Malware Reverse Engineering

Andrea Mambretti (mambro007@gmail.com)

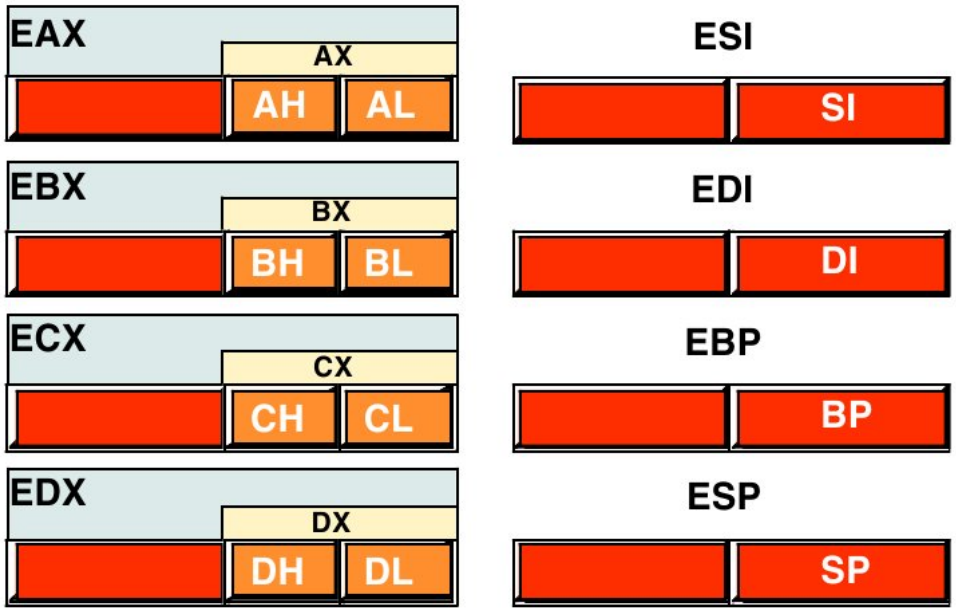
Politecnico di Milano

April 9, 2013

(1) How is the IA made?

