Advanced Artefact Analysis

Introduction to advanced artefact analysis

TOOLSET, DOCUMENT FOR STUDENTS

OCTOBER 2015

www.enisa.europa.eu European Union Agency For Network And Information Security

About ENISA

The European Union Agency for Network and Information Security (ENISA) is a centre of network and information security expertise for the EU, its member states, the private sector and Europe's citizens. ENISA works with these groups to develop advice and recommendations on good practice in information security. It assists EU member states in implementing relevant EU legislation and works to improve the resilience of Europe's critical information infrastructure and networks. ENISA seeks to enhance existing expertise in EU member states by supporting the development of cross-border communities committed to improving network and information security throughout the EU. More information about ENISA and its work can be found at www.enisa.europa.eu.

Authors

This document was created by Yonas Leguesse, Christos Sidiropoulos, Kaarel Jõgi and Lauri Palkmets in consultation with ComCERT[1] (Poland), S-CURE[2] (The Netherlands) and DFN-CERT Services (Germany).

Contact

For contacting the authors please use cert-relations@enisa.europa.eu
For media enquiries about this paper, please use press@enisa.europa.eu.

Acknowledgements

ENISA wants to thank all institutions and persons who contributed to this document. A special 'Thank You' goes to Filip Vlašić, and Darko Perhoc.

[1] Dawid Osojca, Paweł Weżgowiec and Tomasz Chlebowski
[2] Don Stikvoort and Michael Potter

Table of Contents

Main Objective	This training presents the introduction to the advanced artefact analysis and is the first part of a three-day course. At the beginning an introduction to the course is made, setting common terminology and describing different analysis methods. Second and the biggest part of the training is an introduction to the assembly language focusing on Intel x86 family of processors, along with a description of the binary code execution, processor internals and system calls. The material presented in this part is considered as an introduction to the whole course. However, it would be beneficial for trainees to have a prior knowledge of the x86 assembly language so that they could focus on the analysis process rather than learning assembly instructions. The later part of the training introduces a number of tools commonly used for the advanced artefact analysis. Two of them, the IDA Pro Free edition[3] for static and OllyDbg[4] for dynamic analyses, will be used extensively during the rest of the course.
Target Audience	CSIRT staff and incident handlers involved in the technical analysis of incidents, especially those dealing with artefact examination and analysis. Prior knowledge of assembly language and operating systems internals is highly recommended.
Total Duration	3-4 hours

[3]Freeware version of IDA v5.0 https://www.hex-rays.com/products/ida/support/download_freeware.shtml (last accessed 11.09.2015)
[4]OllyDbg http://www.ollydbg.de/ (last accessed 11.09.2015)

1. Introduction to x86 Assembly

1.1 Introduction to assembly language

Assembly language is a low-level programming language whose instructions are almost directly translated into machine code – that is a series of bytes understood by the computer processor (CPU). Nowadays it is mostly used for very specific tasks like programming microcontrollers or writing programs requiring high optimisation in the context of a speed or size. Assembly language is also widely used in a field of reverse engineering, in which the code of executable files is translated into assembly language to get a more human readable form.

It is important to know that different processor families use different instruction sets, which differ in means of available instructions, registers, addressing modes and other aspects. Thus, the assembly language for each of them differs. The most widespread instruction set in the field of a malicious software is undoubtedly the x86 instructions set.

This introduction provides a quick reference for the x86 assembly language. If you would like to learn more, there are plenty of resources freely available online:

- *Intel® 64 and IA-32 Architectures Software Developer's Manuals*[5] - complete reference of the IA-32 architecture (Intel's 32-bit x86 architecture). It consists of three volumes. Volume 1 describes in detail the architecture and programming environment, Volume 2 contains a complete instruction reference and Volume 3 includes the system programming guide. Whenever you don't know some assembly instruction (what it does, which flags it sets, what are its variants) you can take a look at Volume 2 for the most detailed description.
- X86 Assembly guide on Wikibooks[6] – a detailed book covering various aspects of the x86 assembly language (common instructions, different syntaxes and a few more advanced concepts).
- X86 Assembly Guide from University of Virginia[7] – a short guide describing the basics of the 32-bit x86 assembly language with a reference to the most commonly used instructions.
- PC Assembly Language[8] – a course on 32-bit x86 assembly language with references on how to use assembly with programs written in C.

1.2 Instructions, opcodes, operands

When you write a program in a higher level programming language like C or C++ and then compile it, the compiler translates your code into machine code. Machine code is a set of instructions executed directly by the CPU.

