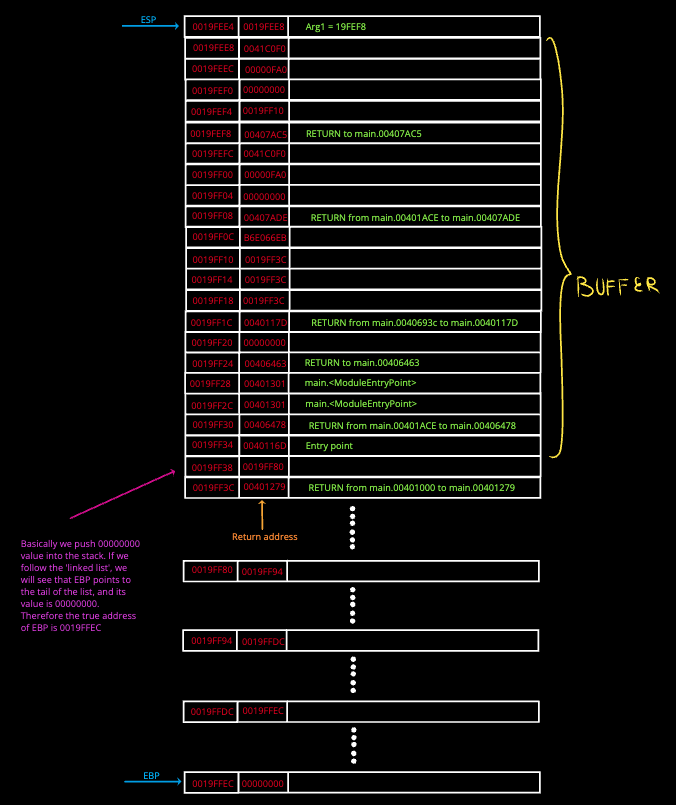
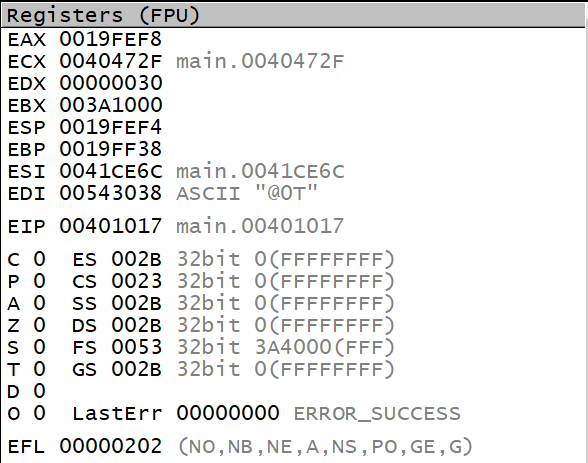
**-Buffer Overflow Homework Documentation**

