Advanced Artefact Analysis

Introduction to advanced artefact analysis

HANDBOOK, DOCUMENT FOR TEACHERS

OCTOBER 2015

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

1. Training introduction 5
1.1 Training overview 5
1.2 Course structure 5
1.3 Samples 6
1.4 Recommended reading and materials 6
2. Advanced artefact analysis 8
2.1 Advanced dynamic analysis 8
2.2 Advanced static analysis 9
2.3 Analysis goal – when to stop 10
3. Introduction to x86 Assembly 11
3.1 Introduction to assembly language 11
3.2 Instructions, opcodes, operands 11
3.3 Registers 13
3.4 Memory organisation 15
3.5 Basic instructions 18
3.5.1 Data transfer instructions 18
3.5.2 Arithmetic operations 19
3.5.3 Logical operations 20
3.5.4 Control flow instructions 21
3.5.5 Jump instructions 22
3.6 Function calls, stack frame and calling conventions 22
4. Tools overview 25
4.1 OllyDbg 25
4.2 Immunity Debugger 26
4.3 WinDbg 28
4.4 WinAppDbg / PyDbg 28
4.5 IDA Pro 29
4.6 Radare2 30
5. Environment preparation 31
6. Training summary 33

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
Frequency	Once per each team member.

[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. Training introduction

Over the years, malware has become a tool for many criminals who use it to spy, steal information and money or to gain persistent access to remote information systems. A continuous race between malware creators and countermeasures developers make the automated defences regularly fail. CSIRTs have an important role to provide expertise and to understand the cause, assess the (potential) damage and develop protection methods. To counter a threat that uses state-of-the-art techniques in order to evade detection mechanisms requires skills, knowledge and often a non-conventional thinking.

This training has been created to deepen an analyst knowledge and to develop such skills in the artefact analysis domain.

1.1 Training overview

This course is designed to extend the existing ENISA *Artefact Analysis training*[5] with more advanced exercises. During this training participants will learn how to analyse artefacts, potentially malicious code at the very code level. Participants will be taught both types of manual artefact analysis – a static, using a disassembler - and the dynamic, using a debugger. This training is intended for security experts working in CSIRT teams who are willing to deepen their knowledge about the artefact analysis.

This training is given using the 32-bit Microsoft Windows platform. It would be beneficial for trainees to have previous knowledge of x86 assembly language and Microsoft Windows system architecture (including Win32 API) so that they keep focused on the analyses rather than on the assembly instructions. A list of a recommended reading is provided in paragraph 1.4.

It is also strongly encouraged to follow the basic ENISA *Artefact Analysis training*, as it gives a broad perspective of the artefact analysis. Techniques and tools presented in that training can help set the goals of advanced analysis while starting it with some knowledge about analysed samples.

1.2 Course structure

The training is split into three separate parts. The first part is *Introduction to Advanced Artefact Analysis*. This part is purely theoretical and its role is to give you an introduction to what the advanced dynamic and static analysis is, as well as to introduce you to fundamentals of the x86 assembly language. In this part you will also learn about free tools commonly used for a reverse engineering and analysis of the malicious code. Finally you will have the opportunity to prepare and test the environment which you will be using in your analyses during the practical part of the training. The knowledge you will get in the first part is essential for completing the second and the third part of the training.

After completing the first part of the training you will proceed to the second part which is *Advanced Dynamic Analysis*. In this part you will be introduced to the OllyDbg debugger[6] which is one of the most commonly used debuggers during malicious code analysis. After that you will learn practical elements of the advanced dynamic analysis including an unpacking of packed samples, followed by the creation of child processes and code injection

[5]Training Courses https://www.enisa.europa.eu/activities/cert/training/courses (last accessed 11.09.2015)
[6] http://www.ollydbg.de/ (last accessed 11.09.2015)

schemes as well as overcoming basic anti-debugging techniques. Finally you will learn the basics of debugging automation using the OllyScript engine.

The last part is *Advanced Static Analysis*. In this part you will learn how to perform a static analysis of the disassembled code using the IDA Pro[7] tool. First you will be taught how to effectively use the IDA Pro tool and the features that it offers. Then you will use IDA to do an initial assessment of the malicious code and find important functions. After finding those important functions you will try to analyse their code. The third part of the training will end with a presentation of various anti-analysis techniques and a presentation on how to use IDA to overcome some of them.

