The Art of Code Cracking

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 most 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, memory and other horrible stuff. The goal of this paper is to over come the fear you might have about the lover level stuff and give you a brief overview of how cracking and patching the binaries.

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.

Moore's and Technology

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 Machine

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 the fetch 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) 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 program. Registers name actually depends on the CPU family you are working with. There are 8-16 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 get 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 EDI 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 (System call Wikipedia). 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

```
Terminal
~ nasm -f elf64 add.asm;ld add.o -o add
~ ./add; echo $?
90
~
~
```

$? is to return the code from the last run process. We can see the output is 90 So we have wrote and compiled the simple program in Assembly Let write the same program in C Language.

```
1 int main()
2 {
3 register int x = 50;
4 x += 40;
5 return x;
6 }
~
```

every C program starts with the main function this is were the progrm start executing on line 3 we have created a variable x which holds the value of 50 the int variable is used to store an integer then on line 40 we have added the 40 into x on line 5 we are simply returning the x.

```
Terminal
~ gcc add.c -o add
~ ./add
~ echo $?
90
~
~
```

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 language and 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 the basic idea of how programs are actually 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.

```
#include <stdio.h>

void main() {
printf("Enter the key: ");
int key;
scanf("%d",&key);
//printf("%d\n", key);

if ( key == 1999) {
printf("Key is correct\n");
printf("\nmessage: 'Valar morghulis'\n");
} else {
printf("key is wrong\n"); }}


```

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 will make the decision if the provided key is equal to 1999 then it will print the message and if the provided 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 to understand how what's really going on under the hood. We can use the object dump to disassemble this program.

command: 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.

```
--
0000000000400646 <main>:
  400646:       55                      push   rbp
  400647:       48 89 e5                mov    rbp,rsp
  40064a:       48 83 ec 10             sub    rsp,0x10
  40064e:       64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
  400655:       00 00
  400657:       48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
  40065b:       31 c0                   xor    eax,eax
  40065d:       bf 54 07 40 00          mov    edi,0x400754
  400662:       b8 00 00 00 00          mov    eax,0x0
  400667:       e8 a4 fe ff ff          call   400510 <printf@plt>
  40066c:       48 8d 45 f4             lea    rax,[rbp-0xc]
  400670:       48 89 c6                mov    rsi,rax
  400673:       bf 64 07 40 00          mov    edi,0x400764
  400678:       b8 00 00 00 00          mov    eax,0x0
  40067d:       e8 ae fe ff ff          call   400530 <__isoc99_scanf@plt>
  400682:       8b 45 f4                mov    eax,DWORD PTR [rbp-0xc]
  400685:       3d cf 07 00 00          cmp    eax,0x7cf
  40068a:       75 16                   jne    4006a2 <main+0x5c>
  40068c:       bf 67 07 40 00          mov    edi,0x400767
  400691:       e8 5a fe ff ff          call   4004f0 <puts@plt>
  400696:       bf 76 07 40 00          mov    edi,0x400776
  40069b:       e8 50 fe ff ff          call   4004f0 <puts@plt>
  4006a0:       eb 0a                   jmp    4006ac <main+0x66>
  4006a2:       bf 92 07 40 00          mov    edi,0x400792
  4006a7:       e8 44 fe ff ff          call   4004f0 <puts@plt>
  4006ac:       48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
  4006b0:       64 48 33 04 25 28 00    xor    rax,QWORD PTR fs:0x28
  4006b7:       00 00
  4006b9:       74 05                   je     4006c0 <main+0x7a>
  4006bb:       e8 40 fe ff ff          call   400500 <__stack_chk_fail@plt>
  4006c0:       c9                      leave
  4006c1:       c3                      ret
  4006c2:       66 2e 0f 1f 84 00 00    nop    WORD PTR cs:[rax+rax*1+0x0]
  4006c9:       00 00 00
  4006cc:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]
```

The Output of the objdump look 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 and and we already know what printf is used for then we have scanf after that we have cmp which just compare the registers value and set the EFLAGS register according to the results after that we have jne 4006a2 <main+0x5c> jne (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 remember the Zero Flag ZF flag) 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 for print the text which is "Enter the key:" then it's take the input using scanf and performed compare operation which set the zero flag if the compression are not matched then we jumped to another print function which print "Key is wrong" and the program closed without printing the secret message.

That's exactly how code it. We match the input with the correct key and use the IF statement to make the decision. But what if somehow we can modify this binary 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 Crack it

Open this binary into the vim (vim is a text editor but we can 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 edit within vim as a normal text file.
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)

in vim we can find the text by entering / so type /7516 and hit enter. Find the jne Opcode and change it to 90.

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

Press I to get into the insert mode and modify the 7516 to 9090

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

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 into the binary data 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 See how easy it is to crack a simple program but It can be more difficult depending on how the key check algorithm is written and some other techniques are also used to protect the binary.

Author: Muhammad Samaak
Email: m.samaak00@gmail.com