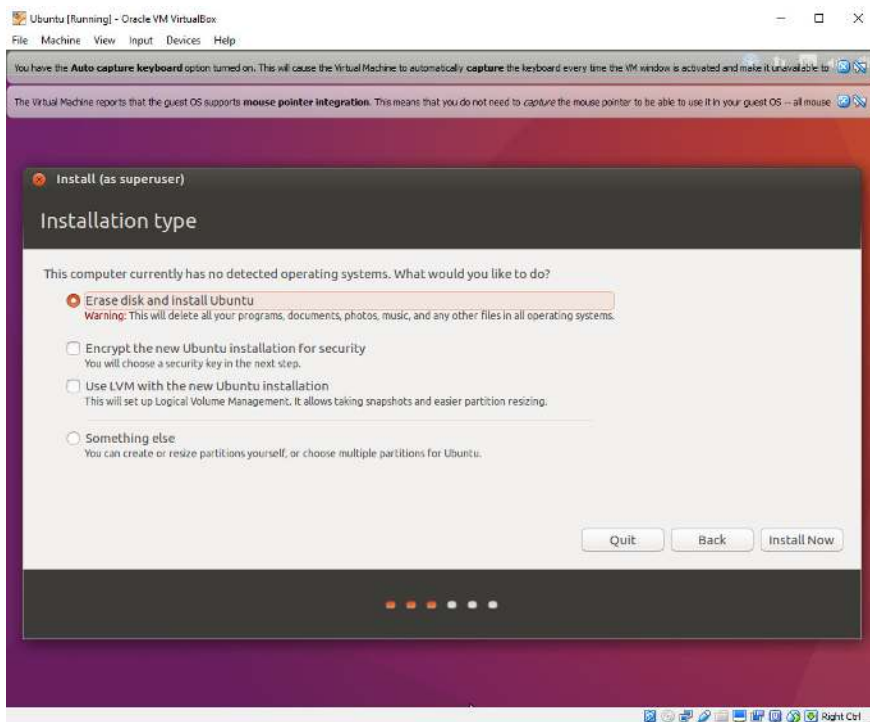
The next step is to click start.



The next step is to let the install begin and click Install Ubuntu.

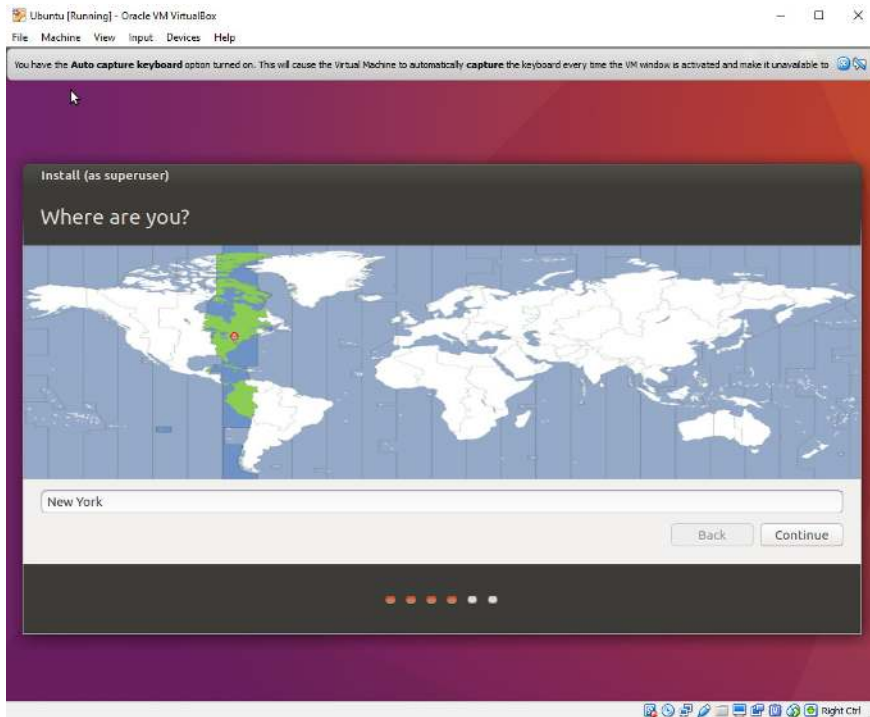


The next step is to check each of the boxes to Download updates while installing Ubuntu and click continue.

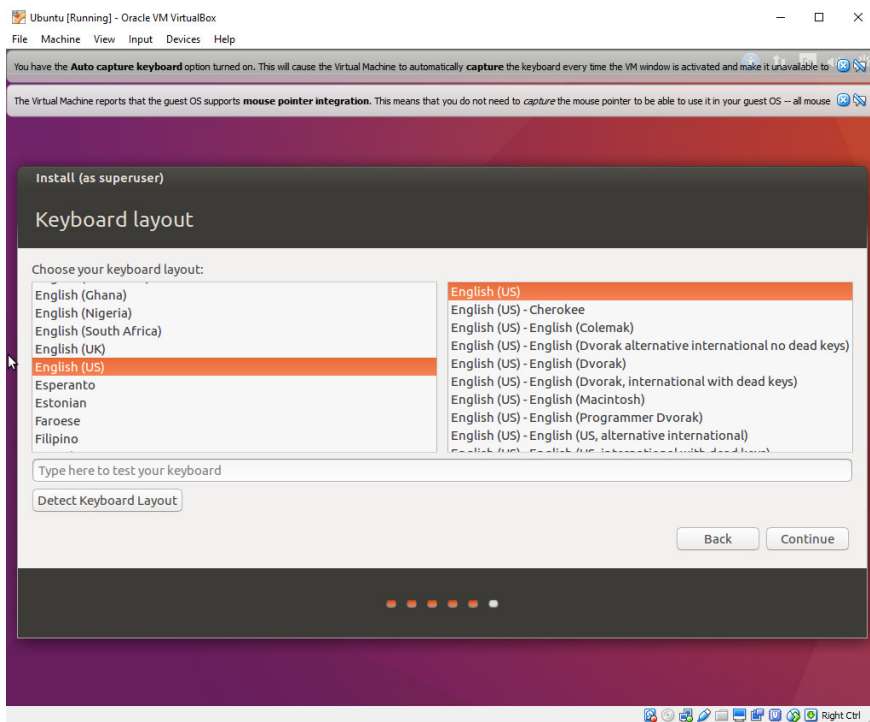


The next step is to select Erase disk and install Ubuntu and click install now.

Part 1: Goals

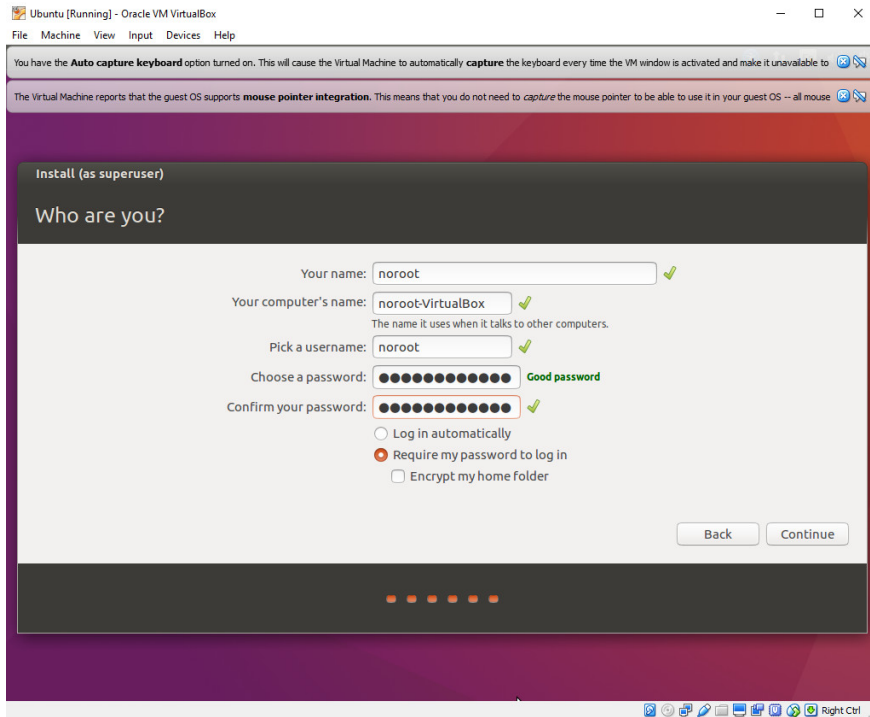


The next step is to click continue and progress forward to the screen where you will select your timezone to which you will select continue.

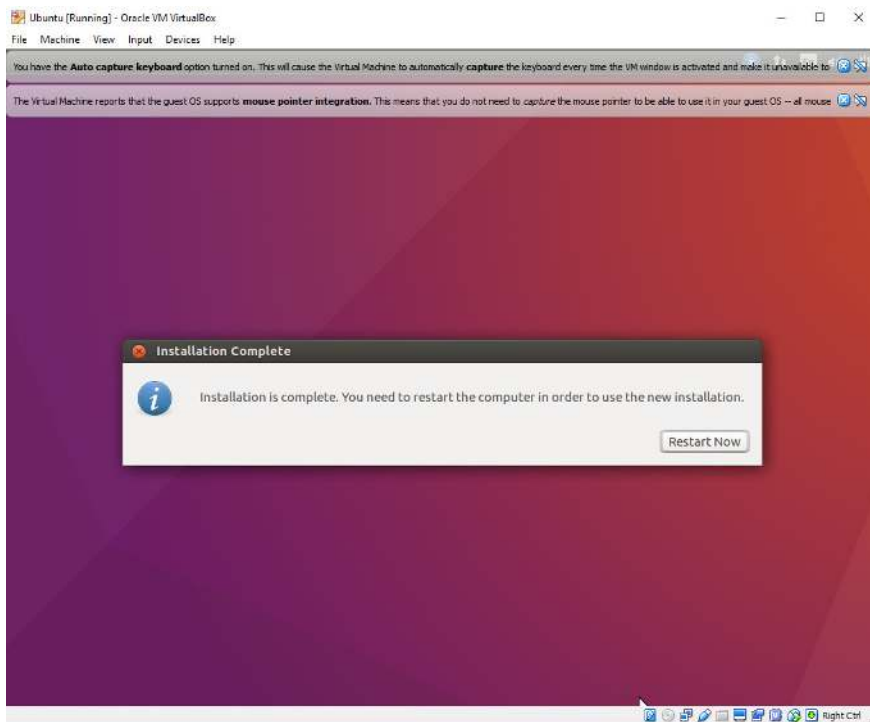


The next step is to select your keyboard layout and click continue.

Part 1: Goals

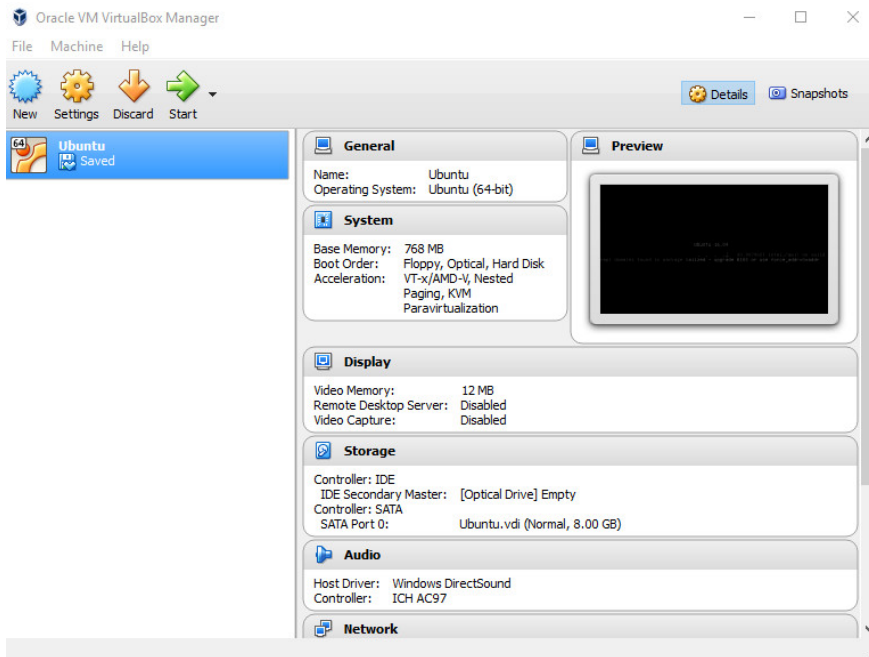


The next step is to create a name for your account. I chose noroot and did the same for the username. In addition, create a password and re-type it for verification and click continue.

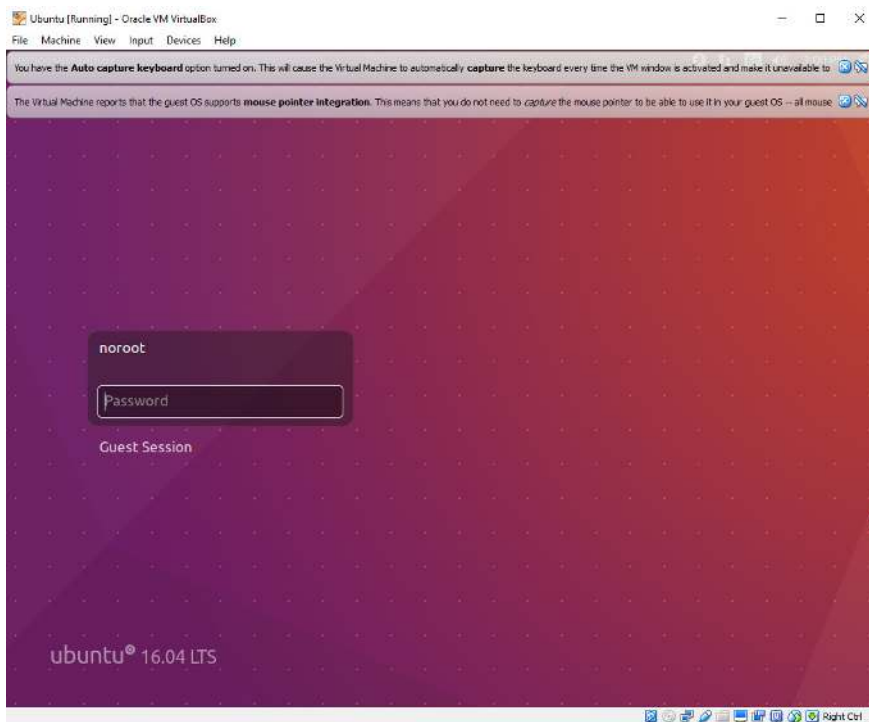


At this point it will take some time to install the operating system. When the process is finished, click restart now. If the window locks up, click Power Off The Machine and click close or next.

Part 1: Goals

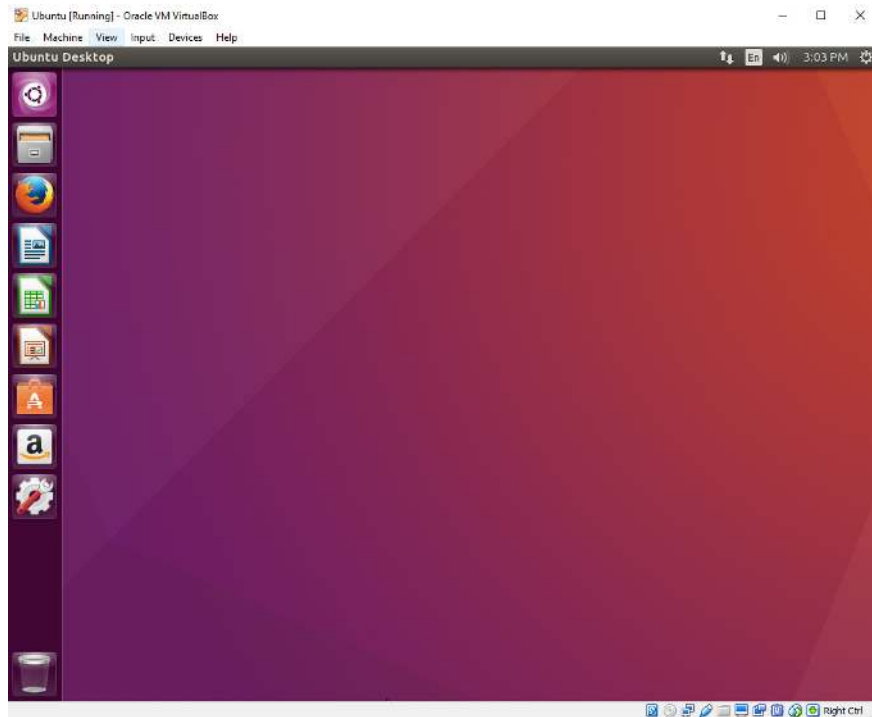


At this point, click on the green start button.

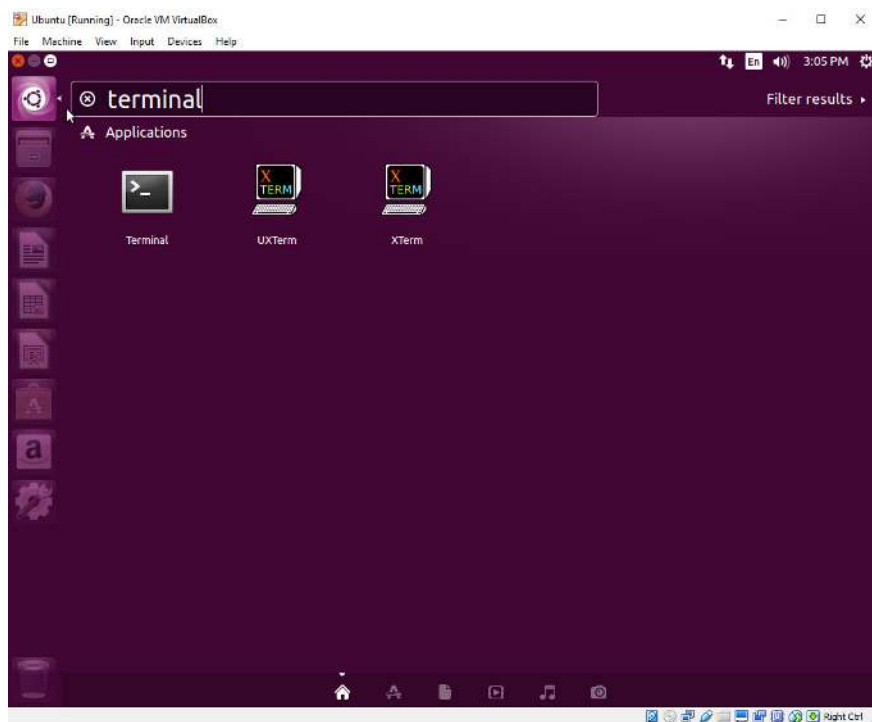


Enter in your password that you created earlier and click enter on your keyboard. You can click on the blue x buttons in the top right corner as they are just some information you can close out.

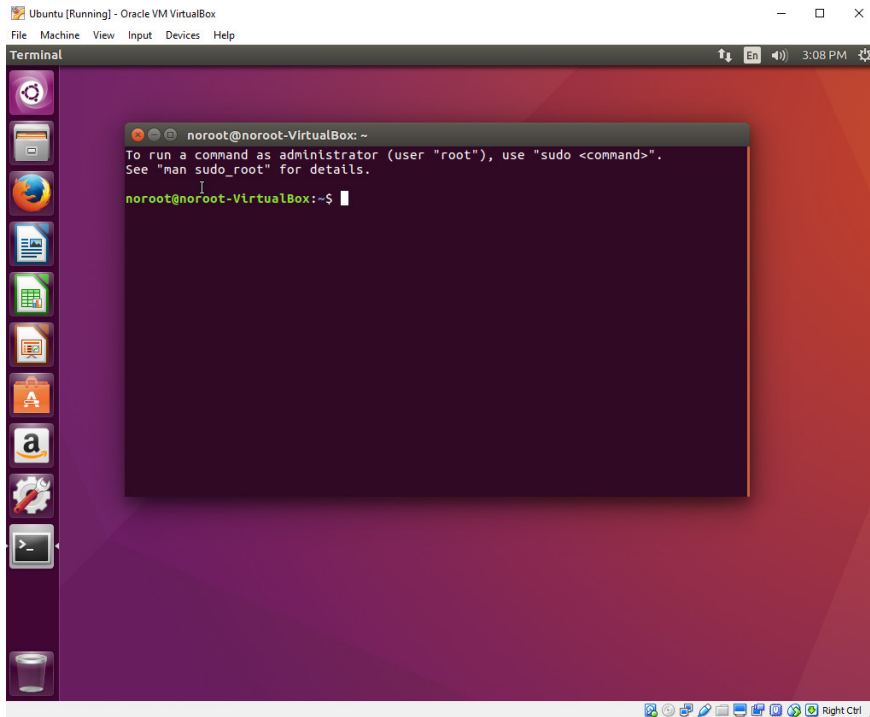
Part 1: Goals



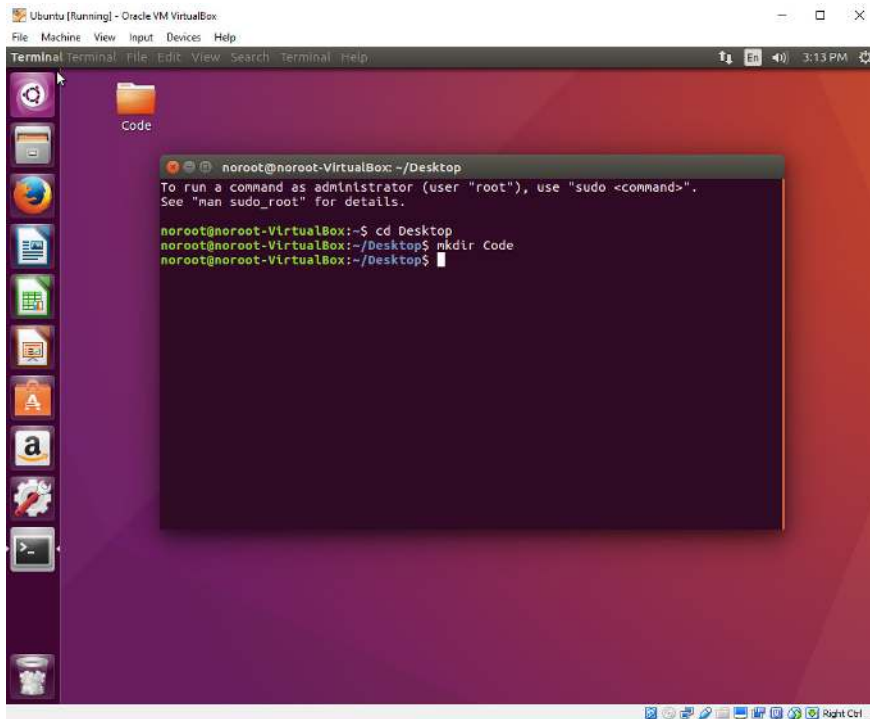
Congratulations! You have a working version of Linux!



Click on the top left icon and type terminal and double-click on the first Terminal icon with the >_ in the window.



You will see a Terminal icon at the bottom left of your screen. Right-click on it and select Lock to Launcher so that it will be available for you once you close the window.



In the terminal window type `cd Desktop` and press Enter. Then type `mkdir Code` and press enter. The first command moves you into the Desktop directory and the `mkdir` command creates a folder on the Desktop called Code so that we have a place to store our software applications that we create.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ sudo apt-get update
```

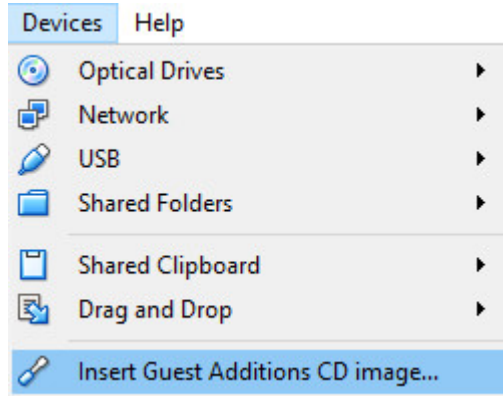
It is important you keep your version of Linux up to date. Every time you login, you should type the following commands. First, `sudo apt-get update` and press enter.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ sudo apt-get upgrade
```

Next you should then type `sudo apt-get upgrade` and press enter.

```
noroot@noroot-VirtualBox:~/Desktop$ sudo apt-get install gcc-multilib
```

In order to work with 32-bit Assembly examination, we need to install the `gcc-multilib` package so that we can compile 32-bit versions of C code for examination. Type `sudo apt-get install gcc-multilib` and press enter.



Finally click on **Devices** and click **Insert Guest Additions CD Image...** in order to get a better working functionality out of your VM.

This has been a very long tutorial however necessary to get you a working copy of Linux so that we can continue with our future tutorials.

I look forward to seeing you all next week when we learn how to use the vim text editor to begin coding!

Part 18 - vim Text Editor

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Now that we have a working version of Linux, we need a text editor that we can work with in the terminal.

To begin, open your terminal and type:

```
noroot@noroot-VirtualBox:~$ cd ~
noroot@noroot-VirtualBox:~$ vi .vimrc
```

This will open up the vi text editor. The first thing you need to type is the letter 'i' to set the editor to insert mode so you may begin typing.

```
set number
set smartindent
set tabstop=4
set shiftwidth=4
set expandtab
```

After you are done typing, press the 'esc' key and type ':wq' and press enter.

Congratulations! You created your first file! This is a one time file that we need to create in order to use our text editor the way we want it to perform.

