THE ART OF CODE CRACKING

Muhammad Samaak

A Good book says Hacker is a term for both those who write code and those who exploit it. Perhaps it's true but these days mostly people think that Exploit Development, Reverse Engineering and all these kind of lower level stuff are really scary because first, you have to understand the computer internals, Assembly Language, memory and other horrible stuffs. The goal of this paper is to give you a brief overview of how things actually worked at the lower level and how a simple password protected program can be cracked.

The secret of Computer Science is the fact that you really don't have to worry about "How". A web developer doesn't need to know how the Internet is working. But In order to find the security related issue in a program, one must have to understand what it really takes to actually run a program on a computer.

People think that technology has really advanced overt the years well its a bit true computers are today more advanced than the computer of 25 or 50 years ago. According to the Moore's Law every two years computers would become twice as powerful they become faster, more powerful but they still work the same way because the architecture hasn't changed much.

The Computers are composed of different parts which include **CPU**, **Memory** , **Secondary Storage** and **Input/Output** devices. The main component of the computer is CPU. It's fetch the instructions and execute them. The CPU has several functional units, including an **Arithmetic** & **Logic Unit** (**ALU**), **control unit** (**CU**) and **Registers**.

ALU perform four kinds of arithmetic operations (**addition**, **subtraction**, **multiplication** and **division**) and logical operations like (**and**, **or**, **xor**) and that's exactly what a computer actually does most of the time. The control unit manages the movement of data and instructions in the CPU basically Control unit is anything else inside the processor except the registers and ALU.

Hard coded Variables

Registers are a small piece of memory they are like internal variables we can use them same like we used the variable in Higher level languages. Registers name actually depends on the CPU family you are working with. There are different registers some of them are General Purpose Registers and some of them are special registers but for the sake of the simplicity for now on we don't want to go into the much details.

Some General Purposes Registers:

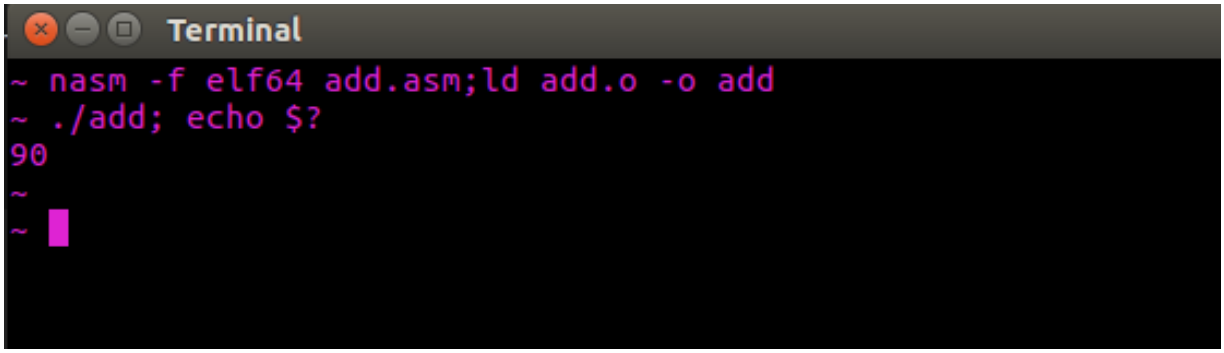
16-bit: AX, BX, CX, DX, SI, DI IP.….
32-bit: EAX, EBX, ECX. EDX. ESI, EDI, EIP….
64-bit: RAX, RBX, RCX, RDX, RSI, RDI, RIP,R8-R15…
we will go into the more details later

Getting Your Hands Dirty

In order to truly understand how the a program worked at the lower level let's write some code in Assembly. There are 2 flavors in x86 assembly AT&T syntax and Intel syntax in this paper we will use the Intel syntax**.**

```
1 global _start
2 _start: mov rdi, 50
3         mov rbx, 40
4         add rdi,rbx
5         mov rax, 0x3c
6         syscall
7
~
```