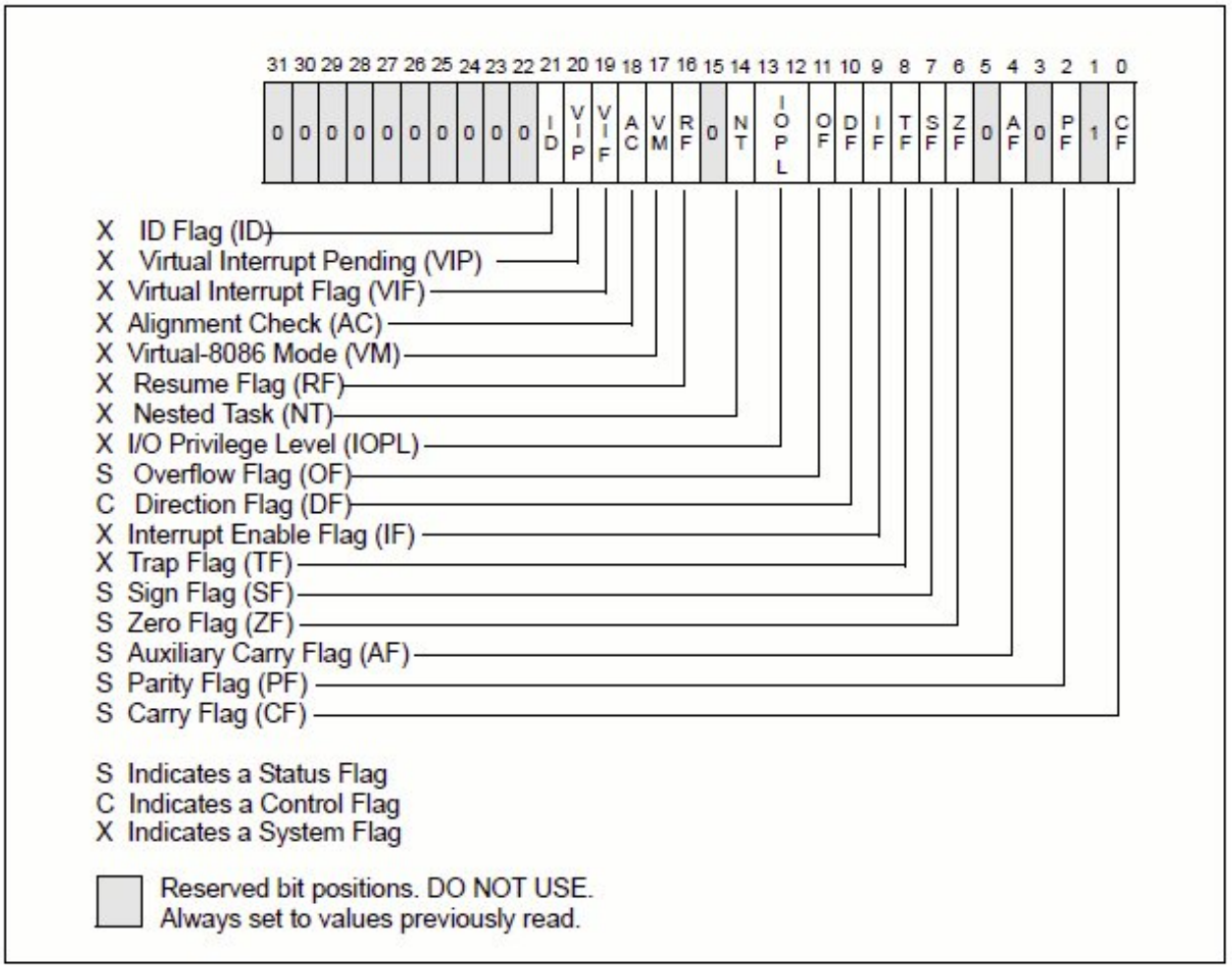
- The processor has 32 bits internal registers to manage and execute operations on data
  They are EAX, EBX, ECX, EDX, ESI, EDI, EBP, EIP and ESP
- Among them we identify EAX, EBX, ECX, EDX are for general purpose
- EBP (BP = base pointer) and ESP (SP = stack pointer) are the stack bounds
- EDI and ESI are extra-registers
- EIP (IP = instruction pointer) is the register that contains the address of the next instruction

(2) How is the IA made?



- The register name system is a porting from 16-bit IA where the registers were called AX, BX and so on.
  "E" means extended. Without it we consider the corrisponding 16-bit register
  There's also the possibility to use AX,BX,CX,DX such as 8-bit registers. For example we can use AX as AH and AL that mean higher and lower 8 bits of AX

(1) What about EFLAGS?



(2) What about EFLAGS?

- It's another 32-bit register
- Only 8 bits out of 32 are of interest for us. The others are either for the kernel mode function or are of little interest for programmers
- The 8 bits are called flags. We consider them singularly. They are boolean (true/false)
- The meaning of each bit is different. They represent overflow, direction, interrupt disable, sign, zero, auxiliary carry, parity and carry flags
- Since they represent information about the instruction last executed, they change at every execution step. They are VERY important for the control flow of the program

Syntax

- In the assembly world we can find two main syntaxes: the AT&T and the Intel
- AT&T syntax is used by all UNIX program (e.g. gdb)
- Intel syntax is used by Microsoft programs (IDApro and others)

(1) Differences in the notation

- Consider the following operation:

  "move the value 0 to EAX"

- AT&T:

  mov $0x0,%eax

- Intel:

  mov eax, 0h

- Comments:
  - As you can see in AT&T syntax the destination is the second operand instead as in the Intel syntax
  - In the AT&T syntax the register are denoted with % and the immediate/costant with $. In the Intel syntax these tokens are not used.