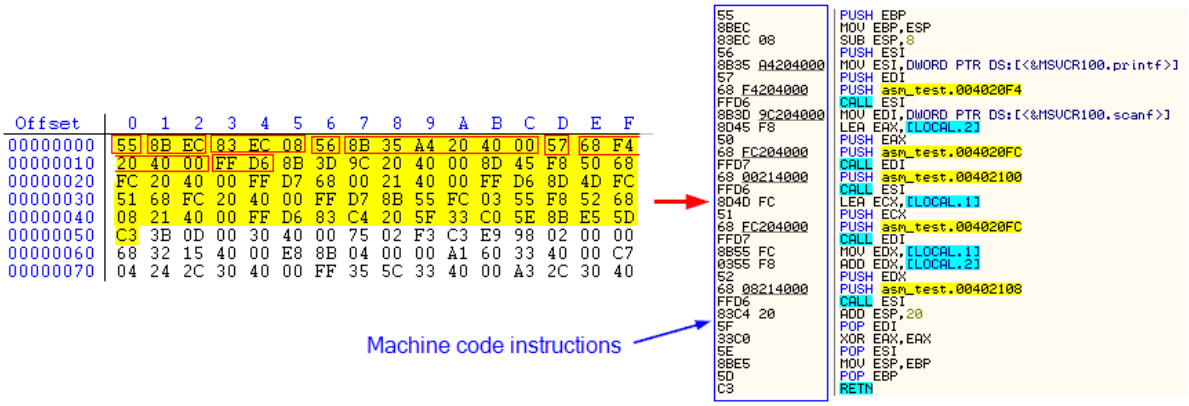
[5]Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (last accessed 11.09.2015)
[6]x86 Assembly https://en.wikibooks.org/wiki/X86_Assembly (last accessed 11.09.2015)
[7]x86 Assembly Guide http://www.cs.virginia.edu/~evans/cs216/guides/x86.html (last accessed 11.09.2015)
[8]PC Assembly Language http://www.drpaulcarter.com/pcasm/ (last accessed 11.09.2015)

```
int _tmain(int argc, _TCHAR* argv[])
{
    int a, b;
    printf("a = ");
    scanf("%d", &a);
    printf("b = ");
    scanf("%d", &b);
    printf("a+b = %d", a+b);
    return 0;
}
```

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000000	55	8B	EC	83	EC	08	56	8B	35	A4	20	40	00	57	68	F4
00000010	20	40	00	FF	D6	8B	3D	9C	20	40	00	8D	45	F8	50	68
00000020	FC	20	40	00	FF	D7	68	00	21	40	00	FF	D6	8D	4D	FC
00000030	51	68	FC	20	40	00	FF	D7	8B	55	FC	03	55	F8	52	68
00000040	08	21	40	00	FF	D6	83	C4	20	5F	33	C0	5E	8B	E5	5D
00000050	C3	3B	0D	00	30	40	00	75	02	F3	C3	E9	98	02	00	00
00000060	68	32	15	40	00	E8	8B	04	00	00	A1	60	33	40	00	C7
00000070	04	24	2C	30	40	00	FF	35	5C	33	40	00	A3	2C	30	40

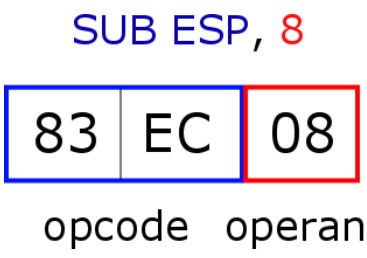
From the perspective of malicious artefacts analysis and reverse engineering, machine code is very hard to read by a human. The problem is that there is no unambiguous and easy way of translating the machine code back to the higher level programming language. Moreover, the machine code is also stripped of valuable information such as comments or variables and functions names.

This is where the assembly language comes in handy. Because the assembly language is almost the exact representation of the machine code, it's also easy to translate the machine code back to assembly instructions.



Red squares in the hex dump were used to mark the first few instructions of the machine code. The same instruction codes can be viewed in the view with the disassembled code.

As you can see from the example, x86 architecture utilises variable length instructions. In a simplified version each instruction consists of opcode and optionally one or more operands.



Opcode specifies what operation should be performed, while operand provides additional "arguments" to the operation (usually specifying values, memory locations or registers for the operation). In this example the 2-byte opcode *83 EC* tells the processor to subtract a value 8 (operand) from the ESP register.

The number and the type of operands depend on a specific instruction. Usually an instruction comes in a few forms allowing the use of different types of operands.

Possible operand types are:

- Immediate value – value encoded in the instruction itself like in *sub esp, 8.*
- Register – operand is one of the registers.
- Memory – operand is in the memory (specified by offset encoded in the instruction).

In reality, the instruction structure is a little more complicated. If you are interested in learning more, please refer to Chapter 2 from Volume 2 of the Intel's manual[9] – *Instruction Format*.

1.3 Registers

A register is a small amount of storage available to the processor. Registers are specific to the given instruction set architecture and differ among processor families. Every contemporary processor has at least several registers available and they are used for different purposes.

The x86 architecture provides 16 basic registers used in general programming:

- 8 general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP).
- 6 segment registers (CS, DS, SS, ES, FS, GS).
- One flags register (EFLAGS).
- One instruction pointer register (EIP).

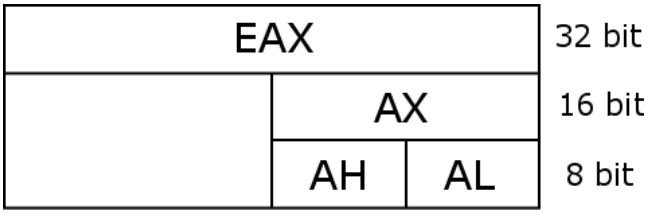
In addition to those registers, there are also several special purpose registers like debug registers, control registers or registers associated with CPU extensions (MMX, SSE, FPU, etc.).

General purpose registers can be used, as the name suggests, as general registers for different types of operations (arithmetic calculations, address calculations or to hold memory pointers). Additionally each one of them also has a special role:

- EAX – accumulator register, used in different arithmetic operations.
- EBX – data pointer.
- ECX – counter register, used in loops.
- EDX – I/O pointer.
- ESI – source data pointer in string operations.
- EDI – destination data pointer used in string operations.
- EBP – base pointer, used to create stack frame in function calls.
- ESP – stack pointer, points to the top of the stack.

General purpose registers are 32-bit in size, however it is possible to access them as 16- or 8-bit registers by using the following pattern.