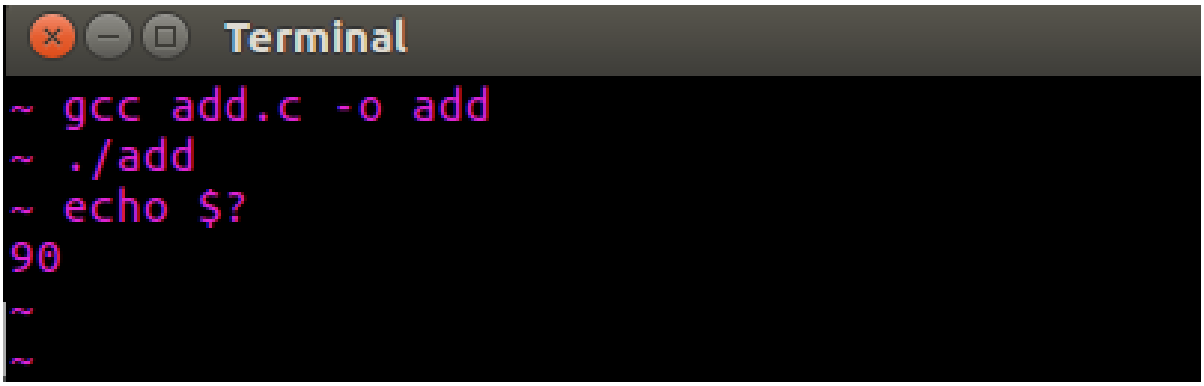
This is the simplest program we can write in assembly all it does is add two numbers together and then exit the program without returning anything. The program actually starts with the line 2 mov rdi, 50 the mov instruction moves data from one location to another in this case we are moving the value 50 into rdi register. RDI (Destination Index) register mostly used as a pointer to a write location during a string operation or loop (forget about this for now just remember RDI is a General Purpose Register and currently holding a value 50) on line 3 we move 40 into the rbx register RBX (Base Register) is a General purpose register it can be used for anything basically it's don't have any special purpose but now holding a value 40.Then on line 4 we use the add instruction which basically performed addition next, we have a mov rax, 0x3c. 0x3c is a hax value equal to 60 which refer to exit system call in Linux every system call is assigned with a static number. EAX holds the System call number and the system calls argument can be put in EBX ECX EDX in alphabetical order but in this case, we don't have any argument on the last we have syscall, system calls are made with the "syscall" instruction. Now compile the program linked it and run it.



$? is to return the code from the last run process. We can see the output is 90

We have wrote and compiled the simple program in Assembly Let write the same program in C

Every C program starts with the main function this is were the program start executing on line 3 we have created a variable x which holds the value of 50 the **int** variables are used to store an integer value then on line 4 we have added the 40 into x variable on line 5 we are simply returning the value which is currently hold by the x variable. Compile it with the GCC compiler and run it.



What's Next?

So far we have wrote a simple program in ASM then we port it into C as we can see the both programs are pretty much similar but instant of assembly the C program codes are much more readable. Now we have a basic idea that how programs are written in lower and higher languages let's use this knowledge to make something more interesting we will make a simple security check program which will ask for the security key and then display the message if the provided key was correct.

```
 1 #include <stdio.h>
 2
 3 void main() {
 4 printf("Enter the key: ");
 5 int key;
 6 scanf("%d",&key);
 7 //printf("%d\n", key);
 8
 9 if ( key == 1999) {
10 printf("Key is correct\n");
11 printf("\nmessage: 'Valar morghulis'\n");
12 } else {
13 printf("key is wrong\n"); }}
14
15
```

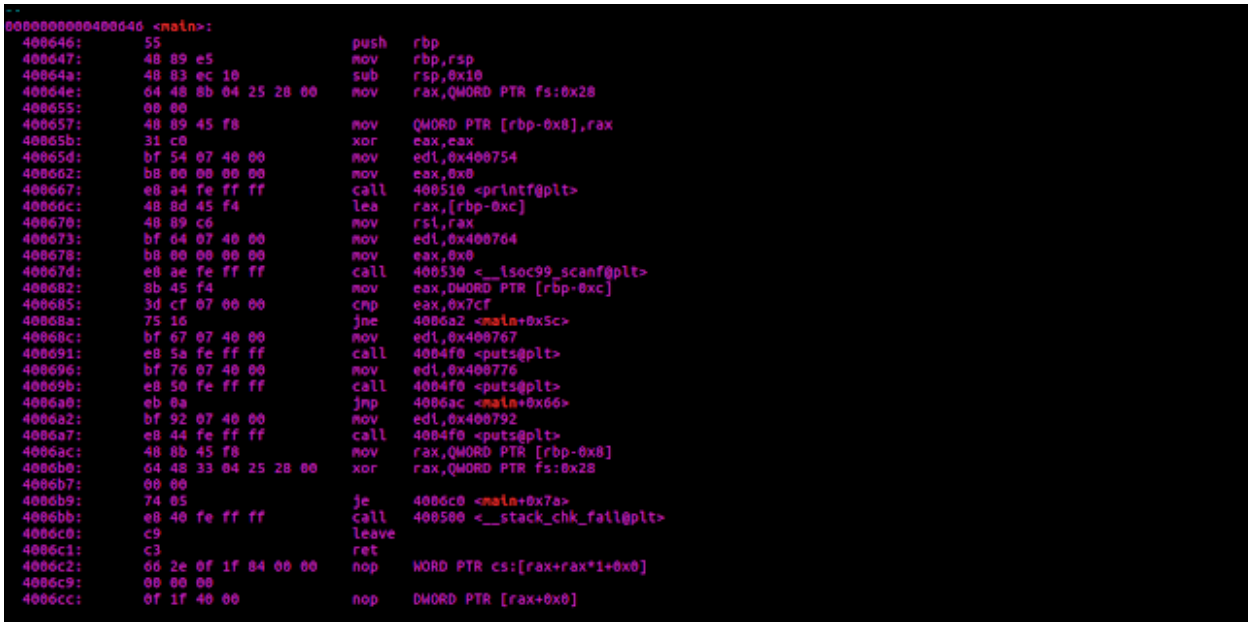
On the line 4 we use the printf which just display the text on the screen scanf used to take the input then we have the simple if statement which just check the key if it's match then i will print the message or if the key is wrong it will terminate the program without printing the message.

```
~ gcc secretmessage.c -o secretmessage
~ ./secretmessage
Enter the key: 1000
key is wrong
~
~ ./secretmessage
Enter the key: 1999
Key is correct

message: 'Valar morghulis'
~
~
```

