Writing Stack Based Overflows on Windows
Part II - Windows Assembly for writing Exploits

Nish Bhalla, 31st May, 2005

(Nish[a-t]SecurityCompass.com)

www.SecurityCompass.com

Windows Assembly For Exploit Writing: Part II / IV

The following section covers the basics of X86 assembly language, that is needed to get a better understanding on writing exploits. This is not an exhaustive lesson in assembly language; however, it gets you started on assembly and reading assembly instructions.

Registers

Before we look at an actual assembly example let us get up-to speed on the terminology and a little background on assembly language.

Assembly language is the symbolic representation of machine code. Machine code a.k.a. Op Code (operational code) are the instructions coded as bit strings. The CPU executes these instructions once the executable is loaded into memory.

The CPU like a typical program needs to store information in a temporary store. The registers are considered temporary store for the CPU. Though the processor can directly operate on the data in memory (RAM), the instructions are executed faster if the data is stored in registers (CPU's temporary storage).

Registers are classified according to the functions they perform. In general there are 16 different types of registers. These registers are classified into four major types; namely the general purpose registers; the registers that hold data for an operation, the segment registers; the registers that hold the address of instructions or data, the status registers; the registers that help keep the current status and the EIP register; the register that stores the pointer to the next instruction to be executed.

The registers we will cover in this section are mainly the registers that would be used in understanding and writing exploits. We will not go into detail on the most of the registers. The ones we will take a close look are mainly the general purpose registers and the EIP register.

There general purpose registers; EAX, EBX, ECX, EDX, EDI, ESI, ESP & EBP are provided for general data manipulation. The "E" in these registers stands for extended to address the full 32-bit registers which can be directly mapped to the 8086 8 bit registers. Mapping of the 8 bit registers to 32 bit registers is displayed below (please note for details of 8 bit or 16 bit registers a good reference point it is the IA-32 Intel Architecture software developer's manual; Basic Architecture, Order Number 245470-012 available from http://developer.intel.com/design/processor/).

32 BIT Registers	**16 BIT Registers**	**8 Bit Mapping (0-7)**	**8 Bit Mapping (8-15)**
EAX	AX	AL	AH
EBX	BX	BL	BH
ECX	CX	CL	CH

EDX	DX	DL	DH
EBP	BP		
ESI	SI		
EDI	DI		
ESP	SP		

Table1 : Mapping Register To 8 Bit Registers

These general purpose registers consist of the indexing registers, the stack registers and additional registers. The 32 bit registers can access the entire 32 bit value. For example if the value 0x41424344 is stored in EAX register, then performing an operation on EAX would be performing an operation on the entire value 0x41424344, however, if instead just AX is accessed, then only 0x4142 will be used in the operation, if AL is accessed, then just 0x41 will be used and if AH is accessed then only 0x42 would be used. This would be useful when writing shellcode.

Indexing Registers

EDI and ESI registers are indexing registers; they are commonly used by string instructions as source (EDI) and destination pointers (EDI) to copy a block of memory.

Stack Registers

The ESP and EBP registers are primarily used for stack manipulation. EBP (as seen in the previous section), points to the base of a stack frame and ESP points to the current location of the stack. EBP, is commonly used as a reference point when storing values on the stack frame (refer example 1, hello.cpp)

Other General Purpose Registers

EAX, also referred as the accumulator register, is one of the most used registers and contains the result of many instructions. EBX is a pointer to the data segment. ECX is commonly used as a counter (for loops etc). EDX is an I/O pointer. These four registers are the only four registers that are byte addressable, i.e. accessible to the byte level.

EIP Register

The EIP register contains the location of the next instruction that needs to be executed. It is updated each time an instruction is executed such that it will point to the next instruction. Unlike all the registers we have discussed thus far which were used for data access and could be manipulated by an instruction, EIP cannot be directly manipulated by an instruction (an instruction can not contain EIP as an operand). This is important to note when writing exploits.

