Penetration Testing Professional

Assemblers, Debuggers and Tools Arsenal

Section 1: System Security – Module 2

v5.0

INTRODUCTION

The previous module showed you that Assembly is a very low-level programming language consisting of a mnemonic code, also known as an **opcode** (operation code).

Although it is a low-level language, it still needs to be converted into machine code in order for the machine to execute.

ASSEMBLER

2.2 Assembler

An assembler is a program that translates the Assembly language to the machine code. There are several different assemblers that depend on the target system's **ISA**:

- Microsoft Macro Assembler (**MASM**), x86 assembler that uses the Intel syntax for MS-DOS and Microsoft Windows
- GNU Assembler (**GAS**), used by the GNU Project, default back-end of GCC.
- Netwide Assembler (**NASM**), x86 architecture used to write 16-bit, 32-bit (IA-32) and 64-bit (x86-64) programs, one of the most popular assemblers for Linux
- Flat Assembler (**FASM**), x86, supports Intel-style assembly language on the IA-32 and x86-64

https://msdn.microsoft.com/en-us/library/afzk3475.aspx
https://www.gnu.org/software/binutils/

https://www.nasm.us/
http://flatassembler.net/

Those are only a few, and in this course, we will use **NASM**.

When a source code file is assembled, the resulting file is called an **object file**. It is a binary representation of the program.

While the assembly instructions and the machine code have a one-to-one correspondence, and the translation process may be simple, the assembler does some further operations such as assigning memory location to variables and instructions and resolving symbolic names.

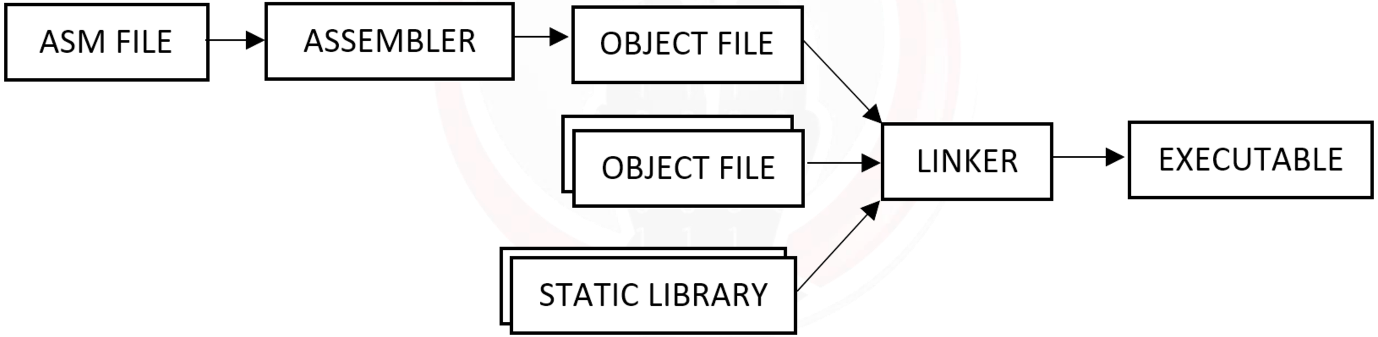
2.2 Assembler

Although exploring the details of the assembling process is not our purpose, it is important to know the process.

Once the assembler has created the object file, a **linker** is needed in order to create the actual executable file. What a linker does is take one or more object files and combine them to create the executable file.

An example of these object files are the *kernel32.dll* and *user32.dll* which are required to create a windows executable that accesses certain libraries.

The process from the Assembly code to the executable file can be represented here:



In the next few slides, you will learn how to perform this process manually and where to acquire the necessary tools.

COMPILER

The complier is similar to the assembler. It converts high-level source code (such as C) into low-level code or directly into an object file. Therefore, once the output file is created, the previous process will be executed on the file. The end result is an executable file.

This is important background knowledge, and although we will not cover the entire, process it is important to know the differences.

NASM

The assembler we are going to use is **NASM**, and to make things easier, we will use the NASM-X project. It is a collection of macros, includes, and examples to help **NASM** programmers develop applications.

You are free to configure your machine the way you want, but the Hera Lab configuration will match the following installation instructions.

https://forum.nasm.us/index.php?topic=1853.0

2.4.1 Installation Instructions

Step 1

Download the .zip file from here.