The first line states **set number** which means we would like each file to show line numbers as this is essential for debugging code. The **set smartindent**, **set tabstop**, **set shiftwidth** and **set expandtab** statements set forth rules to properly format code and allow 4 spaces per tab indent which will help our code to look clean.

There are several commands you need to be aware of. Keep in mind, to go into command mode rather than insert mode you must press the 'esc' key. Below are the most common commands:

j or down-arrow [move cursor down one line]

k or up-arrow [move cursor up one line]

h or left-arrow [move cursor left one character]

l or right-arrow [move cursor right one character]

0 [move cursor to the start of the current line]

\$ [move cursor to the end of the current line]

b [move cursor back to the beginning of preceding word]

dd [deletes the line the cursor is on]

D [deletes from the cursor position to the end of the line]

yy [copies the current line]

p [puts the copied text after the cursor]

u [undo the last change to the file]

:w [save file]

:wq [save file and exit text editor]

:q! [quit text editor and do not save any changes]

You will be consistently moving between command mode '**esc**' and insert mode '**i**'. Remember that when you want to insert characters you need to be in insert mode and when you want to move the cursor other than moving to the next line, you need to be in command mode.

Now that we have vi configured, lets install vim which has some better functionality. Simply type:

```
noroot@noroot-VirtualBox:~/Desktop$ sudo apt-get install vim
```

Once that is installed instead of using vi we will now use vim.

I look forward to seeing you all next week when we talk about why it's important to learn Assembly Language.

Part 19 - Why Learn Assembly

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Why learn Assembly Language? Java is the most in-demand programming language and will get me a job immediately so why in the hell would I ever waste my damn time learning this archaic Assembly Language crap?

So many people ask me this question and it is true, Java is HOT and in the greatest demand and there is nothing wrong with learning Java however the threats that face society more than anything in this world, above everything else, is the Cyber Security threat. With that said, Java offers a great career path and I would encourage you to learn it however Java is not the only game in town.

Most malware is written in higher-level languages however most malware authors do not give the attackers their source code so they can properly deal with their crafted attack.

The hackers use a multitude of high-level languages and the demand for new professional Malware Analyst Reverse Engineers continue to grow daily.

When we examine malware, more than not we get only a compiled binary. The only thing we can do with a compiled binary is to break it down, instruction-by-instruction, in Assembly Language as EVERYTHING ultimately goes down to Assembly Language.

When someone says Assembly Language is a dinosaur I say to those people, lets have that conversation when your entire network is brought to its knees and you can't login to a single terminal or manipulate a single machine on your network. Lets talk about how useless Assembly Language is at that time.

Understanding Assembly Language allows one to open a debugger on an a running process. Each running program has a PID to which is a numerical value which designates a running program. If we open a running process or any bit of malware with a professional or open-source tool like GDB, we can see EXACTLY what is going on and then grab the EIP instruction pointer to go where we need it to go to have COMPLETE control over program flow.

Most malware is written, as I have stated, in a middle-level language and once compiled it can be read by the hardware or OS as it is not human-readable. In order for professional Cyber Security Engineers to understand this, they must learn to read, write and properly debug Assembly.

Assembly Language is low-level and has many more instructions than you would see in a higher-level application.

The prior 18 lessons in this tutorial series gave you the basics of x86 hardware. As I have stated in prior tutorials, we will focus on 32-bit Assembly debugging as most malware is going to try to affect as many systems as possible and although

Part 1: Goals

there is 64-bit malware, 32-bit malware is significantly more destructive and dangerous and will be the focus of this series.

I look forward to seeing you all next week when we learn the basics of instruction code handling.

Part 20 - Instruction Code Handling

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

A CPU reads instruction codes that are stored in memory as each code set can contain one or more bytes of information that guide the processor to perform a very specific task. As each instruction code is read in from memory, any data needed for the instruction code is also stored and read into memory.

Keep in mind, memory that contain instruction codes are no different than the bytes that contain the data used by the CPU and special pointers are used to help the CPU keep track of where in memory data is and where instruction codes are stored.

A data pointer helps the CPU keep track of where the data area in memory starts which is the stack. When new data elements are placed in the stack, the stack pointer moves down in memory and as data is read from the stack the stack pointer moves up in memory. Please review Part 15 – Stack if you don't understand this concept.

The instruction pointer is used to help the CPU keep track of which instruction codes have already been processed and what code is to be processed next. Please review Part 12 – Instruction Pointer Register if you don't understand this concept.

Each and every instruction code must include an opcode that defines the basic function or task to be performed by the CPU to which opcodes are between 1 and 3 bytes in length and uniquely defines the function that is performed.

Lets examine a simple C program called test.c to get started.

```
1 int main(void) {  
2     return 0;  
3 }
```

All we are doing is creating a main function of type integer to which it has a void parameter and returning 0. All this program does is simply exit the OS.

Lets compile and run this program.

```
noroot@noroot-VirtualBox:~/Desktop$ gcc -m32 -ggdb -o test test.c
```

Lets use the objdump tool to and find the main function within it.

```
noroot@noroot-VirtualBox:~/Desktop$ objdump -d -M intel test | grep main.: -A11
```

Here is a snippet of the results you would get by running the above command. Here are the contents of the main function. Keep in mind the below is in Intel syntax as we spoke about in the last tutorial.

```

080483db <main>:
80483db: 55          push    ebp
80483dc: 89 e5      mov     ebp,esp
80483de: b8 00 00 00 00  mov    eax,0x0
80483e3: 5d        pop     ebp
80483e4: c3        ret
80483e5: 66 90     xchg   ax,ax
80483e7: 66 90     xchg   ax,ax
80483e9: 66 90     xchg   ax,ax
80483eb: 66 90     xchg   ax,ax
80483ed: 66 90     xchg   ax,ax
80483ef: 90        nop

```

On the far left we have the corresponding memory addresses. In the center we have the opcodes and finally on the right we have the corresponding assembly language in Intel syntax.

To keep this simple, let's examine memory address **80483de** where we see opcodes **b8 00 00 00 00**. We can see that the **b8** opcode corresponds with the **mov eax, 0x0** instruction on the right. The next series of 00 00 00 00 represents 4 bytes of the value 0. We see **mov eax, 0x0** therefore the value of 0 is moved into `eax` therefore representing the above code. Keep in mind, the IA-32 platform uses what we call little-endian notation which means the lower-value bytes appear first in order when reading right to left.

I want to make sure you have this straight in your head so let's pretend the value above was:

mov eax, 0x1

In this scenario the corresponding opcode would be:

b8 01 00 00 00

If you are confused it is ok. Remember little-endian? Keep in mind `eax` is 32-bits wide therefore that is 4 bytes (8 bits = 1 byte). The values are listed in reverse order therefore we see the above representation.

I look forward to seeing you all next week when we dive into the details about how to compile a program.

Part 21 - How To Compile A Program

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's look again at last weeks C program and take a deeper look at how we turn that source code into an executable file.

```
1 int main(void) {
2     return 0;
3 }
```

To compile this program in C, we simply type:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -m32 -ggdb -o exit exit.c
```

This single step will create **exit.o** which is the binary object file and **exit** which is the binary executable file.

If we wanted to convert this C source code to Assembly, we need to use the GNU compiler in the below fashion. Lets start by running the below command in the terminal:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -S -m32 -O0 exit.c
```

Let's begin with the **-S** switch. The **-S** switch will create comparable AT&T Syntax Assembly source code. The **-m32** will create a 32-bit executable and the **-O0** will tell the compiler how much optimization to use when compiling the binary. That is the capital O and the numeric 0. Numeric 0 in that case means no optimization which means it is at the most human readable instruction set. If you were to substitute a 1, 2 or 3 the amount of optimization increases as the values go up.

```
1  .file   "exit.c"
2  .text
3  .globl main
4  .type   main, @function
5  main:
6  .LFB0:
7      .cfi_startproc
8      pushl   %ebp
9      .cfi_def_cfa_offset 8
10     .cfi_offset 5, -8
11     movl    %esp, %ebp
12     .cfi_def_cfa_register 5
13     movl    $0, %eax
14     popl   %ebp
15     .cfi_restore 5
16     .cfi_def_cfa 4, 4
17     ret
18     .cfi_endproc
19 .LFE0:
20 .size   main, .-main
21 .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.1) 5.4.0 20160609"
22 .section .note.GNU-stack,"",@progbits
```

This step above creates **exit.s** which is the equivalent Assembly Language source code as we mentioned above.

We then need to compile the Assembly source code into a binary object file which will generate a **exit.o** file.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -m32 -c exit.s -o exit.o
```

Finally we need to use a linker to create the actual binary executable code from the binary object file which will create an executable called exit.

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gcc -m32 exit.o -o exit
```

Last week when we examined the executable file exit in a program called objdump, and examined the main area we saw the following below except this time we will use AT&T Assembly Language Syntax:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ objdump -d exit | grep main.: -A11
```

This command above will create the following output below:

```
080483db <main>:
80483db: 55                push   %ebp
80483dc: 89 e5            mov    %esp,%ebp
80483de: b8 00 00 00 00   mov    $0x0,%eax
80483e3: 5d              pop    %ebp
80483e4: c3              ret
80483e5: 66 90           xchg  %ax,%ax
80483e7: 66 90           xchg  %ax,%ax
80483e9: 66 90           xchg  %ax,%ax
80483eb: 66 90           xchg  %ax,%ax
80483ed: 66 90           xchg  %ax,%ax
80483ef: 90              nop
```

Lets examine the code in the debugger. Let's start GDB which is the GNU debugger and first list the source code by typing l, then set a breakpoint on main and run the program. Finally we will disassemble and review the output below:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q exit
Reading symbols from exit...done.
(gdb) l
1      int main(void) {
2          return 0;
3      }
(gdb) b main
Breakpoint 1 at 0x80483de: file exit.c, line 2.
(gdb) r
Starting program: /home/noroot/Desktop/Code/exit

Breakpoint 1, main () at exit.c:2
2      return 0;
(gdb) disas
Dump of assembler code for function main:
   0x080483db <+0>:   push   %ebp
   0x080483dc <+1>:   mov    %esp,%ebp
=> 0x080483de <+3>:   mov    $0x0,%eax
   0x080483e3 <+8>:   pop    %ebp
   0x080483e4 <+9>:   ret
End of assembler dump.
```

In each of the three above examinations, you will essentially see the same set of instructions which we will take a deeper look as to what is exactly going on in future tutorials.

Throughout this tutorial series thus far we have been looking at Intel Syntax Assembly Language. We are going to turn our focus to AT&T Syntax as I have stated above as this is the natural syntax utilized in Linux with the GNU Assembler and GNU Debugger.

The biggest different you will see is that in AT&T Syntax, the source and destinations are reversed.

AT&T Syntax : **movl %esp, %ebp** [This means move esp into ebp.]

Intel Syntax : **mov esp, ebp** [This means move ebp into esp.]

You will also see some additional variances as AT&T uses additional variances which we will cover in a later tutorial.

If we wanted to create a pure Assembly Code program which does the same thing above we would type:

```
1 .section .data
2
3 .section .bss
4
5 .section .text
6     .globl _start
7
8 _start:
9     nop                    # debugging break point
10
11 exit:
12     movl $1, %eax         # sys_exit
13     movl $0, %ebx         # display 0 if normal status
14     int $0x80             # call sys_exit
```

To compile this we would use the GAS Assembler and Linker:

```
noroot@noroot-VirtualBox:~/Desktop/Code$ as --32 -gstabs -o exit__s.o exit__s.s
noroot@noroot-VirtualBox:~/Desktop/Code$ ld -m elf_i386 -o exit__s exit__s.o
noroot@noroot-VirtualBox:~/Desktop/Code$ ./exit
```

To run any executable in Linux you type `./` and the name of the binary executable.

In this case we type `./exit` and press return. When we do so, nothing happens.

That is good as all we did was create a program that exited to the OS.

I look forward to seeing you all next week when we dive into more assembly code!

Part 22 - ASM Program 1 [Moving Immediate Data]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

I appreciate everyone being patient as it has taken 21 lessons to get to our first ASM program however very necessary background had to be covered in order to fully understand where we begin when developing assembly language.

We are going to create 32-bit assembly programs as most malware is written in 32-bit mode in order to attack the maximum amount of systems possible. Keep in mind even though most of us ALL have 64-bit operating systems, 32-bit programs can run on them.

For the most part we have been working with Intel syntax when it comes to assembly however I am going to focus on the native AT&T syntax going forward. It is very easy to convert back and forth between Intel and AT&T syntax as I have demonstrated in prior tutorials.

Every assembly language program is divided into three sections:

1)**Data Section:** This section is used for declaring initialized data or constants as this data does not ever change at runtime. You can declare constant values, buffer sizes, file names, etc.

2)**BSS Section:** This section is used for declaring uninitialized data or variables.

3)**Text Section:** This section is used for the actual code sections as it begins with a global `_start` which tells the kernel where execution begins.

Critical to any development is the use of comments. In the AT&T syntax we use the `#` symbol to declare a comment as any data after that symbol on a respective line will be ignored by the compiler.

Keep in mind, assembly language statements are entered in one statement per line as you do not have to end the line with a semicolon like many other languages. The structure of a statement is as follows:

[label] mnemonic [operands] [comment]

A basic instruction has two parts of which the first one is the name of the instruction or the mnemonic which is executed and the second part is the operands or parameters of the command.

Our first program will demonstrate how to move immediate data to a register and immediate data to memory.

Lets open VIM and create a program called **moving_immediate_data.s** and type the following:

```

1 #moving_immediate_data:  mov immediate data between registers & memory
2
3
4 .section .data
5
6 .section .bss
7     .lcomm buffer 1
8
9 .section .text
10    .globl _start
11
12 _start:
13     nop                                #used for debugging purposes
14
15 mov_immediate_data_to_register:
16     movl $100, %eax                    #mov 100 into the EAX register
17     movl $0x50, buffer                #mov 0x50 into buffer memory location
18
19 exit:
20     movl $1, %eax                      #sys_exit system call
21     movl $0, %ebx                      #exit code 0 successful execution
22     int $0x80                          #call sys_exit

```

To compile you type:

```
as -32 -o moving_immediate_data.o moving_immediate_data.s
```

```
ld -m elf_i386 -o moving_immediate_data moving_immediate_data.o
```

To run you type:

```
./moving_immediate_data
```

I would like to show you what it would look like in Intel syntax as well. Before we examine this part you will need to type **sudo apt-get install nasm** in a command prompt which will install the Netwide Assembler:

```

1 ;moving_immediate_data:  mov immediate data between registers & memory
2
3
4 section .data
5
6 section .bss
7     buffer resb 1
8
9 section .text
10    global _start
11
12 _start:
13     nop                                ;used for debugging purposes
14
15 mov_immediate_data_to_register:
16     mov eax, 100                        ;mov 100 into the EAX register
17     mov byte[buffer], 0x50             ;mov 0x50 into buffer memory location
18
19 exit:
20     mov eax, 1                          ;sys_exit system call
21     mov ebx, 0                          ;exit code 0 successful execution
22     int 0x80                            ;call sys_exit

```

To compile you type:

```
nasm -f elf32 moving_immediate_data.asm
```

```
ld -m elf_i386 -o moving_immediate_data moving_immediate_data.o
```

To run you type:

```
./moving_immediate_data
```

Ok what the heck! There is no output! That is correct and you did not do anything wrong. Many of our programs will not actually do anything as they are not much more than sandbox programs that we will use in GDB for analysis and manipulation.

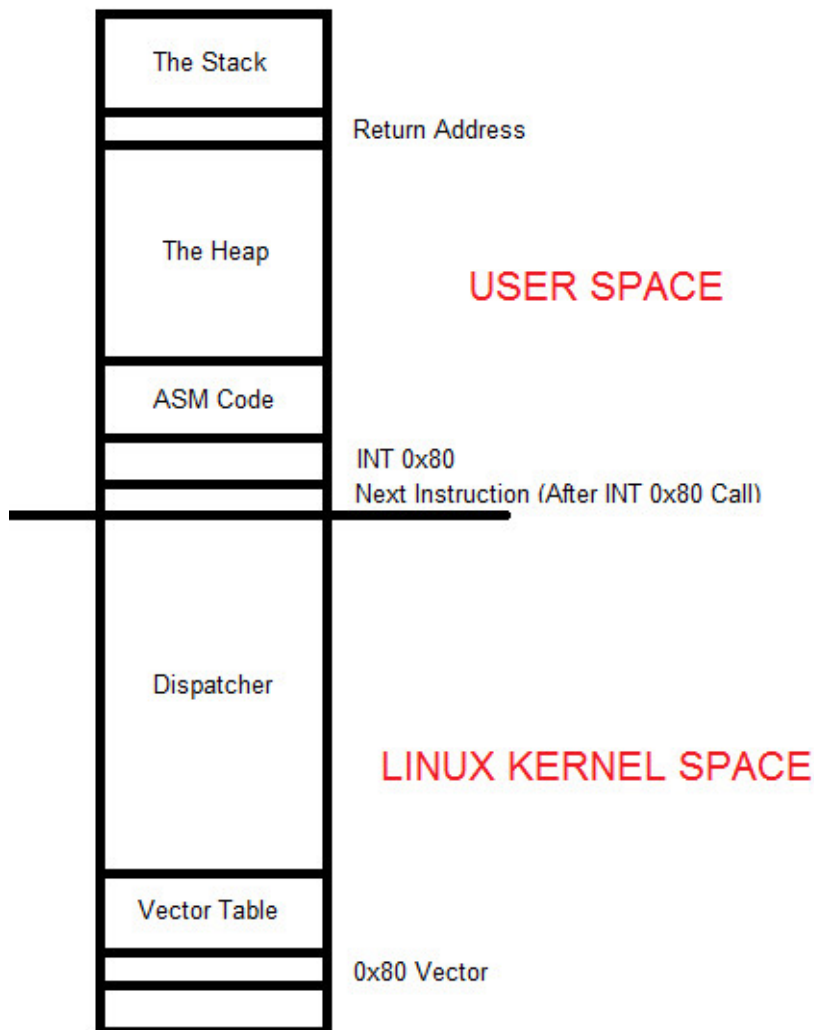
Next week we will dive into the GNU GDB debugger and see what is going on under the hood.

I want to take some time and discuss the code at line 20 – 22 in the AT&T version and the Intel Syntax version as well. This set of instructions takes advantage of what we call a software interrupt. On line 20 in the AT&T Syntax, we **movl \$1, %eax** meaning we move the decimal value of 1 into eax which specifies the `sys_exit` call which will properly terminate program execution back to Linux so that there is no segmentation fault. On line 21, we **movl \$0, %ebx** which moves 0 into ebx to show that the program successfully executed and finally we see **int \$0x80**.

Line 20 and 21 set up the software interrupt which we call on line 22 with the instruction **int \$0x80**. Let's dive into this a little deeper.

In Linux, there are two distinct areas of memory. At the very bottom of memory in any program execution we have the Kernel Space which is made up of the Dispatcher section and the Vector Table.

At the very top of memory in any program execution we have the User Space which is made up of The Stack, The Heap and finally your code all of which can be illustrated in the below diagram:



When we load the values as we demonstrated above and call INT 0x80, the very next instruction's address in the User Space, ASM Code section which is your code, is placed into the Return Address area in The Stack. This is critical so that when INT 0x80 does its work, it can properly know what instruction is to be carried out next to ensure proper and sequential program execution.

Keep in mind in modern versions of Linux, we are utilizing Protected Mode which means you do NOT have access to the Linux Kernel Space. Everything under the long line that runs in the middle of the diagram above represents the Linux Kernel Space.

The natural question is why can't we access this? The answer is very simple, Linux will NOT allow your code to access operating system internals as that would be very dangerous as any Malware could manipulate those components of the OS to track all sorts of things such as user keystrokes, activities and the like.

In addition, modern Linux OS architecture changes the address of these key components constantly as new software is installed and removed in addition to system patches and upgrades. This is the cornerstone of Protected Mode operating systems.

The way that we have our code communicate with the Linux Kernel is through the use of a kernel services call gate which is a protected gateway between User Space where your program is running and Kernel Space which is implemented through the Linux Software Interrupt of 0x80.

At the very, very bottom of memory where segment 0, offset 0 exists is a lookup table with 256 entries. Every entry is a memory address including segment and offset portions which comprise of 4 bytes per entry as the first 1,024 bytes are reserved for this table and NO OTHER CODE can be manipulated there. Each address is called an interrupt vector which comprises the whole called the interrupt vector table where every vector has a number from 0 to 255 to which vector 0 starts off occupying bytes 0 to 3. This continues with vector 1 which contains 4 to 7, etc.

Keep in mind, none of these addresses are part of permanent memory. What is static is vector 0x80 which points to the services dispatcher which point to Linux kernel service routines.

When the return address is popped off the stack returns to the next instruction, the instruction is called the Interrupt Return or IRET which completes the execution of program flow.

Take some time and look at the entire table of system calls by opening up a terminal and typing:

```
cat /usr/include/asm/unistd_32.h
```

Below is a snapshot of just a few of them. As you can see the exit 1 represents the sys_exit that we utilized in our above code.

```
noroot@noroot-VirtualBox:/usr/include/asm$ cat /usr/include/asm/unistd_32.h
#ifdef _ASM_X86_UNISTD_32_H
#define _ASM_X86_UNISTD_32_H 1

#define __NR_restart_syscall 0
#define __NR_exit 1
#define __NR_fork 2
#define __NR_read 3
#define __NR_write 4
```

Starting with this lesson we will take a 3-step approach:

1)Program

2)Debug

3)Hack

Each week we will start with a program like you see here, the following week we will take it into GDB and examine what exactly is going on at the assembly level and finally in the third series of each week we will hack the data in GDB to change it to whatever we want demonstrating the ability to control program flow which includes learning how to hack malware to a point where it is not a threat.

We will not necessarily look at malware directly as I would rather focus on the topics of assembly language programs that will give you the tools and understanding so that ANY program can be debugged and manipulated to your liking. That is the purpose of these tutorials.

The information you will learn in this tutorial series can be used with high-level GUI debuggers like IDA Pro as well however I will focus only on the GNU GDB debugger.

I look forward to seeing you all next week when we dive into creating our first assembly debug!

Part 23 - ASM Debugging 1 [Moving Immediate Data]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's begin by loading the binary into GDB.

To load into GDB type:

gdb -q moving_immediate_dat

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_immediate_data
Reading symbols from moving_immediate_data...(no debugging symbols found)...done
(gdb) b _start
Breakpoint 1 at 0x08048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_immediate_data

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:    nop
End of assembler dump.

```

Let's first set a breakpoint on start by typing **b _start**.

We can then run the program by typing **r**.

To then begin disassembly, we simply type **disas**.

We coded a **nop** which means no operation or 0x90 from an OPCODE perspective for proper debugging purposes which the breakpoint properly hit. This is good practice when creating assembly programs.

```

(gdb) si
0x08048075 in mov_immediate_data_to_register ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_to_register:
=> 0x08048075 <+0>:    mov     $0x64,%eax
   0x0804807a <+5>:    movl   $0x50,0x8049090
End of assembler dump.

```

The native syntax as I have stated many times before is AT&T syntax which you see above. I painfully go back and forth deliberately so that you have comfort in each however going forward I will be sticking to the AT&T syntax however wanted to show you a few examples of both. I will state again that if you ever want to see Intel syntax simply type **set-disassembly-flavor intel** and you will have what you are looking for.

We first use the command **si** which means step-into to advance to the next instruction. What we see here at **_start+0** is you are moving the hex value of **0x64** into **EAX**. This is simply moving decimal **100** or as the computer sees it, hex **0x64** into **EAX** which demonstrates moving an immediate value into a register.

```
(gdb) si
0x0804807a in mov_immediate_data_to_register ()
(gdb) i r
eax          0x64      100
ecx          0x0       0
edx          0x0       0
ebx          0x0       0
esp          0xffffd040 0xffffd040
ebp          0x0       0x0
esi          0x0       0
edi          0x0       0
eip          0x0804807a 0x0804807a <mov_immediate_data_to_register+5>
eflags      0x202     [ IF ]
cs          0x23      35
ss          0x2b      43
ds          0x2b      43
es          0x2b      43
fs          0x0       0
gs          0x0       0
```

We step-into again and then use the command `i r` which keep in mind has a space between them to give us information on the state of the CPU registers. We can see EAX now has the value of 0x64 hex or 100 decimal.

```
(gdb) si
0x08048084 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x08048084 <+0>:      mov     $0x1,%eax
      0x08048089 <+5>:      mov     $0x0,%ebx
      0x0804808e <+10>:     int     $0x80
End of assembler dump.
(gdb) print /x buffer
$1 = 0x50
```

After we step-into again and do a `disas`, we see that we have then moved the value of **0x50** into the **buffer** label as can refer back to the source code from last week to see.

When dealing with non-register data, we can use the print command above as we type `print /x buffer` and it clearly shows us that the value inside buffer is **0x50**. The `/x` designation means show us the value in hex.

```
(gdb) x/xb 0x8049090
0x8049090 <buffer>: 0x50
```

Consequently you can review slide 2 of this tutorial above you see at **_start+5** the immediate value of **0x50** loaded into the **buffer** label or in this case the address of **buffer** which is **0x8049090** and we can examine it by using the examine instruction by typing `x/xb 0x8049090` which shows us one hex byte at that location which yields **0x50**.

We will be doing this with every program example so that we can dive into the debugging process. If there are any questions, please leave them below in the comments.

I look forward to seeing you all next week when we dive into creating our first assembly hack!

Part 24 - ASM Hacking 1 [Moving Immediate Data]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's begin by loading the binary into GDB.

To load into GDB type:

gdb -q moving_immediate_data

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_immediate_data
Reading symbols from moving_immediate_data...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x08048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_immediate_data

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:    nop
End of assembler dump.