1.3 Samples

During the training you will be using various samples. All samples will be 32bit Portable Executable (PE) files dedicated for the Microsoft Windows platform. To learn more about this file format refer to the *ENISA Artefact Analysis training*[8].

While handling samples they should always be executed with caution in an isolated and controlled environment. Whether a sample used in the exercise is malicious or not is explicitly mentioned at the beginning of each exercise. A few samples were obtained from the KernelMode.info[9] forum (many thanks to this very useful community).

To minimise accidental execution risk of samples, the underscore suffix was added to .exe file extensions for all of the samples. You do not need to remove this suffix during the training because all tools used in the training can properly open such files without any issues.

It is strongly suggested to remind students about the principles of building a safe and secure analysis environment such as designed in the *"Building artefact handling and analysis environment"* training scenario[10].

1.4 Recommended reading and materials

Advanced artefact analysis is a complex subject requiring constant learning and practice. If you would like to better prepare for this training or to extend your knowledge after completing it, there are multiple online resources and published books that you can use. The list below contains a recommended further reading covering topics such as assembly language, reverse engineering, Win32 API, dynamic analysis and static analysis.

- ENISA Artefact Analysis training[11] – the previous edition of this training, covering the building of an artefact handling and analysis environment, artefacts acquisition as well as basic static and behavioural (dynamic) analysis. It's recommended to complete this training prior to attending to *Advanced Artefact Analysis* training.
- theForger's Win32 API Programming Tutorial[12] – basic introduction to the programming using Win32 API; explains a few important Windows API concepts (windows, messages, etc.).

[7]The freeware version of IDA v5.0 https://www.hex-rays.com/products/ida/support/download_freeware.shtml (last accessed 11.09.2015)
[8]Training Courses https://www.enisa.europa.eu/activities/cert/training/courses (last accessed 11.09.2015)
[9]KernelMode.info http://www.kernelmode.info/forum/ (last accessed 11.09.2015)
[10]Building artefact handling and analysis environment https://www.enisa.europa.eu/activities/cert/training/training-resources/technical-operational#building (last accessed 11.09.2015)
[11]Training Courses https://www.enisa.europa.eu/activities/cert/training/courses (last accessed 11.09.2015)
[12]theForger's Win32 API Programming Tutorial http://www.winprog.org/tutorial/ (last accessed 11.09.2015)

- The Tenouk's Win32 - Windows 32 bits system (OS) programming tutorial[13] – a group of tutorials covering various Windows architecture and Win32 API topics along with a reference of common API functions.
- Free IDA Pro Binary Auditing Training Material for University Lectures[14] – an excellent set of practical exercises dedicated to reverse engineering and learning of binary auditing.
- OpenSecurityTraining.info[15] – free online training materials covering various security topics including an introduction to x86 assembly, to malware dynamic analysis and reverse engineering of malware.
- Tuts 4 You: Tutorials, Papers, Dissertations, Essays and Guides[16] – source of dozens of papers, tutorials and other materials related to reverse engineering, dynamic and static analysis, unpacking of samples and many more.
- M. Sikorski, A. Honig "*Practical Malware Analysis*" – excellent book, introducing an advanced malware analysis. It starts from the very beginning with an introduction to an x86 assembly language and then gradually moves to more advanced topics covering the malicious code debugging, analysis of disassembled code, malware behaviour and anti-reverse-engineering techniques.
- C. Eagle "*The IDA Pro Book*" – introduction to the IDA Pro disassembler. It serves not only as a program reference but also as a great source of information on reverse engineering, with many practical examples and explaining various concepts.
- B. Dang, A. Gazet, E. Bachaalany, S. Josse "*Practical Reverse Engineering*" – reverse engineering introduction for beginners and more advanced readers. The book explains the x86 architecture, Windows kernel, debugging automation and various code obfuscation techniques.
- J. Seitz "*Grey Hat Python*" – a book focusing on debuggers and dynamic analysis automation using the Python language. It starts by explaining how debuggers work and then moves on to topics such as building your own debugger or writing debugging and reverse engineering automation scripts in Python.
- M. Russinovich, D.Solomon "*Windows Internals (6th Edition)*" – in-depth reference of Microsoft Windows internals explaining in detail the Windows architecture and the operation of various system processes. Recommended for people interested in more advanced dynamic analysis (including debugging on the kernel level).