(2) Differences in the notation

- Consider this new operation:
  "move the value 0 to the address contained in EBX+4"
- AT&T:
  mov $0x0,0x4(%ebx)
- Intel:
  mov [ebx+4h],0h
- Comments:
  - This case shows how the deferentiation is done in assembly
  - In AT&T we use parenthesis. In the Intel syntax we have to use square brackets
  - The way to manage the offset is another syntax difference. In the first case we have to put it out of the parenthesis in the second one inside the square brackets

(1) Basic instructions overview

- Every processor has a huge instruction set (see Intel Manual[1])
- A subset of the whole instructio set is usually processor dipendent
- We will focus on the other subset of instructions that is common among the processors
- We will use the Intel syntax as it is the same syntax used in IDApro by default

[1] http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html

(2) Basic Instruction MOV

- General: MOV **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: Every combination is possible except memloc to memloc!!!It isn't valid here and in all the instructions!!!
- With this instruction, as said in the example above, we move a value from a source place to a destination. There are a ton of different versions. They change in function of the operands ex 32 bits operands, 16 bits operands, immediate to reg, immediate to memory
- Examples

MOV eax, ebx	MOV eax, FFFFFFFFh	MOV ax, bx
MOV [eax],ecx	MOV [eax],[ecx] NO!!!	MOV al, FFh

(3) Basic Instruction ADD

- General: ADD **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: The destination register has to be as big as at least the source or greater
- With this instruction we can add a value from **source** to the destination operand and put the new value inside the destination
- Examples

ADD esp, 44h	ADD eax, ebx	ADD al, dh
ADD edx, cx	ADD [eax],[ecx] NO!!!	ADD [eax],1h

(4) Basic Instruction SUB

- General: SUB **destination**, **source**
  source can be an immediate, a register, a memory location
  destination can be either a register or a memory location
  NB: The destination register has to be as big as at least the source or greater
- With this instruction we can subtract the value **source** from the destination operand and put the new value inside the destination
- Examples

SUB esp, 33h	SUB eax, ebx	SUB al, dh
SUB edx, cx	SUB [eax],[ecx] NO!!!	SUB [eax],1h

(5) Basic Instruction MUL

- General: MUL **Operand**
  Operand can be an immediate, a register, a memory location
- With this instruction we can multiplies **Operand** by the value of corresponding byte-length in the EAX,AX,AL register

OperandSize:	1 byte	2 bytes	4 bytes
Other Operand	AL	AX	EAX
Higher Part of result:	AH	DX	EDX
Lower Part of result:	AL	AX	EAX

- Examples

OperandSize:	1 byte	2 bytes	4 bytes
Immediate	MUL 44h	MUL 4455h	MUL 44556677h
Register	MUL cl	MUL dx	MUL ebx

(6) Basic Instruction DIV

- General: DIV **Operand**
  Operand can be an immediate, a register, a memory location
- With this instruction we can divide the value in the dividend register(s) by "Operand"

OperandSize:	1 byte	2 bytes	4 bytes
Dividend	AX	DX:AX	EDX:EAX
Remainder	AH	DX	EDX
Quotient	AL	AX	EAX