```

Let's first set a breakpoint on start by typing **b _start**.

We can then run the program by typing **r**.

To then begin disassembly, we simply type **disas**.

We coded a **nop** which means no operation or **0x90** from an OPCODE perspective for proper debugging purposes which the breakpoint properly hit. This is good practice when creating assembly programs.

```

(gdb) si
0x08048075 in mov_immediate_data_to_register ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_to_register:
=> 0x08048075 <+0>:    mov     $0x64,%eax
   0x0804807a <+5>:    movl   $0x50,0x8049090
End of assembler dump.

```

Lets have some fun! At this point lets **si** once and do an **i r** to see that **0x64** has in fact been moved into **EAX**.

```
(gdb) si
0x08048075 in mov_immediate_data_to_register ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_to_register:
=> 0x08048075 <+0>:    mov     $0x64,%eax
       0x0804807a <+5>:    movl   $0x50,0x8049090
End of assembler dump.
(gdb) si
0x0804807a in mov_immediate_data_to_register ()
(gdb) i r
eax            0x64      100
ecx            0x0       0
edx            0x0       0
ebx            0x0       0
esp            0xffffd040    0xffffd040
ebp            0x0       0x0
esi            0x0       0
edi            0x0       0
eip            0x804807a    0x804807a <mov_immediate_data_to_register+5>
eflags        0x202     [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0       0
gs             0x0       0
(gdb)
```

We can see **EAX** has the value of **0x64** or **100** decimal. Lets HACK that value now by setting **EAX** to say something like **0x66** by typing **set \$eax = 0x66**.

```
(gdb) set $eax = 0x66
(gdb) i r
eax            0x66      102
ecx            0x0       0
edx            0x0       0
ebx            0x0       0
esp            0xffffd040    0xffffd040
ebp            0x0       0x0
esi            0x0       0
edi            0x0       0
eip            0x804807a    0x804807a <mov_immediate_data_to_register+5>
eflags        0x202     [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0       0
gs             0x0       0
(gdb)
```

BAM! There we go! You can see the **ULTIMATE** power of assembly here! We just hacked the value from **0x64** to **0x66** or **100** to **102** decimal. This is a trivial example however you can clearly see when you learn to master these concepts you develop a greater power over the computer. With each program that we create, we will have a very simple lesson like this where we will hijack at least one portion of the code so we can not only see how the program is created and debugged but how we can manipulate it to whatever we want.

I look forward to seeing you all next week when we dive into creating our second assembly program!

Part 25 - ASM Program 2 [Moving Data Between Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our second program we will demonstrate how we can move data between registers. Moving data from one register to another is the fastest way to move data. It is always advisable to keep data between registers as much as can be engineered for speed.

Specifically we will move the value in EDX into EAX. We will initialize this program with a simple immediate value of 22 decimal which will go into EDX and ultimately into EAX.

```
1 #moving_data_between_registers:  mov data between registers
2
3
4 .section .data
5
6 .section .text
7     .globl _start
8
9 _start:
10     nop                                #used for debugging purposes
11
12     movl $22, %edx                      #mov immediate value into EDX
13
14 mov_data_between_registers:
15     movl %edx, %eax                    #mov the value in EDX into EAX
16
17 exit:
18     movl $1, %eax                      #sys_exit system call
19     movl $0, %ebx                      #exit code 0 successful execution
20     int $0x80                          #call sys_exit
```

Keep in mind you can only move similar registers between each other. We know that EAX and EDX are 32-bit registers. We know that each of these registers can be accessed by their 16-bit values as ax and dx respectively. You can't move a 32-bit value into a 16-bit value and vice-versa.

I look forward to seeing you all next week when we dive into debugging our second assembly program!

Part 26 - ASM Debugging 2 [Moving Data Between Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's debug the second program below:

```

1 #moving_data_between_registers:  mov data between registers
2
3
4 .section .data
5
6 .section .text
7     .globl _start
8
9 _start:
10     nop                                #used for debugging purposes
11
12     movl $22, %edx                    #mov immediate value into EDX
13
14 mov_data_between_registers:
15     movl %edx, %eax                  #mov the value in EDX into EAX
16
17 exit:
18     movl $1, %eax                    #sys_exit system call
19     movl $0, %ebx                    #exit code 0 successful execution
20     int $0x80                        #call sys_exit

```

Lets fire up GDB and break on `_start`, run the binary and disas:

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers
Reading symbols from moving_data_between_registers...(no debugging symbols found)
)..done.
(gdb) b _start
Breakpoint 1 at 0x08048054
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers

Breakpoint 1, 0x08048054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048054 <+0>:    nop
=> 0x08048055 <+1>:    mov     $0x16,%edx
End of assembler dump.
(gdb)
Dump of assembler code for function _start:
=> 0x08048054 <+0>:    nop
=> 0x08048055 <+1>:    mov     $0x16,%edx

```

Now lets `si` twice and `i r`:

```

(gdb) i r
eax             0x0             0
ecx             0x0             0
edx             0x16            22
ebx             0x0             0
esp             0xffffd030        0xffffd030
ebp             0x0             0x0
esi             0x0             0
edi             0x0             0
eip             0x804805a        0x804805a <mov_data_between_registers>
eflags         0x202            [ IF ]
cs              0x23            35
ss              0x2b            43
ds              0x2b            43
es              0x2b            43
fs              0x0             0
gs              0x0             0
(gdb)

```

As we can see the value of `0x16` or `22` decimal did move into EDX successfully. Now lets `si` again.

```
(gdb) i r
eax          0x16      22
ecx          0x0       0
edx          0x16      22
ebx          0x0       0
esp          0xffffd030   0xffffd030
ebp          0x0       0x0
esi          0x0       0
edi          0x0       0
eip          0x804805c   0x804805c <exit>
eflags      0x202      [ IF ]
cs          0x23      35
ss          0x2b      43
ds          0x2b      43
es          0x2b      43
fs          0x0       0
gs          0x0       0
(gdb)
```

As you can see we have successfully moved EDX into EAX.

I look forward to seeing you all next week when we dive into hacking our second assembly program!

Part 27 - ASM Hacking 2 [Moving Data Between Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's hack the second program below:

```

1 #moving_data_between_registers: mov data between registers
2
3
4 .section .data
5
6 .section .text
7     .globl _start
8
9 _start:
10     nop                                #used for debugging purposes
11
12     movl $22, %edx                      #mov immediate value into EDX
13
14 mov_data_between_registers:
15     movl %edx, %eax                     #mov the value in EDX into EAX
16
17 exit:
18     movl $1, %eax                       #sys_exit system call
19     movl $0, %ebx                       #exit code 0 successful execution
20     int $0x80                           #call sys_exit

```

Lets fire up GDB and break on `_start`, run the binary and disas:

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers
Reading symbols from moving_data_between_registers...(no debugging symbols found)
...done.
(gdb) b _start
Breakpoint 1 at 0x08048054
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers

Breakpoint 1, 0x08048054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048054 <+0>:    nop
=> 0x08048055 <+1>:    mov     $0x16,%edx
End of assembler dump.
(gdb)
Dump of assembler code for function _start:
=> 0x08048054 <+0>:    nop
=> 0x08048055 <+1>:    mov     $0x16,%edx

```

Now lets `si` twice and `i r`:

```

(gdb) si
0x08048055 in _start ()
(gdb) si
0x0804805a in mov_data_between_registers ()
(gdb) i r
eax                0x0          0
ecx                0x0          0
edx                0x16         22
ebx                0x0          0
esp                0xffffd030   0xffffd030
ebp                0x0          0x0
esi                0x0          0
edi                0x0          0
eip                0x804805a   0x804805a <mov_data_between_registers>
eflags             0x202       [ IF ]
cs                 0x23         35
ss                 0x2b         43
ds                 0x2b         43
es                 0x2b         43
fs                 0x0          0
gs                 0x0          0
(gdb)

```

As we can see the value of **0x16** or **22** decimal did move into EDX successfully. This is what we did in the last lesson however here we are going to hack that value to something else.

We can set **\$edx = 0x19** for example:

```
(gdb) set $edx = 0x19
(gdb) i r
eax          0x0      0
ecx          0x0      0
edx          0x19     25
ebx          0x0      0
esp          0xffffd030 0xffffd030
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x804805a 0x804805a <mov_data_between_registers>
eflags      0x202    [ IF ]
cs           0x23     35
ss           0x2b     43
ds           0x2b     43
es           0x2b     43
fs           0x0      0
gs           0x0      0
(gdb)
```

As you can see we easily hacked the value of **EDX** to **0x19** or **25** decimal.

Hopefully you see some very simple patterns now that we are diving into very simple assembly language programs. The key is to understand how to manipulate values and instructions so that you have complete control over the binary.

We are going to continue to move at a snails pace throughout the rest of this tutorial as my goal is to give everyone very small bite-size examples of how to understand x86 assembly.

I look forward to seeing you all next week when we dive into writing our third assembly program!

Part 28 - ASM Program 3 [Moving Data Between Memory And Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our third program we will demonstrate how we can move data between memory and registers.

```

1 #moving_data_between_memory_and_registers:  mov data between mem and regs
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                #used for debugging purposes
13
14 mov_immediate_data_between_memory_and_registers:
15     movl constant, %ecx    #mov constant value into EAX register
16
17 exit:
18     movl $1, %eax        #sys_exit system call
19     movl $0, %ebx        #exit code 0 successful execution
20     int $0x80            #call sys_exit

```

Specifically we will move the value of inside the constant integer of 10 decimal into ECX.

Keep in mind to assemble we type:

```
as -32 -o moving_data_between_memory_and_registers.o
moving_data_between_memory_and_registers.s
```

To link the object file we type:

```
ld -m elf_i386 -o moving_data_between_memory_and_registers
moving_data_between_memory_and_registers.o
```

I look forward to seeing you all next week when we dive into debugging our third assembly program!

Part 29 - ASM Debugging 3 [Moving Data Between Memory And Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's debug!

```

1 #moving_data_between_memory_and_registers:  mov data between mem and regs
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                                #used for debugging purposes
13
14 mov_immediate_data_between_memory_and_registers:
15     movl constant, %ecx                #mov constant value into EAX register
16
17 exit:
18     movl $1, %eax                       #sys_exit system call
19     movl $0, %ebx                       #exit code 0 successful execution
20     int $0x80                           #call sys_exit

```

Specifically we will move the value of inside the constant integer of 10 decimal into ECX.

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_memory_and_registers
Reading symbols from moving_data_between_memory_and_registers...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_memory_and_registers
Breakpoint 1, 0x08048074 in _start ()

```

We open GDB in quiet mode and break on `_start` and run by following the commands above.

```

(gdb) i r
eax                0x0          0
ecx                0x0          0
edx                0x0          0
ebx                0x0          0
esp                0xffffd020   0xffffd020
ebp                0x0          0x0
esi                0x0          0
edi                0x0          0
eip                0x8048074   0x8048074 <_start>
eflags            0x202       [ IF ]
cs                 0x23        35
ss                 0x2b        43
ds                 0x2b        43
es                 0x2b        43
fs                 0x0          0
gs                 0x0          0
(gdb)

```

As we can see when we info registers the value of ECX is 0.

```
(gdb) si
0x08048075 in mov_immediate_data_between_memory_and_registers ()
(gdb) si
0x0804807b in exit ()
(gdb) i r
eax          0x0      0
ecx          0xa     10
edx          0x0     0
ebx          0x0     0
esp          0xffffd020 0xffffd020
ebp          0x0     0x0
esi          0x0     0
edi          0x0     0
eip          0x804807b 0x804807b <exit>
eflags      0x202   [ IF ]
cs          0x23    35
ss          0x2b    43
ds          0x2b    43
es          0x2b    43
fs          0x0     0
gs          0x0     0
(gdb)
```

After we step into twice, we now see the value of ECX as 10 decimal of 0xa hex.

I look forward to seeing you all next week when we dive into hacking our third assembly program!

Part 30 - ASM Hacking 3 [Moving Data Between Memory And Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's hack!

```

1 #moving_data_between_memory_and_registers:  mov data between mem and regs
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                                #used for debugging purposes
13
14 mov_immediate_data_between_memory_and_registers:
15     movl constant, %ecx                #mov constant value into EAX register
16
17 exit:
18     movl $1, %eax                       #sys_exit system call
19     movl $0, %ebx                       #exit code 0 successful execution
20     int $0x80                           #call sys_exit

```

Specifically we will move the value of inside the constant integer of 10 decimal into ECX like before.

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_memory_and_registers
Reading symbols from moving_data_between_memory_and_registers...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_memory_and_registers
Breakpoint 1, 0x08048074 in _start ()

```

We open GDB in quiet mode and break on `_start` and run by following the commands above.

```

(gdb) i r
eax                0x0          0
ecx                0x0          0
edx                0x0          0
ebx                0x0          0
esp                0xffffd020   0xffffd020
ebp                0x0          0x0
esi                0x0          0
edi                0x0          0
eip                0x8048074   0x8048074 <_start>
eflags            0x202       [ IF ]
cs                 0x23       35
ss                 0x2b       43
ds                 0x2b       43
es                 0x2b       43
fs                 0x0          0
gs                 0x0          0
(gdb)

```


As we can see when we info registers the value of ECX is 0. Let's do a si and another si.

```
(gdb) si
0x08048075 in mov_immediate_data_between_memory_and_registers ()
(gdb) disas
Dump of assembler code for function mov_immediate_data_between_memory_and_registers:
=> 0x08048075 <+0>:   mov     0x8049087,%ecx
End of assembler dump.
(gdb) si
0x0804807b in exit ()
(gdb) i r
eax          0x0      0
ecx          0xa     10
edx          0x0      0
ebx          0x0      0
esp          0xffffd020  0xffffd020
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x804807b  0x804807b <exit>
eflags      0x202    [ IF ]
cs          0x23     35
ss          0x2b     43
ds          0x2b     43
es          0x2b     43
fs          0x0      0
gs          0x0      0
```

As you can see the value of ECX is 10 decimal or 0xa hex as it was in the prior lesson now lets hack that value to something else.

Let's **set \$ecx = 1337** and do an i r.

```
(gdb) set $ecx = 1337
(gdb) i r
eax          0x0      0
ecx          0x539    1337
edx          0x0      0
ebx          0x0      0
esp          0xffffd020  0xffffd020
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x804807b  0x804807b <exit>
eflags      0x202    [ IF ]
cs          0x23     35
ss          0x2b     43
ds          0x2b     43
es          0x2b     43
fs          0x0      0
gs          0x0      0
(gdb)
```

As you can clearly see we have hacked the value of ECX to 0x539 hex or 1337 decimal.

As I have stated throughout this series. Each of these lessons are very bite-sized examples so that you get the hard muscle memory on how to hack through a variety of situations so that you ultimately have a complete mastery of processor control.

I look forward to seeing you all next week when we dive into creating our fourth assembly program!

Part 31 - ASM Program 4 [Moving Data Between Registers And Memory]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our fourth program we will demonstrate how we can move data between registers and memory.

```

1 #moving_data_between_registers_and_memory: mov data between regs and mem
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                                #used for debugging purposes
13
14 mov_immediate_data_between_registers_and_memory:
15     movl $777, %eax                    #mov immediate value 777 to eax
16     movl %eax, constant                #mov eax value into constant mem value
17
18 exit:
19     movl $1, %eax                      #sys_exit system call
20     movl $0, %ebx                      #exit code 0 successful execution
21     int $0x80                          #call sys_exit

```

Specifically we will move the immediate value of 777 decimal into EAX. We then move that value stored in EAX into the constant value in memory which initially had the value of 10 decimal at runtime. Keep in mind we could have called the value anything however I called it constant as it was set up as a constant in the .data section.

You can clearly see it can be manipulated so it is NOT a constant. I chose constant deliberately as if it was in pure form the value would stay 10 decimal or 0xa hex.

This code is purely an academic exercise as variable data normally would be set up under the .bss section however I wanted to demonstrate that the above is possible to show the absolute flexibility of assembly language.

Keep in mind to assemble we type:

```
as -32 -o moving_data_between_registers_and_memory.o
moving_data_between_registers_and_memory.s
```

To link the object file we type:

```
ld -m elf_i386 -o moving_data_between_registers_and_memory
moving_data_between_registers_and_memory.o
```

I look forward to seeing you all next week when we dive into debugging our fourth assembly program!

Part 32 - ASM Debugging 4 [Moving Data Between Registers And Memory]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our fourth program we will demonstrate how we can move data between registers and memory.

```

1 #moving_data_between_registers_and_memory: mov data between regs and mem
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                #used for debugging purposes
13
14 mov_immediate_data_between_registers_and_memory:
15     movl $777, %eax    #mov immediate value 777 to eax
16     movl %eax, constant #mov eax value into constant mem value
17
18 exit:
19     movl $1, %eax     #sys_exit system call
20     movl $0, %ebx     #exit code 0 successful execution
21     int $0x80         #call sys_exit

```

Specifically we will move the immediate value of 777 decimal into EAX. We then move that value stored in EAX into the constant value in memory which initially had the value of 10 decimal at runtime. Keep in mind we could have called the value anything however I called it constant as it was set up as a constant in the .data section.

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers_and_memory
Reading symbols from moving_data_between_registers_and_memory...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers_and_memory
Breakpoint 1, 0x08048074 in _start ()
(gdb) si
0x08048075 in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807a in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807f in exit ()
(gdb) print constant
$1 = 777
(gdb)

```

As you can see above we go into GDB and clearly see that the value of constant has been replaced with 777 decimal where in the code it was clearly set at 10 decimal in line 6 of the code at the beginning of this tutorial.

We can clearly see that in line 16 of the code the value of 777 decimal was successfully moved into EAX and into the memory value of constant.

I look forward to seeing you all next week when we dive into hacking our fourth assembly program!

Part 33 - ASM Hacking 4 [Moving Data Between Registers And Memory]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's re-examine the source code.

```

1 #moving_data_between_registers_and_memory: mov data between regs and mem
2
3
4 .section .data
5     constant:
6         .int 10
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                #used for debugging purposes
13
14 mov_immediate_data_between_registers_and_memory:
15     movl $777, %eax    #mov immediate value 777 to eax
16     movl %eax, constant #mov eax value into constant mem value
17
18 exit:
19     movl $1, %eax      #sys_exit system call
20     movl $0, %ebx      #exit code 0 successful execution
21     int $0x80          #call sys_exit

```

We again can see above that we will move the immediate value of 777 decimal into EAX. We then move that value stored in EAX into the constant value in memory which initially had the value of 10 decimal at runtime. Keep in mind we could have called the value anything however I called it constant as it was set up as a constant in the .data section.

```

noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers_and_memory
Reading symbols from moving_data_between_registers_and_memory...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers_and_memory

Breakpoint 1, 0x08048074 in _start ()
(gdb) si
0x08048075 in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807a in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807f in exit ()
(gdb) print constant
$1 = 777
(gdb)

```

As you can see above we go into GDB and clearly see that the value of constant has been replaced with 777 decimal where in the code it was clearly set at 10 decimal in line 6 of the code at the beginning of this tutorial.

We can clearly see that in line 16 of the code the value of 777 decimal was successfully moved into EAX and into the memory value of constant.

Now lets hack this thing!

Part 1: Goals

```
noroot@noroot-VirtualBox:~/Desktop/Code$ gdb -q moving_data_between_registers_and_memory
Reading symbols from moving_data_between_registers_and_memory...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/moving_data_between_registers_and_memory

Breakpoint 1, 0x08048074 in _start ()
(gdb) si
0x08048075 in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807a in mov_immediate_data_between_registers_and_memory ()
(gdb) si
0x0804807f in exit ()
(gdb) print constant
$1 = 777
(gdb) set constant = 666
(gdb) print constant
```

We took the very steps as we did last time with the debugging lesson. Here we hack the value of constant to which we hack the value from 777 to 666.

I look forward to seeing you all next week when we dive into creating our fifth assembly program!

Part 34 - ASM Program 5 [Indirect Addressing With Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our fifth program we will demonstrate how we can manipulate indirect addressing with registers.

```

1 #indirect_addressing_with_registers:  accessing data in mem using pointers
2
3
4 .section .data
5     constants:
6         .int 5, 8, 17, 44, 50, 52, 60, 65, 70, 77, 80
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                #used for debugging purposes
13
14 indirect_addressing_with_registers:
15     movl constants, %eax    #mov constants mem value into eax
16     movl $constants, %edi   #mov mem addr into edi
17     movl $25, 4(%edi)      #mov immediate val 4b after edi ptr
18     movl $1, %edi          #load 2nd index constants label
19     movl constants(, %edi, 4), %ebx #mov above value 4 bytes from constants
20
21 exit:
22     movl $1, %eax          #sys_exit system call
23     movl $0, %ebx          #exit code 0 successful execution
24     int $0x80              #call sys_exit

```

We can place more than one value in memory as indicated above. In the past, our memory location contained one single value. In the above as you can see the value of constants contains 11 separate values.

This creates a sequential series of data values placed in memory. Each data value occupies one unit of memory which is an integer or 4 bytes.

We must use an index system to determine these values as what we have created above is an array.

We will utilize the indexed memory mode where the memory address is determined by a base address, an offset address to add to the base address and the size of the data element, in our case an integer of 4 bytes and an index to determine which data element to select.

Keep in mind an array starts with index 0. Therefore in the above code we see 1 moving into edi which is the 2nd index which ultimately goes into ebx.

We will dive deeper into this in the next lesson we debug however I want you to take some time to study the code above and get a good grasp of what is going on.

Keep in mind to assemble we type:

```

as -32 -o indirect_addressing_with_registers.o
indirect_addressing_with_registers.s

```

To link the object file we type:

```
ld -m elf_i386 -o indirect_addressing_with_registers  
indirect_addressing_with_registers.o
```

I look forward to seeing you all next week when we dive into debugging our fifth assembly program!

Part 35 - ASM Debugging 5 [Indirect Addressing With Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our fifth program we demonstrated how we can manipulate indirect addressing with registers.

```

1 #indirect_addressing_with_registers:  accessing data in mem using pointers
2
3
4 .section .data
5     constants:
6         .int 5, 8, 17, 44, 50, 52, 60, 65, 70, 77, 80
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                    #used for debugging purposes
13
14 indirect_addressing_with_registers:
15     movl constants, %eax    #mov constants mem value into eax
16     movl $constants, %edi  #mov mem addr into edi
17     movl $25, 4(%edi)      #mov immediate val 4b after edi ptr
18     movl $1, %edi         #load 2nd index constants label
19     movl constants(, %edi, 4), %ebx #mov above value 4 bytes from constants
20
21 exit:
22     movl $1, %eax          #sys_exit system call
23     movl $0, %ebx         #exit code 0 successful execution
24     int $0x80             #call sys_exit

```

I want to start by addressing the question of why I use AT&T syntax. In previous lessons I provided many ways to easily convert back and forth between AT&T syntax and Intel syntax.

I deliberately choose this path so that it forces you to be comfortable with the most complex flavor of x86. If you are confused with this syntax please review the prior lessons as I go through the differences between both.

Let's recap. We will use objdump to take a compiled binary such as the one above that we compiled in our last lesson and show how we can view it's Intel source code.

objdump -d -M intel indirect_addressing_with_registers | grep _start.: -A24

```

noroot@noroot-gamma:~/Desktop/Code$ objdump -d -M intel indirect_addressing_with_registers | grep _start.: -A24
08048074 <_start>:
8048074:    90                    nop

08048075 <indirect_addressing_with_registers>:
8048075:    a1 9e 90 04 08       mov     eax,ds:0x804909e
804807a:    bf 9e 90 04 08       mov     edi,0x804909e
804807f:    c7 47 04 19 00 00 00 mov     DWORD PTR [edi+0x4],0x19
8048086:    bf 01 00 00 00       mov     edi,0x1
804808b:    8b 1c bd 9e 90 04 08 mov     ebx,DWORD PTR [edi*4+0x804909e]

08048092 <exit>:
8048092:    b8 01 00 00 00       mov     eax,0x1
8048097:    bb 00 00 00 00       mov     ebx,0x0
804809c:    cd 80                int     0x80

```

Now back to our regularly scheduled program.

Let's load the binary into GDB and break on `_start`, step a few steps and examine 6 of the 11 values inside the constants label.


```

noroot@noroot-gamma:~/Desktop/Code$ gdb -q indirect_addressing_with_registers
Reading symbols from indirect_addressing_with_registers...(no debugging symbols
found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/indirect_addressing_with_registers

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:    nop
End of assembler dump.
(gdb) si
0x08048075 in indirect_addressing_with_registers ()
(gdb) disas
Dump of assembler code for function indirect_addressing_with_registers:
=> 0x08048075 <+0>:    mov     0x804909e,%eax
0x0804807a <+5>:    mov     $0x804909e,%edi
0x0804807f <+10>:   movl   $0x19,0x4(%edi)
0x08048086 <+17>:   mov     $0x1,%edi
0x0804808b <+22>:   mov     0x804909e(,%edi,4),%ebx
End of assembler dump.
(gdb) si
0x0804807a in indirect_addressing_with_registers ()
(gdb) x/6 &constants
0x804909e:    5      8      17     44
0x80490ae:    50     52

```

We then move the memory address of the constants label into edi and move the immediate value of 25 decimal into the second index of our array. This is in essence a source code hack as we are changing the original value of 8 to 25.

If you examine the source code you see line 18 where we load the value of 1 into edi. Keep in mind this is the second value as arrays are 0 based.