Step 2

Extract the files and save it to a folder on your computer. We will extract it in `C:\nasmx`. Make sure that your configuration does not have any spaces in the path.

https://sourceforge.net/projects/nasmx/

2.4.1 Installation Instructions

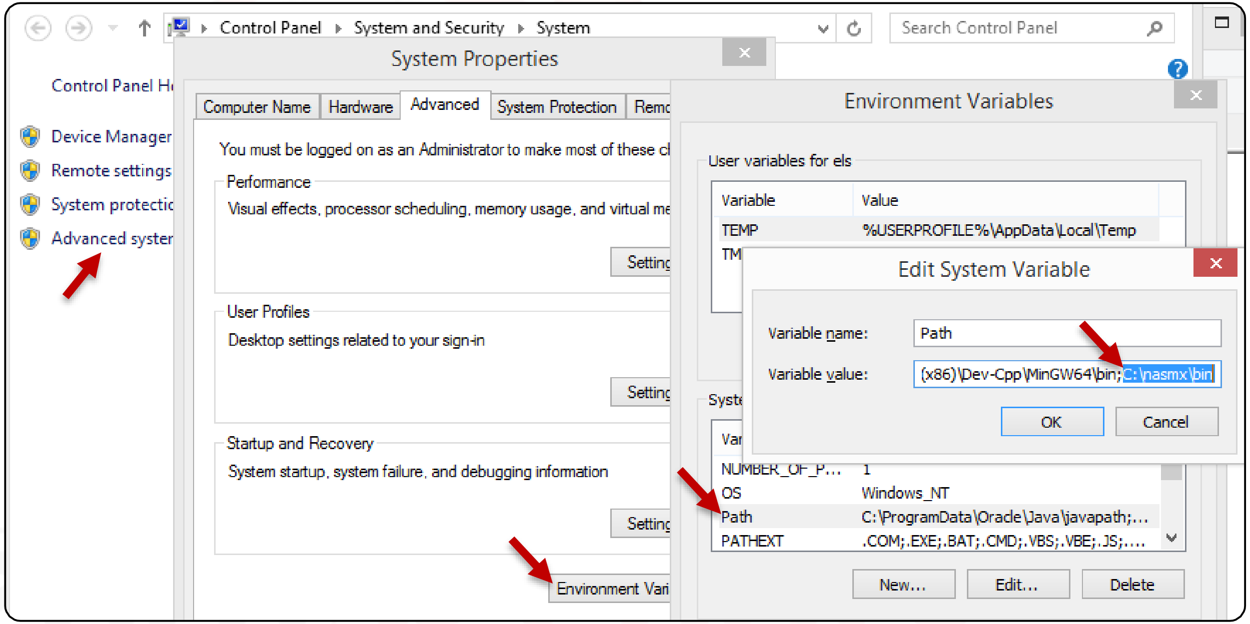
Step 3

Set Windows environment variables. Add in the **Path** variable the path to the **NASM-X** binaries.

This is done by opening the **Environment Variables** window, selecting the **Path** entry within the **System variable** section, at the bottom of the window, and adding the path `C:\nasmx\bin`.

2.4.1 Installation Instructions

This will allow us to use `nasm.exe` and all the other binaries from any path in our command line prompt.



2.4.1 Installation Instructions

Next, open the command prompt and navigate the `nasmx` folder. Once there, let us run `setpath.bat`.

You will see something like this:

```
C:\nasmx>setpaths.bat
"NASMX Development Toolkit"

C:\nasmx>
```

2.4.1 Installation Instructions

In order to work with the `nasmx` demos, we need to change one last thing. Navigate to the `C:\nasmx\demos` folder and edit the file named `windemos.inc`.

Comment the following line:

```
%include 'nasmx.inc'
```

And add this right after:

```
%include 'C:\nasmx\inc\nasmx.inc'
```

2.4.1 Installation Instructions

Finally, to verify that everything is configured properly, open a new terminal and navigate to the folder `C:\nasmx\demos\win32\DEMO1`.

Once in the folder, you should see three files:

- `demo1.asm` is the file containing the assembly code
- `demo1.bat` is a script that will automatically assemble and link the demo1.asm file to obtain the executable file
- `makefile` contains all the data and commands needed to transform the source code files to an executable program, but for now, we don't need it

To assemble the `demo1.asm` file we have to run the following command:

```
nasm -f win32 demo1.asm -o demo1.obj
```

With the `-f` option we are telling **NASM** what the output file format we want is. In this case, it is Microsoft object file format for 32-bit OS. If everything works correctly, it will create a new file in the folder called `demo1.obj`.