[9] Intel® 64 and IA-32 Architectures Software Developer Manuals: Vol. 2
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (last accessed 11.09.2015)



This means that in order to access the lower 16 bits of the EAX register you should refer to it as AX register. You can also access the lower and higher 8 bits of AX register by referring to them adequately as AL and AH.

The same scheme applies to EBX, ECX and EDX registers. It also applies to ESI, EDI, EBP and ESP except you cannot access them as 8 bit registers.

The next group of registers are six segment registers (CS, DS, SS, ES, FS and GS). In the early days of x86 processors, they were used to hold 16-bit segment selectors, for use in memory segmentation. Since most modern operating systems use paging and flat memory model segment registers, they are rarely used anymore or only for special purposes[10]. Example of a special usage is the FS register which on Win32 systems points to *Thread Information Block* structure[11].

The last two registers are flag register (EFLAGS) and instruction pointer (EIP).

EFLAGS register is used to store flags values holding information about results of previous operations or other system information. The following basic flags are available:

- **CF – Carry Flag**
- **PF – Parity Flag**
- **AF – Auxiliary Carry Flag**
- **ZF – Zero Flag**
- **SF – Sign Flag**
- TF – Trap Flag
- IF – Interrupt Enable Flag
- DF – Direction Flag
- **OF – Overflow Flag**
- IOPL – I/O Privilege Level
- NT – Nested Task
- RF – Resume Flag
- VM – Virtual-8086 Mode
- AC – Alignment Check
- VIF – Virtual Interrupt Flag
- VIP – Virtual Interrupt Pending
- ID – ID Flag

Flags in **bold** are so called status flags. They store information about results of arithmetic operations and there are primarily used for conditional branch instructions. For example the Zero flag (ZF) informs that the result of

[10]What has happened to the segment registers? http://www.lshift.net/blog/2010/03/31/what-has-happened-to-the-segment-registers/ (last accessed 11.09.2015)
[11]Under the Hood https://www.microsoft.com/msj/archive/S2CE.aspx (last accessed 11.09.2015)

operation was zero, while the Overflow flag (OF) indicates that there was an overflow of integer number (resulting value was either too big or too small for negative numbers). To get detailed information about each flag role, refer to Volume 1 of Intel's reference manual[12].

Different assembly instructions can set or clear different flags. For example the ADD instruction can set OF, SF, ZF, AF, CF and PF flags. To get information which flags can be set by specific instruction refer to the x86 assembly instructions reference[13].

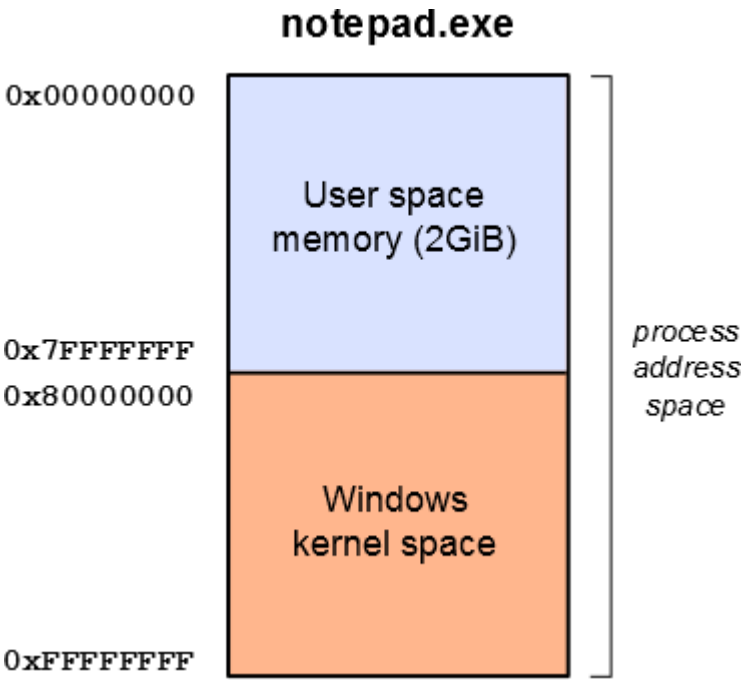
Finally, the instruction pointer register (EIP) is used to hold the address of the next instruction to be executed. This register however cannot be directly accessed by software – neither for read not for write purposes.

1.4 Memory organisation

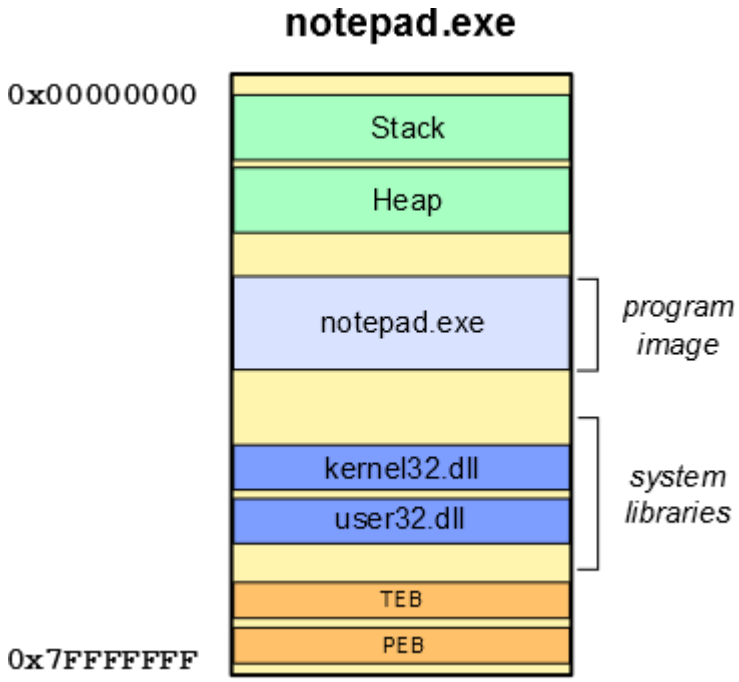
The Microsoft Windows system (as well as most other contemporary operating systems) uses a flat memory model in which programs see memory as a contiguous and linear address space. Moreover, thanks to the virtual memory concept, each process running on Microsoft Windows gets access to its own virtual address space[14]. One of the outcomes of this is process isolation. Two different processes can have different data blocks loaded at the same address in their virtual address space and none of them would be able to directly access memory of the second process without the help of the operating system.