```

(gdb) si
0x0804807f in indirect_addressing_with_registers ()
(gdb) disas
Dump of assembler code for function indirect_addressing_with_registers:
0x08048075 <+0>:    mov     0x804909e,%eax
0x0804807a <+5>:    mov     $0x804909e,%edi
=> 0x0804807f <+10>:   movl   $0x19,0x4(%edi)
0x08048086 <+17>:   mov     $0x1,%edi
0x0804808b <+22>:   mov     0x804909e(,%edi,4),%ebx
End of assembler dump.
(gdb) si
0x08048086 in indirect_addressing_with_registers ()
(gdb) x/6 &constants
0x804909e:    5      25     17     44
0x80490ae:    50     52

```

You can see we changed the value of 8 decimal into 25 as explained.

This is our first introduction to arrays in assembly language. It is critical that you understand how they work as you may someday be a Malware Analyst or Reverse Engineer looking at the compiled binary of any number of higher-level program compiled arrays.

In our next lesson we will manually hack one of the values in GDB. Keep in mind, we will have to overwrite the contents inside an actual memory address with an immediate value. The fun is only beginning!

I look forward to seeing you all next week when we dive into hacking our fifth assembly program!

Part 36 - ASM Hacking 5 [Indirect Addressing With Registers]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's reexamine the source once more.

```

1 #indirect_addressing_with_registers:  accessing data in mem using pointers
2
3
4 .section .data
5     constants:
6         .int 5, 8, 17, 44, 50, 52, 60, 65, 70, 77, 80
7
8 .section .text
9     .globl _start
10
11 _start:
12     nop                #used for debugging purposes
13
14 indirect_addressing_with_registers:
15     movl constants, %eax    #mov constants mem value into eax
16     movl $constants, %edi   #mov mem addr into edi
17     movl $25, 4(%edi)      #mov immediate val 4b after edi ptr
18     movl $1, %edi          #load 2nd index constants label
19     movl constants(, %edi, 4), %ebx #mov above value 4 bytes from constants
20
21 exit:
22     movl $1, %eax          #sys_exit system call
23     movl $0, %ebx          #exit code 0 successful execution
24     int $0x80              #call sys_exit

```

Let's once again load the binary into GDB and break on `_start`.

```

noroot@noroot-gamma:~/Desktop/Code$ gdb -q indirect_addressing_with_registers
Reading symbols from indirect_addressing_with_registers...(no debugging symbols
found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/indirect_addressing_with_registers

Breakpoint 1, 0x08048074 in _start ()
(gdb) print *0x804909e
$1 = 5
(gdb) print *0x80490a2
$2 = 8

```

As we look above we see the command `print *0x804909e`. We see that it yields a value of 5 decimal. The binary at runtime puts the values inside the constants label to a respective memory address.

In this case we see that the pointer to `0x804909e` or `*0x804909e` holds 5 decimal as we have stated above. An integer holds 4 bytes of data. The next value in our array will be stored in `0x80490a2`. This memory location will hold the value of 8.

If we were to continue to advance through the array we would move 4 bytes to the next value and so forth. Remember each memory location in x86 32-bit assembly holds 4 bytes of data.

Let's hack!

```
noroot@noroot-gamma:~/Desktop/Code$ gdb -q indirect_addressing_with_registers
Reading symbols from indirect_addressing_with_registers...(no debugging symbols
found)...done.
(gdb) b _start
Breakpoint 1 at 0x8048074
(gdb) r
Starting program: /home/noroot/Desktop/Code/indirect_addressing_with_registers

Breakpoint 1, 0x08048074 in _start ()
(gdb) x/6d &constants
0x804909e:   5      8      17     44
0x80490ae:  50     52
(gdb) set *0x80490a2 = 66
(gdb) x/6d &constants
0x804909e:   5      66     17     44
0x80490ae:  50     52
```

After we broke on `_start` and ran, we examined the array like we did in our prior lesson. Here we hack the value at `0x80490a2` to 66 decimal instead of 8 decimal and we can see that we successfully changed one element of the array.

This lesson is very important to understand how arrays are ultimately stored in memory and how to manipulate and hack them. If you have any questions, please leave them in the comments below.

I look forward to seeing you all next week when we dive into programming our sixth assembly program!

Part 37 - ASM Program 6 [CMOV Instructions]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

In our sixth program we will demonstrate how we can work with CMOV instructions.

Before we dive into some code lets talk about CMOV is. CMOV can prevent the processor from utilizing the JMP instructions and speeds up the respective binary.

There are unsigned CMOV instructions such as:

CMOVA or CMOVNBE = Above [Carry Flag or Zero Flag = 0]

CMOVAE or CMOVNB = Above Or Equal [Carry Flag = 0]

CMOVNC = Not Carry [Carry Flag = 0]

CMOVB or CMOVNAE = Below [Carry Flag = 1]

CMOVC = Carry [Carry Flag = 1]

CMOVBE or CMOVNA = Below Or Equal [Carry Flag or Zero Flag = 1]

CMOVE or CMOVZ = Equal [Zero Flag = 1]

CMOVNE or CMOVNZ = Not Equal [Zero Flag = 0]

CMOVPE or CMOVPO = Parity [Parity Flag = 1]

CMOVNP or CMOVPO = Not Parity [Parity Flag = 0]

There are also signed CMOV instructions such as:

CMOVGE or CMOVNL = Greater Or Equal [Sign Flag xor Overflow Flag = 0]

CMOVL or CMOVNGE = Less [Sign Flag xor Overflow Flag = 1]

CMOVLE or CMOVNG = Less Or Equal [Sign Flag xor Overflow Flag or ZF = 1]

CMOVO = Overflow [Overflow Flag = 1]

CMOVNO = Not Overflow [Overflow Flag = 0]

CMOVS = Sign NEGATIVE [Sign Flag = 1]

CMOVNS = Not Sign POSITIVE [Sign Flag = 0]

Keep in mind to review the relationships between the unsigned and signed operations. The unsigned instructions utilize the CF, ZF and PF to determine the difference between the two operands where the signed instructions utilize the SF and OF to indicate the condition of the comparison between the operands.

If you need a refresher on the flag please review Part 14 on Flags in this series.

The CMOV instructions rely on a mathematical instruction that sets the EFLAGS register to operate and therefore saves the programmer to use JMP statements after the compare statement. Lets examine some source code.

```

1 #cmov_instructions: conditional move instruction
2
3
4 .section .data
5     result:
6     .asciz "The smallest value is "
7     lr:
8     .ascii ".\n"
9     constants:
10    .int 43, 144, 32, 549, 600, 7, 10, 11
11
12 .section .bss
13     .comm answer, 1
14
15 .section .text
16     .globl _start
17
18 _start:
19     nop                                #used for debugging purposes
20
21     movl constants, %ebx                #mov array values into ebx
22     movl $1, %edi                       #load 2nd index constants label
23
24 find_smallest_value:
25     movl constants(, %edi, 4), %eax     #mov value 4 bytes from constants
26     cmp %ebx, %eax                      #compare ebx to eax
27     cmovb %eax, %ebx                    #compare below eax to ebx
28     inc %edi                             #increment edi to move through array
29     cmp $8, %edi                         #check where we are in array
30     jne find_smallest_value            #jne to beginning of loop
31     addl $0x30, %ebx                     #convert int to ascii
32     movl %ebx, answer                   #move new value of ebx to answer label
33
34     movl $4, %eax                        #sys_write
35     movl $1, %ebx                        #stdout
36     movl $result, %ecx                   #mov result into ecx
37     movl $23, %edx                       #mov 23 bytes into edx
38     int $0x80                            #call sys_write
39
40     movl $4, %eax                        #sys_write
41     movl $1, %ebx                        #stdout
42     movl $answer, %ecx                   #mov answer label into ecx
43     movl $1, %edx                       #mov 1 byte into edx
44     int $0x80                            #call sys_write
45
46     movl $4, %eax                        #sys_write
47     movl $1, %ebx                        #stdout
48     movl $lr, %ecx                      #mov lr label into ecx
49     movl $2, %edx                       #mov 1 byte into edx
50     int $0x80                            #call sys_write
51
52 exit:
53     movl $1, %eax                        #sys_exit system call
54     movl $0, %ebx                       #exit code 0 successful execution
55     int $0x80                            #call sys_exit

```

Ok lets begin with lines 21 and 22. This is nothing new that we have experienced as we are simply moving the array into ebx.

On line 24 we see the find_smallest_value function to where we are cycling through the array and using the CMOVB to find the lowest value ultimately.

We see **cmp %ebx, %eax** to which cmp subtracts the first operand from the second and sets the EFLAGS register appropriately. At this point the cmovb is used to replace the value in ebx with the value in eax if the value is smaller than what was originally in the ebx register.

After we exit the loop we see three sets of sys_writes to first display our message, second to display our converted integer to ascii value and then finally a period and line feed.

Keep in mind to assemble we type:

Part 1: Goals

```
as -32 -o cmov_instructions.o cmov_instructions.s
```

To link the object file we type:

```
ld -m elf_i386 -o cmov_instructions cmov_instructions.o
```

I look forward to seeing you all next week when we dive into debugging our sixth assembly program!

Part 38 - ASM Debugging 6 [CMOV Instructions]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Lets re-examine some source code.

```

1 #cmov_instructions: conditional move instruction
2
3
4 .section .data
5     result:
6         .asciz "The smallest value is "
7         lr:
8         .ascii ".\n"
9     constants:
10        .int 43, 144, 32, 549, 600, 7, 10, 11
11
12 .section .bss
13     .comm answer, 1
14
15 .section .text
16     .globl _start
17
18 _start:
19     nop                                #used for debugging purposes
20
21     movl constants, %ebx                #mov array values into ebx
22     movl $1, %edi                       #load 2nd index constants label
23
24 find_smallest_value:
25     movl constants(, %edi, 4), %eax      #mov value 4 bytes from constants
26     cmp %ebx, %eax                      #compare ebx to eax
27     cmovb %eax, %ebx                    #compare below eax to ebx
28     inc %edi                             #increment edi to move through array
29     cmp $8, %edi                         #check where we are in array
30     jne find_smallest_value             #jne to beginning of loop
31     addl $0x30, %ebx                     #convert int to ascii
32     movl %ebx, answer                   #move new value of ebx to answer label
33
34     movl $4, %eax                        #sys_write
35     movl $1, %ebx                        #stdout
36     movl $result, %ecx                   #mov result into ecx
37     movl $23, %edx                       #mov 23 bytes into edx
38     int $0x80                             #call sys_write
39
40     movl $4, %eax                        #sys_write
41     movl $1, %ebx                        #stdout
42     movl $answer, %ecx                   #mov answer label into ecx
43     movl $1, %edx                         #mov 1 byte into edx
44     int $0x80                             #call sys_write
45
46     movl $4, %eax                        #sys_write
47     movl $1, %ebx                        #stdout
48     movl $lr, %ecx                       #mov lr label into ecx
49     movl $2, %edx                         #mov 1 byte into edx
50     int $0x80                             #call sys_write
51
52 exit:
53     movl $1, %eax                        #sys_exit system call
54     movl $0, %ebx                        #exit code 0 successful execution
55     int $0x80                             #call sys_exit

```

Lets break on 0x08048092 which is line 31. Lets do a r to run and then type **print \$ebx**. We can see the value of 7.

```

pc@pc-delta:~/Desktop/Code$ gdb -q ./cmov_instructions
Reading symbols from ./cmov_instructions...(no debugging symbols found)...done.
(gdb) b *0x08048092
Breakpoint 1 at 0x8048092
(gdb) r
Starting program: /home/pc/Desktop/Code/cmov_instructions

Breakpoint 1, 0x08048092 in find_smallest_value ()
(gdb) print $ebx
$1 = 7

```

Part 1: Goals

Ok now lets break on 0x080480b1 which is line 46. Remember when we are examining the value of **answer**, it has been converted to its ascii printable equivalent so in order to see the value of '7' you would type **x/1c &answer**.

```
(gdb) b *0x080480b1
Breakpoint 2 at 0x80480b1
(gdb) s
Single stepping until exit from function find_smallest_value,
which has no line number information.
The smallest value is
Breakpoint 2, 0x080480b1 in find_smallest_value ()
(gdb) x/1c &answer
0x8049122 <answer>:      55 '7'
```

I look forward to seeing you all next week when we dive into hacking our sixth assembly program!

Part 39 - ASM Hacking 6 [CMOV Instructions]

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's bring the binary into gdb.

```
Reading symbols from cmov_instructions...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x08048074
(gdb) r
Starting program: /home/pc/Desktop/cmov_instructions

Breakpoint 1, 0x08048074 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x08048074 <+0>:    nop
      0x08048075 <+1>:    mov     0x8049102,%ebx
      0x0804807b <+7>:    mov     $0x1,%edi
End of assembler dump.
(gdb) si
0x08048075 in _start ()
(gdb) si
0x0804807b in _start ()
(gdb) si
0x08048080 in find_smallest_value ()
```

Let's now run the binary. We see that the smallest value is 7 which is expected.

Our final bit of instruction in this tutorial will teach you how to jump to any part of the execution that you so choose.

```
(gdb) s
Single stepping until exit from function find_smallest_value,
which has no line number information.
The smallest value is 7.
0x080480dd in exit ()
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/pc/Desktop/cmov_instructions

Breakpoint 1, 0x08048074 in _start ()
(gdb) set $eip = 0x080480dd
(gdb) s
Single stepping until exit from function exit,
which has no line number information.
[Inferior 1 (process 22023) exited normally]
(gdb)
```

We set **\$eip = 0x080480dd** which is the exit routine. We see now that it bypasses all of the code from the nop instruction when we broke on `_start`. You now can use this command to jump anywhere inside of any binary within the debugger.

I look forward to seeing you all next week when we wrap up our tutorial series.

Part 40 - Conclusion

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

This has been an extensive and hopefully beneficial tutorial series for you all. Understanding assembly language is so important to everyone when trying to understand how Malware works in addition to programming no matter bare-metal assembly, c, c++ or even Java, Python or iOS or Android development.

If you are looking to pursue a career in Reverse Engineering, assembly will be second nature to you. Most of us will pursue higher-level language development as computers and devices are significantly more powerful today which allows for rapid development languages.

I want to thank you all for joining me on this tutorial series and look forward to you all making an impact in the future of tomorrow!

The 32-bit ARM Architecture (Part 1)

Let's dive in rightaway!

Part 1 - The Meaning Of Life

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover.

https://github.com/mytechtalent/hacking_c-_arm64

Why C++? I primarily develop in Python professionally as an Automator however with every day passing we see another Ransomware attack that further cripples society in a catastrophic way.

This course is a comprehensive series where we learn every facet of C++ and how it relates to the ARM 64 architecture as we will reverse engineer each step in ARM 64 assembly language to get a full understanding of the environment.

There are roughly over 2,000 hacks a day world-wide and so few who truly understand how the hacks are executed on a fundamental level. This course is going to take a very basic and step-by-step approach to understanding low-level architecture as it relates to the ARM 64.

In our next lesson we will set up our development environment.

Part 2 – Number Systems

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

At the core of the microprocessor are a series of binary numbers which are either +5V (on or 1) or 0V (off or 0). Each 0 or 1 represents a bit of information within the microprocessor. A combination of 8 bits results in a single byte.

Before we dive into binary, lets examine the familiar decimal. If we take the number 2017, we would understand this to be two thousand and seventeen.

Value	1000s	100s	10s	1s
Representation	10^3	10^2	10^1	10^0
Digit	2	0	1	7

Let's take a look at the binary system and the basics of how it operates.

Bit Number	b7	b6	b5	b4	b3	b2	b1	b0
Representation	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Decimal Weight	128	64	32	16	8	4	2	1

If we were to convert a binary number into decimal, we would very simply do the following. Lets take a binary number of 0101 1101 and as you can see it is 93 decimal.

Bit	Weight	Value
0	128	0
1	64	64
0	32	0
1	16	16
1	8	8
1	4	4
0	2	0
1	1	1

Adding the values in the value column gives us $0 + 64 + 0 + 16 + 8 + 4 + 0 + 1 = 93$ decimal.

If we were to convert a decimal number into binary, we would check to see if a subtraction is possible relative to the highest order bit and if so, a 1 would be placed into the binary column to which the remainder would be carried into the next row. Let's consider the example of the decimal value of 120 which is 0111 1000 binary.

128	64	32	16	8	4	2	1
0	1	1	1	1	0	0	0

1) Can 128 fit inside of 120: No, therefore 0.

2) Can 64 fit inside of 120: Yes, therefore 1, then $120 - 64 = 56$.

3) Can 32 fit inside of 56: Yes, therefore 1, then $56 - 32 = 24$.

4) Can 16 fit inside of 24: Yes, therefore 1, then $24 - 16 = 8$.

5) Can 8 fit inside of 8: Yes, therefore 1, then $8 - 8 = 0$.

6) Can 4 fit inside of 0: No, therefore 0.

7) Can 2 fit inside of 0: No, therefore 0.

8) Can 1 fit inside of 0: No, therefore 0.

When we want to convert binary to hex we simply work with the following table.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Lets convert a binary number such as 0101 1111 to hex. To do this we very simply look at the table and compare each nibble which is a combination of 4 bits. Keep in mind, 8 bits is equal to a byte and 2 nibbles are equal to a byte.

0101 = 5

1111 = F

Therefore 0101 1111 binary = 0x5f hex. The 0x notation denotes hex.

To go from hex to binary it's very simple as you have to simply do the opposite such as:

0x3a = 0011 1010

3 = 0011

A = 1010

It is important to understand that each hex digit is a nibble in length therefore two hex digits are a byte in length.

To convert from hex to decimal we do the following:

0x5f = 95

5 = $5 \times 16^1 = 5 \times 16 = 80$

F = $15 \times 16^0 = 15 \times 1 = 15$

Therefore we can see that $80 + 15 = 95$ which is 0x5f hex.

Finally to convert from decimal to hex. Lets take the number 850 decimal which is 352 hex.

Division	Result (No Remainder)	Remainder	Remainder Multiplication
850 / 16	53	0.125	$0.125 \times 16 = 2$
53 / 16	3	0.3125	$0.3125 \times 16 = 5$
3 / 16	0	0.1875	$0.1875 \times 16 = 3$

We put the numbers together from bottom to the top and we get 352 hex.

“Why the hell would I waste my time learning all this crap when the computer does all this for me!”

If you happen to know any reverse engineers please if you would take a moment and ask them the above question.

The reality is, if you do NOT have a very firm understanding of how all of the above works, you will have a hard time getting a grasp on how the ARM processor registers hold and manipulate data. You will also have a hard time getting a grasp on how the ARM processor deals with a binary overflow and it's effect on how carry operations work nor will you understand how compare operations work or even the most basic operations of the most simple assembly code.

I am not suggesting you memorize the above, nor am I suggesting that you do a thousand examples of each. All I ask is that you take the time to really understand that literally everything and I mean everything goes down to binary bits in the processor.

Whether you are creating, debugging or hacking an Assembly, Python, Java, C, C++, R, JavaScript, or any other new language application that hits the street, ultimately everything MUST go down to binary 0 and 1 to which represent a +5V or 0V.

We as humans operate on the base 10 decimal system. The processor works on a base 16 (hex) system. The registers we are dealing with in conjunction with Linux are addressed in 32-bit sizes. When we begin discussion of the processor registers, we will learn that each are 32-bits wide (technically the BCM2837 are 64-bit wide however our version of Linux that we are working with is 32-bit therefore we only address 32-bits of each register).

Next week we will dive into binary addition! Stay tuned!

Part 3 – Binary Addition

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Binary addition can occur in one of four different fashions:

```
0 + 0 = 0
1 + 0 = 1
0 + 1 = 1
1 + 1 = 0 (1) [One Plus One Equals Zero, Carry One]
```

Keep in mind the (1) means a carry bit. It very simply means an overflow.

Lets take the following 4-bit nibble example:

```
0111
+ 0100
= 1011
```

We see an obvious carry in the 3rd bit. If the 8th bit had a carry then this would generate a carry flag within the CPU.

Let's examine an 8-bit number:

```
01110000
+ 01010101
= 11000101
```

If we had:

```
11110000
+ 11010101
= (1)11000101
```

Here we see a carry bit which would trigger the carry flag within the CPU to be 1 or true. We will discuss the carry flag in later tutorials. Please just keep in mind this example to reference as it is very important to understand.

Next week we will dive into binary subtraction! Stay tuned!

Part 4 – Binary Subtraction

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Binary subtraction is nothing more than adding the negative value of the number to be subtracted. For example $8 + -4$, the starting point would be zero to which we move 8 points in the positive direction and then four points in the negative direction yielding a value of 4.

We represent a sign bit in binary to which bit 7 indicates the sign of number where 0 is positive and 1 is negative.

Sign Bit 7	Bits 0 – 6
1	000011

The above would represent -2.

We utilize the concept of twos complement which inverts each bit and then finally adding 1.

Lets examine binary 2.

0000010

Invert the bits.

1111101

Add 1.
1111101
+ 0000001
1111110

Let's examine a subtraction operation:

0000100	4 decimal	-
+ 1111110	-2 decimal	
(1)0000010	2 decimal	

So what is the (1) you may ask, that is the overflow bit. In future tutorials we will examine what we refer to as the overflow flag and carry flag.

Next week we will dive into word lengths! Stay tuned!

Part 5 – Word Lengths

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

The system on chip we are working with has a 32-bit ARM CPU. 32-bits is actually 4 bytes of information which make up a word.

If you remember my prior tutorial on x86 Assembly, a word was 16-bits. Every different architecture defines a word differently.

The most significant bit of a word for our ARM CPU is located at bit 31 therefore a carry is generated if an overflow occurs there.

The lowest address in our architecture starts at 0x00000000 and goes to 0xFFFFFFFF. The processor sees memory in word blocks therefore every 4 bytes. A memory address associated with the start of a word is referred to as a word boundary and is divisible by 4. For example here is our first word:

```
0x00000000
0x00000004
0x00000008
0x0000000C
```

So why is this important? There is the concept of fetching and executing to which the processor deals with instructions to which it must work in this fashion for proper execution.

Before we dive into coding assembly it is critical that you understand some basics of how the CPU operates. There will be a number of more lectures going over the framework so I appreciate everyone hanging in there!

Next week we will dive into registers! Stay tuned!

Part 6 – Registers

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Our ARM microprocessor has internal storage which make any operation must faster as there is no external memory access needed. There are two modes, User and Thumb. We will be focusing on User Mode as we are ultimately focused on developing for a system on chip within a Linux OS rather than bare-metal programming which would be better suited on a microcontroller device.

In User Mode we have 16 registers and a CPSR register to which have a word length each which is 32-bits each or 8 bytes each.

Registers R0 to R12 are multi-purpose registers to which R13 – R15 have a unique purpose as well as the CPSR. Lets take a look at a simple table to illustrate.

```
R0 GPR (General-Purpose Register)
R1 GPR (General-Purpose Register)
R2 GPR (General-Purpose Register)
R3 GPR (General-Purpose Register)
R4 GPR (General-Purpose Register)
R5 GPR (General-Purpose Register)
R6 GPR (General-Purpose Register)
R7 GPR (General-Purpose Register)
R8 GPR (General-Purpose Register)
R9 GPR (General-Purpose Register)
R10 GPR (General-Purpose Register)
R11 GPR (General-Purpose Register)
R12 GPR (General-Purpose Register)
R13 Stack Pointer
R14 Link Register
R15 Program Counter
CPSR Current Program Status Register
```

It is critical that we understand registers in a very detailed way. At this point we understand R0 – R12 are general purpose and will be used to manipulate data as we build our programs and additionally when you are hacking apart or reverse engineering binaries from a hex dump on a cell phone or other ARM device, no matter what high-level language it is written in, it must ultimately come down to assembly which you need to understand registers and how they work to grasp and understand of any such aforementioned operation.

The chip we are working with is known as a load and store machine. This means we load a register with the contents of a register or memory location and we can store a register with the contents of a memory or register location. For example:

```
ldr, r4, [r10] @
    load r4 with the contents of r10, if r10 had the
decimal value of
    say 22, 22 would go to r4

str, r9, [r4] @
    store r9 contents into location in r4, if r9 had 0x02
hex,
    0x02 would be stored into location r4
```

The @ simply indicates to the compiler that what follows it on a given line is a comment and to be ignored.

The next few weeks we will take our time and look at each of the special purpose registers so you have a great understanding of what they do.

Next week we will dive into more information on the program counter! Stay tuned!

Part 7 – Program Counter

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

We will dive into the registers over the coming weeks to make sure you obtain a firm understand of their role and what they can do.

We begin with the PC or program counter. The program counter is responsible for directing the CPU to what instruction will be executed next. The PC literally holds the address of the instruction to be fetched next.

When coding you can refer to the PC as PC or R15 as register 15 is the program counter. You **MUST** treat it with care as you can set it wrong and crash the executable quite easily.

You can control the PC directly in code:

```
mov r15, 0x00000000
```

I would not suggest trying that as we are not in Thumb mode and that will cause a fault as you would be going to an OS area rather than designated program area.

Regarding our ARM processor, we follow the standard calling convention meaning params are passed by placing the param values into regs R0 – R3 before calling the subroutine and the subroutine returns a value by putting it in R0 before returning.

This is important to understand when we think about how execution flows when dealing with a stack operation and the link register which we will discuss in future tutorials.

When you are hacking or reversing a binary, controlling the PC is essential when you want to test for subroutine execution and learning about how the program flows in order to break it down and understand exactly what it is doing.

Next week we will dive into more information on the CPSR! Stay tuned!

Part 8 - CPSR

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

The CPSR register stores information about the program and the results of a particular operation. Bits that are in the respective registers have pre-assigned conditions that are tested for an occurrence which are flags.

There are 32-bits that total this register. The highest 4 we are concerned with most which are:

Bit 31 – N = Negative Flag

Bit 30 – Z = Zero Flag

Bit 29 – C = Carry Flag (UNSIGNED OPERATIONS)

Bit 28 – V = Overflow flag (SIGNED OPERATIONS)

When the instruction completes the CPSR can get updated if it falls into one of the aforementioned scenarios. If one of the conditions occurs, a 1 goes into the respective bits.

There are two instructions that directly effect the CPSR flags which are CMP and CMN. CMP is compare such as:

```
CMP R1, R0 @ notational subtraction where R1 - R0 and if
the result is 0, bit 30 Z would be set to 1
```

The most logical command that usually follows is BEQ = branch if equal, meaning the zero flag was set and branches to another label within the code.

Regarding CMP, if two operands are equal then the result is zero. CMN makes the same comparison but with the second operand negated for example:

```
CMN R1, R0 @ R1 - (-R0) or R1 + R0
```

When dealing with the SUB command, the result would NOT update the CPSR you would have to use the SUBS command to make any flag update respectively.

Next week we will dive into more information on the Link Register! Stay tuned!

Part 9 - Link Register

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The Link Register, R14, is used to hold the return address of a function call.

When a BL (branch with link) instruction performs a subroutine call, the link register is set to the subroutine return address. BL jumps to another location in the code and when complete allows a return to the point right after the BL code section. When the subroutine returns, the link register returns the address back to the program counter.

The link register does not require the writes and reads of the memory containing the stack which can save a considerable percentage of execution time with repeated calls of small subroutines.

When BL has executed, the return address which is the address of the next instruction to be executed, is loaded into the LR or R14. When the subroutine has finished, the LR is copied directly to the PC (Program Counter) or R15 and code execution continues where it was prior in the sequential code source.

CODE TIME! Don't be discouraged if you don't understand everything in the code example here. It will become clear over the next few lessons.