2.4.1 Installation Instructions

The final step to obtain the executable file is to use the linker, which is stored in the same folder of `nasm.exe`, and it is called `GoLink.exe`.

```
GoLink.exe /entry _main demo1.obj kernel32.dll user32.dll
```

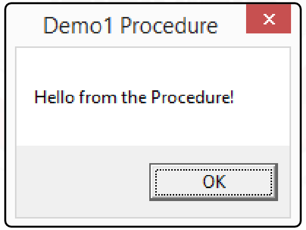
With this command, we are instructing the linker to link `demo1.obj` with the two `dll` files, `kernel32`, and `user32`.

These libraries are required for the program to work correctly because the program calls functions like `MessageBox`, which are defined in these libraries. Moreover, we are telling the linker that the entry point for our program is `_main`.

We will introduce this shortly.

2.4.1 Installation Instructions

If the command succeeds, you should see a new file named `demo1.exe`; this is an executable that we can run. Now we can execute the file and see the program running on our machine. If everything went well, you should see the following window:



If not, don't worry, just go back and verify that the setup configuration matches what is here.

2.4.2 ASM Basics

Now that we have assembled our first assembly file, let's first introduce **ASM** and x86 basics.

Our purpose is not to teach you how to write assembly programs, but it is essential that we introduce a few instructions and concepts that will help you better understand the few next modules.

2.4.2 ASM Basics

High-level functions such as `strcpy()` are made of multiple **ASM** instructions put together to perform the given operation (copy two strings).

The simplest assembly instruction is **MOV** that moves data from one location to another in memory.

2.4.2 ASM Basics

Most instructions have two operands and fall into one of the following classes:

Instructions

Data Transfer: `MOV, XCHG, PUSH, POP`	**Arithmetic:** `ADD, SUB, MUL, XOR, NOT`
Control Flow: `CALL, RET, LOOP, Jcc` (where cc is any condition)	**Other:** `STI, CLI, IN, OUT`

The following is an example of a simple assembly code that sums two numbers:

```
&lt;/&gt;
MOV EAX,2          ; store 2 in EAX
MOV EBX,5          ; store 5 in EBX
ADD EAX,EBX        ; do EAX = EAX + EBX operation
                   ; now EAX contains the results
```

As you already know, **ASM** deals directly with the registers and memory locations, and there are certain rules for each assembly language.

For example, each opcode is represented in one line and contains the instruction (ex: **MOV**, **ADD**, etc.) and the operands used by the opcode. The number of operands may vary depending on the instruction.

2.4.2.1 Intel vs AT&T

More importantly, depending on the architectural syntax, instructions and rules may vary. For example, the source and the destination operands may be in different position.

Here's an example:

	Intel (Windows)	AT&T (Linux)
Assembly	MOV EAX, 8	MOVL $8, %EAX
Syntax	`<instruction><destination><source>`	`<instruction><source><destination>`

2.4.2.1 Intel vs AT&T

As you can see, AT&T puts a percent sign (`%`) before registers names and a dollar sign (`$`) before numbers.

Another thing to notice is that AT&T adds a suffix to the instruction, which defines the operand size: `Q` (quad – 64 bits), `L` (long – 32 bits), `W` (word -16 bits), `B` (byte – 8 bits).