On Microsoft Windows, the system memory of 32-bit processes is addressed through 32-bit addresses starting from 0 up to 0xFFFFFFFF (4GB). Though not all address space is available to the user-mode processes[15]. User-mode processes can access freely only memory from 0 up to 0x7FFFFFFF (2GB)[16]. The second half, that is addresses from 0x80000000 up to 0xFFFFFFFF, is reserved for the operating system.

[12] Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (last accessed 11.09.2015)
[13] Intel® 64 and IA-32 Architectures Software Developer Manuals: Vol. 2 http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf (last accessed 11.09.2015)
[14] Virtual address spaces https://msdn.microsoft.com/en-us/library/windows/hardware/hh439648%28v=vs.85%29.aspx (last accessed 11.09.2015)
[15] User mode and kernel mode https://msdn.microsoft.com/en-us/library/windows/hardware/ff554836%28v=vs.85%29.aspx (last accessed 11.09.2015)
[16] It's possible to let process address more than 2GiB of memory using special /LARGEADDRESSAWARE linker option (https://msdn.microsoft.com/en-us/library/vstudio/wz223b1z%28v=vs.100%29.aspx) (last accessed 11.09.2015)



When a new PE executable is started on a Windows system, a new process is created and the system loader maps the PE file into the process's address space as well as loads all DLL libraries needed by the program. Process heap and stack are also created.



The Thread Environment Block (TEB) and the Process Environment Block (PEB) are system structures providing information about the current thread's context and the process itself. For a process there is only one PEB structure but separate TEBs, one for each application thread.

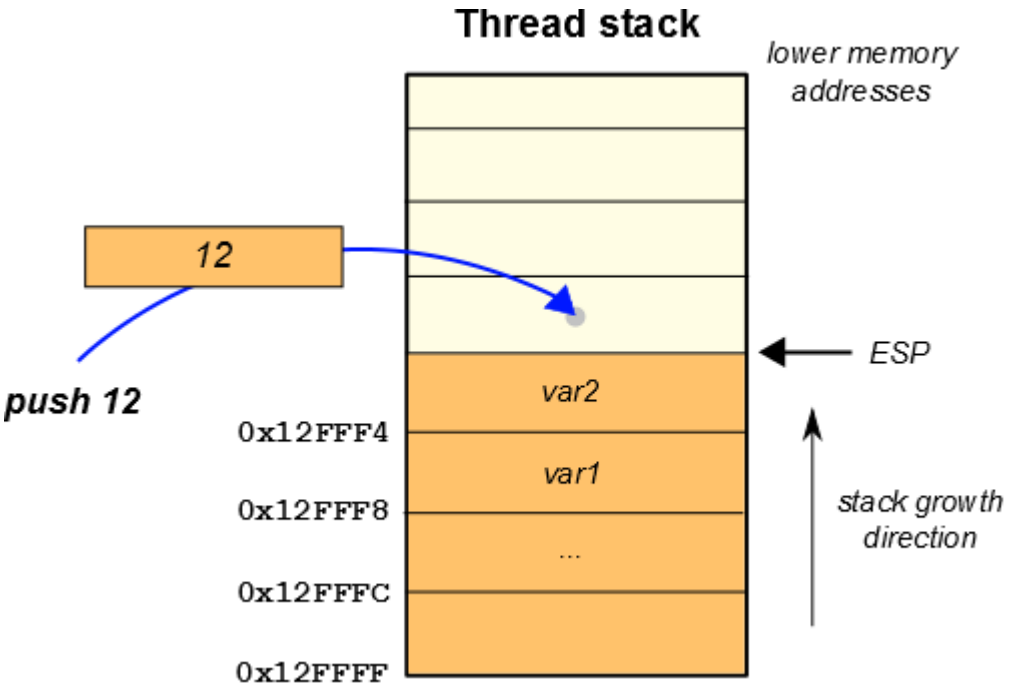
This is a simplified version of the process address space because normally it would also contain other memory blocks (e.g. block with environment variables[17] or multiple heaps). You can view a detailed memory map for any process using VMMap tool from Sysinternals[18].