- Examples

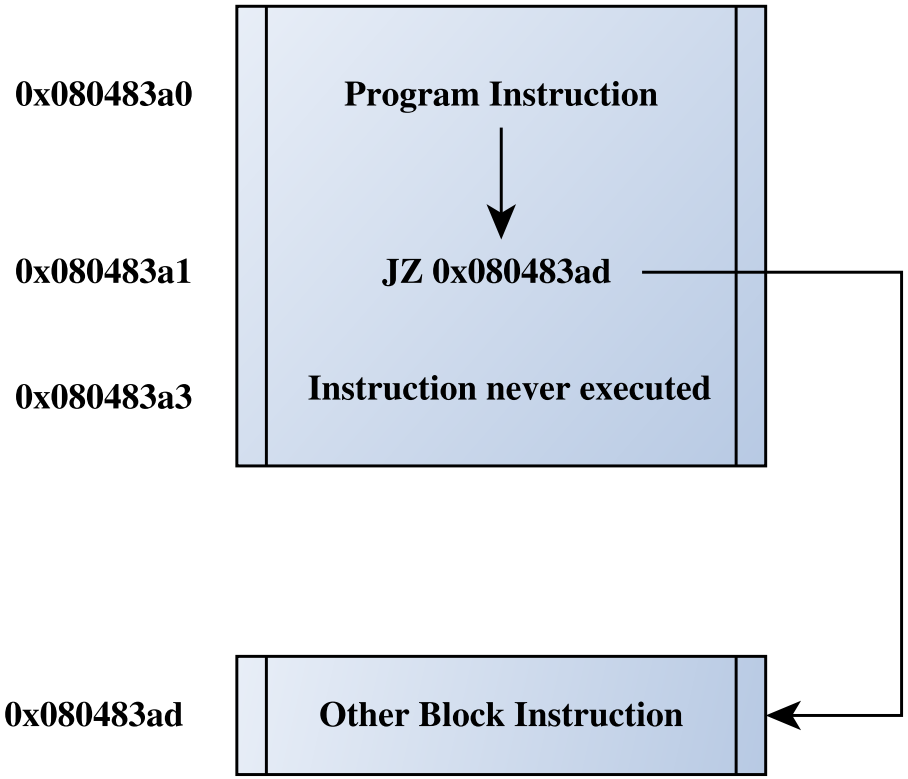
OperandSize:	1 byte	2 bytes	4 bytes
Register	DIV bl	DIV bx	DIV ebx
Immediate	DIV 66h	DIV 6677h	DIV 66778899h

(7) Basic Instruction CMP

- General: CMP **Operand_1**, **Operand_2**
- This instruction performs a subtraction between two operands and sets the flags, it doesn't store the result
- Examples

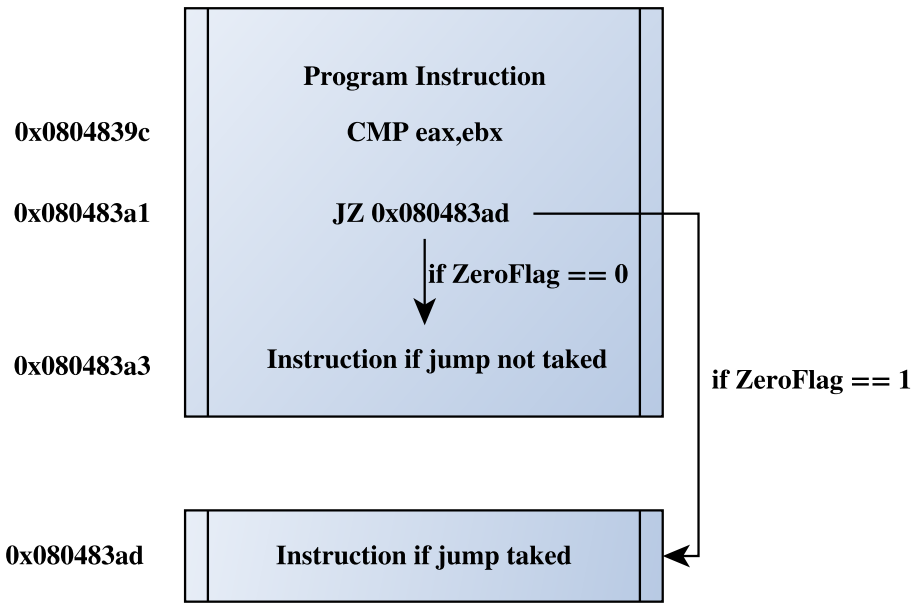
CMP eax, ebx	CMP eax, 44BBCCDDh	CMP al, dh
CMP al, 44h	CMP ax,FFFFh	CMP [eax],4h

(8) Basic Instruction JMP



- General: JMP **address**
- This instruction is called unconditional jump and when is executed put in the eip (the next instruction address) the **address** passed as operand. We say that the execution jumps to **address** and it's unconditional because always the execution jump.