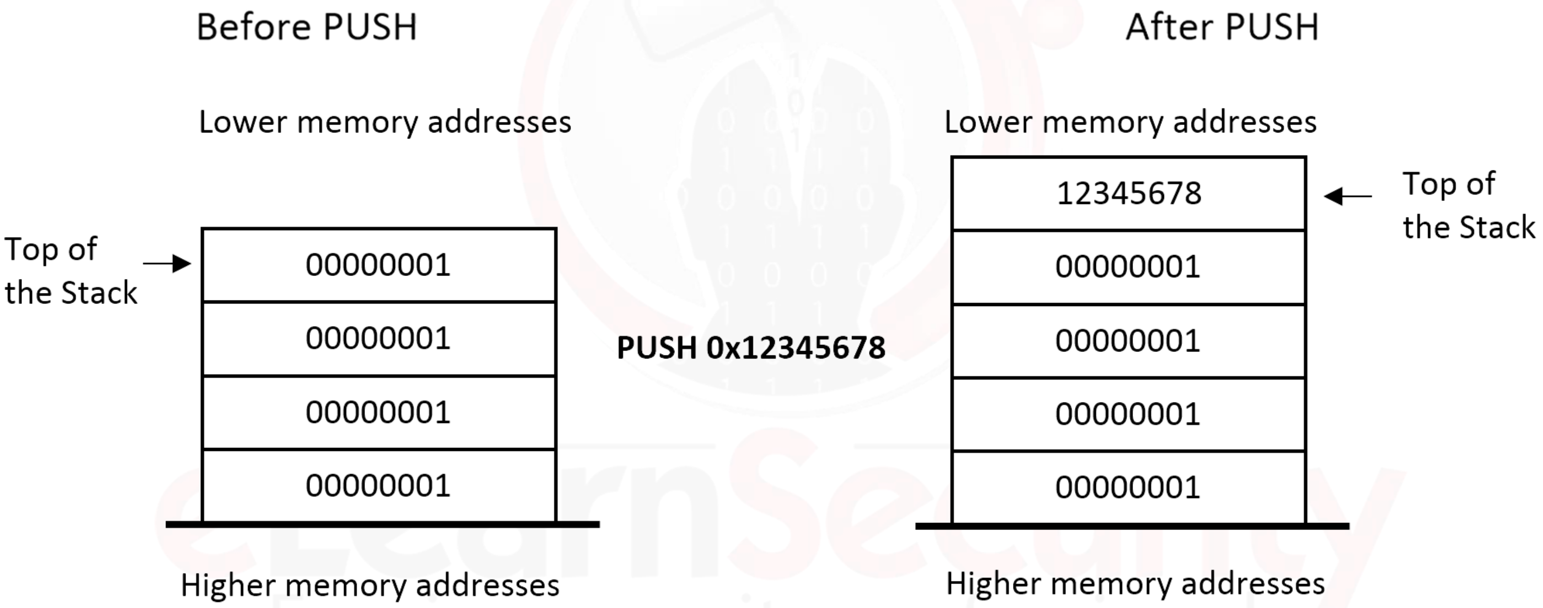
2.4.2.2 PUSH Instruction

In the previous module, we introduced two very important assembly instructions that allow us to work with the stack: **PUSH** and **POP**.

Since these are very important operations, let us inspect them more in details.

2.4.2.2 PUSH Instruction

PUSH stores a value to the top of the stack, causing the stack to be adjusted by -4 bytes (on 32 bit systems): `-0x04`.



2.4.2.2 PUSH Instruction

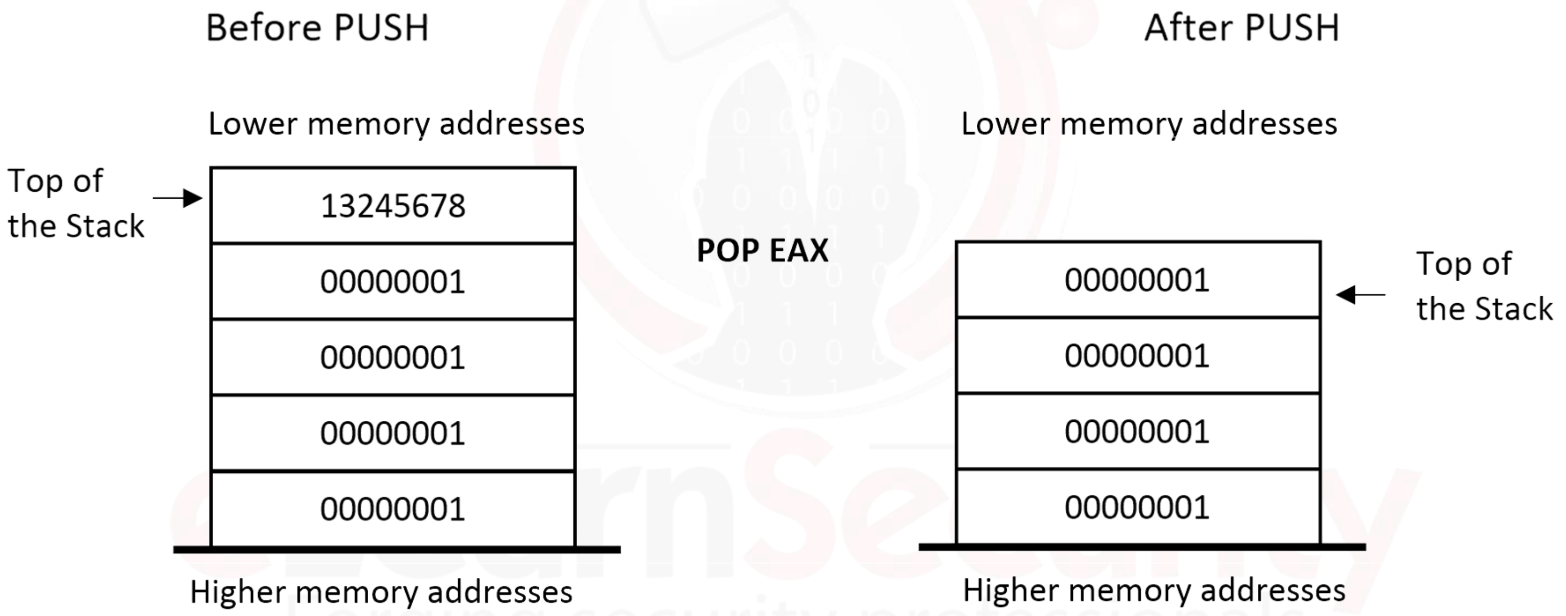
Another interesting fact is that the same operation (`PUSH 0x12345678`) can be achieved in a different way.