Address	Type	Size	Privat...	Protection	Details
00160000	Mapped File	412 K		Read	C:\Windows\System32\locale.nls
001D0000	Private Data	4 K	4 K	Read/Write	
001E0000	Heap (Private Data)	64 K	12 K	Read/Write	Heap ID: 3 [COMPATABILITY]
001F0000	Shareable	800 K		Read	
002C0000	Shareable	1,028 K		Read	
003D0000	Shareable	4 K		Read/Write	
003E0000	Shareable	8 K		Read	
003F0000	Private Data	4 K	4 K	Read/Write	
00400000	Image	528 K	28 K	Execute/Read	C:\Users\ENISA\Desktop\putty.exe
00490000	Private Data	512 K	4 K	Read/Write	
00540000	Heap (Private Data)	256 K	64 K	Read/Write	Heap ID: 7 [LOW FRAGMENTATION]
00620000	Heap (Private Data)	64 K	4 K	Read/Write	Heap ID: 4 [COMPATABILITY]
00680000	Heap (Private Data)	1,024 K	224 K	Read/Write	Heap ID: 1 [LOW FRAGMENTATION]
77770000	Image (ASLR)	1,264 K	20 K	Execute/Read	C:\Windows\System32\ntdll.dll
778B0000	Image (ASLR)	212 K	8 K	Execute/Read	C:\Windows\System32\ws2_32.dll
77980000	Image (ASLR)	100 K	8 K	Execute/Read	C:\Windows\System32\sechost.dll
779B0000	Image (ASLR)	4 K		Read	C:\Windows\System32\apisetschema.dll
7F6F0000	Shareable	1,024 K		Read	
7FFB0000	Shareable	140 K		Read	
7FFDE000	Private Data	4 K	4 K	Read/Write	Thread Environment Block ID: 596
7FFDF000	Private Data	4 K	4 K	Read/Write	Process Environment Block
7FFE0000	Private Data	64 K		Read	

Two important memory structures are stack and heap. The process heap is a memory region where dynamically allocated variables (e.g. using malloc()) are put. The stack on the other hand is used for storing local variables and tracing function calls in the current thread. The stack is the last in first out (LIFO) data structure and there is a separate stack for each thread.

The top of the stack is always pointed to by the ESP register. What's important is that the stack grows towards lower memory addresses. This means that whenever a new value is pushed onto the stack, the ESP register is decremented.

[17] Changing Environment Variables https://msdn.microsoft.com/en-us/library/windows/desktop/ms682009%28v=vs.85%29.aspx (last accessed 11.09.2015)
[18] VMMap https://technet.microsoft.com/en-us/library/dd535533.aspx (last accessed 11.09.2015)



Except for storing local variables, the stack is also used for passing function arguments and tracing function calls. This will be described in a later section.

1.5 Basic instructions

There are plenty of different instructions in the x86 instruction set. Additionally, each instruction usually comes in a few forms allowing to use it with different operand types (registers, immediate values, memory addresses). This section will list most common x86 Assembly instructions with a brief description of each one.

The following notation is assumed for operands:

- <reg> - one of the general-purpose registers
- <mem> - memory location
- <imm> - immediate value
- <rel> - address relative to the current instruction

If bit suffix is added to the operand type, this means that only an operand of that bit size is allowed in the operation, for example:

- <reg32> - only double-word general-purpose registers (EAX, EBX, ECX, EDX)
- <reg16> - only word general-purpose registers (AX, BX, CX, DX)

1.5.1 Data transfer instructions

Instruction	Description	Affected flags
mov <reg>, <reg> **mov <reg>, <mem>** **mov <reg>, <imm>** **mov <mem>, <imm>** **mov <mem>, <reg>**	Copies data from the second operand to the first operand.	None
movsb	Moves byte from address ESI to address EDI and increases/decreases ESI and EDI according to DF flag	None
movsw	Moves word from address ESI to address EDI and	None

	increases/decreases ESI and EDI according to DF flag.	
xchg <reg>, <reg> **xchg <reg>, <mem>** **xchg <mem>, <reg>**	Exchanges contents of the first operand and the second operand.	None
stosb	Stores byte AL at address EDI and increases/decreases EDI according to DF flag.	None
stosw	Stores word AX at address EDI and increases/decreases EDI according to DF flag.	None
lodsb	Loads byte from address ESI into AL and increases/decreases ESI according to DF flag.	None
lodsw	Loads word from address ESI into AX and increases/decreases ESI according to DF flag.	None
push <reg32/reg16> **push <mem32/mem16>** **push <imm>**	Decrements stack pointer and stores source operand at top of the stack.	None
pushfd	Decrements stack pointer by 4 and stores entire EFLAGS register at top of the stack.	None
pushad	Pushes general purpose registers onto the stack in following order: EAX, ECX, EDX, EBX, ESP (original), EBP, ESI and EDI.	None
pop <reg32/reg16> **pop <mem32/mem16>**	Pops value from top of the stack and store into destination operand. Then increment stack pointer adequately.	None
popfd	Pops top of the stack into EFLAGS register (restores flags values).	All flags.
popad	Pops EDI, ESI, EBP, EBX, EDX, ECX and EAX registers. Value of ESP register on the stack is ignored.	None
lea <reg>, <mem>	Computes effective address of the second operand (memory offset) and stores it in the first operand.	None

1.5.2 Arithmetic operations

Instruction	Description	Affected flags
add <reg>, <imm> **add <reg>, <mem>** **add <reg>, <reg>** **add <mem>, <imm>** **add <mem>, <reg>**	Adds the second operand to the first operand and stores result in the first operand.	OF, SF, ZF, AF, CF, PF
sub <reg>, <imm> **sub <reg>, <mem>** **sub <reg>, <reg>** **sub <mem>, <imm>** **sub <mem>, <reg>**	Subtracts the second operand from the first operand.	OF, SF, ZF, AF, CF, PF
div <reg> **div <mem>**	Unsigned divide. Divides value stored in EDX:EAX by the source operand and stores quotient in EAX and remainder in EDX.	CF, OF, SF, ZF, AF, PF
idiv <reg> **idiv <mem>**	Signed divide. Divides value stored in EDX:EAX by the source operand and stores quotient in	CF, OF, SF, ZF, AF, PF