(9) Basic Instruction JZ,JNZ and so on



- General: JX **address**
  $X \in \{O, NO, S, NS, E, Z, NE...\}$
- This set of instruction are called conditional jump. It means that the execution will go to **address** if and only if the specific flag of the condition jump given is verified. For example: jz jumps if zero flag is 1 if no is not taked [2]

[2] http://www.unixwiz.net/techtips/x86-jumps.html

(10) Basic Instruction INT

- General: INT **VALUE**
- **VALUE** is the software interrupt that should be generated (0-255)
- Famous values are 21h for call service under windows and 80 for linux
- look the manual for the other

How much is different x86_64 from x86?

- The prefix of the register is r instead of e so we have (rip, rax etc.)
- There are 8 new registers (r8 to r15)
- each of them can be consider at 8, 16, 32, 64 bits
  with $X \in \{8..15\}$ we have

bits	8	16	32	64
reg	rXb	rXw	rXd	rX

- for better syntax information look at
  `http://www.x86-64.org/documentation/assembly.html`

Different Binary File Format

- PE (Portable Executable): This kind of binary file format is used by Microsoft binary executable.
- ELF: This is the common binary format for Unix, Linux, FreeBSD and others
- Other

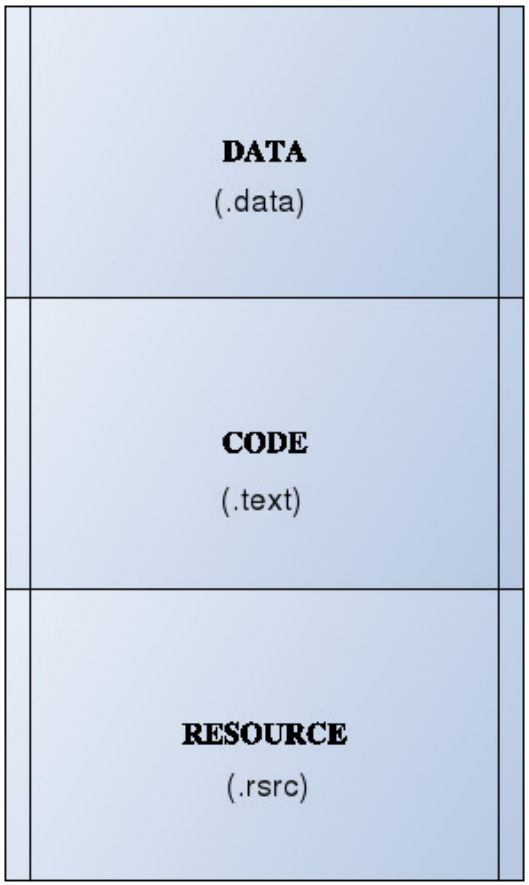
How a program is seen in memory in linux (ELF)

Executable	Description
.bss	This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run.
.comment	This section holds version control information.
.data/.data1	These sections hold initialized data that contribute to the program's memory image
.debug	This section holds information symbolic debugging.
.text	This section holds the "text," or executable instructions, of a program.
.init	This section holds executable instructions that contribute to the process initialization code.
	That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called main for C programs).
.got	This section holds the global offset table.

How a program is seen in memory in windows (PE)

Executable	Description
.text	Contains the executable code
.rdata	Holds read-only data that is globally accessible within the program
.data	Stores global data accessed throughout the program
.idata	Sometimes present and stores the import function information; if this section is not present, the import function information is stored in the .rdata section
.edata	Sometimes present and stores the export function information; if this section is not present, the export function information is stored in the .rdata section
.pdata	Present only in 64-bit executables and stores execption-handling information
.rsrc	Stores resources needed by the executable
.reloc	Contains information for relocation of library files