For example, we can use the following two instructions:

```
SUB ESP, 4                 ;subtract 4 to ESP -> ESP=ESP-4
MOVE [ESP], 0x12345678 ;store the value 0x12345678 to the location
                           ;pointed by ESP. Square brackets indicates to
                           ;address pointed by the register.
```

2.4.2.3 POP Instruction

POP reads the value from the top of the stack, causing the stack to be adjusted `+0x04`.



2.4.2.3 POP Instruction

The **POP** operation can also be done in several other instructions:

```
MOV EAX, [ESP]   ;store the value pointed by ESP into EAX
                 ;the value at the top of the stack
ADD ESP,4        ;Add 4 to ESP – adjust the top of the stack
```

2.4.2.4 CALL instruction

Subroutines are implemented by using the **CALL** and **RET** instruction pair.

The **CALL** instruction pushes the current instruction pointer (**EIP**) to the stack and jumps to the function address specified. Whenever the function executes the **RET** instruction, the last element is popped from the stack, and the CPU jumps to the address.

2.4.2.4 CALL instruction

Here is an example of a **CALL** in assembly:

```
MOV EAX,1               ; store 1 in EAX
MOV EBX,2               ; store 2 in EBX
CALL ADD_sub ; call the subroutine named ADD_sub
INC EAX   ; Increment EAX: now EAX holds "4"
                        ; 2(EBX)+1(EAX)+1(INC)
JMP end_sample
ADD_sub:
ADD EAX, EBX
RETN                    ; Function completed so return
                        ; back to the caller function
end_sample:
```

2.4.2.4 CALL instruction

The following is an example of how to call a procedure. This example begins at `proc_2`:

```
proc proc_1
        Locals none
        MOV ECX,[EBP+8] ;ebp+8 is the
        PUSH ECX                ;function argument
        POP ECX
endproc
proc proc_2
        Locals none
        invoke proc_1, 5 ;invoke proc_1 proc
endproc
```

Teaching the assembly language is out of the scope of this course, but the previous example will help you to get familiar with it.

There are tons of additional resources on the web if you are interested in learning more.

Tools Arsenal

2.5 Tools Arsenal

During this course, we will use different tools and software. The most important are the C/C++ compilers and the debuggers. Although we will not go into great details about the compiler, we will closely inspect the debuggers and their features.

Moreover, we will give you different tool options, so you are free to use the one that best fits your needs.

2.5.1 Compilers

Since we will write some C and C++ code, you will need a compiler. If you are not confident in programming in C or C++, don't worry. The coding we will do will not be that complicated, and as long as you follow along and understand the concepts, you will do fine.

To make things easier, we will use an IDE to manage all the files.

2.5.1 Compilers

We have different options, such as the commercial Microsoft Visual C/C++ (Visual Studio) or free software like Orwell Dev-C++, or Code::Blocks and so on.

Ideally, it should be one that you are familiar with, but they perform the same functions.

https://www.microsoft.com/en-us/download/details.aspx?id=48145
http://orwelldevcpp.blogspot.com/
http://www.codeblocks.org/

For our purposes, Dev-C++ is enough. It offers a good IDE and all the required compiling and decompiling tools.

Although compiling an application from the IDE is very simple and straightforward, you can also compile an application from the command line. As you will see, this will be very useful later with exploit and payloads.

2.5.1 Compilers

Dev-C++ creates a directory named MinGW64 where all the compiling tools are stored.

MinGW64 is an improved version of the MinGW (Minimalist GNU compiler for Windows), which provides programming tools for developing native MS-Windows applications.

http://mingw-w64.org/doku.php
http://mingw.org/

2.5.1 Compilers

Once you install Orwell Dev-C++, remember to add the bin folder to your windows environment variables so that you can run `gcc` from any path in the command prompt.

To compile `.c` or `.cpp` files we can use the `gcc.exe` compiler found in the `bin` folder:

http://orwelldevcpp.blogspot.com/

2.5.1 Compilers

Once the environment variables are set, extract the file

- `HelloStudents.c`

contained in the `2_Assembler_Debuggers_Tools.rar` package from the *Members area* and save it locally.