[13]The Tenouk's Win32 - Windows 32 bits system (OS) programming tutorial http://www.tenouk.com/cnwin32tutorials.html (last accessed 11.09.2015)
[14]Free IDA Pro Binary Auditing Training Material for University Lectures http://www.binary-auditing.com/ (last accessed 11.09.2015)
[15]OPEN SECURITY TRAINING http://www.opensecuritytraining.info/Training.html (last accessed 11.09.2015)
[16] Tuts 4 You https://tuts4you.com/download.php?list.19 (last accessed 11.09.2015)

2. Advanced artefact analysis

2.1 Advanced dynamic analysis

Dynamic analysis is a type of analysis in which a suspicious file is intentionally executed in a dedicated and controlled environment in order to observe its behaviour. Then, based on the observed results, the analyst seeks to assess whether the sample was malicious, what changes were made to the operating system and if possible to determine the malware family.

There are two types of the dynamic analysis:

- Behavioural analysis;
- Debugging of malicious code.

During the behavioural analysis, the analyst executes a sample in a controlled environment and observes what changes are made to the operating system:

- Changes in the filesystem, registry or if execution resulted in creation of new processes.
- Network traffic is also a valuable source of information.

This type of analysis is covered in ENISA *Artefact Analysis training*[1].

The second type of analysis, referred to later on as advanced dynamic analysis, is to execute a suspicious code under the control of the debugger. This activity is commonly named *sample debugging* as it shares some tools and techniques with the process of hunting bugs in the code. Such an approach gives the analyst an opportunity to track the code execution, instruction after instruction, providing much more detailed information than the behavioural analysis would give. Moreover, unlike the behavioural analysis, debugging also allows to change at runtime how a malicious code executes – for example it allows to check what would happen if the malicious code will receive different commands from a C&C server. This analysis is however not without drawbacks. It requires from the analyst a deep knowledge of reverse engineering and good understanding of how the operating system works. It requires also by definition executing at least parts of the potentially malicious code. This may have some unpredictable consequences such as unintended infections or leaving tracks on other systems (including the C&C servers) if allowed to connect. On top of this, debugging malicious code, depending on its complexity, might often take a significant amount of time.

Advanced dynamic analysis is especially useful in the following scenarios:

Unpacking malware samples

Packing is a process in which the original malicious code is heavily obfuscated in order to avoid a detection by anti-virus engines. This makes the static analysis of such a code nearly impossible. Using a debugger, it is possible to unpack a sample by letting it execute until the code in the memory is restored to its *original* form.

Following process injection

Modern malware often tries to hide its presence in the system. Two popular methods of achieving this are (1) injecting malicious code into other processes, or (2) starting a new process of legitimate applications and overwriting their memory with malicious code (process hollowing[17]). If the process injection or process hollowing techniques were used you can use a debugger to analyse what malicious code was injected.

Analysing complex algorithms

In contrast to a static analysis of the disassembled code, debugging allows to view the runtime state of the memory and registers. This is useful in the analysis of more complex algorithms which are often easier to understand when you follow their execution step by step, checking at the same time how the registers and variables are changing.

Analysing data decrypted at runtime

To hinder analysis, a lot of valuable information, like the addresses of C&C servers or base configurations, are often initially encrypted in the binary file and decrypted only at runtime. To obtain this data (in decrypted form) it is usually easier to use a debugger than the static analysis techniques.

Analysing communications with C&C server

Quite often the most interesting part of the malicious code analysis is learning how it communicates with the C&C server – what data it sends to the remote server and what data it receives. Since the most modern malware encrypts the traffic with the C&C server, capturing such traffic isn't very helpful. However, using a debugger, you can let the malicious application execute until it will try to execute the code controlling network communication. Then in many cases you should be able to read the original data from the memory before the encryption or after the decryption.

