Contents

1 Introduction

Welcome to Game Hacking for Noobs. This aim of this book is to give you a *really* brief introduction into an art of examining computer processes and more precisely, computer games.

By reading this modest book you should

- understand what computer programs are and how their run at our computers
- find variables in alien programs you are interested in
- create simple hacks

2 Architecture of computers

Before we can start with examining applications such as computer games, we will need to understand at least some basics of *computer architecture*. Following section will give you rather brief overview of the architecture. In case you are familiar with terms such as *stack*, *heap*, *instructions* and *instruction set*, you can skip this chapter.

2.1 What are computer programs made of

So you probably knows that computers are running computer programs, written by fanatic programmers who sacrifice their life to create the best program ever doing their voo-doo with mystery computers.

In reality, **computer programs** are just sequentions of code. But what's code ? Well, that depends on what the program is written in. Usually, programs consist of **statements** and **expressions**,

which determine model the flow of program.

According to Dijkstra, *statements* can be listed into three general categories:

- **sequence** allows to define a block of commands which are processed one-by-one
- **control** statements can choose which way to go according to some conditions
- **iteration** defines commands which are executed repeatedly either for fixed-iteration times or till

condition for iteration is valid - for example, **for**, **while** or **do while**

- **assignment** copies values from one place to another

Expressions are made of operators and can be evaluated into value of some type. For example,

expression (sin(90 deg)*5.0 + 1.0) evaluates roughly to 6.*

The text state above can is demostrated in following C program:

```
// this is a simple function which takes a single parameter N
int fibonnaci(int n)
{
    // if N is lower than or equal to  1, return 1
    if(n <= 1) // IF is a control-flow statement
        return 1; // return is just a statement which assigns return
           value
    else {
        // otherwise compute value as a sum of previous two elements of
           Fibbonaci series
        // n-2 is an expression which computes a temporate value whis
           passed to fibonnaci function
        return fibonnaci(n-1)+fibonnaci(n-2);
    }
}
```

2.2 Stack vs heap

While implementing an algorithm, one may not be certain about how much of data he will need to store. The amout of data may be determined by users or other outter factors. To store the data in the program, computer programs utilize two memory models:

- **stack** is special region in global memory which has a specific mode of access. It can be

accessed randomly as for reading, but data can only be appended or poped from the top of stack. It is used implicitely when we create local variables in functions. Each thread has its own stack and stack is usually used to pass arguments to functions.

- **heap** is another specialized region in global memory which has its memory manager called

allocator which assignes memory subsection on demand. Heap can either be managed by standard library (that's that malloc does), operating system or even the program itself.

2.3 A CPU point of view

As you can see, programms are usually programmed using programming languages with *higher abstraction*. Hold on, why higher ? Is there any lower abstraction ? Actually, there is as programs are perceived differently by computers and will get to this in following section.

When programmes are done in abstract languages, they must be converted into form which can be interpreted by CPU. Most of modern CPUs are actually dumb - they only use a set of simple instructions which express simple operations with memory, registers and ALU unit.

CPUS are **state-machines**. That means their activity is controled by *state*, which they store in so-called **flags**. This machines have access to global memory (called RAM) and to **registers** that stores the data that CPU is currently working with. Actually, CPUs are such simple that there are no instructions to express iterations or selections explicitely. Instead, CPUs work with **jumps** and **conditioned jumps** which allow to express these high level abstractions.

What CPU does is that it has a special register called **instruction pointer** which stores address to global memory and denotes the current position in code. Before CPU runs, it **fetches** the instruction at pointed address, **decodes** and **executes** the instruction afterwards. Execution manipulates **data path** - it may lead to loading additional data from memory to register, manipulating with registers using arithemetics and logics or storing data back to memory.

Consider the example of calculating `fibonnaci` function. We have already seen its C version. So how would a C function look like in CPU-instruction version ?