To compile the C file using `gcc` we can run the following command:

```
gcc -m32 HelloStudents.c -o HelloStudents.exe
```

2.5.1 Compilers

With the previous command, we are telling the compiler (`gcc`) to compile the file `HelloStudents.c` for 32-bit environments (`–m32`) and then output the compiled version to a file named `HelloStudents.exe`.

2.5.1 Compilers

If the command succeeds, you should be able to run the executable from your command prompt and see the following message:

Command Prompt

```
C:\samples>gcc -m32 HelloStudents.c -o hellostudents.exe

C:\samples>hellostudents.exe
Hello students!

C:\samples>_
```

2.5.1 Compilers

While all the compiling options are out of the scope of this course, you can check the **gcc** manual here, to learn about all the different possibilities.

It is important to know that every compiler will produce a slightly different output. Therefore, the same source code compiled with different compilers (such as *Microsoft Visual Studio*, *MinGW*, *GCC*, etc.) may produce different machine codes.

https://gcc.gnu.org/onlinedocs/

2.5.2 Debuggers

A debugger is a program which runs other programs, in a way that we can exercise control over the program itself. In our specific case, the debugger will help us to write exploits, analyze programs, reverse engineer binaries and much more. As we will see, the debugger allows us to:

- Stop the program while it is running
- Analyze the stack and its data
- Inspect registers
- Change the program or program variables and more

2.5.2 Debuggers

There are several debugs available. During this course, we will use Immunity Debugger. It is a good suggestion to have a general overview of a couple of these debuggers because some companies may prefer one over another. Here is a small list of some of the most popular ones:

- IDA (Windows, Linux, MacOS)
- GDB (Unix, Windows)
- X64DBG (Windows)
- EDB (Linux)
- WinDBG (Windows)
- OllyDBG (Windows)
- Hopper (MacOS, Linux)

https://www.immunityinc.com/products/debugger/
https://www.hex-rays.com/products/ida/
https://www.gnu.org/software/gdb/
https://x64dbg.com/#start
http://codef00.com/projects#debugger
https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk
http://www.ollydbg.de/
https://www.hopperapp.com/