```

1  LR Demo - Link Register Demo
2
3  .global _start
4
5  _start:
6      mov r7, #0x30      @ mov hex 30 into r7
7      b no_return      @ branch to no_return function which never returns
8
9  no_return:
10     mov r7, #1        @ mov decimal 1 into r7
11     bl my_function   @ go to my_function which after execution
12                       @ will after this
13                       @ bl loads the return address of wrap_up into lr
14 wrap_up:
15     mov r7, #0x12     @ mov hex 12 into r7
16     b exit           @ branch to exit
17
18 my_function:
19     mov r0, #10       @ mov decimal 10 into r10
20     mov pc, lr       @ mov the ret address into the program counter
21                       @ here is where we return back to where we first
22                       @ bl therefore we go to wrap_up
23
24 exit:
25     mov r7, #1        @ sys_exit
26     svc 0

```

To compile:

```
as -o lr_demo.o lr_demo.s
```

```
ld -o lr_demo lr_demo.o
```

The simple example I created here is pretty self-explanatory. We start and proceed to the **no_return** subroutine and proceed to the **my_function** subroutine then to the **wrap_up** subroutine and finally **exit**.

```

pi@pi-beta:~/Code $ gdb -q lr_demo
Reading symbols from lr_demo...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/Code/lr_demo

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r7, #48 ; 0x30
    0x00010058 <+4>:   b      0x1005c <no_return>
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r7, #48 ; 0x30
=> 0x00010058 <+4>:   b      0x1005c <no_return>
End of assembler dump.
(gdb) si
0x0001005c in no_return ()
(gdb) disas
Dump of assembler code for function no_return:
=> 0x0001005c <+0>:   mov     r7, #1
    0x00010060 <+4>:   bl     0x1006c <my_function>
End of assembler dump.
(gdb) si
0x00010060 in no_return ()
(gdb) disas
Dump of assembler code for function no_return:
=> 0x0001005c <+0>:   mov     r7, #1
=> 0x00010060 <+4>:   bl     0x1006c <my_function>
End of assembler dump.
(gdb) si
0x0001006c in my_function ()
(gdb) disas
Dump of assembler code for function my_function:
=> 0x0001006c <+0>:   mov     r0, #10
    0x00010070 <+4>:   mov     pc, lr
End of assembler dump.
(gdb) si
0x00010070 in my_function ()
(gdb) disas
Dump of assembler code for function my_function:
=> 0x0001006c <+0>:   mov     r0, #10
=> 0x00010070 <+4>:   mov     pc, lr
End of assembler dump.

```

It is necessary that we jump into GDB which is our debugger to see exactly what happens with each step:

As you can see with every step inside the debugger it shows you exactly the progression from **no_return** to **my_function** skipping **wrap_up** until the program counter gets the address from the link register.


```

(gdb) si
0x00010064 in wrap_up ()
(gdb) disas
Dump of assembler code for function wrap_up:
=> 0x00010064 <+0>:    mov     r7, #18
      0x00010068 <+4>:    b      0x10074 <exit>
End of assembler dump.
(gdb) si
0x00010068 in wrap_up ()
(gdb) disas
Dump of assembler code for function wrap_up:
      0x00010064 <+0>:    mov     r7, #18
=> 0x00010068 <+4>:    b      0x10074 <exit>
End of assembler dump.
(gdb) si
0x00010074 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x00010074 <+0>:    mov     r7, #1
      0x00010078 <+4>:    svc     0x00000000
End of assembler dump.
(gdb) si
0x00010078 in exit ()
(gdb) disas
Dump of assembler code for function exit:
      0x00010074 <+0>:    mov     r7, #1
=> 0x00010078 <+4>:    svc     0x00000000
End of assembler dump.
(gdb) si
[Inferior 1 (process 20229) exited with code 012]

```

Here we see the progression from **wrap_up** to **exit**.

This is a fundamental operation when we see next week how the stack operates as the LR is an essential part of this process.

Next week we will dive into the Stack Pointer! Stay tuned!

Part 10 - Stack Pointer

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The Stack is an abstract data type to which is a LIFO (Last In First Out). When we push a value onto the stack it goes into the Stack Pointer and when it is popped off of the stack it pops the value off of the stack and into a register of your choosing.

CODE TIME! Again, don't be discouraged if you don't understand everything in the code example here. It will become clear over the next few lessons.

```
1 Stack Pointer Demo - Stack Pointer Demo
2
3     .global _start
4
5 _start:
6     mov r7, #0x30      @ mov hex 30 into r7
7     push {r7}         @ push 0x30 value onto the Stack
8     mov r7, #0x10     @ mov hex 10 into r7
9     pop {r7}          @ pop the value on the Stack, 0x30 back into r7
10
11 exit:
12     mov r7, #1        @ sys_exit
13     svc 0
14
```

To compile:

```
as -o sp_demo.o sp_demo.s
```

```
ld -o sp_demo sp_demo.o
```

Once again lets load the binary into GDB to see what is happening.

```

pi@pi-beta:~/Code $ gdb -q sp_demo
Reading symbols from sp_demo...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/Code/sp_demo

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r7, #48 ; 0x30
      0x00010058 <+4>:   push   {r7}           ; (str r7, [sp, #-4]!)
      0x0001005c <+8>:   mov     r7, #16
      0x00010060 <+12>:  pop    {r7}           ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x0      0
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3a0   0x7efff3a0
lr          0x0      0
pc          0x10054   0x10054 <_start>
cpsr       0x10     16

```

Lets step into one time.

```

(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r7, #48 ; 0x30
      0x00010058 <+4>:   push   {r7}           ; (str r7, [sp, #-4]!)
      0x0001005c <+8>:   mov     r7, #16
      0x00010060 <+12>:  pop    {r7}           ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x0      0
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x30     48
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3a0   0x7efff3a0
lr          0x0      0
pc          0x10058   0x10058 <_start+4>
cpsr       0x10     16

```

We see **hex 30** or **48 decimal** moved into r7. Lets step into again.

```
(gdb) si
0x0001005c in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:   mov     r7, #48 ; 0x30
   0x00010058 <+4>:   push   {r7}           ; (str r7, [sp, #-4]!)
=> 0x0001005c <+8>:   mov     r7, #16
   0x00010060 <+12>:  pop    {r7}           ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x0      0
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x30     48
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff39c   0x7efff39c
lr          0x0      0
pc          0x1005c   0x1005c <_start+8>
cpsr       0x10     16
```

We see the value of the **sp** change from **0x7efff3a0** to **0xefff39c**. That is a movement backward **4 bytes**. Why the heck is the stack pointer going backward you may ask!

The answer revolves around the fact that the stack grows **DOWNWARD**. When we say the top of the stack you can imagine a series of plates being placed **BENEATH** of each other.

Originally the **sp** was at **0x7efff3a0**.



When we pushed **r7** onto the stack, the new value of the **Stack Pointer** is now **0x7efff39c** so we can see the Stack truly grows **DOWNWARD** in memory.



Now lets step into again.

```
(gdb) si
0x00010060 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:   mov     r7, #48 ; 0x30
   0x00010058 <+4>:   push   {r7}          ; (str r7, [sp, #-4]!)
   0x0001005c <+8>:   mov     r7, #16
=> 0x00010060 <+12>:  pop     {r7}          ; (ldr r7, [sp], #4)
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x0      0
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x10     16
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff39c    0x7efff39c
lr          0x0      0
pc          0x10060   0x10060 <_start+12>
cpsr       0x10     16
```

We can see the value of **hex 10** or **decimal 16** moved into r7. Notice the **sp** did not change.


Before we step into again, lets look at the value inside the **sp**.

```
(gdb) x/x $sp
0x7efff39c: 0x30
```

Lets step into again.

```
(gdb) si
0x00010064 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x00010064 <+0>:   mov     r7, #1
   0x00010068 <+4>:   svc     0x00000000
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x0      0
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x30     48
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3a0    0x7efff3a0
lr          0x0      0
pc          0x10064   0x10064 <exit>
cpsr       0x10     16
```

We see the value in the stack was popped off the stack and put back into r7 therefore the value of **hex 30** is back in r7 as well as the **sp** is back at **0x73ff3a0**.

SP  0x7efff3a0

Part 1: Goals

Please take the time to type out the code, compile and link it and then step through the binary in GDB. Stack operations are critical to understanding Reverse Engineering and Malware Analysis as well as any debugging of any kind.

Next week we will dive into ARM Firmware Boot Procedures.

Part 11 - ARM Firmware Boot Procedures

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's take a moment to talk about what happens when we first power on our Raspberry Pi device.

As soon as the Pi receives power, the graphics processor is the first thing to run as the processor is held in a reset state to which the GPU starts executing code. The ROM reads from the SD card and reads **bootcode.bin** to which gets loaded into memory in C2 cache and turns on the rest of the RAM to which **start.elf** then loads.

The **start.elf** is an OS for the graphics processor and reads **config.txt** to which you can mod. The **kernel.img** then gets loaded into **0x8000** in memory which is the Linux kernel.

Once loaded, **kernel.img** turns on the CPU and starts running at **0x8000** in memory.

If we wanted, we could create our own **kernel.img** to which we can hard code machine code into a file and replace the original image and then reboot. Keep in mind the ARM word size is 32 bit long which go from bit 0 to 31.

As stated, when **kernel.img** is loaded the first byte, which is 8-bits, is loaded into address **0x8000**.

Lets open up a hex editor and write the following:

FE FF FF EA

Save the file as **kernel.img** and reboot.

"Ok nothing happens, this sucks!"

Actually something did happen, you created your first bare-metal firmware! Time to break out the champagne!

When the Pi boots, the below code when it reached **kernel.img** loads the following:

FE FF FF EA

@ address 0x8000, 0xfe gets loaded.

@ address 0x8001, 0xff gets loaded.

@ address 0x8002, 0xff gets loaded.

@ address 0x8003, 0xea gets loaded.

"So what the hell is really going on?"

This set of commands simply executes an infinite loop.

Review the datasheet:

<https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>

The above code has 3 parts to it:

- 1) Conditional – Set To Always
- 2) Op Code – Branch
- 3) Offset – How Far To Move Within The Current Location

Condition – bits 31-28: 0xe or 1110

Op Code – bits 27-24: 0xa or 1010

Offset – bits 23-0 -2

I know this may be a lot to wrap your mind around however it is critical that you take the time and read the datasheet linked above. Do not cut corners if you truly have the passion to understand the above. READ THE DATASHEET!

I will go through painstaking efforts to break everything down step-by-step however there are exercises like the above that I am asking you to review the datasheet above so you learn how to better understand where to look when you are stuck on a particular routine or set of machine code. This is one of those times I ask you to please read and research the datasheet above!

“I’m bored! Why the hell does this crap matter?”

Glad you asked! The single most dangerous malware on planet earth today is that of the root-kit variety. If you do not have a basic understanding of the above, you will never begin to even understand what a root-kit is as you progress in your understanding.

Anyone can simply replace the **kernel.img** file with their own hacked version and you can have total control over the entire process from boot.

Next week we will dive into the Von Neumann Architecture.

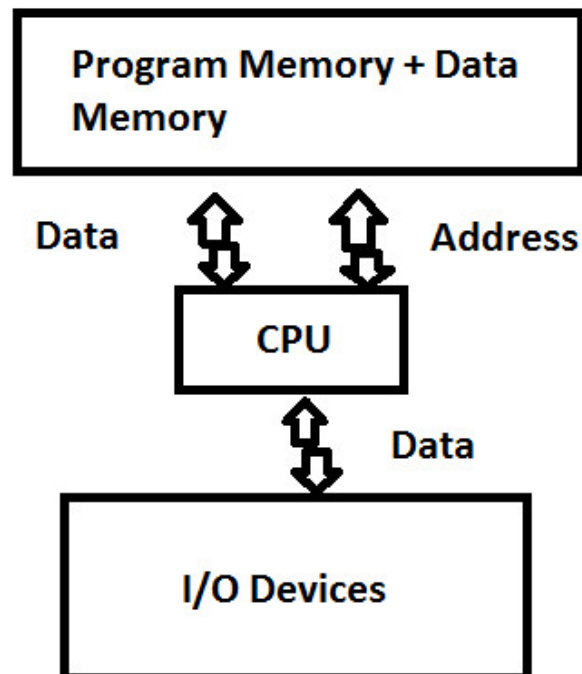
Part 12 - Von Neumann Architecture

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

ARM is a load and store machine to which the Arithmetic Logic Unit only operates on the registers themselves and any data that needs to be stored out to RAM, the control unit moves the data between memory and the registers which share the same data bus.

Von Neumann Architecture



The CPU chip of this architecture holds a control unit and the arithmetic logic unit (along with some local memory) and the main memory is in the form of RAM sticks located on the motherboard.

A stored-program digital computer is one that keeps its program instructions, as well as its data, in read-write, random-access memory or RAM.

Next week we will dive into the Instruction Pipeline.

Part 13 - Instruction Pipeline

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The processor works with three separate phases which are:

1)**Fetch Phase** – The control unit grabs the instruction from memory and loads it into the instruction register.

2)**Decode Phase** – The control unit configures all of the hardware within the processor to perform the instruction.

3)**Execute Phase** – The processor computes the result of the instruction or operation.

When the processor processes instruction 1 we refer to it as being in the fetch phase. When the processor processes instruction 2, instruction 1 goes into the decode phase and instruction 2 goes into the fetch phase. When the processor processes instruction 3, instruction 2 goes into the decode stage and instruction 1 goes into the execute stage.

Instruction:	1st Cycle	2nd Cycle	3rd Cycle
1	Fetch	Decode	Execute
2		Fetch	Decode
3			Fetch

Keep in mind, if a branch instruction occurs, the pipeline might be flushed and start over again with a fresh set of cycles.

You now have a strong basis and background of ARM Assembly and how it works regarding its load and store capability between memory and the respective registers and the basics of how the instruction set flows.

Next week we will dive into our first C++ program!

Part 14 - ADD

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

In ARM Assembly, we have three instructions that handle addition, the first being ADD, the second ADC (Add With Carry) and the final ADDS (Set Flag). This week we will focus on ADD.

Let's look at an example to illustrate:

```
#add - simple program to demonstrate the add command

        .global _start
_start:
    mov r1, #67           @ mov 67 decimal into r1
    mov r2, #53           @ mov 53 decimal into r2
    add r0, r1, r2        @ add r1 + r2 and store in r0

exit:
    mov r7, #1            @ exit syscall
    swi 0
```

Here we see that we move decimal **67** into **r1** and decimal **53** into **r2**. We then **add r1** and **r2** and put the result into **r0**.

"So what the heck is all that and why should I care?"

This series is going to be unlike any other in it's class. The goal is to take small pieces of code and see exactly what it does. If you are going to understand how to reverse a binary or malware of any kind, it is critical that you understand the basics. Learning ARM Assembly basics will help you when reversing an iPhone or Android. This tutorial series is going to work to take extremely small bites of code and talk about:

- 1)**The Code:** (Here) we speak briefly about what the code does.
- 2)**The Debug:** We break down the binary in the GDB Debugger and step through each instruction and see what specifically it does to program flow, register values and flags.
- 3)**The Hack:** We hack a piece of the code to make it do whatever WE want!

This approach will allow you to spend just a few minutes each week to get a good grasp on what is going on behind the scenes.

Next week we will dive into Debugging ADD.

Part 15 - Debugging ADD

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's review our ADD example below:

```
#add - simple program to demonstrate the add command

        .global _start
_start:
    mov r1, #67          @ mov 67 decimal into r1
    mov r2, #53          @ mov 53 decimal into r2
    add r0, r1, r2      @ add r1 + r2 and store in r0

exit:
    mov r7, #1          @ exit syscall
    swi 0
```

Again we see that we move decimal **67** into **r1** and decimal **53** into **r2**. We then **add r1** and **r2** and put the result into **r0**.

Let's compile:

```
as -o add.o add.s
```

```
ld -o add add.o
```

Let's bring into GDB to debug:

```
gdb -q add
```

```

Reading symbols from add...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) q
pi@pi-alpha:~/code $ gdb -q add
Reading symbols from add...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/add

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r1, #67 ; 0x43
      0x00010058 <+4>:   mov     r2, #53 ; 0x35
      0x0001005c <+8>:   add     r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0         0
r1             0x0         0
r2             0x0         0
r3             0x0         0
r4             0x0         0
r5             0x0         0
r6             0x0         0
r7             0x0         0
r8             0x0         0
r9             0x0         0
r10            0x0         0
r11            0x0         0
r12            0x0         0
sp             0x7efff3b0      0x7efff3b0
lr             0x0         0
pc             0x10054     0x10054 <_start>
cpsr          0x10         16
(gdb) █

```

We can see that when we `b _start`, break on start and `r`, run we see the disassembly. If you do an `i r` we see the info registers where we notice our `cpsr` is **0x10**.

As we step again and info registers:

```
(gdb) si
0x00010058 in _start ()
(gdb) i r
r0          0x0      0
r1          0x43     67
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x10058 0x10058 <_start+4>
cpsr       0x10     16
(gdb)
```

We notice **0x43** hex or **67** decimal into **r1**. We also notice that the flags are unchanged (**cpsr 0x10**).

Let's step again and info registers:

```
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:   mov     r1, #67 ; 0x43
   0x00010058 <+4>:   mov     r2, #53 ; 0x35
=> 0x0001005c <+8>:   add     r0, r1, r2
End of assembler dump.
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0          0x78     120
r1          0x43     67
r2          0x35     53
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x10060 0x10060 <exit>
cpsr       0x10     16
(gdb)
```

We can see **r0** now holds **0x78** hex or **120** decimal. We successfully saw the add instruction in place and we again notice that the flags register (**cpsr**) remains unchanged by this operation.

Part 1: Goals

Next week we will dive into Hacking ADD.

Part 16 - Hacking ADD

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's again review our ADD example below:

```
#add - simple program to demonstrate the add command

.global _start

_start:
    mov r1, #67          @ mov 67 decimal into r1
    mov r2, #53          @ mov 53 decimal into r2
    add r0, r1, r2      @ add r1 + r2 and store in r0

exit:
    mov r7, #1          @ exit syscall
    swi 0
```

Let's debug:

```
pi@pi-alpha:~/code $ gdb -q add
Reading symbols from add...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/add

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:    mov     r1, #67 ; 0x43
    0x00010058 <+4>:    mov     r2, #53 ; 0x35
    0x0001005c <+8>:    add     r0, r1, r2
End of assembler dump.
```

We see the value of **67** decimal is being moved into **r1** below:


```

(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:    mov     r1, #67 ; 0x43
=> 0x00010058 <+4>:    mov     r2, #53 ; 0x35
   0x0001005c <+8>:    add     r0, r1, r2
End of assembler dump.
(gdb) i r
r0                0x0        0
r1                0x43       67
r2                0x0        0
r3                0x0        0
r4                0x0        0
r5                0x0        0
r6                0x0        0
r7                0x0        0
r8                0x0        0
r9                0x0        0
r10               0x0        0
r11               0x0        0
r12               0x0        0
sp                0x7efff3b0    0x7efff3b0
lr                0x0        0
pc                0x10058    0x10058 <_start+4>
cpsr              0x10       16

```

Let's hack! Lets set **r1 = 66!**

```

(gdb) set $r1 = 66
(gdb) i r
r0                0x0        0
r1                0x42       66
r2                0x0        0
r3                0x0        0
r4                0x0        0
r5                0x0        0
r6                0x0        0
r7                0x0        0
r8                0x0        0
r9                0x0        0
r10               0x0        0
r11               0x0        0
r12               0x0        0
sp                0x7efff3b0    0x7efff3b0
lr                0x0        0
pc                0x10058    0x10058 <_start+4>
cpsr              0x10       16

```

Now we see we have hacked the program so when it adds the values it will have a different output. If you remember back to the last lecture, **r0 = 120**. Here we see we have hacked r1 and now the value of **r0** is **119!**

```

(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:    mov     r1, #67 ; 0x43
=>  0x00010058 <+4>:    mov     r2, #53 ; 0x35
   0x0001005c <+8>:    add     r0, r1, r2
End of assembler dump.
(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0          0x77      119
r1          0x42      66
r2          0x35      53
r3          0x0       0
r4          0x0       0
r5          0x0       0
r6          0x0       0
r7          0x0       0
r8          0x0       0
r9          0x0       0
r10         0x0       0
r11         0x0       0
r12         0x0       0
sp          0x7efff3b0   0x7efff3b0
lr          0x0       0
pc          0x10060   0x10060 <exit>
cpsr       0x10      16

```

This is the power of understanding assembly. This is a VERY simple example however with each new series as I have stated we will create a program, debug and hack it.

This combination of instructions will help you to get hands on experience when learning how to have absolute control over an application and in the case of malware reverse engineering gives you the ability to make the binary do exactly what you want!

Next week we will dive into ADDS.

Part 17 - ADDS

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

ADDS is the same as ADD except it sets the flags accordingly in the CPSR.

Let's look at an example to illustrate:

```

#adds - simple program to demonstrate the adds command

.global _start

_start:
    mov r1, #100           @ mov 100 decimal into r1
    mov r2, #4294967295   @ mov 4294967295 decimal into r2

    adds r0, r1, r2       @ adds r1 + r2 and store in r0

exit:
    mov r7, #1            @ exit syscall
    swi 0
  
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**. We then **add r1** and **r2** and place in **r0**.

We see **adds** which sets the flags in the CPSR. We have to remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start to debug and hack this example in the coming tutorials.

You can compile the above by:

```

as -o adc.o adc.s
ld -o adc adc.o
  
```

We need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

Therefore if the value in binary was **0110** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag SET

Carry Flag SET

Overflow Flag NOT Set

Part 1: Goals

It is critical that you compile, debug and hack each exercise in order to understand what is going on here.

Next week we will dive into Debugging ADDS.

Part 18 – Debugging ADDS

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's re-examine our code:

```

adds - simple program to demonstrate the adds command

.global _start

_start:
    mov r1, #100           @ mov 100 decimal into r1
    mov r2, #4294967295    @ mov 4294957295 decimal into r2

    adds r0, r1, r2       @ adds r1 + r2 and store in r0

exit:
    mov r7, #1           @ exit syscall
    swi 0

```

We again **add 100** decimal into **r1**, **4,294,967,295** into **r2**. We then **add r1** and **r2** and place in **r0**.

Lets debug:

```

pi@pi-alpha:~/code $ gdb -q adds
Reading symbols from adds...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adds

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:      mov     r1, #100           ; 0x64
    0x00010058 <+4>:      mvn    r2, #0
    0x0001005c <+8>:      adds   r0, r1, r2
End of assembler dump.
(gdb) i r
r0          0x0          0
r1          0x0          0
r2          0x0          0
r3          0x0          0
r4          0x0          0
r5          0x0          0
r6          0x0          0
r7          0x0          0
r8          0x0          0
r9          0x0          0
r10         0x0          0
r11         0x0          0
r12         0x0          0
sp          0x7efff3b0     0x7efff3b0
lr          0x0          0
pc          0x10054     0x10054 <_start>
cpsr       0x10         16
(gdb)

```

We again see **adds** which sets the flags in the CPSR. We have to remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start

to debug and hack this example in the coming tutorials.

We need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **10 hex**. **10 hex** in binary is **00010000**.

Therefore if the value in binary was **00010000** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag NOT SET

Overflow Flag Set

There is nothing in code above which set the **Overflow Flag** however in it's natural state upon executing this binary it is set.