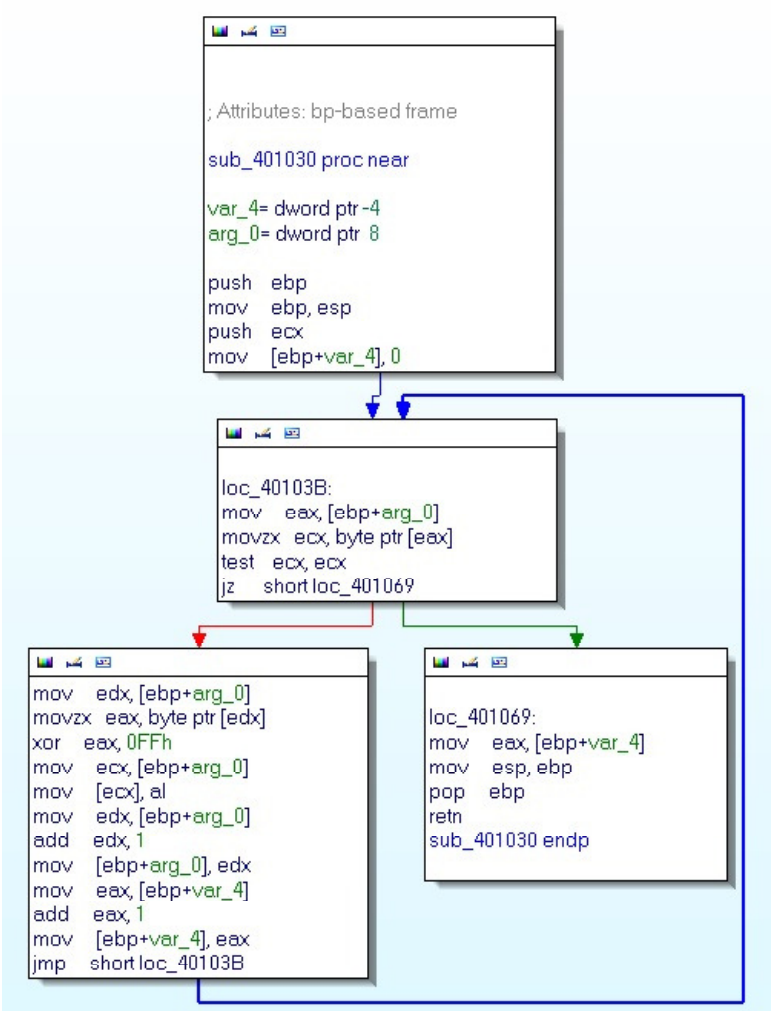
A General Schema



Function

- The concept of function in assembly is the same of the common function in almost all the programming languages
- Is a piece of code that receive data from the caller a return some value after the elaboration
- Differently from all high level languages the way to pass parameters to a function can be done in more than one way
- Is different also the way how to call a specific function

Function, An example



Call Convention

- It's the way how a program receives parameters,how a function returns its return value and who cleans the stack
- There are different implementation of the Call Convention that dictates exactly where a caller should place any parameters that a function requires
- Everything is dipendent by the compiler(gcc/g++, visual studio c++ etc.) and by the high-level language from which the assembly comes from (c, c++, visualbasic and so on)
- Let's look at some of those

The C Calling Convention

- It's the default calling convention used by most C compilers for the x86 arch
- When a compiler doesn't use this convention we can force it using the modifier _cdecl
- It specifies that the **caller place** parameters to a function on a stack in the right to left order and that the **caller remove** the parameters from the stack after the called function completes

The C Calling Convention example

```
void demo_cdecl(int w, int x, int y, int z);
```

```
; demo_cdecl(1, 2, 3, 4);   //programmer calls demo_cdecl
push   4            ; push parameter z
push   3            ; push parameter y
push   2            ; push parameter x
push   1            ; push parameter w
call   demo_cdecl   ; call the function
add    esp, 16      ; adjust esp to its former value
```

```
demo_cdecl(1, 2, 3, 4);   //programmer calls demo_cdecl
 mov   [esp+12], 4   ; move parameter z to fourth position on stack
 mov   [esp+8], 3    ; move parameter y to third position on stack
 mov   [esp+4], 2    ; move parameter x to second position on stack
 mov   [esp], 1      ; move parameter w to top of stack
 call   demo_cdecl  ; call the function
```