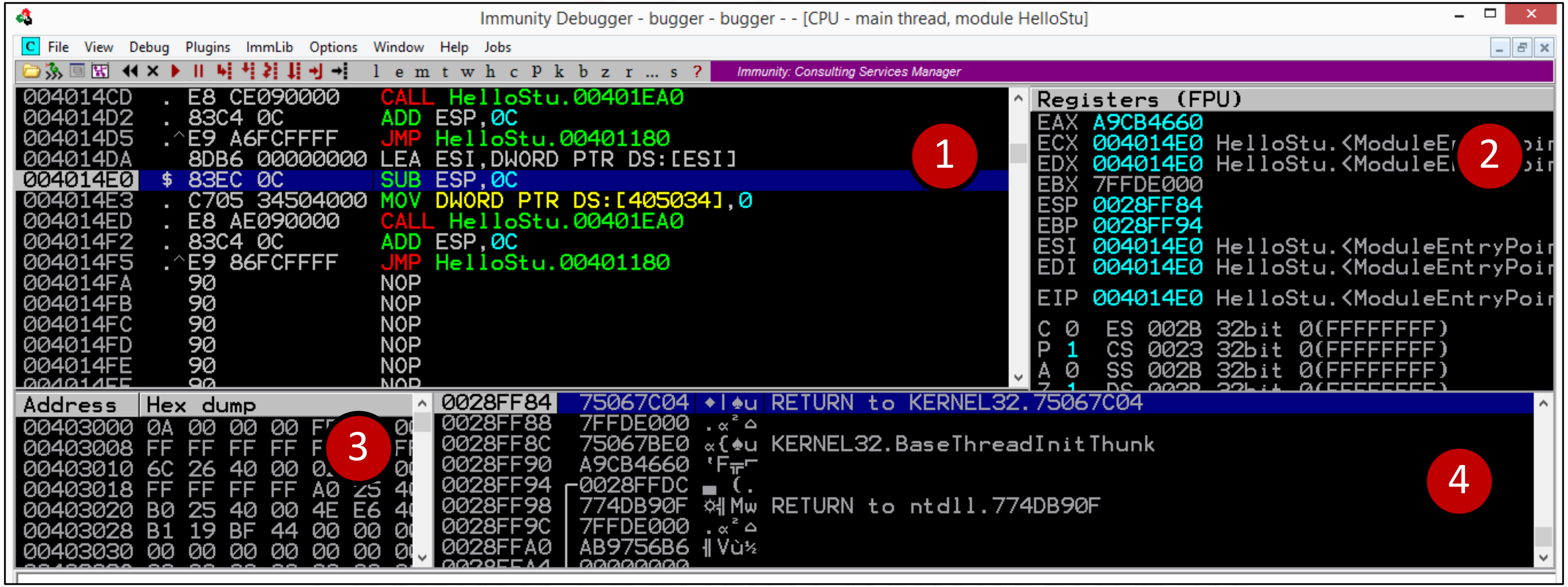
2.5.2 Debuggers

First, we'll give you a quick breakdown of how the Immunity Debugger GUI looks and operates. We will then use the GUI by running the samples previously built and provided by the Immunity Debugger.

This will help you understand how programs are interpreted on your system, which is the main prerequisite for enjoying real exploitation.

2.5.2 Debuggers

In the following slide we will examine the Immunity Debugger GUI and the meaning of each panel:



Panel 1

The Disassembler Panel is not only the main window, but it is the most important panel. Here is where all the Assembler code is produced or viewed when you are debugging a module.

- In the **first column** is the address location
- In the **second column** is the machine code
- In the **third column** is the assembly language
- In the **fourth column** is the debugger comments

Panel 1

A quick review of the assembly language column should provide you with some familiar terms: **PUSH, MOV, CALL, POP,** and **RETN.**

2.5.2 Debuggers

Panel 2

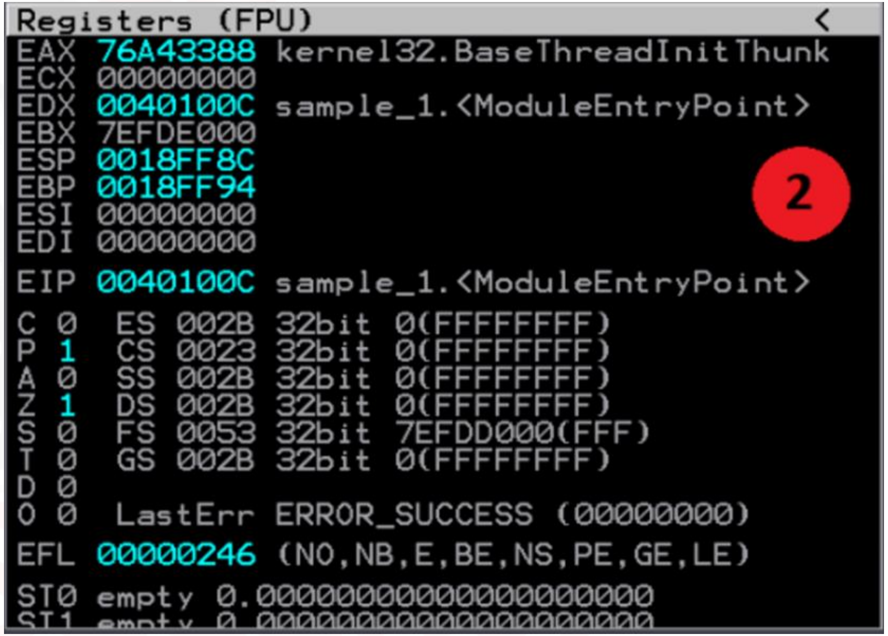
The Register Panel holds information on standard registers. Here you can see:

- Names of registers
- Their content
- If a register points to an ASCII string, the value of the string

Panel 2

You can think of this as working similar to a train or plane schedule.

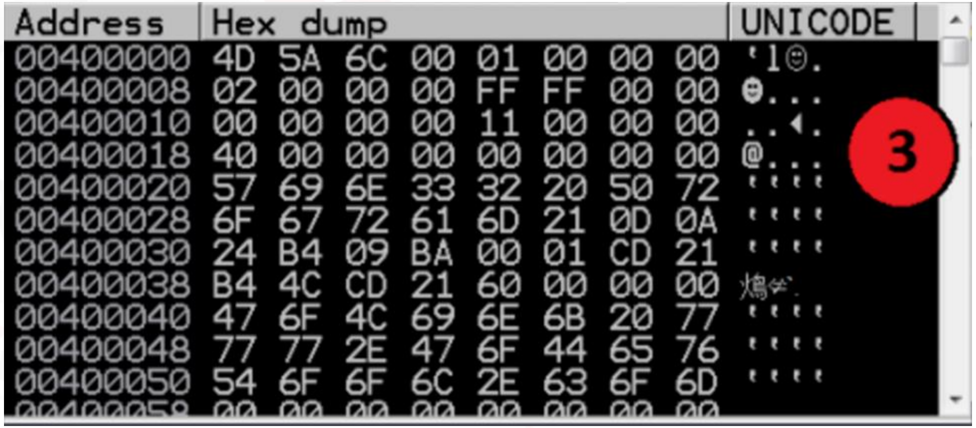
As the program progresses and registers are changed or updated, it is easily noted and observed in this panel.