Lets step through the program:

```
(gdb) si
0x00010058 in _start ()
(gdb) i r
r0          0x0          0
r1          0x64          100
r2          0x0          0
r3          0x0          0
r4          0x0          0
r5          0x0          0
r6          0x0          0
r7          0x0          0
r8          0x0          0
r9          0x0          0
r10         0x0          0
r11         0x0          0
r12         0x0          0
sp          0x7efff3b0     0x7efff3b0
lr          0x0          0
pc          0x10058     0x10058 <_start+4>
cpsr       0x10          16
(gdb)
```

We see **64 hex** or **100 decimal** moved into **r1** as expected. No change in the **CPSR**. Lets step some more.

```

(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x00010060 <+0>:      mov     r7, #1
      0x00010064 <+4>:      svc     0x00000000
End of assembler dump.
(gdb) i r
r0          0x63      99
r1          0x64      100
r2          0xffffffff 4294967295
r3          0x0       0
r4          0x0       0
r5          0x0       0
r6          0x0       0
r7          0x0       0
r8          0x0       0
r9          0x0       0
r10         0x0       0
r11         0x0       0
r12         0x0       0
sp          0x7efff3b0 0x7efff3b0
lr          0x0       0
pc          0x10060 0x10060 <exit>
cpsr       0x20000010 536870928
(gdb)

```

We see the addition that transpires above and notice the value in **r0** is **99 decimal** after **100 decimal** and **4294967295 decimal** were added together. How is that possible? The answer is simple, we overflowed the 32-bit register of **r0** from this addition.

If we examine the **CPSR** we now see **20000010 hex** or **0010 0000 0000 0000 0000 0000 0001 0000 binary**. We only have to focus on the most significant bits which are **0010**:

The value in binary is **0010** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag SET

Overflow Flag NOT Set

We see that the **Carry Flag** was set and the **Overflow Flag** was NOT set. Why is that?

The **Carry Flag** is a flag set when two **unsigned numbers** were added and the result is larger than the register where it is saved. We are dealing with a 32-bit register. We are also dealing with unsigned numbers therefore the **CF** is set and the **OF** was not as the **OF** flag deals with **signed numbers**.

Next week we will dive into Hacking ADDS.

Part 19 – Hacking ADDS

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's once again re-examine our code:

```

adds - simple program to demonstrate the adds command

.global _start

_start:
    mov r1, #100           @ mov 100 decimal into r1
    mov r2, #4294967295    @ mov 4294957295 decimal into r2

    adds r0, r1, r2       @ adds r1 + r2 and store in r0

exit:
    mov r7, #1           @ exit syscall
    swi 0

```

We again **add 100** decimal into **r1**, **4,294,967,295** into **r2**. We then **add r1** and **r2** and place in **r0**.

Lets debug:

```

pi@pi-alpha:~/code $ gdb -q adds
Reading symbols from adds...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adds

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:      mov     r1, #100           ; 0x64
    0x00010058 <+4>:      mvn    r2, #0
    0x0001005c <+8>:      adds   r0, r1, r2
End of assembler dump.
(gdb) i r
r0                0x0          0
r1                0x0          0
r2                0x0          0
r3                0x0          0
r4                0x0          0
r5                0x0          0
r6                0x0          0
r7                0x0          0
r8                0x0          0
r9                0x0          0
r10               0x0          0
r11               0x0          0
r12               0x0          0
sp                0x7efff3b0      0x7efff3b0
lr                0x0          0
pc                0x10054      0x10054 <_start>
cpsr              0x10       16
(gdb)

```

We again see **adds** which sets the flags in the CPSR. We have to remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start

to debug and hack this example in the coming tutorials.

We need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **10 hex**. **10 hex** in binary is **0001**.

Therefore if the value in binary was **0001** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag NOT SET

Overflow Flag Set

Lets take a look if we step again:

```

pi@pi-alpha:~/code $ gdb -q adds
Reading symbols from adds...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adds

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:    mov     r1, #100          ; 0x64
       0x00010058 <+4>:    mvn    r2, #0
       0x0001005c <+8>:    adds   r0, r1, r2
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) disas
Dump of assembler code for function _start:
       0x00010054 <+0>:    mov     r1, #100          ; 0x64
       0x00010058 <+4>:    mvn    r2, #0
=> 0x0001005c <+8>:    adds   r0, r1, r2
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x64    100
r2          0xffffffff 4294967295
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x1005c 0x1005c <_start+8>
cpsr       0x10     16

```

We see **4294967295 decimal** or **0xffffffff** in r2. We know if we step again we will cause the CPSR to change from 0001 to 0010 which means:

The value in binary is **0010** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag SET

Overflow Flag NOT Set

This action sets the carry flag. However lets hack:

```
(gdb) set $r2 = 1
(gdb) i r
r0          0x0      0
r1          0x64     100
r2          0x1      1
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x1005c 0x1005c <_start+8>
cpsr       0x10     16
```

We hacked **r2** and changed the value to **1 decimal** and **0x1 hex**. NOW we know before the **CPSR** went to **0010** last time however now that we hacked this, lets see what happens to the **CPSR** when we step.

```
(gdb) i r
r0          0x65     101
r1          0x64     100
r2          0x1      1
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x10060 0x10060 <exit>
cpsr       0x10     16
```

BAM! We hacked it and see **r0** is **101** and therefore did NOT trigger the carry flag and kept the **CPSR** at **0x10 hex** which means **0001 binary** which means:

Therefore if the value in binary was **0001** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT SET

Carry Flag NOT SET

Overflow Flag Set

It is so important that you understand this lesson in its entirety. If not, please review the last two weeks lessons.

Part 1: Goals

Next week we will dive into ADC.

Part 20 – ADC

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

ADC is the same as ADD except it adds a 1 if the carry flag is set. We need to pay particular attention to the CPSR or Status Register when we work with ADC.

Let's look at an example to illustrate:

```

adc - simple program to demonstrate the adc command

.global _start

_start:
    mov r1, #100           @ mov 100 decimal into r1
    mov r2, #4294967295   @ mov 4294957295 decimal into r2
    mov r3, #100          @ mov 100 decimal into r3
    mov r4, #100          @ mov 100 decimal into r4

    adds r0, r1, r2       @ adds r1 + r2 and store in r0
    adc r5, r3, r4        @ adc r3 + r4 and store in r5

exit:
    mov r7, #1            @ exit syscall
    swi 0
  
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**, **100** decimal into **r3** and **100** decimal into **r4**. We then **add r1** and **r2** and place in **r0** and then **add r3** and **r4** and place into **r5**.

We see **adds** which sets the flags in the CPSR. We have to once again remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start to debug and hack this example in the coming tutorials.

You can compile the above by:

```

as -o adc.o adc.s
ld -o adc adc.o
  
```

I want you to ask yourself what is going to happen when **r3(100 decimal)** is added to **r4(100 decimal)**? What do you think the value of **r5** will be with the above example of setting the flags with the adds result? Think about the first sentence in this tutorial and keep this in mind for the next tutorial.

Next week we will dive into Debugging ADC.

Part 21 – Debugging ADC

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

To recap, ADC is the same as ADD except it adds a 1 if the carry flag is set. We need to pay particular attention to the CPSR or Status Register when we work with ADC.

Let's review our code:

```

adc - simple program to demonstrate the adc command

.global _start
_start:
mov r1, #100           @ mov 100 decimal into r1
mov r2, #4294967295   @ mov 4294967295 decimal into r2
mov r3, #100          @ mov 100 decimal into r3
mov r4, #100          @ mov 100 decimal into r4

adds r0, r1, r2        @ adds r1 + r2 and store in r0
adc r5, r3, r4         @ adc r3 + r4 and store in r5

exit:
mov r7, #1             @ exit syscall
swi 0

```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**, **100** decimal into **r3** and **100** decimal into **r4**. We then **add r1** and **r2** and place in **r0** and then **add r3** and **r4** and place into **r5**.

We see **adds** which sets the flags in the CPSR. We have to once again remember when we debug in GDB, the value of the CPSR is in hex. In order to see what flags are set, we must convert the hex to binary. This will make sense as we start to debug and hack this example in the coming tutorials.

Last week I raised a question where I wanted you to ask yourself what is going to happen when **r3(100 decimal)** is added to **r4(100 decimal)**? What do you think the value of **r5** will be with the above example of setting the flags with the adds result?

```

pi@pi-alpha:~/code $ gdb -q adc
Reading symbols from adc...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adc

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r1, #100      ; 0x64
      0x00010058 <+4>:   mvn    r2, #0
      0x0001005c <+8>:   mov    r3, #100      ; 0x64
      0x00010060 <+12>:  mov    r4, #100      ; 0x64
      0x00010064 <+16>:  adds  r0, r1, r2
      0x00010068 <+20>:  adc   r5, r3, r4
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in _start ()
(gdb) si
0x00010064 in _start ()
(gdb) si
0x00010068 in _start ()
(gdb) si
0x0001006c in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x0001006c <+0>:   mov    r7, #1
      0x00010070 <+4>:   svc   0x00000000
End of assembler dump.
(gdb) i r
r0          0x63      99
r1          0x64     100
r2          0xffffffff 4294967295
r3          0x64     100
r4          0x64     100
r5          0xc9     201
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x1006c 0x1006c <exit>
cpsr       0x20000010 536870928
(gdb)

```

Ok so we add **100 decimal** and **100 decimal** together in **r3** and **r4** and we get **201 decimal** in **r5**! Is something broken? ADC is the same as ADD except it adds a 1 if the carry flag is set. Therefore we get the extra 1 in **r5**.

We again need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **20000010 hex**. The most significant bits of **20000010 hex** in binary is **0010**.

Therefore if the value in binary was **0010** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT Set

Carry Flag SET

Overflow Flag NOT Set

As we can clearly see the carry flag was set. I hope you can digest and understand each of these very simple operations and how they have an effect on the CPSR.

Next week we will dive into Hacking ADC.

Part 22 – Hacking ADC

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

To recap again, ADC is the same as ADD except it adds a 1 if the carry flag is set. We need to pay particular attention to the CPSR or Status Register when we work with ADC.

Let's again review our code:

```

adc - simple program to demonstrate the adc command

.global _start

_start:
    mov r1, #100           @ mov 100 decimal into r1
    mov r2, #4294967295    @ mov 4294957295 decimal into r2
    mov r3, #100           @ mov 100 decimal into r3
    mov r4, #100           @ mov 100 decimal into r4

    adds r0, r1, r2        @ adds r1 + r2 and store in r0
    adc r5, r3, r4         @ adc r3 + r4 and store in r5

exit:
    mov r7, #1             @ exit syscall
    swi 0
  
```

We **add 100** decimal into **r1**, **4,294,967,295** into **r2**, **100** decimal into **r3** and **100** decimal into **r4**. We then **add r1** and **r2** and place in **r0** and then **add r3** and **r4** and place into **r5**.

```

pi@pi-alpha:~/code $ gdb -q adc
Reading symbols from adc...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/adc

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:    mov     r1, #100      ; 0x64
    0x00010058 <+4>:    mvn    r2, #0
    0x0001005c <+8>:    mov    r3, #100     ; 0x64
    0x00010060 <+12>:   mov    r4, #100    ; 0x64
    0x00010064 <+16>:   adds   r0, r1, r2
    0x00010068 <+20>:   adc    r5, r3, r4
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) disas
Dump of assembler code for function _start:
    0x00010054 <+0>:    mov     r1, #100      ; 0x64
=> 0x00010058 <+4>:    mvn    r2, #0
    0x0001005c <+8>:    mov    r3, #100     ; 0x64
    0x00010060 <+12>:   mov    r4, #100    ; 0x64
    0x00010064 <+16>:   adds   r0, r1, r2
    0x00010068 <+20>:   adc    r5, r3, r4
End of assembler dump.
(gdb) si
  
```

We run the program and step to where we move **4,294,967,295** into **r2**. Let's hack that value in **r2** and change it to **100 decimal**.

```

0x0001005c in _start ()
(gdb) set $r2 = 100
(gdb) disas
Dump of assembler code for function _start:
   0x00010054 <+0>:   mov     r1, #100      ; 0x64
   0x00010058 <+4>:   mvn    r2, #0
=> 0x0001005c <+8>:   mov     r3, #100     ; 0x64
   0x00010060 <+12>:  mov     r4, #100     ; 0x64
   0x00010064 <+16>:  adds   r0, r1, r2
   0x00010068 <+20>:  adc    r5, r3, r4
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x64    100
r2          0x64    100
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0 0x7efff3b0
lr          0x0      0
pc          0x1005c 0x1005c <_start+8>
cpsr       0x10     16

```

Let's step a few more times:

```

(gdb) si
0x00010060 in _start ()
(gdb) si
0x00010064 in _start ()
(gdb) si
0x00010068 in _start ()
(gdb) si
0x0001006c in exit ()
(gdb) disas
Dump of assembler code for function exit:
=> 0x0001006c <+0>:      mov     r7, #1
      0x00010070 <+4>:      svc     0x00000000
End of assembler dump.
(gdb) i r
r0          0xc8      200
r1          0x64      100
r2          0x64      100
r3          0x64      100
r4          0x64      100
r5          0xc8      200
r6          0x0       0
r7          0x0       0
r8          0x0       0
r9          0x0       0
r10         0x0       0
r11         0x0       0
r12         0x0       0
sp          0x7efff3b0    0x7efff3b0
lr          0x0       0
pc          0x1006c   0x1006c <exit>
cpsr       0x10      16

```

Ok so now we add **100 decimal** and **100 decimal** together in **r3** and **r4** and we get **200 decimal** in **r5**! Do you remember last week when we had **201**? Let's examine the CPSR below.

We again need to remember that bits 31, 20, 29 and 28 in the CPSR indicate the following:

bit 31 - N = Negative Flag

bit 30 - Z = Zero Flag

bit 29 - C = Carry Flag

bit 28 - V = Overflow Flag

We see the **CPSR** at **10 hex**. The most significant bits of **10 hex** in binary is **0001**.

Therefore if the value in binary was **0001** of bit 31, 30, 29 and 28 (**NZCV**) that would mean:

Negative Flag NOT Set

Zero Flag NOT Set

Carry Flag NOT SET

Overflow Flag Set

Part 1: Goals

As we can clearly see the carry flag was NOT set. I hope you can digest and understand each of these very simple operations and how they have an effect on the CPSR. Please take the time and review last weeks lesson for comparison.

Next week we will dive into SUB.

Part 23 – SUB

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Subtraction in ARM has four instructions which are SUB, SBC, RSB and RSC. We will start today with SUB.

Please keep in mind when you add the S suffix on the end of each such as SUBS, SBSC, RSBS, RSCS, it will affect the flags. We have spent enough time on flags in the prior lessons so that you should have a firm grasp on this now.

Let's examine an example of SUB:

```
sub - simple program to demonstrate the sub command

.global _start
_start:
    mov r1, #67          @ mov 67 decimal into r1
    mov r2, #53          @ mov 53 decimal into r2
    sub r0, r1, r2       @ sub r1 - r2 and store in r0

exit:
    mov r7, #1           @ exit syscall
    swi 0
```

To compile:

```
as -o sub.o sub.s
ld -o sub sub.o
```

We simply take **67 decimal** and move into **r1** and **53 decimal** and move into **r2** and subtract $r1 - r2$ and put the result in **r0**.

Next week we will dive into SUB debugging.

Part 24 – Debugging SUB

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

As stated, subtraction in ARM has four instructions which are SUB, SBC, RSB and RSC. We will start today with SUB.

Please keep in mind when you add the S suffix on the end of each such as SUBS, SBCS, RSBS, RSCS, it will affect the flags. We have spent enough time on flags in the prior lessons so that you should have a firm grasp on this now.

Let's re-examine our example of SUB:

```
sub - simple program to demonstrate the sub command

.global _start
_start:
    mov r1, #67          @ mov 67 decimal into r1
    mov r2, #53          @ mov 53 decimal into r2
    sub r0, r1, r2       @ sub r1 - r2 and store in r0

exit:
    mov r7, #1           @ exit syscall
    swi 0
```

We simply take **67 decimal** and move into **r1** and **53 decimal** and move into **r2** and subtract $r1 - r2$ and put the result in **r0**.

Let's debug.

```

pi@pi-alpha:~/code $ gdb -q sub
Reading symbols from sub...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/sub

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r1, #67 ; 0x43
      0x00010058 <+4>:   mov     r2, #53 ; 0x35
      0x0001005c <+8>:   sub     r0, r1, r2
End of assembler dump.
(gdb) i r
r0             0x0      0
r1             0x0      0
r2             0x0      0
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0   0x7efff3b0
lr             0x0      0
pc             0x10054  0x10054 <_start>
cpsr          0x10     16

```

As we can see the registers are clear. Lets step through and see what the value of **r0** becomes.

```

(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0             0xe      14
r1             0x43     67
r2             0x35     53
r3             0x0      0
r4             0x0      0
r5             0x0      0
r6             0x0      0
r7             0x0      0
r8             0x0      0
r9             0x0      0
r10            0x0      0
r11            0x0      0
r12            0x0      0
sp             0x7efff3b0   0x7efff3b0
lr             0x0      0
pc             0x10060  0x10060 <exit>
cpsr          0x10     16

```

As you can see above **r0** now has **decimal 14** which works as expected.

Next week we will dive into SUB hacking.

Part 25 – Hacking SUB

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

As stated, subtraction in ARM has four instructions which are SUB, SBC, RSB and RSC. We will start today with SUB.

Please keep in mind when you add the S suffix on the end of each such as SUBS, SBCS, RSBS, RSCS, it will affect the flags. We have spent enough time on flags in the prior lessons so that you should have a firm grasp on this now.

Let's re-examine our example of SUB:

```
sub - simple program to demonstrate the sub command

.global _start
_start:
    mov r1, #67          @ mov 67 decimal into r1
    mov r2, #53          @ mov 53 decimal into r2
    sub r0, r1, r2       @ sub r1 - r2 and store in r0

exit:
    mov r7, #1           @ exit syscall
    swi 0
```

We simply take **67 decimal** and move into **r1** and **53 decimal** and move into **r2** and subtract $r1 - r2$ and put the result in **r0**.

Let's hack.


```

pi@pi-alpha:~/code $ gdb -q sub
Reading symbols from sub...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/sub

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r1, #67 ; 0x43
      0x00010058 <+4>:   mov     r2, #53 ; 0x35
      0x0001005c <+8>:   sub     r0, r1, r2
End of assembler dump.
(gdb) i r
r0          0x0      0
r1          0x0      0
r2          0x0      0
r3          0x0      0
r4          0x0      0
r5          0x0      0
r6          0x0      0
r7          0x0      0
r8          0x0      0
r9          0x0      0
r10         0x0      0
r11         0x0      0
r12         0x0      0
sp          0x7efff3b0   0x7efff3b0
lr          0x0      0
pc          0x10054   0x10054 <_start>
cpsr       0x10     16

```

As we can see the registers are clear. Lets step through and see what the value of **r0** becomes when we do a little hacking.

```

pi@pi-alpha:~/code $ gdb sub -q
Reading symbols from sub...(no debugging symbols found)...done.
(gdb) b _start
Breakpoint 1 at 0x10054
(gdb) r
Starting program: /home/pi/code/sub

Breakpoint 1, 0x00010054 in _start ()
(gdb) disas
Dump of assembler code for function _start:
=> 0x00010054 <+0>:   mov     r1, #67 ; 0x43
       0x00010058 <+4>:   mov     r2, #53 ; 0x35
       0x0001005c <+8>:   sub     r0, r1, r2
End of assembler dump.
(gdb) si
0x00010058 in _start ()
(gdb) si
0x0001005c in _start ()
(gdb) set $r2 = 50
(gdb) si
0x00010060 in exit ()
(gdb) i r
r0          0x11      17
r1          0x43      67
r2          0x32      50
r3          0x0       0
r4          0x0       0
r5          0x0       0
r6          0x0       0
r7          0x0       0
r8          0x0       0
r9          0x0       0
r10         0x0       0
r11         0x0       0
r12         0x0       0
sp          0x7efff3b0   0x7efff3b0
lr          0x0       0
pc          0x10060  0x10060 <exit>
cpsr       0x10      16
(gdb) █

```

As you can see above **r0** now has **decimal 17** which works as expected as we hacked the value of **r2** to **decimal 50** instead of **decimal 53**.

I want to thank you all for taking this journey to learn ARM Assembly. This is the end of the series as I encourage you all to take what you have learned and continue to work through the ARM instruction set and continue your progress.

This tutorial's purpose was to provide you a solid foundation in ARM Assembly and I believe we have done that. Thank you all and I look forward to seeing you all become future Reverse Engineers!

The 32-bit ARM Architecture (Part 2)

Let's dive in rightaway!

Part 1 – The Meaning Of Life Part 2

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Welcome to the ARM Reverse Engineering tutorial. This is the third tutorial series that I have done focusing on Assembly Language and Reverse Engineering.

The first series was on x86 Assembly and the second was on ARM Assembly. This series will be an expansion series on ARM focusing on ARM Reverse Engineering so rather than create programs directly in Assembly alone and then Reverse Engineer the binary in Assembly we will work with Assembly and C together and Reverse Engineer in Assembly so that you will get a flavor for a real-world series of applications and what it looks like disassembled.

We will not be working with GUI tools such as IDA Pro as we will be working with GDB in CLI shell. We will not be working in a traditional lab environment where we are going to put a binary into a debugger rather we are going to SSH into the ARM device and actually attach to a running process (PID) and Reverse Engineer the process as it is running.

The first 13 weeks will be an exact review of the ARM Assembly series as it is critical that we re-examine these concepts so that we have a very firm grasp when it comes time to reverse our binaries.

I wanted to bring back the original quote below before we get started...

“So if I go to college and learn Java will I make a million dollars and have nice things?”

I felt it necessary to start out this tutorial series with such a statement. This is NOT an attack on Java as I have used Java in Android Development, Spring and JavaEE. In today's Agile environment, rapid-development is reality. With the increased challenges in both the commercial market and the government sector, software development will continue to focus on more robust libraries that will do more with less. React, Python, Java, C# and the like will continue to grow not shrink as the race for project completion augments with each passing second of time.

Like it or not, hardware is getting smaller and smaller and the trend is going from CISC to RISC. A CISC is your typical x86/x64 computer with a complex series of instructions. CISC computers will always exist however with the trend going toward cloud computing and the fact that RISC machines with a reduced instruction set are so enormously powerful today, they are the obvious choice for consumption.

How many cell phones do you think exist on earth today? Most of them are RISC machines. How many of you have a Smart TV or Amazon Echo or any number of devices considered part of the IOT or Internet Of Things? Each of these devices have one thing in common – they are RISC and all are primarily ARM based.

ARM is an advanced RISC machine. Compared to the very complex architecture of a CISC, most ARM systems today are what is referred to as a SoC or system on chip which is an integrated circuit which has all of the components of a computer and electronic system on a single chip. This includes RF functionality as well. These low-power embedded devices can run versions of Windows, Linux and many other advanced operating systems.

“Well who cares about ARM, you can call it anything you want, I know Java and that’s all I need to know cause when I program it works everywhere so I don’t have to worry about anything under the hood.”

I again just want you to reflect on the above statement for a brief moment. As every day continues to pass, more and more systems are becoming vulnerable to attack and compromise. Taking the time to understand what is going on under the hood can only help to curb this unfortunate reality.

This series will focus on ARM Reverse Engineering. We will work with a Raspberry Pi 3 which contains the Broadcom BCM2837 SoC with a 4x ARM Cortex-A53, 1.2GHz CPU and 1 GB LPDDR2 RAM. We will work with the Raspbian Jessie, Linux-based operating system. If you don’t own a Raspberry Pi 3, they are usually available for \$35 on Amazon or any number of retailers. If you would like to learn more visit <https://www.raspberrypi.org>.

We will work solely in the terminal so no pretty pictures and graphics as we are keeping it to the hardcore bare-bones utilizing the GNU toolkit to compile and debug our code base.

Next week we will dive into the binary number system and compare and contrast it with decimal and hexadecimal so we have a proper framework of understanding to move forward.

Part 11 - Firmware Boot Procedures

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's take a moment to talk about what happens when we first power on our Raspberry Pi device.

As soon as the Pi receives power, the graphics processor is the first thing to run as the processor is held in a reset state to which the GPU starts executing code. The ROM reads from the SD card and reads **bootcode.bin** to which gets loaded into memory in C2 cache and turns on the rest of the RAM to which **start.elf** then loads.

The **start.elf** is an OS for the graphics processor and reads **config.txt** to which you can mod. The **kernel.img** then gets loaded into **0x8000** in memory which is the Linux kernel.

Once loaded, **kernel.img** turns on the CPU and starts running at **0x8000** in memory.

If we wanted, we could create our own **kernel.img** to which we can hard code machine code into a file and replace the original image and then reboot. Keep in mind the ARM word size is 32 bit long which go from bit 0 to 31.

As stated, when **kernel.img** is loaded the first byte, which is 8-bits, is loaded into address **0x800**.

Lets open up a hex editor and write the following:

FE FF FF EA

Save the file as **kernel.img** and reboot.

"Ok nothing happens, this sucks!"

Actually something did happen, you created your first bare-metal firmware! Time to break out the champagne!

When the Pi boots, the below code when it reached **kernel.img** loads the following:

FE FF FF EA

@ address 0x8000, 0xfe gets loaded.

@ address 0x8001, 0xff gets loaded.

@ address 0x8002, 0xff gets loaded.

@ address 0x8003, 0xea gets loaded.

"So what the hell is really going on?"

This set of commands simply executes an infinite loop.

Review the datasheet:

<https://www.raspberrypi.org/wp-content/uploads/2012/02/BCM2835-ARM-Peripherals.pdf>

The above code has 3 parts to it:

- 1) Conditional – Set To Always
- 2) Op Code – Branch
- 3) Offset – How Far To Move Within The Current Location

Condition – bits 31-28: 0xe or 1110

Op Code – bits 27-24: 0xa or 1010

Offset – bits 23-0 -2

I know this may be a lot to wrap your mind around however it is critical that you take the time and read the datasheet linked above. Do not cut corners if you truly have the passion to understand the above. READ THE DATASHEET!

I will go through painstaking efforts to break everything down step-by-step however there are exercises like the above that I am asking you to review the datasheet above so you learn how to better understand where to look when you are stuck on a particular routine or set of machine code. This is one of those times I ask you to please read and research the datasheet above!

“I’m bored! Why the hell does this crap matter?”

Glad you asked! The single most dangerous malware on planet earth today is that of the root-kit variety. If you do not have a basic understanding of the above, you will never begin to even understand what a root-kit is as you progress in your understanding.

Anyone can simply replace the **kernel.img** file with their own hacked version and you can have total control over the entire process from boot.

Next week we will dive into the Von Neumann Architecture.

Part 14 - Hello World

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Today we begin our journey into the world of C++ and gaining a better understanding of how C++ interacts with our ARM processor.

The prior lessons in this series focus on the basics of the ARM processor and touch upon its architecture and how everything ultimately translates down to Assembly Language and then ultimately opcodes into machine language.

We start with our first program in C++ which is our "Hello World" program. Let's dive in and break each line down step-by-step and see how this language works. We will call this **example1.cpp** and save it to our device.