	EAX and remainder in EDX.	
mul <reg> **mul <mem>**	Unsigned multiply. Multiples value in EAX (destination) and operand. Result stores in EDX:EAX.	OF, CF
imul <reg> **imul <mem>**	Signed multiply. Multiples value in EAX (destination) and operand. Result stores in EDX:EAX. There is also two and three operand version of signed multiply. Refer to instruction reference for more information.	OF, CF
inc <reg> **inc <mem>**	Adds 1 to destination operand.	OF, SF, ZF, AF, PF
dec <reg> **dec <mem>**	Subtracts 1 from destination operand.	OF, SF, ZF, AF, PF
neg <reg> **neg <mem>**	Two's complement negation of the operand.	CF, OF, SF, ZF, AF, PF
sal <reg>, <imm8> **sal <reg>, CL** **sal <reg>, 1** **sal <mem>, <imm8>** **sal <mem>, CL** **sal <mem>, 1**	Arithmetic shift left of the first operand by <imm8>/CL/1.	CF, OF, SP, ZP, PP
sar <reg>, <imm8> **sar <reg>, CL** **sar <reg>, 1** **sar <mem>, <imm8>** **sar <mem>, CL** **sar <mem>, 1**	Arithmetic shift right of the first operand by <imm8>/CL/1.	CF, OF, SP, ZP, PP
cmp <reg>, <reg> **cmp <reg>, <mem>** **cmp <reg>, <imm>** **cmp <mem>, <reg>** **cmp <mem>, <imm>**	Compares first operand with the second operand by subtracting second operand from the first and setting appropriate flags. Operands values are not changed.	CF, OF, SF, ZF, AF, PF

1.5.3 Logical operations

Instruction	Description	Affected flags
and <reg>, <reg> **and <reg>, <mem>** **and <reg>, <imm>** **and <mem>, <reg>** **and <mem>, <imm>**	Bitwise AND operation of the first (destination) operand and the second (source) operand. Result is stored in the first operand.	OF, CF cleared. SF, ZF, PF set appropriately.
or <reg>, <reg> **or <reg>, <mem>** **or <reg>, <imm>** **or <mem>, <reg>** **or <mem>, <imm>**	Bitwise inclusive OR operation of the first (destination) operand and the second (source) operand. Result is stored in the first operand.	OF, CF cleared. SF, ZF, PF set appropriately.
not <reg> **not <mem>**	Bitwise NOT operation.	None

shl <reg>, <imm8> **shl <reg>, CL** **shl <reg>, 1** **shl <mem>, <imm8>** **shl <mem>, CL** **shl <mem>, 1**	Logical shift left of the first operand by <imm8>/CL/1.	OF, SP, ZP, PP
shr <reg>, <imm8> **shr <reg>, CL** **shr <reg>, 1** **shr <mem>, <imm8>** **shr <mem>, CL** **shr <mem>, 1**	Logical shift right of the first operand by <imm8>/CL/1.	OF, SP, ZP, PP
xor <reg>, <reg> **xor <reg>, <mem>** **xor <reg>, <imm>** **xor <mem>, <reg>** **xor <mem>, <imm>**	Bitwise exclusive OR (XOR) operation of the first (destination) operand and the second (source) operand. Result is stored in the first operand.	OF, CF cleared. SF, ZF, PF set appropriately.
test <reg>, <reg> **test <reg>, <imm>** **test <mem>, <reg>** **test <mem>, <imm>**	Logical compare operation by performing bitwise AND operation on the first and the second operand and setting appropriate flags.	OF, CF cleared. SF, ZF, PF set appropriately.

1.5.4 Control flow instructions

Instruction	Description	Affected flags
call <rel> **call <reg>** **call <mem>**	Procedure call. Saves return address on the stack and branches to the called procedure.	None
ret **ret <imm16>**	Return to the return address popped from the stack. Optionally releases <imm16> bytes from the stack.	None
leave	Releases stack frame. Copies the frame pointer (EBP) into stack pointer register (ESP) and pops old frame pointer from the stack.	None
int <imm8>	Generates interrupt specified by immediate value in operand (calls interrupt handler).	EFLAGS register is pushed onto the stack. Certain flags might be affected depending on the interrupt.
nop	No operation. Does nothing. Machine code 0x90, useful in debugging.	None
loop <imm8>	Performs loop operation. Jumps short until ECX=0 decrementing ECX with each iteration.	None

1.5.5 Jump instructions

Instruction	Description	Affected flags
jmp <rel>	Always jumps to the address specified by the	None

jmp <reg> **jmp <mem>**	operand.	
je/jz <rel>	Jumps if equal/zero (ZF=1)	None
ja <rel>	Jumps if above (CF=0 and ZF=0)	None
jb <rel>	Jumps if below (CF=1)	None
jae <rel>	Jumps if above or equal (CF=0)	None
jbe <rel>	Jumps if below or equal (CF=1 or ZF=1)	None
jne/jnz <rel>	Jumps if not equal/zero (ZF=0)	None
jna <rel>	Jumps if not above (CF=1 or ZF=1)	None
jnb <rel>	Jumps if not below (CF=0)	None

1.6 Function calls, stack frame and calling conventions

A function is a part of the code which can be called multiple times from different locations and which has a very specific task to perform. The function concept is also present in the assembly language. X86 assembly supports function calls by introducing special instructions (e.g. call, ret, leave) and registers like EBP used to hold address of the current stack frame (described in more detail later).

Functions are called using a *call* instruction. When a function is called, the address of an instruction following a *call* (return address) is pushed onto the stack. This address will later be used by the *ret* instruction to return back to the code from where the function was called.