2.5.2 Debuggers

Panel 3

The Memory Dump Panel shows the memory locations and relative contents in multiple formats (i.e., hex, UNICODE, etc.).



2.5.2 Debuggers

Panel 4

The Stack Panel shows the current thread stack.

- In the **first column** is the addresses
- In the **second column** is the value on the stack at that address
- In the **third column** is an explanation of the content (if it's an address or a UNICODE string, etc.)
- In the **fourth column** is the debugger comments

2.5.2 Debuggers

Now that you know the main parts of Immunity Debugger, we can proceed with the analysis of one of our samples:

- Compile the file *demo1.asm* (located in the NASMX folder)
- Watch the following video that will show you how to load and inspect files in Immunity Debugger.
- Load exe file in Immunity Debugger
- Explore, question, think and finally, understand how it works on your machine.

Video: Immunity Debugger

If you have a **FULL** or **ELITE** plan you can click on the image on the left to start the video

2.5.3 Decompiling

The last topic in this module is how to decompile applications. So far we have seen how to compile and assemble applications. But, in order to be a successful pen tester, you need to have the knowledge to reverse that operation.

What do you do if you are given an executable and asked how it works? How can you disassemble it in order to obtain the assembly code?

2.5.3 Decompiling

We have different options and tools that allow us to do this. In the same Dev-C++ folder where the `gcc` executable is located, is a file called `objdump.exe`.

The purpose of this file is to disassemble executable programs.

Decompiling example:

File: `HelloStudents.exe` (file we compiled earlier)

Command:

```
objdump -d -Mintel HelloStudents.exe > disasm.txt
```

Switches:

- `-d` option tells the tool to disassemble the input file
- `-Mintel` is a disassembler option that allows us to select disassembly for the given architecture (Intel in our case)
- `>` tells the command what the output file is to be called

2.5.3 Decompiling

If the command succeeds, we should have a new file named `disasm.txt`, containing the assembly code of the program.

This will be very useful later to identify the parts of the program that you want to test or exploit.

2.5.3 Decompiling

Notice that this is something that a debugger like Immunity Debugger automatically does. The following is the assembly code of the function `main()` in `HelloStudents`.

The first column contains the addresses in memory, the second column contains the opcodes (machine code), while the third column contains the assembly code.

2.5.3 Decompiling

The following image shows what we obtain:

```
00401500 <_main>:
  401500:   55                        push   ebp
  401501:   89 e5                     mov    ebp,esp
  401503:   83 e4 f0                  and    esp,0xfffffff0
  401506:   83 ec 10                  sub    esp,0x10
  401509:   e8 72 09 00 00            call   401e80 <___main>
  40150e:   c7 04 24 00 40 40 00      mov    DWORD PTR [esp],0x404000
  401515:   e8 de 10 00 00            call   4025f8 <_puts>
  40151a:   b8 00 00 00 00            mov    eax,0x0
  40151f:   c9                        leave
  401520:   c3                        ret
```

2.5.3 Decompiling

Congrats on making it through the fundamentals required to work with executables, debuggers, ASM languages.

We can finally get our hands dirty and start working on buffer overflows!

REFERENCES

References

NASM
https://www.nasm.us/

MASM
https://www.microsoft.com/en-us/download/details.aspx?id=12654

GAS
https://www.gnu.org/software/binutils/

FASM
http://flatassembler.net/

NASM-X
https://forum.nasm.us/index.php?topic=1853.0

Microsoft Visual C/C++
https://www.microsoft.com/en-us/download/details.aspx?id=48145

Orwell Dev-C++
http://orwelldevcpp.blogspot.com/

Code::Blocks
http://www.codeblocks.org/

MinGW64
http://mingw-w64.org/doku.php

MinGW
http://mingw.org/

Immunity Debugger
https://www.immunityinc.com/products/debugger/

IDA
https://www.hex-rays.com/products/ida/

OllyDBG
http://www.ollydbg.de/

GDB
https://www.gnu.org/software/gdb/

Hopper
https://www.hopperapp.com/

WinDBG
https://docs.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk

Immunity Debugger