```
#include <iostream>

int main(void) {
    std::cout << "Hello World" std::endl;

    return 0;
}
```

```
1 #include <iostream>
2
3 int main(void) {
4     std::cout << "Hello World!" << std::endl;
5
6     return 0;
7 }
```

To compile this we simply type:

```
g++ example1.cpp -o example1
```

We simply then type:

```
./example1
```

```
pi@pi-alpha:~/code $ g++ example1.cpp -o example1
pi@pi-alpha:~/code $ ./example1
Hello World!
```

SUCCESS! We see "Hello World" printed to the standard output or terminal!

Lets break it down line by line:

#include <iostream> is referred to as a preprocessor statement. These preprocessor statements happen just before the compilation of the rest of the code. The **#include** keyword will find a file called **iostream** and take all of the contents of that file and paste it into the existing code we just created. These files are also called header files.

We call **iostream** because we need a declaration for a function called **cout** and **endl**. The **cout** function allows us to print text to the standard output or terminal and the **endl** function creates a new line after the text has been displayed.

The main section which is of type integer is the entry point into the main application or binary. You will notice a **void** inside the **()** which indicates that it does not have any parameters which will be passed into the function.

The **std** indicates a namespace which is quite simply a mechanism to organize code into logical groups in order to prevent name collisions when you are dealing with multiple libraries.

You will see many examples where they declare a using namespace std; however I will NEVER utilize this approach as it can cause naming collisions in more complex applications.

The **<<** operator is referred to as an overloaded operator. They are essentially a function very similar to **printf** in the C language. We are simply moving the “**Hello World**” string into the **cout** function through the use of the **<<** overloaded operator. We then push the **endl** which creates a new line to the console.

The final line is the return 0. Since our main function is of type int, we have to return something. In C++ 11 there is no need for this in the main function however is required for every other function. I will stick to tradition and simply include it.

The next stage is that we compile the file. The first thing that occurs is the entire contents of the **iostream** header goes into the source file as we discussed. The compile process is where the C++ code gets translated into machine code. The next stage of compilation occurs when the rest of the lines of our existing code are parsed through. Essentially we have all of the contents of **iostream** into a new file and then all of the contents of our existing file added to a single file.

Compiling takes our text file the **cpp** file and converts it into an intermediate format called an **obj** file. An abstract syntax tree is created which is a conversion of constant data, variables and instructions.

Once the tree is created the code is generated. This means we now have machine code that our ARM CPU will execute. Every **cpp** file (translation units) which will have its own respective **obj** file associated with it.

Linking takes our **obj** files, our compiled files, in addition to the C++ Standard Library and finds where each symbol and function is and link them all together into one executable.

The concepts above may appear a bit confusing if you are new to programming however as you code and compile and later debug and hack in Assembly Language it will all become very clear and you will learn to master the processor.

Part 1: Goals

Next week we will dive into Debugging Hello World.

Part 15 - Debugging Hello World

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our code from last week.

```

1 #include <iostream>
2
3 int main(void) {
4     std::cout << "Hello World!" << std::endl;
5
6     return 0;
7 }

```

Let's debug! Let's fire up GDB which is the GNU Debugger to which we will break down the C++ binary and step through it line-by-line in ARM Assembly.

```

pi@pi-alpha:~/code $ gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:   push   {r11, lr}
0x00010720 <+4>:   add    r11, sp, #4
0x00010724 <+8>:   ldr    r0, [pc, #32] ; 0x1074c <main+48>
0x00010728 <+12>:  ldr    r1, [pc, #32] ; 0x10750 <main+52>
0x0001072c <+16>:  bl     0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
0x00010730 <+20>:  mov    r3, r0
0x00010734 <+24>:  mov    r0, r3
0x00010738 <+28>:  ldr    r1, [pc, #20] ; 0x10754 <main+56>
0x0001073c <+32>:  bl     0x105dc <_ZNSt8ios_base4InitD1Ev+36>
0x00010740 <+36>:  mov    r3, #0
0x00010744 <+40>:  mov    r0, r3
0x00010748 <+44>:  pop    {r11, pc}
0x0001074c <+48>:  ldrdeq r0, [r2], -r0 ; <UNPREDICTABLE>
0x00010750 <+52>:  andeq  r0, r1, r8, asr #16
0x00010754 <+56>:  andeq  r0, r1, r8, ror #11
End of assembler dump.

```

This is the ARM disassembly that we are seeing. No matter what language you program in, it ultimately will go down to this level.

This might be a bit scary to you if you did not take my prior course on ARM Assembly. If you need to do a refresher, please link back to that series.

You are probably asking yourself why we are not debugging with the original source code and seeing how it matches nicely to the assembly. The answer is when you are a professional Reverse Engineer, you do not get the luxury of seeing source code when you are reversing binaries.

This is a childishly simple example and we will continue through the series with very simple examples so that you can learn effective techniques. We are using a text-based debugger here so that you fully understand what is going on and to

also get some training if you had to ever attach yourself to a running process inside a foreign machine you will know how to properly debug or hack.

I will focus SOLELY on this method rather than using a nice graphical debugger like IDA or the like so that you are able to manipulate at a very low-level.

We start with loading the link register into **r11** and adding **4** to the stack pointer and then adding it to **r11**. This is simply a routine which will allow the binary to preserve the link register and setting up space on the stack.

We notice memory address **0x10750** being loaded from memory to the register **r1**. Let's do a string examination and see what is located at that address.

```
(gdb) x/s *0x10750
0x10848: "Hello World!"
```

Voila! We see our string. **"Hello World!"** located at that memory address.

Let's set a breakpoint at **main+16**.

```
(gdb) b *main+16
Breakpoint 2 at 0x1072c
(gdb) s
Single stepping until exit from function main,
which has no line number information.

Breakpoint 2, 0x0001072c in main ()
(gdb) disas
Dump of assembler code for function main:
   0x0001071c <+0>:  push   {r11, lr}
   0x00010720 <+4>:  add    r11, sp, #4
   0x00010724 <+8>:  ldr    r0, [pc, #32] ; 0x1074c <main+48>
   0x00010728 <+12>: ldr    r1, [pc, #32] ; 0x10750 <main+52>
=>  0x0001072c <+16>:  bl     0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:  mov    r3, r0
   0x00010734 <+24>:  mov    r0, r3
   0x00010738 <+28>:  ldr    r1, [pc, #20] ; 0x10754 <main+56>
   0x0001073c <+32>:  bl     0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:  mov    r3, #0
   0x00010744 <+40>:  mov    r0, r3
   0x00010748 <+44>:  pop    {r11, pc}
   0x0001074c <+48>:  ldrdeq r0, [r2], -r0 ; <UNPREDICTABLE>
   0x00010750 <+52>:  andeq  r0, r1, r8, asr #16
   0x00010754 <+56>:  andeq  r0, r1, r8, ror #11
End of assembler dump.
```

Let's take a look at our register values.

```
(gdb) i r
r0      0x209d0  133584
r1      0x10848  67656
r2      0x7efff39c  2130703260
r3      0x1071c  67356
r4      0x0      0
r5      0x0      0
r6      0x105f4  67060
r7      0x0      0
r8      0x0      0
r9      0x0      0
r10     0x76fff000  1996484608
r11     0x7efff23c  2130702908
r12     0x76e1f000  1994518528
sp      0x7efff238  0x7efff238
lr      0x76cfa294  1993319060
pc      0x1072c  0x1072c <main+16>
cpsr    0x60000010  1610612752
(gdb) x/s $r1
0x10848: "Hello World!"
```

Let's now take a look at what is inside the **r1** register and then step through the binary.

```
(gdb) x/s $r1
0x10848:      "Hello World!"
(gdb) s
Single stepping until exit from function main,
which has no line number information.
Hello World!
__libc_start_main (main=0x7efff394, argc=1994518528,
  argv=0x76cfa294 <__libc_start_main+276>, init=<optimized out>,
  fini=0x10838 <__libc_csu_fini>, rtdl_fini=0x76fdfa14 <_dl_fini>,
  stack_end=0x7efff394) at libc-start.c:321
321      libc-start.c: No such file or directory.
```

We see the **“Hello World!”** string now residing inside of **r1** which resides at memory address **0x10848**. Finally let's continue through the binary.

```
(gdb) c
Continuing.
[Inferior 1 (process 1069) exited normally]
```

Understanding assembly and step-by-step debugging allows you to have complete and ultimate control over any binary! More complex binaries can cause you hours, days or weeks to truly Reverse Engineer however the techniques are the same just more time consuming.

Reverse Engineering is the most sophisticated form of analysis in advanced Computer Engineering. There are many tools that a professional Reverse Engineer uses however each of those tools have a usage and purpose however this technique is the most sophisticated and comprehensive.

Next week we will dive into Hacking Hello World.

Part 16 - Hacking Hello World

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our code from two weeks ago.

```

1 #include <iostream>
2
3 int main(void) {
4     std::cout << "Hello World!" << std::endl;
5
6     return 0;
7 }

```

Let's debug once again.

```

pi@pi-alpha:~/code $ gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:    push    {r11, lr}
   0x00010720 <+4>:    add     r11, sp, #4
   0x00010724 <+8>:    ldr    r0, [pc, #32] ; 0x1074c <main+48>
   0x00010728 <+12>:   ldr    r1, [pc, #32] ; 0x10750 <main+52>
   0x0001072c <+16>:   bl     0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:   mov    r3, r0
   0x00010734 <+24>:   mov    r0, r3
   0x00010738 <+28>:   ldr    r1, [pc, #20] ; 0x10754 <main+56>
   0x0001073c <+32>:   bl     0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:   mov    r3, #0
   0x00010744 <+40>:   mov    r0, r3
   0x00010748 <+44>:   pop    {r11, pc}
   0x0001074c <+48>:   ldrdeq r0, [r2], -r0 ; <UNPREDICTABLE>
   0x00010750 <+52>:   andeq  r0, r1, r8, asr #16
   0x00010754 <+56>:   andeq  r0, r1, r8, ror #11
End of assembler dump.

```

Let's once again examine the contents of the string at memory address **0x10750** and continue through the execution of the program.

```

(gdb) x/s *0x10750
0x10848:    "Hello World!"
(gdb) c
Continuing.
Hello World!
[Inferior 1 (process 1038) exited normally]

```

As you can see it holds the **"Hello World!"** string and when we continue through it echo's back to the terminal as such.

Let's hack! Let's now overwrite the value inside of the memory address with the string, **"Hacked World!"** and continue execution.

```
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) x/s *0x10750
0x10848:      "Hello World!"
(gdb) set *0x10750 = "Hacked World!"
(gdb) c
Continuing.
Hacked World!
[Inferior 1 (process 1045) exited normally]
```

Woohoo! Our first hack! As you can see as you understand Assembly you have ABSOLUTE control over the entire binary no matter what language it is written in. In this very simple example we were able to hack the value inside the memory address of **0x10750** to which when executed it echoed, "**Hacked World!**" to the terminal or standard output.

Let's again run the binary and do a disassembly.

```
(gdb) r
Starting program: /home/pi/code/example1

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:      push   {r11, lr}
0x00010720 <+4>:      add    r11, sp, #4
0x00010724 <+8>:      ldr   r0, [pc, #32] ; 0x1074c <main+48>
0x00010728 <+12>:     ldr   r1, [pc, #32] ; 0x10750 <main+52>
0x0001072c <+16>:     bl    0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
0x00010730 <+20>:     mov   r3, r0
0x00010734 <+24>:     mov   r0, r3
0x00010738 <+28>:     ldr   r1, [pc, #20] ; 0x10754 <main+56>
0x0001073c <+32>:     bl    0x105dc <_ZNSt8ios_base4InitD1Ev+36>
0x00010740 <+36>:     mov   r3, #0
0x00010744 <+40>:     mov   r0, r3
0x00010748 <+44>:     pop   {r11, pc}
0x0001074c <+48>:     ldrdeq r0, [r2], -r0 ; <UNPREDICTABLE>
0x00010750 <+52>:     andeq r0, r1, r8, asr #16
0x00010754 <+56>:     andeq r0, r1, r8, ror #11
End of assembler dump.
```

Let's now do the same procedure however lets **si** 3x and examine the string inside of **r1**. We see that it contains, "**Hello World!**" as it has been successfully **ldr** (load from memory into the register) at **main+12**.

Let's now set **r1** to "**Hacked World!**" and continue execution. As you can see we now hacked it coming out of the register rather than in memory. You can clearly begin to see there are a number of ways to hack anything and here is a simple example of two such ways.

```

(gdb) si
0x00010720 in main ()
(gdb) si
0x00010724 in main ()
(gdb) si
0x00010728 in main ()
(gdb) disas
Dump of assembler code for function main:
   0x0001071c <+0>:   push   {r11, lr}
   0x00010720 <+4>:   add    r11, sp, #4
   0x00010724 <+8>:   ldr    r0, [pc, #32] ; 0x1074c <main+48>
=> 0x00010728 <+12>:  ldr    r1, [pc, #32] ; 0x10750 <main+52>
   0x0001072c <+16>:  bl     0x105c4 <_ZNSt8ios_base4InitD1Ev+12>
   0x00010730 <+20>:  mov    r3, r0
   0x00010734 <+24>:  mov    r0, r3
   0x00010738 <+28>:  ldr    r1, [pc, #20] ; 0x10754 <main+56>
   0x0001073c <+32>:  bl     0x105dc <_ZNSt8ios_base4InitD1Ev+36>
   0x00010740 <+36>:  mov    r3, #0
   0x00010744 <+40>:  mov    r0, r3
   0x00010748 <+44>:  pop    {r11, pc}
   0x0001074c <+48>:  ldrdeq r0, [r2], -r0 ; <UNPREDICTABLE>
   0x00010750 <+52>:  andeq  r0, r1, r8, asr #16
   0x00010754 <+56>:  andeq  r0, r1, r8, ror #11
End of assembler dump.
(gdb) si
0x0001072c in main ()
(gdb) x/s $r1
0x10848: "Hello World!"
(gdb) set $r1 = "Hacked World!"
(gdb) c
Continuing.
Hacked World!
[Inferior 1 (process 1050) exited normally]

```

Reverse Engineering is all about understanding how a program executes and hijacking execution flow and changing values to suit our purpose! Today you took your first step into this amazing journey!

Next week we will dive into constants.

Part 17 - Constants

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

So far we have created, debugged and hacked a simple string echo to the standard terminal. We will expand upon that example by adding a constant.

A constant in C++ is a value that will not change throughout program execution (unless hacked). It is used such that you have a declaration early in the code so that if your future program architecture ever changes you can redefine the constant in one place rather than having to update code all through your code base.

It is standard practice to code our constants in all CAPS so that when we see it referenced somewhere in the code we know that value is a constant.

We start with our second program in C++ which is our “Constant” program. Let’s dive in and break each line down step-by-step and see how this language works. We will call this **example2.cpp** and save it to our device.

```
#include <iostream>

int main(void) {
    const int YEAR = 2017;

    std::cout << YEAR << std::endl;

    return 0;
}
```

```
1 #include <iostream>
2
3 int main(void) {
4     const int YEAR = 2017;
5
6     std::cout << YEAR << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

```
g++ example2.cpp -o example2
```

We simply then type:

```
./example2
```

```
pi@pi-alpha:~/code $ g++ example2.cpp -o example2
pi@pi-alpha:~/code $ ./example2
2017
```

SUCCESS! We see “2017” printed to the standard output or terminal!

Part 1: Goals

Let's break it down:

We utilize the **const** keyword to indicate a constant to which we assign it the integer value of 2017.

We then utilize the **cout** function to print it to the standard output or terminal and add a new line with the **endl** function.

That's it! Very simple.

Next week we will dive into Debugging Constants.

Part 18 – Debugging Constants

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review last week's code.

```

1 #include <iostream>
2
3 int main(void) {
4     const int YEAR = 2017;
5
6     std::cout << YEAR << std::endl;
7
8     return 0;
9 }

```

Let's debug!

```

pi@pi-alpha:~/code $ gdb -q example2
Reading symbols from example2...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example2

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:  push   {r11, lr}
    0x000106f4 <+4>:  add    r11, sp, #4
    0x000106f8 <+8>:  sub    sp, sp, #8
    0x000106fc <+12>: ldr    r3, [pc, #44] ; 0x10730 <main+64>
    0x00010700 <+16>: str    r3, [r11, #-8]
    0x00010704 <+20>: ldr    r0, [pc, #40] ; 0x10734 <main+68>
    0x00010708 <+24>: ldr    r1, [pc, #32] ; 0x10730 <main+64>
    0x0001070c <+28>: bl     0x1055c
    0x00010710 <+32>: mov    r3, r0
    0x00010714 <+36>: mov    r0, r3
    0x00010718 <+40>: ldr    r1, [pc, #24] ; 0x10738 <main+72>
    0x0001071c <+44>: bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
    0x00010720 <+48>: mov    r3, #0
    0x00010724 <+52>: mov    r0, r3
    0x00010728 <+56>: sub    sp, r11, #4
    0x0001072c <+60>: pop    {r11, pc}
    0x00010730 <+64>: andeq  r0, r0, r1, ror #15
    0x00010734 <+68>: andeq  r0, r2, r0, lsr #19
    0x00010738 <+72>:                ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print *0x10730
$1 = 2017

```

As we can see the value in the memory address **0x10730** is equal to **2017**. Let's continue and watch the value print to the standard output (terminal) as it did last week when we ran it.

```

(gdb) c
Continuing.
2017
[Inferior 1 (process 1008) exited normally]

```

We can see very clearly that we move the value from memory into **r1** and then we branch to our **cout** function to print to the terminal. At this stage you should feel a little more comfortable with understanding what the assembly is doing above.

Next week we will dive into Hacking Constants.

Part 19 – Hacking Constants

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our original code.

```

1 #include <iostream>
2
3 int main(void) {
4     const int YEAR = 2017;
5
6     std::cout << YEAR << std::endl;
7
8     return 0;
9 }

```

Let's hack!

```

pi@pi-alpha:~/code $ gdb -q example2
Reading symbols from example2...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example2

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:  push    {r11, lr}
    0x000106f4 <+4>:  add     r11, sp, #4
    0x000106f8 <+8>:  sub     sp, sp, #8
    0x000106fc <+12>: ldr     r3, [pc, #44] ; 0x10730 <main+64>
    0x00010700 <+16>: str     r3, [r11, #-8]
    0x00010704 <+20>: ldr     r0, [pc, #40] ; 0x10734 <main+68>
    0x00010708 <+24>: ldr     r1, [pc, #32] ; 0x10730 <main+64>
    0x0001070c <+28>:  bl     0x1055c
    0x00010710 <+32>:  mov     r3, r0
    0x00010714 <+36>:  mov     r0, r3
    0x00010718 <+40>:  ldr     r1, [pc, #24] ; 0x10738 <main+72>
    0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
    0x00010720 <+48>:  mov     r3, #0
    0x00010724 <+52>:  mov     r0, r3
    0x00010728 <+56>:  sub     sp, r11, #4
    0x0001072c <+60>:  pop     {r11, pc}
    0x00010730 <+64>:  andeq  r0, r0, r1, ror #15
    0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
    0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print *0x10730
$1 = 2017
(gdb) set *0x10730 = 1981
(gdb) c
Continuing.
1981
[Inferior 1 (process 1046) exited normally]

```

As we can see the value in the memory address **0x10730** is equal to **2017**. Let's change that value in memory to **1981**. Let's continue and watch the value turn to **1981**! Successful hack!

Let's hack a second way! Re-start the program and set a breakpoint at main+28 and continue to the breakpoint.

```
(gdb) r
Starting program: /home/pi/code/example2

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  ldr    r1, [pc, #32] ; 0x10730 <main+64>
0x0001070c <+28>:  bl     0x1055c
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  andeq  r0, r0, r1, ror #15
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()
```

Let's continue and we see the value in **r1** is **2017**. Let's change the value in **r1** to **1981**. We continue and see the program successfully hacked to **1981**!

```
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  ldr    r1, [pc, #32] ; 0x10730 <main+64>
=> 0x0001070c <+28>:  bl     0x1055c
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  andeq  r0, r0, r1, ror #15
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) print $r1
$r2 = 2017
(gdb) set $r1 = 1981
(gdb) c
Continuing.
1981
[Inferior 1 (process 1049) exited normally]
```

Next week we will dive into Character Variables.

Part 20 – Character Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The next stage in our journey is that of character variables. Unlike the strings we have dealt with thus far, a character only takes up one byte of data.

Keep in mind, when we deal with any character data, we deal with literally two hex digits which are the ASCII code that represents an actual character that we see on our respective terminals.

Remember that each hex digit is 4 bits in length. Therefore two hex digits are 8 bits in length or a byte long.

To recap, each character translates down to an ASCII code in hex which the processor understands. The value of **n** is **0x6e** hex or **110** decimal. You can review any ASCII table to see where we derived this value. This will come in handy in the next lesson.

We start with our third program in C++ which is our “Character Variable” program. Let’s dive in and break each line down step-by-step and see how this language works. We will call this example3.cpp and save it to our device.

```
#include <iostream>

int main(void) {

    char yes_no = 'n';

    std::cout << yes_no << std::endl;

    return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     char yes_no = 'n';
5
6     std::cout << yes_no << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

Part 1: Goals

```
g++ example3.cpp -o example3
```

We simply then type:

```
./example3
```

```
pi@pi-alpha:~/code $ g++ example3.cpp -o example3
pi@pi-alpha:~/code $ ./example3
n
```

SUCCESS! We see “n” printed to the standard output or terminal!

Let’s break it down:

We utilize the **char** keyword to indicate a character variable to which we assign it the value of **n**.

We then utilize the **cout** function to print it to the standard output or terminal and add a new line with the **endl** function.

That’s it! Very simple.

Next week we will dive into Debugging Character Variables.

Part 21 – Debugging Character Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our code.

```

1 #include <iostream>
2
3 int main(void) {
4     char yes_no = 'n';
5
6     std::cout << yes_no << std::endl;
7
8     return 0;
9 }

```

Let's debug!

```

pi@pi-alpha:~/code $ gdb -q example3
Reading symbols from example3...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example3

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:   push   {r11, lr}
0x00010720 <+4>:   add    r11, sp, #4
0x00010724 <+8>:   sub    sp, sp, #8
0x00010728 <+12>:  mov    r3, #110           ; 0x6e
0x0001072c <+16>:  strb  r3, [r11, #-5]
0x00010730 <+20>:  ldrb  r3, [r11, #-5]
0x00010734 <+24>:  ldr   r0, [pc, #36]      ; 0x10760 <main+68>
0x00010738 <+28>:  mov   r1, r3
0x0001073c <+32>:  bl   0x105b8
0x00010740 <+36>:  mov   r3, r0
0x00010744 <+40>:  mov   r0, r3
0x00010748 <+44>:  ldr   r1, [pc, #20]     ; 0x10764 <main+72>
0x0001074c <+48>:  bl   0x105dc <_ZNSt8ios_base4InitD1Ev+24>
0x00010750 <+52>:  mov   r3, #0
0x00010754 <+56>:  mov   r0, r3
0x00010758 <+60>:  sub   sp, r11, #4
0x0001075c <+64>:  pop   {r11, pc}
0x00010760 <+68>:  ldrdeq r0, [r2], -r0   ; <UNPREDICTABLE>
0x00010764 <+72>:  andeq r0, r1, r8, ror #11
End of assembler dump.

```

Woah! This is confusing. I don't see any clear memory addresses being loaded into a register to manipulate the data.

Let's keep in mind that we are dealing with a single byte character variable.

If you remember from last week each character translates down to an ASCII code in hex which the processor understands. The value of **n** is **0x6e** hex or **110** decimal. You can review any ASCII table to see where we derived this value.

We do see **0x6e** at **main+12** which is the character 'n'.


```
(gdb) si
0x00010720 in main ()
(gdb) si
0x00010724 in main ()
(gdb) si
0x00010728 in main ()
(gdb) si
0x0001072c in main ()
(gdb) i r
r0          0x1          1
r1          0x7efff394      2130703252
r2          0x7efff39c      2130703260
r3          0x6e         110
r4          0x0          0
r5          0x0          0
r6          0x105f4      67060
r7          0x0          0
r8          0x0          0
r9          0x0          0
r10         0x76fff000      1996484608
r11         0x7efff23c      2130702908
r12         0x76e1f000      1994518528
sp          0x7efff230      0x7efff230
lr          0x76cfa294      1993319060
pc          0x1072c      0x1072c <main+16>
cpsr       0x60000010      1610612752
(gdb) print/c $r3
$1 = 110 '\n'
```

If we step into a few times we notice the value has been placed into **r3**. When we print the value in **r3** we now see our 'n' character.

Let's continue.