Consider this example:

```
00401005 xor     eax, eax
00401007 call    func_1
0040100C mov     ebx, eax
0040100E cmp     ebx, 500h
```

In this example, when instruction *call func_1* is executed, address *0x40100C* will be pushed onto the stack. When the function returns, execution will resume from this address.

A typical function call looks like this.

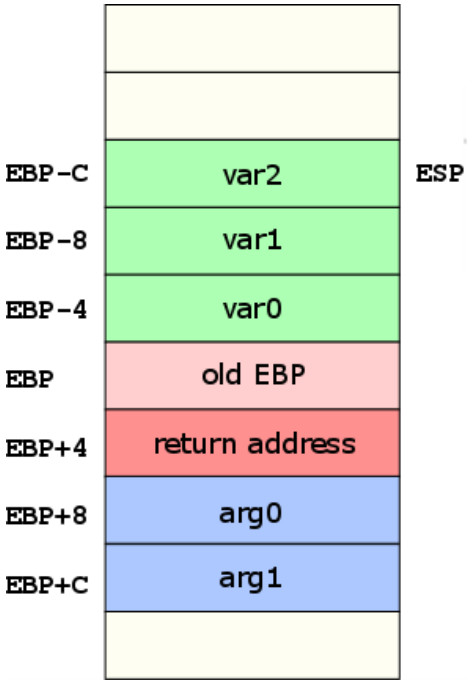
1. Passing parameters for the function (if any).
2. Calling function.
3. Reserving stack memory for local variables.
4. Function operations.
5. Restoring stack.
6. Function return and optionally cleaning function arguments.

To ease referencing parameters, local variables and restoring stack functions frequently use EBP based stack frames. The stack frame is created at the beginning of the function by pushing the previous EBP value onto the stack and then saving the current stack pointer (ESP) value in the EBP register.

Creation of a stack frame typically looks like the following.

```
0040100F push    ebp             ; saving EBP
00401010 mov     ebp, esp        ; saving current ESP
00401012 sub     esp, 12         ; reserving memory for local variables
```

This is often called a function prologue. After that, the stack should look like this (assuming there were also two arguments passed to the function on the stack).



Using EBP based stack frame local variables (var0…var2) and function arguments (arg0…arg1) can be addressed relatively to the EBP register (compiler doesn't need to track all stack pointer changes in the function body).

If a function is using an EBP based stack frame, then restoration of the original stack at function end is also relatively easy and takes just two steps: first EBP is copied to ESP (restoring ESP value from the function beginning) and then the old EBP is popped from the top of the stack.

```
00401022 mov     esp, ebp         ; restoring esp
00401024 pop     ebp              ; restoring ebp
00401025 retn
```

This is often referred to as the function epilogue. Restoration of the stack is necessary because when the *ret* instruction is reached, the top of the stack should contain the return address.

In the example above parameters were passed to the function on the stack. This does not always need to be the case. The exact way how a function is called and how arguments are passed is defined by so called calling conventions. There are a few popular calling conventions in use, depending on the compiler and the code.

In general, a calling convention defines:

- How parameters are passed to the function.
- The order in which function parameters are passed (left to right or right to left).
- How the function is returning a value.
- Which registers should be preserved by called function. Such registers can still be used in the function but before the function returns, their value should be restored.
- What should be done with parameters passed to the function via the stack? Should they be cleaned by the called function, (callee clean-up) or by the caller?

Below you find a short description of popular calling conventions.

cdecl (__cdecl):
This is the default calling convention for C and C++ programs[19].

- Arguments are passed on the stack right to left.
- Function result is returned in EAX register.
- All registers except EAX, ECX and EDX should be preserved by callee.
- Caller cleans arguments from the stack.

stdcall (__stdcall)
Standard calling convention for Windows Win32 API functions[20].

- Arguments are passed on the stack right to left.
- Function result is returned in EAX register.
- All registers except EAX, ECX and EDX should be preserved by callee.
- Callee cleans arguments from the stack when returning.

fastcall (__fastcall)
This is a less commonly used calling convention[21].

- The first two arguments from left to right are passed in ECX and EDX registers, other arguments from right to left are passed on the stack.
- The function result is returned in EAX register.
- All registers except EAX, ECX and EDX should be preserved by callee.
- Callee cleans arguments from the stack when returning (if there are any arguments).

thiscall (__thiscall)
This calling convention is used by C++ for non-static member functions[22]. Its implementation slightly differs among the compilers (GCC and Microsoft Visual C++). Similarly to other calling conventions arguments are passed on the stack right to left. Additionally, as a first argument *'this'* pointer is also passed. In GCC it's passed on the stack (as a first argument), in Microsoft Visual C++ it's passed in the ECX register. The function result is returned in EAX.

[19] __cdecl https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx (last accessed 11.09.2015)
[20] __stdcall https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx (last accessed 11.09.2015)
[21] __fastcall https://msdn.microsoft.com/en-us/library/6xa169sk.aspx (last accessed 11.09.2015)
[22] __thiscall https://msdn.microsoft.com/en-us/library/ek8tkfbw.aspx (last accessed 11.09.2015)