2.2 Advanced static analysis

Static analysis is a method of analysis during which an analyst tries to learn about a suspicious file without executing it. In a basic static analysis of a PE file, the analyst will study PE headers, embedded strings and attached resources. He may also do a signature scanning or look for information about the file via online resources. If the sample is not packed, or if the analysis is performed on an unpacked sample, the analyst can learn about some of the functionality of the executable and sometimes recognise the malware family. Basic static analysis is described in detail in the *ENISA Artefact Analysis training*.

More thorough or, in other words, *advanced* static analysis involves disassembling an executable file and then performing an analysis of the disassembled code. Similarly to advanced dynamic analysis it requires good reverse engineering skills and a prior knowledge of operating system internals. Depending on your goal it might also be quite time consuming.

When analysing a disassembled code, you usually do not know which parts of the code would be executed nor what would be the state of the memory and registers. You can only learn this through a slow analysis process. On the other hand, during the static analysis, you are not limited to one execution flow as you would be during the dynamic analysis. This makes the static analysis very useful for learning about functions that would be executed only on a specific condition. A good example is a function that is executed only after receiving a specific command from the C&C server.

Additionally, thanks to the fact that static analysis does not require executing a sample, it is much safer. You are also not limited in the analysis of executable files to those created for the same architecture and the operation system you are running. For example, there is no problem with conducting an analysis of ARM binaries or Linux ELF

[17] Process hollowing http://www.autosectools.com/process-hollowing.pdf (last accessed 11.09.2015)

executable files on Windows. The only requirement is that the tools you will be using should support this particular format.

2.3 Analysis goal – when to stop

Both advanced dynamic and static analysis of malicious files can be time consuming. To minimise the risk of spending too much time on one task, you should always have a clear goal of the analysis in your mind. Examples of such goals are:

- Checking if the sample is really a malicious file.
- Finding infection indicators – changes made to the environment as a result of an execution of the malicious file that can be used further to detect infected workstations.
- Analysis of malware capabilities and functionality.
- Analysis of a network traffic and protocol used to communicate with the C&C.
- Decrypting base configuration of the malicious file to find addresses of C&C servers.
- Preparing a malicious file signature.
- Recognising the malware family.
- Full analysis of the malicious code.

Depending on the profile of your work, a good practice might be also to have a predetermined time limit you can spend on the analysis. Sometimes you will find complex samples and then after reaching the time limit there will probably be plenty of things still unknown to you, and things you would like to check out. At that moment you would need to decide if spending more time on that is worth the effort or if it may be better to finish the analysis and move on to the next sample.

Finally, always remember the golden rule of problem solving: *when you reach an impasse at some point, take a break, do something else, and then re-think your problem*. At the beginning of learning malware analysis you will often find that you don't know how to do something. Just don't spend too much time on thinking how to solve the problem. In the field of malware analysis and reverse engineering most of the things can be done in various ways. Maybe your approach just wasn't the best one and after a short break you will find a better one.

3. Introduction to x86 Assembly

3.1 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.

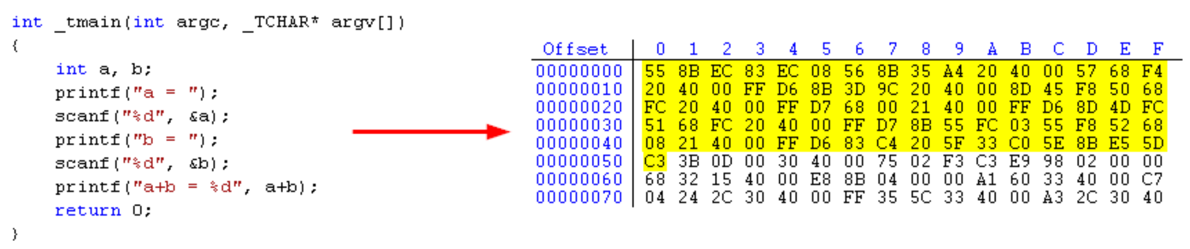
This introduction provides a quick reference for the x86 assembly language. If you would like to learn more, there are plenty of resources freely available online:

- *Intel® 64 and IA-32 Architectures Software Developer's Manuals*[18] - 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[19] – 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[20] – 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[21] – a course on 32-bit x86 assembly language with references on how to use assembly with programs written in C.

3.2 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.