Guidelines Section

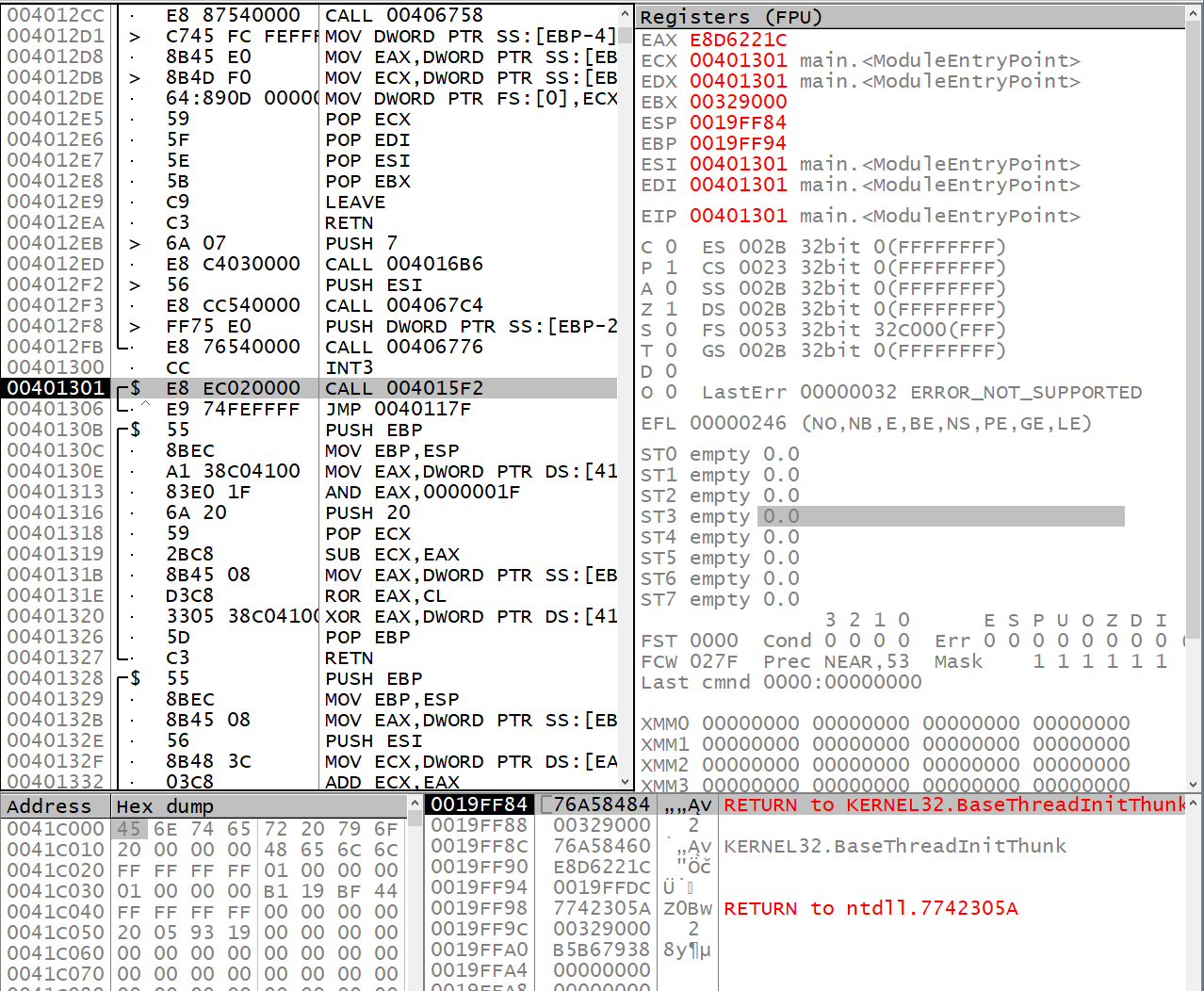
**===TASK\_ONE\_START===**

This is the stack layout of our main.c

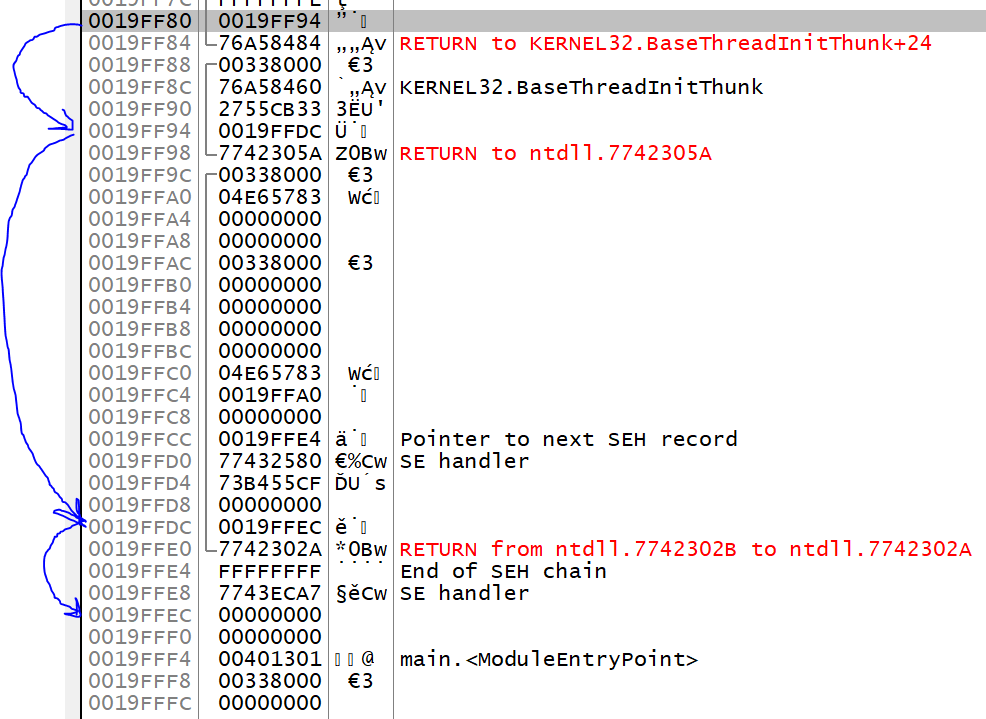


And the register’s data:  


The address of the stack that stores a return address is 0019FF3C.



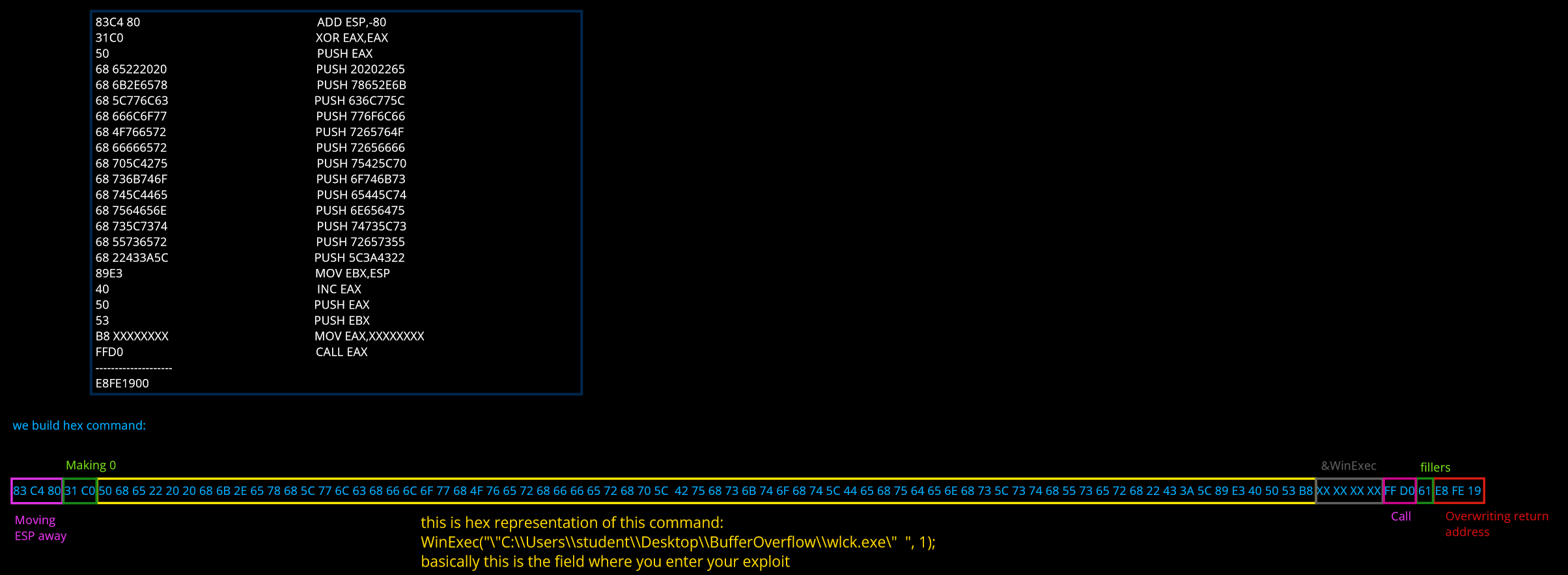
Regarding the EBP register, whenever we start our application, EBP points to 0019FF94. After calling 004018D0, EBP re-points to 0019FF80 which points to 0019FF94, which also points to 0019FFDC which also points to 0019FFEC which ALSO points to 00000000. That is the border of the program’s stack.