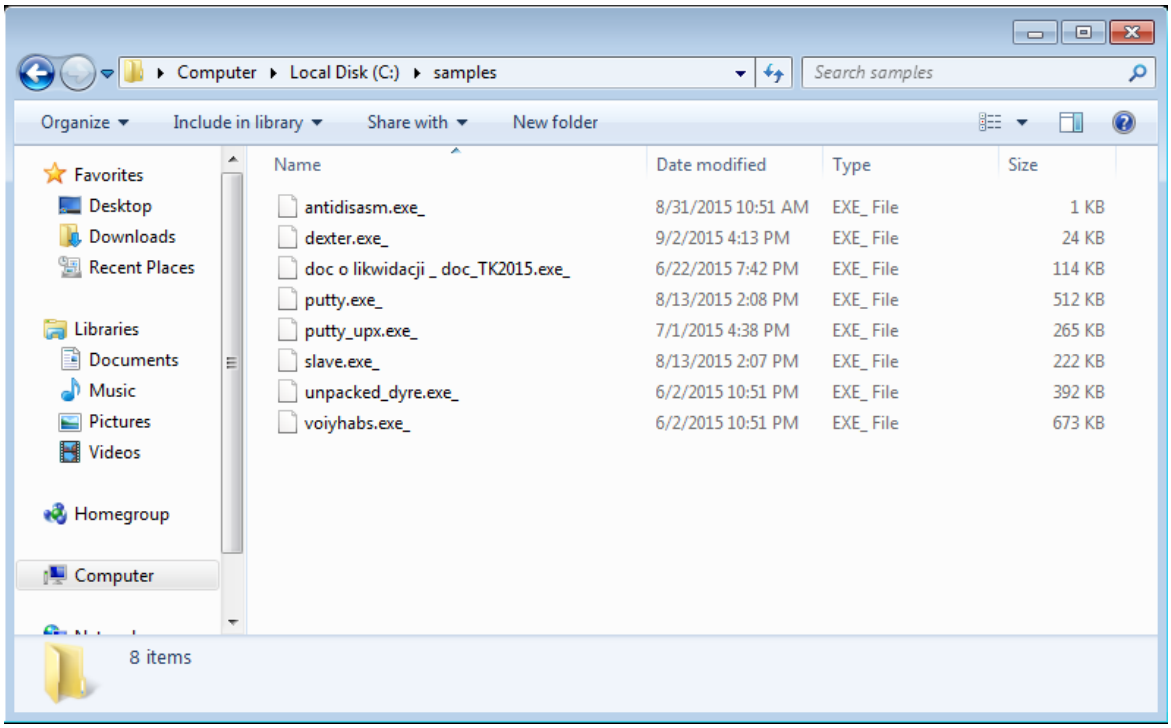
2. Environment preparation

All analyses of malicious files should be performed on dedicated and isolated environments. Most often this would be a group of properly configured virtual machines. The role of this environment is to prevent malicious code from accessing your private data or infecting other hosts on your local network. An example of such an environment is described in *Building artefact handling and analysis environment* exercise from the *ENISA Artefact Analysis training*[23].

In this training you will be using a single virtual machine with the Microsoft Windows operating system (Microsoft Windows 7 32-bit) without network connectivity. To learn how to create such virtual machine and how to install necessary tools, refer to *Building artefact handling and analysis environment* training (description of *Winbox* VM). Note that since no network connection is necessary you don't need to configure networking (just make sure there is no network connection at the end of the process).

Samples used in this training will be provided in a separate archive. Password to the archive is: *infected*, which is a commonly used password for archives containing malicious code. Please note that the purpose of archiving with a password is to make the user aware of the maliciousness of the code and avoid running it accidentally. It is not intended as a confidentiality measure.

Unpacking of the archive should reveal the following samples.



Do not execute any of those samples before creating a snapshot of the clean virtual machine!

Before you proceed to the second part of the training, make sure that:

[23] Training Courses https://www.enisa.europa.eu/activities/cert/training/courses (last accessed 11.09.2015)

- There is no access from the *Virtual Machine* to the Internet (nor access to your local network).
- You have installed all necessary tools and copied malware samples.
- You can start the following programs: OllyDbg, Process Hacker, and IDA Pro Free. Note that OllyDbg and Process Hacker should be always run as an administrator.
- You have created a clean snapshot of your system (before executing any sample).

Finally, because in this training you will be dealing with live malware samples, you should always remember to take proper security precautions such as:

- Analyse malicious files only in a dedicated and controlled environment, which is isolated from your local network, private files or any other sensitive data. If you are using virtualisation technology, make sure that you are using the newest stable version of that software. Also, you should not install *Guest Additions* on your virtual machine.
- If it's not necessary for the analysis, disable the Internet connection on the virtual machine. Otherwise, malicious code running on the virtual machine might start sending spam or attacking hosts on the Internet.
- Restore a snapshot of the clean system after each analysis involving execution of malicious code (unless it's specified otherwise in the exercise). This is necessary because the previously run malicious code may have made changes to the operating system which may prevent the next sample from executing correctly.
- Remember that dynamic analysis of the malicious code with a debugger is really the same as executing malicious code, just slower! A debugger lets the processor execute machine commands and the results affect the environment in the same way as when they would run without a debugger. Safety of the operation depends solely on selecting cautiously which parts of the code may be run.
- Don't copy samples of malicious files onto your personal computer. If it is really necessary move malicious files into a password protected archive first. This will protect you from accidentally executing a sample.
- When you store malicious files make sure that everyone having access to those files will know what they are. A good idea is to put malicious files in a directory with a clear and suggestive name.

Follow all instructions of the trainer.

ENISA

European Union Agency for Network
and Information Security
Science and Technology Park of Crete (ITE)
Vassilika Vouton, 700 13, Heraklion, Greece

Athens Office

1 Vass. Sofias & Meg. Alexandrou
Marousi 151 24, Athens, Greece

PO Box 1309, 710 01 Heraklion, Greece
Tel: +30 28 14 40 9710
info@enisa.europa.eu
www.enisa.europa.eu