looks perfect!. Let's try to disassemble this binary and read though the assembly so we can actually understand how it's working and what's really going on under the hood. We can use the object dump to disassemble this program.

Command for disassembling: *objdump -D -M intel secretmessage | grep 'main' -A 20*



-M intel is to disassemble the code in Intel syntax and remember every C program must have a main function so for now on we will just stick to the main function.

Ok thats took really horrible but we can still ignore most of the stuff here let's just focus on the actual flow in the main function at line 11 we have a **call printf** call instruction is used for calling the function in this case it's calling the printf function and we already know what printf is used for then we have **scanf** after that we have ***cmp*** instruction which just compare the registers value and set the **FLAGS** registers according to the results after that we have **jne 4006a2 <main+0x5c>** jne means Jump not equal so we will jump to the address **4006a2** if the zf (the "zero" flag) is equal to 0 (FLAGS registers are basically the status registers which contain the current status of the process there are different flag registers like CF "***carry flag***" ZF "***zero flag***" etc but for now we just focus on the Zero Flag ZF) so if the zero flag is not set we will proceed to the puts call puts is just an other function used for printing after that the program will terminate.

Let's break it again but simply

The program first call the printf function which will display the text which is "Enter the key:" then it's take the input using scanf function and performed compare operation which set the zero flag if the compression are not matched then we will jumped to the another print function which will print "Key is wrong" and then the program will closed without printing the secret message.

That's exactly what we did when we were writing this program we use the if statement so we can make the decisions if the key is correct it will print the message other wise it will terminate the program without printing the message.

But what if somehow we can modify this binary file and remove this jump statement so it won't jump to the address 4006a2 even if the key is wrong…... let's see if it's possible.

Let’s Hack it

Open this binary into the vim (vim is a text editor but we can also use it as a hex editor or you can use any other text editor they all are same)
Commands: *vim secretmessage*
Then type: ESC + :%!xxd + ENTER

By these commands we will get the hexdump of the binary that you can modify within vim..
If you check the objdump output and look at the jne instruction the opcode is 75 16 let's find these opcode and change it to 90 (90 is just a opcode of nop instruction that means we are removing the instruction)

```
00000680: ffff 8b45 f43d cf07 0000 7516 bf67 0740  ...E.=....u..g.@
```

in vim we can find the text by entering / so type /7516 and hit enter. Find the jne Opcode and change it to nop.

```
00000680: ffff 8b45 f43d cf07 0000 9090 bf67 0740  ...E.=.......g.@
```

press **I** to get into the insert mode and modify the 7516 to 9090
So we have just remove the jne statement in order to save it first exit form the Insert mode by pressing ESC key then we have to convert it back so enter **:%!xxd -r** and press enter then save it by typing **:wq**
Run the program and enter any random key.

```
~ ./secretmessage
Enter the key: 000000
Key is correct

message: 'Valar morghulis'
~
```

And that's worked we just bypass the key checkup that we have wrote to protect the message.

It was just a simple program we didn’t wrote any complex algorithm for checking the key we didn’t use any types of protection so Cracking a program can not be easy most of the time it can take hours or may be days of work and lot’s of knowledge but I hope you will still get the basic idea of how a simple program can be cracked.