**===TASK\_ONE\_END===**

**===TASK\_TWO\_START===**

Here is the design of our exploit



It is just happened, that the buffer we had is plenty enough for our exploit. In other cases, you must fill the rest of the buffer with some fillers like NOP instructions or any other random characters, in our case ‘a’.

Now, speaking of our assembly code:

00401001 83C4 80 ADD ESP,-80

00401004 31C0 XOR EAX,EAX

00401006 50 PUSH EAX

00401007 68 65222020 PUSH 20202265

0040100C 68 6B2E6578 PUSH 78652E6B

00401011 68 5C776C63 PUSH 636C775C

00401016 68 666C6F77 PUSH 776F6C66

0040101B 68 4F766572 PUSH 7265764F

00401020 68 66666572 PUSH 72656666

00401025 68 705C4275 PUSH 75425C70

0040102A 68 736B746F PUSH 6F746B73

0040102F 68 745C4465 PUSH 65445C74

00401034 68 7564656E PUSH 6E656475

00401039 68 735C7374 PUSH 74735C73

0040103E 68 55736572 PUSH 72657355

00401043 68 22433A5C PUSH 5C3A4322

00401048 89E3 MOV EBX,ESP

0040104A 40 INC EAX

0040104B 50 PUSH EAX

0040104C 53 PUSH EBX

0040104D B8 8039AD76 MOV EAX,76AD3980

00401052 FFD0 CALL EAX

This assembly code is designed to call the WinExec function with a specific path to an executable file as its argument. I will break down the code step by step:

ADD ESP,-80: Adjust the stack pointer by subtracting 80. This reserves some stack space for local variables or other uses.

XOR EAX,EAX: Clear the EAX register (set it to 0).

PUSH EAX: Push the value 0 onto the stack. This will be the second argument for the WinExec function call, which corresponds to a hidden window.

The following series of PUSH instructions push the path of the executable file onto the stack, in reverse order. The path is split into DWORD-sized chunks which are pushed onto the stack one by one.

These instructions push the string: "C:\Users\student\Desktop\BufferOverflow\wlck.exe".

MOV EBX, ESP: Move the current stack pointer value into the EBX register.

INC EAX: Increment the EAX register (set it to 1).

PUSH EAX: Push the value 1 onto the stack. This will be the second argument for the WinExec function call, which corresponds to a normal window without focus.

PUSH EBX: Push the address of the path string onto the stack. This will be the first argument for the WinExec function call.

MOV EAX,76AD3980: Load the address of the WinExec function into the EAX register. Note that this address may be different on different systems or runs, so it needs to be updated dynamically.

CALL EAX: Call the WinExec function using the address stored in the EAX register.

In summary, this assembly code calls the WinExec function with a hard-coded path to an executable file as its argument and a normal window without focus as the window style.

There are several characters that are considered "impossible" because they cannot be easily printed or displayed on a screen. Some of these characters include:

Carriage Return (CR): This character moves the cursor to the beginning of a line of text.

Line Feed (LF): This character moves the cursor to the next line of text.

Tab (TAB): This character moves the cursor to the next tab stop.

Backspace (BS): This character moves the cursor one position to the left and erases the character there.

Escape (ESC): This character signals the start of an escape sequence that can be used to change text color or other formatting options.

Null (NUL): This character has no visual representation and is used to terminate strings or data.

These characters are often represented in text files using special escape sequences, such as \r for carriage return and \n for line feed, to make them visible and distinguish them from printable characters.

When reading user input using functions like gets() in C programming language, non-printable characters can cause several issues. For example, if the user enters a non-printable character like a carriage return (\r) or line feed (\n) as part of the input string, it may interfere with how the program processes the input. Overall these are the issues with those non-printable characters:

Input validation issues: Non-printable characters can interfere with input validation, making it difficult to ensure that user input is in the expected format or meets certain criteria.