Here is an example snippet in a pseudo-Assembly language:

```
# Beggining of fibonnaci
fibonnaci:
COMPARE n, 1
JUMP IF EQUAL zeroAndOneCase
# If JUMP is not taken, this section is executed

# Store our current n argument
PUSH n
# Execude n-1
SUBTRACT n, 1
# Call fibonnaci function with new argument n
CALL fibonnaci
ADD sum, returnValue
# Get the original argument
POP n
# SUBTRACT n, 2
# Call fibonnaci function with new argument n
CALL fibonnaci
ADD sum, returnValue
POP n
MOVE returnValue, sum
RETURN
# zeroAndOneCase section of code is only accessed using JUMP above
zeroAndOneCase:
MOVE returnValue, 1
RETURN
```

Please, note that I made up the names of instructions in order to make the example more concise.

Except of lines starting with #, each line stand for an **instruction** which is executed by CPU. The execution of instruction is influenced by flags (for example, JUMP IF EQUAL would check EQUAL

flag before deciding where to continue.

So, as you could see, in reality, CPUs are executing simple instructions which has no knowledge of the program itself - instructions only access registers, flags and global memory.

2.4 Compilation & Linking & Execution

Up to now, we've seen how programs are written in languages supporting abstract constructions and we've noted the fact that CPU can't work this way - it requires rather dumb sequence of instructions, encode in CPU's specific instruction set code. So, how do we get there ? We use *compilers*.

Compilers are computer programs which analyze other's program source codes and converts (compiles) program from one computer language to another (possibly CPU sequence of instructions). Naturally, the conversion preserves program's functionality so that both programs act equally. Compilers analyze source code at different levels. Just briefly - characters which made the code are gathered and analyzed to detect what kind of *lemma* (sentence member) are. Subsequently, syntax trees are construct using lemmas and result in *abstract syntax trees* which can be either exectuced (that's how interpreted languages such as Java/Python might work) or stored in different language representing the parsed nodes of AST.

As a game hacker, you should understand that compilation may transform the source code into something which looks completely different than the original. This covers stripping the names of varibles, functions and commentaries stored in the code.

3 Most common Assembly instructions

Hey ! This text covers the most common x86 instructions that you can encounter with while hacking our beloved games ! Before we can start with enumerating and explainging instructions, we will need to

discuss at least basics of how CPU works.

3.1 About CPUs

CPU stands for *central processing unit*. What a regular CPU does is expressed in 5 instruction process stages:

1. Fetch
2. Decode
3. Execute
4. Read
5. Write back

During the first stage, *fetch*, the CPU loads the instuction from **RAM**. Which instruction should it load ? The address is stored inside so called **instruction pointer** or **program counter**, which is refered as **EIP** at x86 or **RIP** in 64-bit world.

When instruction is fetched into *instruction register*, the CPU must determine the semantics and actions which it should sort out for given instruction. This stage is called **decode** and causes that required signals inside CPU circuit are set up for the next stage.

In the next stage, the instruction semantics is procceeded. This usually covers doing *arithmetic-logic operations* such as summing, multiplying, dividing or *bitwise operations* such as logical AND or OR.

During this stage, instruction can also manipulate with **program counter** and conditionally change the address of the next instruction. That's how programs are branched and how it all works.

But, hold on, what is being added or divided ? What values does it operate on ? Well, the processor has either two options how to get data it works with - either data are stored inside **registers**, which are specialized and size-bounded variables and which can be filled in using specialized CPU instructions.

The second option is that the instruction somehow referes the value outside of CPU. The value which we operate on can be hard-coded inside the instruction code (and thus already retrieved during *fetch* stage), or it's stored at addresses, which is mentioned in the instruction. In that case, the whole instrucion process must stall for a single cycle in order to load data from RAM to immediate register.

Then, when result is retrieved after successfully operating over *ALU unit*, the result must be written back, which happens in **write back** stage.

To sum it, processors process *instructions* to find out what should happens to data stored in *registers* and in RAM.

3.2 Stack

 {#svgexample-figure}

To be honest with you, there is still something I haven't mentioned yet - stack. **Stack** is a range of addresses inside RAM, allocated for it. As the name says, this addresses are usually being manipulated in terms of *stacking* - new values are **pushed** onto stack, and afterwards **popped** from the top.

What can be this good for ?