```
(gdb) c
Continuing.
n
[Inferior 1 (process 1567) exited normally]
```

We now see the 'n' printed to the standard output as expected.

It is important that you understand this process and understand that each character translates into an ASCII value to which the processor loads directly into a respective register. Our previous experience we have seen a string loaded directly into a memory location and this is not the case here.

Next week we will dive into Hacking Character Variables.

Part 22 – Hacking Character Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our code.

```

1 #include <iostream>
2
3 int main(void) {
4     char yes_no = 'n';
5
6     std::cout << yes_no << std::endl;
7
8     return 0;
9 }

```

Let's hack!

```

pi@pi-alpha:~/code $ gdb -q example3
Reading symbols from example3...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x1071c
(gdb) r
Starting program: /home/pi/code/example3

Breakpoint 1, 0x0001071c in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x0001071c <+0>:    push   {r11, lr}
   0x00010720 <+4>:    add    r11, sp, #4
   0x00010724 <+8>:    sub    sp, sp, #8
   0x00010728 <+12>:   mov    r3, #110      ; 0x6e
   0x0001072c <+16>:   strb  r3, [r11, #-5]
   0x00010730 <+20>:   ldrb  r3, [r11, #-5]
   0x00010734 <+24>:   ldr   r0, [pc, #36] ; 0x10760 <main+68>
   0x00010738 <+28>:   mov   r1, r3
   0x0001073c <+32>:   bl    0x105b8
   0x00010740 <+36>:   mov   r3, r0
   0x00010744 <+40>:   mov   r0, r3
   0x00010748 <+44>:   ldr   r1, [pc, #20] ; 0x10764 <main+72>
   0x0001074c <+48>:   bl    0x105dc <_ZNSt8ios_base4InitD1Ev+24>
   0x00010750 <+52>:   mov   r3, #0
   0x00010754 <+56>:   mov   r0, r3
   0x00010758 <+60>:   sub   sp, r11, #4
   0x0001075c <+64>:   pop   {r11, pc}
   0x00010760 <+68>:   ldrdeq r0, [r2], -r0 ; <UNPREDICTABLE>
   0x00010764 <+72>:   andeq r0, r1, r8, ror #11
End of assembler dump.

```

We again see the direct value of **0x6e** moved into **r3** at **main+12** which is our **'n'**.

```

(gdb) si
0x00010720 in main ()
(gdb) si
0x00010724 in main ()
(gdb) si
0x00010728 in main ()
(gdb) si
0x0001072c in main ()
(gdb) print/c $r3
$1 = 110 'n'

```

After stepping into 4 times and verify the value in **r3** which we clearly see as **'n'**.

```

(gdb) set $r3 = 'y'
(gdb) print/c $r3
$2 = 121 'y'

```

Let's hack the value in **r3** to a **'y'** and then reexamine the value in **r3**. We can now clearly see it has been changed to **'y'**.

Part 1: Goals

```
(gdb) c
Continuing.
y
[Inferior 1 (process 1587) exited normally]
```

As we continue we successfully see our hack worked! We see the value of 'y' printing to the standard output.

Next week we will dive into Boolean Variables.

Part 23 – Boolean Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The next stage in our journey is that of Boolean variables. The name goes back to the great George Boole to which all modern computer science has derived.

At the lowest level a value is either 0 or 1, false or true, + < 5 volts or +5 volts, etc.

Let's examine our code.

```
#include <iostream>

int main(void) {

    bool isHacked = false;

    std::cout << isHacked << std::endl;

    return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     bool isHacked = false;
5
6     std::cout << isHacked << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

```
g++ example4.cpp -o example4

./example4
```

```
pi@pi-alpha:~/code $ g++ example4.cpp -o example4
pi@pi-alpha:~/code $ ./example4
0
```

SUCCESS! We see **0** printed to the standard output or terminal!

Let's break it down:

Part 1: Goals

We create a boolean variable called **isHacked** to which we assign a value of **false** or **0**. When we run the binary we clearly see the value **0** that successfully was echoed to the standard output.

Next week we will dive into Debugging Boolean Variables.

Part 24 – Debugging Boolean Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's re-examine our code.

```

1 #include <iostream>
2
3 int main(void) {
4     bool isHacked = false;
5
6     std::cout << isHacked << std::endl;
7
8     return 0;
9 }

```

Let's debug.

```

pi@pi-alpha:~/code $ gdb -q example4
Reading symbols from example4...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example4

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  mov    r3, #0
0x00010700 <+16>:  strb  r3, [r11, #-5]
0x00010704 <+20>:  ldrb  r3, [r11, #-5]
0x00010708 <+24>:  ldr   r0, [pc, #36] ; 0x10734 <main+68>
0x0001070c <+28>:  mov   r1, r3
0x00010710 <+32>:  bl    0x10598 <_ZNSt8ios_base4InitD1Ev+12>
0x00010714 <+36>:  mov   r3, r0
0x00010718 <+40>:  mov   r0, r3
0x0001071c <+44>:  ldr   r1, [pc, #20] ; 0x10738 <main+72>
0x00010720 <+48>:  bl    0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010724 <+52>:  mov   r3, #0
0x00010728 <+56>:  mov   r0, r3
0x0001072c <+60>:  sub   sp, r11, #4
0x00010730 <+64>:  pop   {r11, pc}
0x00010734 <+68>:  andeq r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.

```

Let's step 4 times and disassemble.

```
(gdb) si
0x000106f4 in main ()
(gdb) si
0x000106f8 in main ()
(gdb) si
0x000106fc in main ()
(gdb) si
0x00010700 in main ()
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  mov    r3, #0
=> 0x00010700 <+16>:  strb  r3, [r11, #-5]
0x00010704 <+20>:  ldrb  r3, [r11, #-5]
0x00010708 <+24>:  ldr   r0, [pc, #36] ; 0x10734 <main+68>
0x0001070c <+28>:  mov   r1, r3
0x00010710 <+32>:  bl   0x10598 <_ZNSt8ios_base4InitD1Ev+12>
0x00010714 <+36>:  mov   r3, r0
0x00010718 <+40>:  mov   r0, r3
0x0001071c <+44>:  ldr   r1, [pc, #20] ; 0x10738 <main+72>
0x00010720 <+48>:  bl   0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010724 <+52>:  mov   r3, #0
0x00010728 <+56>:  mov   r0, r3
0x0001072c <+60>:  sub   sp, r11, #4
0x00010730 <+64>:  pop   {r11, pc}
0x00010734 <+68>:  andeq r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's examine what is now in **r3**.

```
(gdb) print $r3
$r1 = 0
(gdb) c
Continuing.
0
[Inferior 1 (process 21451) exited normally]
```

As we can clearly see the value in **isHacked** is **0** or **false** which makes sense based on our c++ source code.

I know these lessons may seem trivial however Reverse Engineering is all about breaking things down in their most basic components. Reverse Engineering is about patience and logical flow. It is critical that you take the time and work through all of these examples with a Raspberry Pi device so that you can have a proper appreciation for how the process actually works.

Next week we will dive into Hacking Boolean Variables.

Part 25 – Hacking Boolean Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's re-examine our code.

```

1 #include <iostream>
2
3 int main(void) {
4     bool isHacked = false;
5
6     std::cout << isHacked << std::endl;
7
8     return 0;
9 }

```

Let's hack!

```

pi@pi-alpha:~/code $ gdb -q example4
Reading symbols from example4...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example4
disas
Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:   push    {r11, lr}
0x000106f4 <+4>:   add     r11, sp, #4
0x000106f8 <+8>:   sub     sp, sp, #8
0x000106fc <+12>:  mov     r3, #0
0x00010700 <+16>:  strb   r3, [r11, #-5]
0x00010704 <+20>:  ldrb   r3, [r11, #-5]
0x00010708 <+24>:  ldr    r0, [pc, #36] ; 0x10734 <main+68>
0x0001070c <+28>:  mov    r1, r3
0x00010710 <+32>:  bl     0x10598 <_ZNSt8ios_base4InitD1Ev+12>
0x00010714 <+36>:  mov    r3, r0
0x00010718 <+40>:  mov    r0, r3
0x0001071c <+44>:  ldr    r1, [pc, #20] ; 0x10738 <main+72>
0x00010720 <+48>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010724 <+52>:  mov    r3, #0
0x00010728 <+56>:  mov    r0, r3
0x0001072c <+60>:  sub    sp, r11, #4
0x00010730 <+64>:  pop    {r11, pc}
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
(gdb) si
0x000106f4 in main ()
(gdb) si
0x000106f8 in main ()
(gdb) si
0x000106fc in main ()
(gdb) si
0x00010700 in main ()

```

Let's break at main, run and disas in addition to step into four times.


```
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  mov    r3, #0
=> 0x00010700 <+16>:  strb   r3, [r11, #-5]
0x00010704 <+20>:  ldrb   r3, [r11, #-5]
0x00010708 <+24>:  ldr    r0, [pc, #36] ; 0x10734 <main+68>
0x0001070c <+28>:  mov    r1, r3
0x00010710 <+32>:  bl     0x10598 <_ZNSt8ios_base4InitD1Ev+12>
0x00010714 <+36>:  mov    r3, r0
0x00010718 <+40>:  mov    r0, r3
0x0001071c <+44>:  ldr    r1, [pc, #20] ; 0x10738 <main+72>
0x00010720 <+48>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010724 <+52>:  mov    r3, #0
0x00010728 <+56>:  mov    r0, r3
0x0001072c <+60>:  sub    sp, r11, #4
0x00010730 <+64>:  pop    {r11, pc}
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We see that **0** or **FALSE** is moved into **r3** at main+12.

```
(gdb) set $r3 = 1
(gdb) print $r3
$r1 = 1
(gdb) c
Continuing.
1
[Inferior 1 (process 21457) exited normally]
```

Very simply we set **r3** to **1** or **TRUE** and continue execution to which we notice that the Boolean variable **isHacked** is now **TRUE**.

It's that simple folks! These elementary examples will help build your mental library of examples of how to approach everything in code and understanding how to take control of code execution no matter what!

Next week we will dive into Integer Variables.

Part 26 – Integer Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

The next stage in our journey is that of Integer variables.

A 32-bit register can store 2^{32} different values. The range of integer values that can be stored in 32 bits depends on the integer representation used. With the two most common representations, the range is 0 through 4,294,967,295 ($2^{32} - 1$) for representation as an (unsigned) binary number, and $-2,147,483,648$ (-2^{31}) through 2,147,483,647 ($2^{31} - 1$) for representation as two's complement.

Keep in mind with 32-bit memory addresses you can directly access a maximum of 4 GB of byte-addressable memory.

Let's examine our code.

```
#include <iostream>

int main(void) {

    int myNumber = 777;

    std::cout << myNumber << std::endl;

    return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     int myNumber = 777;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

```
g++ example5.cpp -o example5
```

```
./example5
```

```
pi@pi-alpha:~/code $ g++ example5.cpp -o example5
pi@pi-alpha:~/code $ ./example5
777
```

Part 1: Goals

SUCCESS! We see **777** printed to the standard output or terminal!

Let's break it down:

We assign the integer **777** directly into the variable **myNumber** and then print it out to the terminal with the c++ **cout** function.

Next week we will dive into Debugging Integer Variables.

Part 27 – Debugging Integer Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our code. I again want to include the below information from last week's lesson to emphasize what is going on regarding integers.

A 32-bit register can store 2^{32} different values. The range of integer values that can be stored in 32 bits depends on the integer representation used. With the two most common representations, the range is 0 through 4,294,967,295 ($2^{32} - 1$) for representation as an (unsigned) binary number, and $-2,147,483,648$ (-2^{31}) through 2,147,483,647 ($2^{31} - 1$) for representation as two's complement.

Keep in mind with 32-bit memory addresses you can directly access a maximum of 4 GB of byte-addressable memory.

```

1 #include <iostream>
2
3 int main(void) {
4     int myNumber = 777;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }

```

Let's debug!

```

pi@pi-alpha:~/code $ gdb -q example5
Reading symbols from example5...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example5

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:   push    {r11, lr}
0x000106f4 <+4>:   add     r11, sp, #4
0x000106f8 <+8>:   sub     sp, sp, #8
0x000106fc <+12>:  ldr     r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str     r3, [r11, #-8]
0x00010704 <+20>:  ldr     r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  ldr     r1, [r11, #-8]
0x0001070c <+28>:  bl      0x1055c
0x00010710 <+32>:  mov     r3, r0
0x00010714 <+36>:  mov     r0, r3
0x00010718 <+40>:  ldr     r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl      0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
0x00010720 <+48>:  mov     r3, #0
0x00010724 <+52>:  mov     r0, r3
0x00010728 <+56>:  sub     sp, r11, #4
0x0001072c <+60>:  pop     {r11, pc}
0x00010730 <+64>:  andeq  r0, r0, r9, lsl #6
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.

```

We see at **main+12** the address at **0x10730** loading data into **r3**. Let's take a closer look.

```

(gdb) x/d 0x10730
0x10730 <main+64>:   777
(gdb) c
Continuing.
777
[Inferior 1 (process 1141) exited normally]

```

Part 1: Goals

When we examine the data inside **0x10730** we clearly see the integer **777** present. When we continue we see **777** echoed back to the terminal which makes sense as we utilized the **cout** function within c++.#linux #arm #asm #cplusplus #reverseengineering

Next week we will dive into Hacking Integer Variables.

Part 28 – Hacking Integer Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's review our code.

```
1 #include <iostream>
2
3 int main(void) {
4     int myNumber = 777;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

Let's hack!

```
pi@pi-alpha:~/code $ gdb -q example5
Reading symbols from example5...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example5

Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:      push    {r11, lr}
0x000106f4 <+4>:      add     r11, sp, #4
0x000106f8 <+8>:      sub     sp, sp, #8
0x000106fc <+12>:     ldr     r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:     str     r3, [r11, #-8]
0x00010704 <+20>:     ldr     r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:     ldr     r1, [r11, #-8]
0x0001070c <+28>:     bl     0x1055c
0x00010710 <+32>:     mov     r3, r0
0x00010714 <+36>:     mov     r0, r3
0x00010718 <+40>:     ldr     r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:     bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+24>
0x00010720 <+48>:     mov     r3, #0
0x00010724 <+52>:     mov     r0, r3
0x00010728 <+56>:     sub     sp, r11, #4
0x0001072c <+60>:     pop     {r11, pc}
0x00010730 <+64>:     andeq  r0, r0, r9, lsl #6
0x00010734 <+68>:     andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:     ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

Let's take a look again inside the memory location of **0x10730**.

```
(gdb) x/d 0x10730
0x10730 <main+64>:      777
(gdb) c
Continuing.
777
[Inferior 1 (process 1141) exited normally]
```

As we can clearly see the integer value of **777** appears and when we continue it echoes out to the terminal the value of **777** which corresponds with our c++ function **cout**.

Let's hack the value inside of **0x10730** and set the value to **666** and then reexamine the value inside **0x10730** and continue.

```
(gdb) x/d 0x10730
0x10730 <main+64>:      777
(gdb) set *0x10730 = 666
(gdb) x/d 0x10730
0x10730 <main+64>:      666
(gdb) c
Continuing.
666
[Inferior 1 (process 1145) exited normally]
```

Part 1: Goals

Success! As we can see we hacked the value to **666** as we continue we see it echoed out to stdout.

Next week we will dive into Float Variables.

Part 29 – Float Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

The next stage in our journey is that of Floating-Point variables.

A floating-point variable is different from an integer as it has a fractional value attached to which we designate with a period.

Let's examine our code.

```
#include <iostream>

float main(void) {

    int myNumber = 1337.1;

    std::cout << myNumber << std::endl;

    return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     float myNumber = 1337.1;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

To compile this we simply type:

```
g++ example6.cpp -o example6
```

```
./example6
```

SUCCESS! We see 1337.1 printed to the standard output or terminal!

Let's break it down:

We assign the floating-point variable directly into the variable **myNumber** and then print it out to the terminal with the c++ **cout** function.

Part 1: Goals

Thus far we have a good understanding of the ARM registers however next week we will introduce the registers within the math co-processor that work with floating-point variables. The registers you have worked with up to now only store whole numbers or integers and at the Assembly level, any fractional value must be manipulated through the math co-processor registers.

Next week we will dive into Debugging Float Variables.

Part 30 – Debugging Float Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will

cover. <https://github.com/mytechnotalent/Reverse-Engineering-Tutorial>

Let's re-examine our code.

```
#include <iostream>

int main(void) {

    float myNumber = 1337.1;

    std::cout << myNumber << std::endl;

    return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     float myNumber = 1337.1;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

Let's debug!

```

pi@pi-alpha:~/code $ gdb -q example6
Reading symbols from example6...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example6
disas
Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  vldr  s0, [r11, #-8]
0x0001070c <+28>:  bl     0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  strtmi r2, [r7], #819 ; 0x333
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.

```

Let's break on **main+20** and continue to that point.

```

(gdb) b *main+20
Breakpoint 2 at 0x10704
(gdb) c
Continuing.

Breakpoint 2, 0x00010704 in main ()
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
=> 0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  vldr  s0, [r11, #-8]
0x0001070c <+28>:  bl     0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  strtmi r2, [r7], #819 ; 0x333
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.

```

Let's examine what value is inside **r11-8**. We clearly see it is **1337.09998** which approximates our value in our original c++ code. Keep in mind a float has roughly 7 decimal digits of precision and that is why we do not see **1337.1** so please remember that as we go forward.

```

(gdb) x/f $r11-8
0x7efff234:   1337.09998

```

We can also see this value in high memory.

```

(gdb) x/f 0x7efff234
0x7efff234:   1337.09998

```

Let's break on **main+28** and continue.

```

(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()

```

We see a strange new instruction. We see **vldr** and the value within **r11, #8** being moved into **s0**. So what is **s0**? We have a math co-processor which has a series of additional registers that work with decimal or floating-point numbers. Here we see an example of such to which the value of **1337.09998** is being moved into **s0**. The **vldr** instruction loads a constant value into every element of a single-precision or double-precision register such as **s0**.

```
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:  push   {r11, lr}
0x000106f4 <+4>:  add    r11, sp, #4
0x000106f8 <+8>:  sub    sp, sp, #8
0x000106fc <+12>: ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>: str    r3, [r11, #-8]
0x00010704 <+20>: ldr    r0, [pc, #40] ; 0x10734 <main+68>
=> 0x00010708 <+24>: vldr  s0, [r11, #-8]
0x0001070c <+28>: bl    0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
0x00010710 <+32>: mov    r3, r0
0x00010714 <+36>: mov    r0, r3
0x00010718 <+40>: ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>: bl    0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010720 <+48>: mov    r3, #0
0x00010724 <+52>: mov    r0, r3
0x00010728 <+56>: sub   sp, r11, #4
0x0001072c <+60>: pop   {r11, pc}
0x00010730 <+64>: strtmi r2, [r7], #819 ; 0x333
0x00010734 <+68>: andeq r0, r2, r0, lsr #19
0x00010738 <+72>:      ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We can only see these special registers if we do a `info registers all` command as we do below.

```
(gdb) i r a
```

Below we see the value now being moved into **s0**.

```
s0      1337.09998      (raw 0x44a72333)
s1      0              (raw 0x00000000)
s2      0              (raw 0x00000000)
s3      0              (raw 0x00000000)
s4      0              (raw 0x00000000)
s5      0              (raw 0x00000000)
s6      0              (raw 0x00000000)
s7      0              (raw 0x00000000)
s8      0              (raw 0x00000000)
s9      0              (raw 0x00000000)
s10     0              (raw 0x00000000)
s11     0              (raw 0x00000000)
s12     0              (raw 0x00000000)
s13     2              (raw 0x40000000)
s14     0              (raw 0x00000000)
s15     0              (raw 0x00000000)
s16     0              (raw 0x00000000)
s17     0              (raw 0x00000000)
s18     0              (raw 0x00000000)
s19     0              (raw 0x00000000)
s20     0              (raw 0x00000000)
s21     0              (raw 0x00000000)
s22     0              (raw 0x00000000)
s23     0              (raw 0x00000000)
s24     0              (raw 0x00000000)
s25     0              (raw 0x00000000)
s26     0              (raw 0x00000000)
s27     0              (raw 0x00000000)
s28     0              (raw 0x00000000)
s29     0              (raw 0x00000000)
s30     0              (raw 0x00000000)
s31     0              (raw 0x00000000)
```

Next week we will dive into Hacking Float Variables.

Part 31 – Hacking Float Variables

For a complete table of contents of all the lessons please click below as it will give you a brief of each lesson in addition to the topics it will cover. <https://github.com/mytechtalent/Reverse-Engineering-Tutorial>

Let's re-examine our code.

```
#include <iostream>

int main(void) {

    int myNumber = 1337.1;

    std::cout << myNumber << std::endl;

    return 0;

}
```

```
1 #include <iostream>
2
3 int main(void) {
4     float myNumber = 1337.1;
5
6     std::cout << myNumber << std::endl;
7
8     return 0;
9 }
```

Let's review last week's tutorial.

```

pi@pi-alpha:~/code $ gdb -q example6
Reading symbols from example6...(no debugging symbols found)...done.
(gdb) b main
Breakpoint 1 at 0x106f0
(gdb) r
Starting program: /home/pi/code/example6
disas
Breakpoint 1, 0x000106f0 in main ()
(gdb) disas
Dump of assembler code for function main:
=> 0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  vldr  s0, [r11, #-8]
0x0001070c <+28>:  bl     0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  strtmi r2, [r7], #819 ; 0x333
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.

```

Let's break on **main+20** and continue to that point.

```

(gdb) b *main+20
Breakpoint 2 at 0x10704
(gdb) c
Continuing.

Breakpoint 2, 0x00010704 in main ()
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
=> 0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  vldr  s0, [r11, #-8]
0x0001070c <+28>:  bl     0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  strtmi r2, [r7], #819 ; 0x333
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.

```

Let's examine what value is inside **r11-8**. We clearly see it is **1337.09998** which approximates our value in our original c++ code. Keep in mind a float has roughly 7 decimal digits of precision and that is why we do not see **1337.1** so please remember that as we go forward.

```

(gdb) x/f $r11-8
0x7efff234:   1337.09998

```

We can also see this value in high memory.

```

(gdb) x/f 0x7efff234
0x7efff234:   1337.09998

```

Let's break on **main+28** and continue.

```

(gdb) b *main+28
Breakpoint 2 at 0x1070c
(gdb) c
Continuing.

Breakpoint 2, 0x0001070c in main ()

```

We see a strange new instruction. We see **vldr** and the value within **r11, #8** being moved into **s0**. So what is **s0**? We have a math co-processor which has a series of additional registers that work with decimal or floating-point numbers. Here we see an example of such to which the value of **1337.09998** is being moved into **s0**. The **vldr** instruction loads a constant value into every element of a single-precision or double-precision register such as **s0**.

```
(gdb) disas
Dump of assembler code for function main:
0x000106f0 <+0>:   push   {r11, lr}
0x000106f4 <+4>:   add    r11, sp, #4
0x000106f8 <+8>:   sub    sp, sp, #8
0x000106fc <+12>:  ldr    r3, [pc, #44] ; 0x10730 <main+64>
0x00010700 <+16>:  str    r3, [r11, #-8]
0x00010704 <+20>:  ldr    r0, [pc, #40] ; 0x10734 <main+68>
0x00010708 <+24>:  vldr  s0, [r11, #-8]
=> 0x0001070c <+28>:  bl     0x105a4 <_ZNSt8ios_base4InitD1Ev+24>
0x00010710 <+32>:  mov    r3, r0
0x00010714 <+36>:  mov    r0, r3
0x00010718 <+40>:  ldr    r1, [pc, #24] ; 0x10738 <main+72>
0x0001071c <+44>:  bl     0x105b0 <_ZNSt8ios_base4InitD1Ev+36>
0x00010720 <+48>:  mov    r3, #0
0x00010724 <+52>:  mov    r0, r3
0x00010728 <+56>:  sub    sp, r11, #4
0x0001072c <+60>:  pop    {r11, pc}
0x00010730 <+64>:  strtmi r2, [r7], #819 ; 0x333
0x00010734 <+68>:  andeq  r0, r2, r0, lsr #19
0x00010738 <+72>:  ; <UNDEFINED> instruction: 0x000105bc
End of assembler dump.
```

We can only see these special registers if we do a `info registers all` command as we do below.

```
(gdb) i r a
```

Below we see the value now being moved into **s0**.

```
s0      1337.09998      (raw 0x44a72333)
s1      0              (raw 0x00000000)
s2      0              (raw 0x00000000)
s3      0              (raw 0x00000000)
s4      0              (raw 0x00000000)
s5      0              (raw 0x00000000)
s6      0              (raw 0x00000000)
s7      0              (raw 0x00000000)
s8      0              (raw 0x00000000)
s9      0              (raw 0x00000000)
s10     0              (raw 0x00000000)
s11     0              (raw 0x00000000)
s12     0              (raw 0x00000000)
s13     2              (raw 0x40000000)
s14     0              (raw 0x00000000)
s15     0              (raw 0x00000000)
s16     0              (raw 0x00000000)
s17     0              (raw 0x00000000)
s18     0              (raw 0x00000000)
s19     0              (raw 0x00000000)
s20     0              (raw 0x00000000)
s21     0              (raw 0x00000000)
s22     0              (raw 0x00000000)
s23     0              (raw 0x00000000)
s24     0              (raw 0x00000000)
s25     0              (raw 0x00000000)
s26     0              (raw 0x00000000)
s27     0              (raw 0x00000000)
s28     0              (raw 0x00000000)
s29     0              (raw 0x00000000)
s30     0              (raw 0x00000000)
s31     0              (raw 0x00000000)
```

Let's hack!

```
(gdb) set $s0 = 666.666
```

Let's now look at the registers and see what has transpired.

```
(gdb) i r a
```