The Standard Calling Convention

- This is the Miscrosoft Calling Convention standard
- When a compiler doesn't use this convention we can force it using the modifier _stdcall
- Also here the parameters are passed all using only the stack, the difference is that the called function is responsible for clearing the function parameters from the stack when the function has finished. To do this the function has to know the right number of parameter passed. It's valid only with function with fixed number of parameters so such as printf can't use it.

```
void _stdcall demo_stdcall(int w, int x, int y);

ret 12      ; return and clear 12 bytes from the stack
```

fastcall Convention for x86

- It's a variation of the stdcall convention, the fastcall calling convention passes up to two parameters in CPU registers rather than on the program stack.
- The Microsoft Visual C/ C++ and GNU gcc/g++ compilers recognize the fastcallmodifier in function declaration.
- In this case the first two parameters are passed in the register (ECX and EDX), the remaining parameters are place on the stack in right to left order similar to stdcall.The function is responsible for removing parameters from the stack when they return to their caller.

```
void fastcall demo_fastcall(int w, int x, int y, int z);
```

```
; demo_fastcall(1, 2, 3, 4);   //programmer calls demo_fastcall
   push   4              ; move parameter z to second position on stack
   push   3              ; move parameter y to top position on stack
   mov    edx, 2         ; move parameter x to edx
   mov    ecx, 1         ; move parameter w to ecx
   call   demo_fastcall  ; call the function
```

C++ Calling Convention

- This Call Convention is for non static member function in c++ to male available the **this** pointer-
- The address of the object used to invoke the function must be supplied by the caller and is therefore provided as a parameter when calling nonstatic member functions.
- The c++ language standard does not specify how **this** should be passed to nonstatic member function
- So here arise the problem that every c++ compiler does something different from another one
- LOOK TO THE SPECIFIC COMPILER WHEN YOU REVERSE C++ PROGRAM!!!

Other Calling Convention

- A Ton of other convention are implemented...They are often language-, compiler- and CPU-specific.
- Actually all the people that write assembly by hand use the own convention...
- ...Try you to write some little program

Background of a good Reverser

- Assembly Language
- Compilers
- Virtual Machine and Bytecodes (ex Java)
- Operative Systems

Tools of a good Reverser

- System-Monitoring Tools
- Disassemblers
- Debuggers
- Decompilers

IDA Pro

- **Type:** Disassembler
- **Description:** IDApro, without doubts, is the most powerful reverse tool. I will try to resume all the properties of it. It has three view of the code (graph, text and hex view), a smart version of strings that filter the output to allow a better look at it, two version of the toolbar (advanced and basic), either auto finds data structures inside the code or you can specify it (the same for enums), import/export windows, hexrail extension to get a c-like code of some function, you can specify more than one external debugger to debug your program and much more!

OllyDbg

- **Type:** Disassembler and Debugger
- **Description:** OllyDbg is a well-known Windows debugger, is really useful on binary code analysis. With OllyDbg is very simple patch the binary to get the desidered flaw of the program. It's really intuitive and you can get it for free online.

WinDbg

- **Type:** Kernel Debugger
- **Description:** This tool is provided by Microsoft, we can use it to debug drivers, applications and services on Windows systems.

PEview

- **Type:** Portable Executable(PE) viewer
- **Description:** This program is used actually to get a first idea of what is going on. It allows us to see inside the header of the file and understand things such as it's packed or not, it has the resource section with something bad, which are the imports (that's equal to say "What it will try to do in theory on our system")

Wireshark

- **Type:** Network protocol analyzer
- **Description:** WS is a traffic sniffer with a huge set of protocol filter that gives us the full control of what goes on the network. Why do we need it? Actually the reverse engineering process is joint with a dynamic analysis of the malware that maybe uses the network to communicate with remote servers or other bad things. Using wireshark, we can catch and look into every packet that flows in the network.