[18]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)
[19]x86 Assembly https://en.wikibooks.org/wiki/X86_Assembly (last accessed 11.09.2015)
[20]x86 Assembly Guide http://www.cs.virginia.edu/~evans/cs216/guides/x86.html (last accessed 11.09.2015)
[21]PC Assembly Language http://www.drpaulcarter.com/pcasm/ (last accessed 11.09.2015)

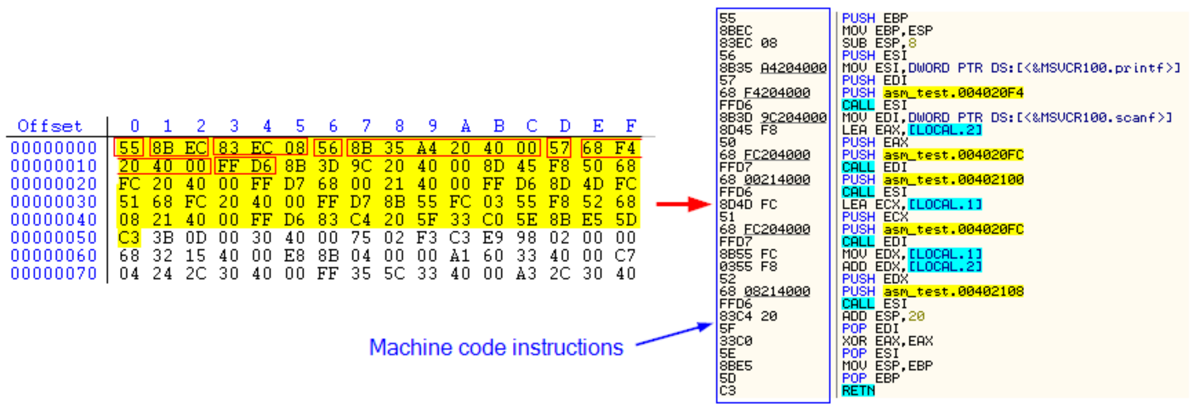


```
int _tmain(int argc, _TCHAR* argv[])
{
    int a, b;
    printf("a = ");
    scanf("%d", &a);
    printf("b = ");
    scanf("%d", &b);
    printf("a+b = %d", a+b);
    return 0;
}
```

```
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.

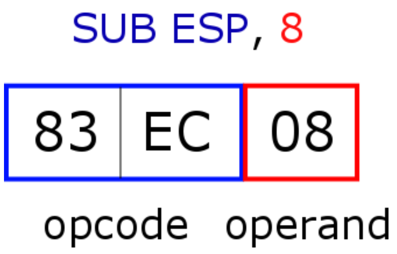


```
Offset    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00000000 55 8B EC 83 EC 08 56 8B 35 A4 20 40 00 57 68 F4
00000010 20 40 00 FF D6 8B 3D 9C 20 40 00 8D 45 F8 50 68
00000020 FC 20 40 00 FF D7 68 00 21 40 00 FF D6 8D 4D FC
00000030 51 68 FC 20 40 00 FF D7 8B 55 FC 03 55 F8 52 68
00000040 08 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
```