Buffer overflow vulnerabilities: Non-printable characters can cause buffer overflows if the input buffer is not properly bounded or if the program does not account for them in its processing.

Command injection vulnerabilities: Non-printable characters can be used in command injection attacks, where an attacker injects malicious commands into user input and then tricks the program into executing them.

String termination issues: Non-printable characters can interfere with string termination, causing unexpected behavior or crashes in the program.

Encoding issues: Non-printable characters can cause encoding issues when reading or writing text files, particularly if the encoding is not compatible with the characters being used.

About those XXXX, this is the address of the WinExec function from kernel32.dll library.

**===TASK\_TWO\_END===**

**TASKS 3 AND 4 HAS BEEN UNCHACHED**

**===TASK\_THREE\_START===**

We have a problem regarding the WinExec.   
After each reboot the address of the function in Kernel32.dll changes, therefore we need to know the address first.   
  
I have made another simple app that gives you the address of the function, called: getWinExec.exe  
  
When launched - it will ask you to enter either "reload" or "exit". "Reload" will give you the address, "exit" will close the app.

After getting the address, we pass it through hexDex.exe and get the alt+decimal value in reversed order and paste it into xxxx chunk.

The reason WinExec was selected – is its simplicity in comparison with other process-creating functions, since they require complex assembly instructions to assemble and call a function. In addition, WinExec is called by a single line of code or 2 lines of instructions. (i guess).

The only issue is that the executable name is treated as the first white space-delimited string in lpCmdLine. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls WinExec using the Program Files directory will run this application instead of the intended application.

**===TASK\_THREE\_END===**

**===TASK\_FOUR\_START===**

**WARNING, TURN OFF YOUR ANTIVIRUS, OR ELSE THE ANNOYING WINDOWS PROTECTION WILL JUST ERASE WLCK.EXE!!!**

The exploit launches wlck.exe which is a self-made winlocker. The source file is provided as well.   
The documentation regarding the wlck.exe is not presented, but it consists of several phases:  
  
1. Stopping explorer.exe

2. Closing all running apps, except for itself.

3. Injecting the copy of the executable into startup folder. (I actually thought to inject there something else)

4. A window pops up with the message about windows block.  
5. Entering password.  
6. Restarting the PC.  
  
To restore everything in emergency -> ctrl+shift+esc, create new process and enter explorer and close the virus.  
  
I actually had an idea to inject a NJRat file, but something went wrong and I decided to stop on the winlocker.

Regarding the guide to perform the buffer attack. After researching and compiling all information into one file, the guide would look like this:  
IF THE APP IS YOURS THEN FIRST DO THIS:  
compile it using a flag /GS- this will turn off buffer security, and after compiling, link it against the /DYNAMICBASE:NO and /NXCOMPAT:NO flags. Those will turn off random base address creation and turn off data execution prevention.

Do not forget to use old terminal!  
  
1. Define the size of the buffer allocated by the executable for inputting a string.

2. At the moment when an app requests your input, check the location of ESP and EBP.

3. Calculate the distance between ESP+4 and EBP -> DISTANCE = cast<int>((ESP+4) - EBP), where each stack frame consists of a hex value lenght of 8 (divide by 2 and you get 4 characters in each stack frame). The larger is buffer, the longer your and simplier it would be to enter an exploint.

4. You fill anything of lenght of DISTANCE + 4. The rest 4 characters are for the readressing a return address. That return address should point to the beginning of your exploit (ESP+4). So when function completes its work, the next instruction to be executed will be your exploit.

5. There might be a problem with esp. When reaching RETN and executing each line of you exploit, the ESP register will move downwards and re-writing assembly code. That might break your exploit, therefore it would be prefferable to move ESP somewhere far away, or place it at lower addresses than the address of your exploit start.

6. Draw a stack frame of your exploit and debug it so it could execute without any problems.

**===TASK\_FOUR\_END===**