Strings

- **Type:** Binary analyzer
- **Description:** Strings is a very small program but it can provide us very important information. It looks inside the binary and prints all the printable charaters on the screen. The strings of the binary can be very usuful to understand what is the binary. NB: Ok malware writers are not stupid they will try to put wrong strings inside but if you look at things such as "9.23.111.43" **maybe** you can suppose that is a ipv4 address and look at it

Snort

- **Type:** Intrusion detection system
- **Description:** It's an open source network intrusion prevention and detection system. We can use it to take under control some event during the execution of a malware. It will give us fast information about new event coming from the network such as a botmaster that send something to our infected test-machine

Tor

- **Type:** Proxy for Anonymity
- **Description:** Tor allows a ton of things, everyone is strictly connected to the **Anonymity**. The program is a simple proxy that get you access to the "hidden internet". Tor is so anonymous that bad guys, of course, use it to do bad things. Some malware works using tor so we need it to try to undestand its behaviour inside the hidden internet.

UPX (Ultimate Packer for Executable)

- **Type:** Packer/Unpacker
- **Description:** With this program we can detect and unpack malware that use upx packing to avoid antivirus detection. It works on a huge collection of binary format and work on a lot of OS

Regshot

- **Type:** Windows register analyzer
- **Description:** This tool is designed to make a really simple thing. It allows you to do a snapshot of a the whole set of registers in two different moment and it gives us the differences between snapshot one and two. The result can be used to understand which register are touched by a malware

Process Monitor

- **Type:** Monitor
- **Description:** Process Monitor is an advanced monitoring tool for Windows that shows real-time file system, Registry and process/thread activity. It combines the features of two legacy Sysinternals utilities, Filemon and Regmon, and adds an extensive list of enhancements including rich and non-destructive filtering, comprehensive event properties such session IDs and user names, reliable process information, full thread stacks with integrated symbol support for each operation, simultaneous logging to a file, and much more. Its uniquely powerful features will make Process Monitor a core utility in your system troubleshooting and malware hunting toolkit.

Process Explorer

- **Type:** Monitor
- **Description:** This Program gives us information about either file or directory that are opened by a specific process. The unique capabilities of Process Explorer make it useful for tracking down DLL-version problems or handle leaks, and provide insight into the way Windows and applications work.

Process Hacker

- **Type:** Monitor
- **Description:** Process Hacker is a free and open source process viewer. This multi-purpose tool will assist you with debugging, malware detection and system monitoring. It includes powerful process termination, memory viewing/editing and other unique and specialized features.

Resource Hacker

- **Type:** Resource section manager
- **Description:** Resource HackerTM is a freeware utility to view, modify, rename, add, delete and extract resources in 32bit & 64bit Windows executables and resource files (*.res). It incorporates an internal resource script compiler and decompiler and works on all (Win95 - Win7) Windows operating systems.

Reverse on Malware and Antireversing Techniques

Today's malware and commercial program use techniques against Reversing

- Anti-Disassembly (Linear and Flow Oriented Disassembly)
  - Jump instructions with the same target
  - Jump instruction with a Costant Condition
  - Impossible disassembly
- Anti-Debugging
  - They understand that are executed in a debugger and change their behaviour either crashing itself, the debugger or totally doing other stuff
- Anti-Virtual Machine Techniques
- Packers and Unpacking

Bibliography

- The Ida Pro Book 2 Edition
- The Shellcoder Handbook
- Reverse Engineering Code with IDA Pro
- Secrets of Reverse Engineering

Conclusion

- Software Reverse Engineering is a very powerfull instrument but it requires a lot of lowlevel-knowledge
- Appling this technique on malware analysis is not optional if we want undestand how the malware works
- Can be very time consuming and if all the tools used for the analysis are not setted correctly there's no way to reverse the malware

End

- Thanks Folk...Questions?