```
55              PUSH EBP
8BEC            MOV EBP,ESP
83EC 08         SUB ESP,8
56              PUSH ESI
8B35 A4204000   MOV ESI,DWORD PTR DS:[<&MSVCR100.printf>]
57              PUSH EDI
68 F4204000     PUSH asm_test.004020F4
FFD6            CALL ESI
8B3D 9C204000   MOV EDI,DWORD PTR DS:[<&MSVCR100.scanf>]
8D45 F8         LEA EAX,[LOCAL.2]
50              PUSH EAX
68 FC204000     PUSH asm_test.004020FC
FFD7            CALL EDI
68 00214000     PUSH asm_test.00402100
FFD6            CALL ESI
8D4D FC         LEA ECX,[LOCAL.1]
51              PUSH ECX
68 FC204000     PUSH asm_test.004020FC
FFD7            CALL EDI
8B55 FC         MOV EDX,[LOCAL.1]
0355 F8         ADD EDX,[LOCAL.2]
52              PUSH EDX
68 08214000     PUSH asm_test.00402108
FFD6            CALL ESI
83C4 20         ADD ESP,20
5F              POP EDI
33C0            XOR EAX,EAX
5E              POP ESI
8BE5            MOV ESP,EBP
5D              POP EBP
C3              RETN
```

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[22] – *Instruction Format*.

3.3 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).

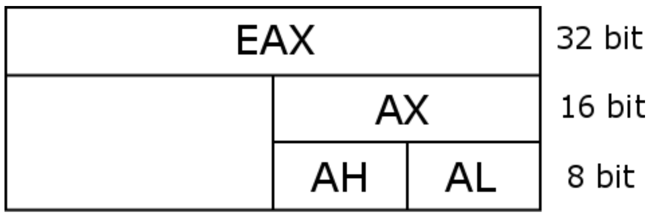
In addition to those registers, there are also several special purpose registers like debug registers, control registers or registers associated with CPU extensions (MMX, SSE, FPU, etc.).

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.

[22] 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 ca not access them as 8 bit registers.

The next group of registers are six segment registers (CS, DS, SS, ES, FS, 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[23]. Example of a special usage is the FS register which on Win32 systems points to *Thread Information Block* structure[24].