Data Type

The fundamental data types are; a byte of eight bits, a word of 2 bytes (16 bits) and a double word of 4 bytes (32 bits). For performance purposes, the data structures (specially stack) require that the words and double-words be aligned. A word or double word that crosses an 8 byte boundary requires two separate memory bus cycles to be accessed. When writing exploits the code sent to the remote system requires the instructions to be aligned to ensure fully functional / executable exploit code.

Operations

Now that we have a basic understanding of some of the registers and data types, let us take a look at some of the commonly seen instructions. Below are some of the common instructions and a brief explanation of what each of the instruction does. These instructions are instructions that are commonly seen in decompiled code. This of-course is just a very small list of instructions, there are many excellent reference points including the references guide provided by Intel (available for download from http://developer.intel.com/design/processor/.) that detail all the instructions.

Assembly Instructions	**Explanation**
CALL EAX	EAX contains the address to call
CALL 0x77e7f13a	Call WriteFile process from kernel32.dll
MOV EAX, 0FFH	Load EAX with 255
CLR EAX	Clear the register EAX.
INC ECX	ECX=ECX +1 or increment counter
DEC ECX	ECX= ECX-1 or decrement counter
ADD EAX, 2	ADD 2 to EAX
SUB EBX, 2	Subtract 2 bytes from EBX
RET 4	The current value of the stack is put into EIP.
INT 3	Interrupt 3 is typically a breakpoint; INT instructions allow a program to explicitly raise a specified interrupt.
JMP 80483f8	JMP simply sets EIP to the address following the instructions. Nothing is saved on the stack. Most if-then-else requires a minimum of one JMP instruction.

JNZ	Jump Not Zero
XOR EAX, EAX	Clear EAX register by performing an XOR set the value to 0
LEA EAX	Load Effective address stored in EAX
PUSH EAX	Pushes the values stored in EAX onto the stack
POP EAX	Pops the value stored in EAX

Hello World

To better understand the stack layout and assembly instructions let us take a standard hello world example and look into it in more detail.

The code below is the code for a simple hello world program. You can get a listing of this program from the website. Below is part of the listing that shows the main function. The locations shown here are relative to the beginning of the module. The listing below is not yet linked.

```
1  1://helloworld.cpp : Defines the entry point for the console
2  2://application. This example has been compiled on Visual Studio
3  3://.NET so the “/GS” flag results can be seen on line 37.
4  4:    #include "stdafx.h"
5  5:
6  6:    int main(int argc, char* argv[])
7  7:    {

8  //Prologue Begins
9  00401010   push        ebp        //Save EBP on the stack
10 00401011   mov         ebp,esp    //Save Current Value of ESP in EBP
11 00401013   sub         esp,40h    //Make space for 64 bytes (40h) var
12 00401016   push        ebx        //store the value of registers
13 00401017   push        esi        //on to the
14 00401018   push        edi        //stack
15 00401019   lea         edi,[ebp-40h] //load ebp-64 bytes into edi
16 //the location where esp was before it started storing the values of
    //ebx etc on the stack.
17
18 0040101C   mov         ecx,10h //store 10h into ecx register
19 00401021   mov         eax,0CCCCCCCCh
20 00401026   rep stos    dword ptr [edi]
21 //Prologue Ends
22 //Body Begins
23 8:        printf("Hello World!\n");
24 00401028   push        offset string "Hello World!\n" (0042001c)
25 0040102D   call        printf (00401060)
26 00401032   add         esp,4
27 9:        return 0;
28 00401035   xor         eax,eax
29 10:   }
```

```
30 //End Body
31 //Epilogue Begins
32 00401037   pop         edi               // restore the value of
33 00401038   pop         esi               //all the registers
34 00401039   pop         ebx
35 0040103A   add         esp,40h           //Add up the 64 bytes to esp
36 0040103D   cmp         ebp,esp
37 0040103F   call        __chkesp (004010e0) // "/GS FLAG"
38 00401044   mov         esp,ebp
39 00401046   pop         ebp               //restore the old EBP
40 00401047   ret 3                   //restore and run to saved EIP
```

Line numbers 9 through 21 are parts of prologue and lines 31 through 40 are the epilogue. Prologue and Epilogue code is generated automatically by a compiler in order to setup a stack frame, preserve registers and maintain a stack frame (All the information put on the stack related to one function is called a stack frame) after a function call is completed. The body contains the actual code to the function call. Prologue and Epilogue are architecture and compiler specific.