1. Temporary variables and midresults are stored at stack, because there is a fixed number of registers

and it's easier to organize data in layers which are pushed and poped

1. Stack frames for functions. Whenever a function is called, the caller saves the current function state onto stack

and the address for returning back. This also allows us to call the same function recursivelly - each time the function is called the variables are stored at stack, so the next recursive call won't overwrite our current values

CAUTION !!! In convention, the stack works in reversed. When pushing data onto stack, the stack address **decreases**, and when popping it **increases**. The idea is that your **heap** and **stack** are stored inside the same memory and they both grow towards each other.

3.3 Instructions

3.3.1 mov dest, src

This instruction stands for moving - actually, it *copies* data from *source* to *dest*.

3.3.2 jmp dst

This instruction changes **program counter** to dst uncoditionally. That means the program will continue at location *dest*.

3.3.3 CMP a,b

Compares integer values of both resources. It result of comparison is stored in **flag register**.

3.3.4 jz/jnz/je dest

This is a conditional version of *jmp* instruction. The jump is done if and only if condition is fullfilled. The fullfillness is stored inside so-called **flag register** which can be set by comparison instructions such as **CMP**, **AND**.

Character 'N' means negation of condition.

3.3.5 call dest

Calls function at dest. Usually, arguments are either pushed at stack using **push** instruction, or stored inside devoted **regiters** (e.g. ECX is used for storing class instance's address). If you are insterested, search for **calling conventions**.

3.3.6 ret

Jump to the next instruction after previous **CALL**.

3.3.7 push value

Pushes value onto stack. The stack address after completing

3.3.8 label:

Stand for marked addresses which we can use for jumping and calling.

3.4 A simple program

Consider following programm:

```
mov eax, 0xF
cycle:
dec eax
cmp eax, 0x0
jne cycle
end:
```

We will examine the program code step-by-step:

```
mov eax, 0xF
```

The first line uses *mov* instruction for setting the initial value in *EAX* register. The initial value is *0xF*, which is 16 in decimal form. In computer world, 0x prefix usually *implies hexadecimal notation.*

```
cycle:
```

cycle label denotes the address of cycle beginning.

```
dec eax
```

Each time the instruction is executed the value in EAX get decreased.

```
cmp eax, 0x0
```

The instruction compares the current value in EAX register with 0. If both values are equal, then **zero flag** is set. If EAX is greater, **above flag** is set.

```
jne cycle
```

Finally, **jne** stands for *jump if not equal. The instruction checks for zero flag. If the flag isn't set, it jumps.*

As you can see, the previous program iterates over values 16, 15, 14 ... 0 and then it terminates.

4 #1 - Turning on godmode

The following text will reveal you how to become a god in almost any game easily. We will choose Mafia as it is my favourite game.

At first, start the game and start playing. When your game is ready and you are free to move with your character, then start up your favourite memory searching program. We will use *Cheat Engine* for this purposes.

Now here is the plan. Your character is most likely just a object inside the game's engine, which handles its attributes. To become a god, we will need to *prevent game rewriting object's attribute*, which stands for player's health. But we need to find the address of heath at first !

4.1 A trivial algorithm how to find object's health

1. Find out what's the value of your current health. Usually, this is displayed somewhere in HUD.
2. Search for all addresses in game which has the exactly same value as your health is.
3. Decrease your healt ! Bump into objects using a car, get yourself shot by a mobster or fall down from the hill !
4. Now use *Cheat Engine* to remain only those addresses which has the same value as your current health.

5. Repeat from step 3 until the set of addresses is small-enough and can be examined manually.

After a few iterations of the algorithm, two scenarious can occur:

1. You have found right address which when written to handles the value you

put inside and actually controls the behaviour of the game.

1. Your set has become empty - in this case, either you spoiled something

while looking for changed addresses, or the game doesn't keep the health value the way you thought so. In that case, you can use an alterned algorithm - instead of looking for an exact value, you can actually search for addresses, whose value has decreased / increased.

4.2 When all is done and said

Hurray ! You have just found an address which allows you to change your health. To become a god, you can use *Cheat Engine*'s feature which freezes the address.

4.3 What can I use this hack for

What we've just learnt is how simple attributes can be revelead using memory comparison methods. This algorithm can in general be used for:

- finding addresses for different objects and their attributes (health, armor, ammunition)