The above example (lines 9 – 21) displays a typical prologue seen under visual studio 6.0. The first instruction saves the old EBP (parent base pointer/frame pointer) address on to the stack(inside the newly created stack frame). The next instruction copies the value of the register ESP into the register EBP thus setting the new base pointer it to point to the new base pointer (EBP). The third instruction then reserves room on the stack for local variables, in this example a total of 64 bytes space is created. It is important to remember that typically arguments are passed from right to left and the calling function is responsible for the stack clean up.

The above epilogue code restores the state of the registers before the stack frame is cleaned up. The content of all the registers that was pushed onto the stack frame in the prologue are now popped and restored to their original value in reverse (line 31 – 33), the next three lines appear only in debug version (line 34- 36), where by 64 bytes are added to the stack pointer to point to the base pointer which is checked in the next line.

The instruction at line 37 validates the return address that was popped back onto the stack before returning to the original function [detailed explanation of "/GS Flag implementation" available on MSDN]. Once the return address is validated, it is moved into ESP and the contents of EBP are popped and finally the return instruction is executed. The return instruction pops the value on top of the stack, which is now the return address, into the EIP register.

Note: Calling Convention

The standard "calling convention" under visual studio is CDECL. The stack layout changes very little if this standard is not used. We will not be discussing the other calling conventions such as fastcall or stdcall.

Glossary:

- **Registers**: Registers are a high speed storage area inside the CPU. It is used to store data temporarily.

- **Op Code**: Op Code is the symbolic representation of assembly language.

Utilities:

- **Visual Studio C++**

Links For Additional Reading:

- http://spiff.tripnet.se/~iczelion/tutorials.html This site has numerous assembly language tutorials.
- http://board.win32asmcommunity.net/ An excellent bulletin board where people discuss common problems with assembly programming.

<u>To Do (Mini Exercise – Paper Part II):</u>

Compile the following code in VC++ (either Visual Studio 7.0 or 7.1) and view the disassembled code.

```
#include <stdio.h>
void main(int argc, char* argv[])
{
        if (argc < 2)
                {
                        printf("Usage: %s string\n", argv[0]);
                        exit(1);
                }
        printf("%s is the argument received", argv[1]);
}
```

1. Locate the prologue and epilogue section in the code.

2. If you can, find the location of the printf function used.

3. Dump the imported functions from the executable and locate the printf function. If the printf function exists locate the address and explain why there is a difference between that address (the address printed in front of the dump) and the address you found in the above question. If the printf function doesn't exists explain why it doesn't exists
   Hint1: Use dumpbin / ollydbg to dump.
   Hint2: Answer to the printf function is available on MSDN, search for different compiler options

Solution (Mini Exercice – Part II):

Dump the imported functions from the executable and locate the printf function. If the printf function exists locate the address and explain why there is a difference between that address (the address printed in front of the dump) and the address you found in the above question. If the printf function doesn't exists explain why it doesn't exists [hint: answer available on MSDN].

```
#include <stdio.h>
void main(int argc, char* argv[])
{
        if (argc < 2)
              {
                     printf("Usage: %s string\n", argv[0]);
                     exit(1);
              }
        printf("%s is the argument received", argv[1]);
}
```

Solution 3:

Browse to "Project/Properties/C C++/Optimization/Optimization" and disable it ("Disabled /Od"). The compiler optimizes the printf calls and replaces them, thus the printf function is not visible when dumpbin is run on the executable.