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 operation was zero, while the Overflow flag (OF) indicates that there was an overflow of integer number (resulting value was

[23]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)
[24]Under the Hood https://www.microsoft.com/msj/archive/S2CE.aspx (last accessed 11.09.2015)

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[25].

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[26].

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.

3.4 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[27]. 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[28]. User-mode processes can access freely only memory from 0 up to 0x7FFFFFFF (2GB)[29]. The second half, that is addresses from 0x80000000 up to 0xFFFFFFFF, is reserved for the operating system.

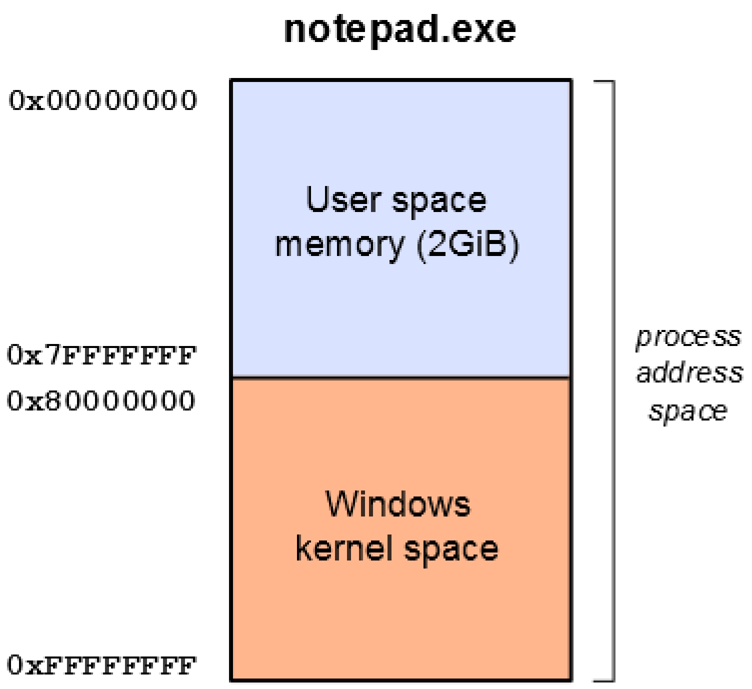
[25] Intel® 64 and IA-32 Architectures Software Developer Manuals http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html (last accessed 11.09.2015)

[26] 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)

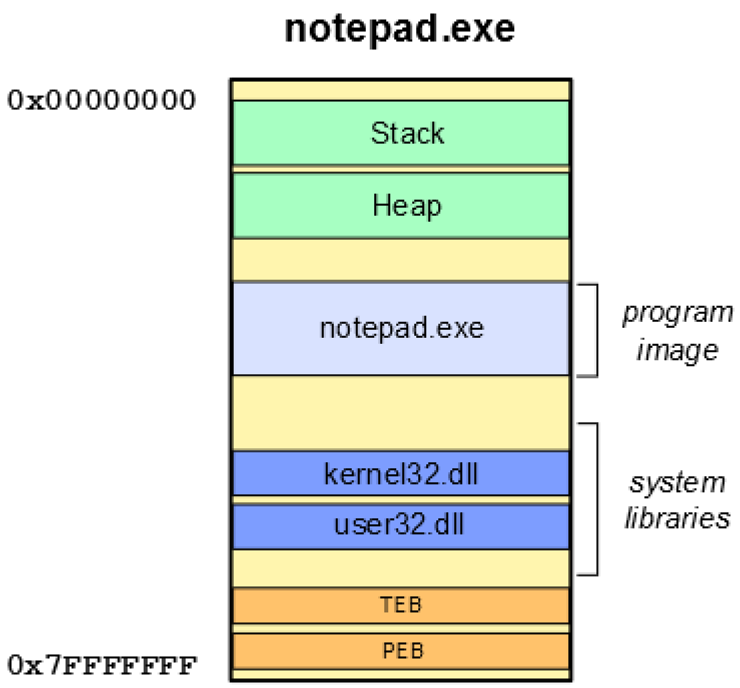
[27] Virtual address spaces https://msdn.microsoft.com/en-us/library/windows/hardware/hh439648%28v=vs.85%29.aspx (last accessed 11.09.2015)

[28] 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)

[29] 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)

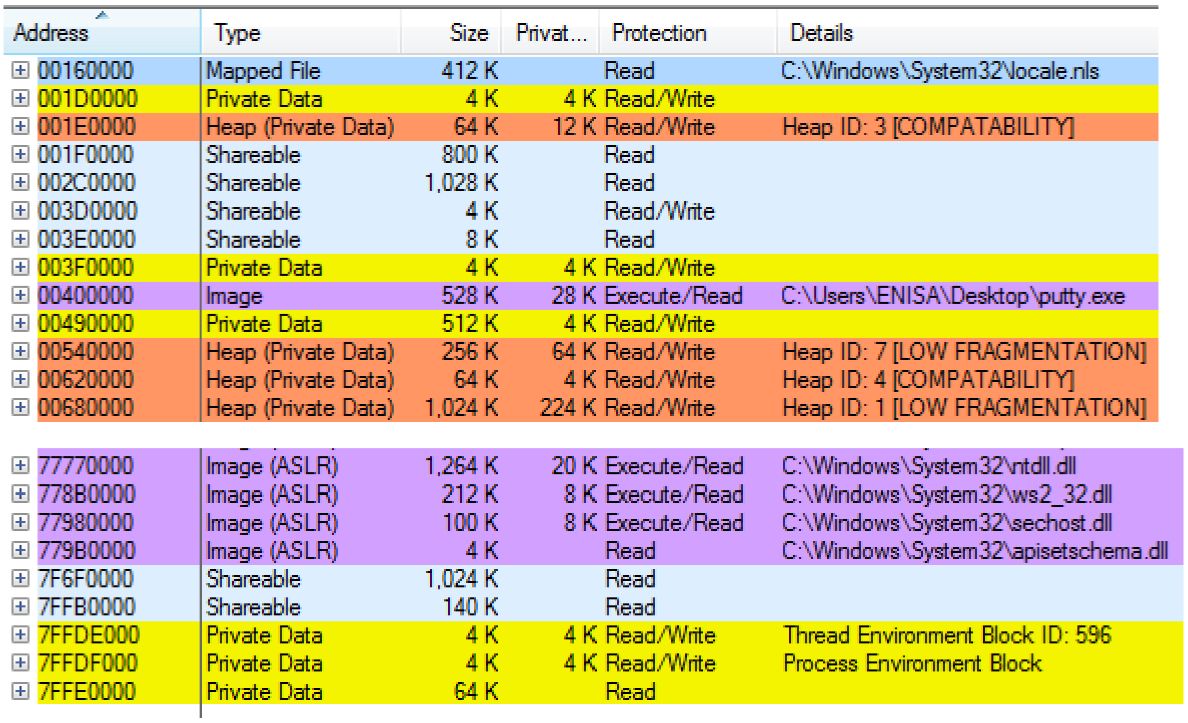


When a new PE executable is started on a Windows system, a new process is created and the system loader maps the PE file into the process's address space as well as loads all DLL libraries needed by the program. Process heap and stack are also created.



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[30] or multiple heaps). You can view a detailed memory map for any process using VMMap tool from Sysinternals[31].

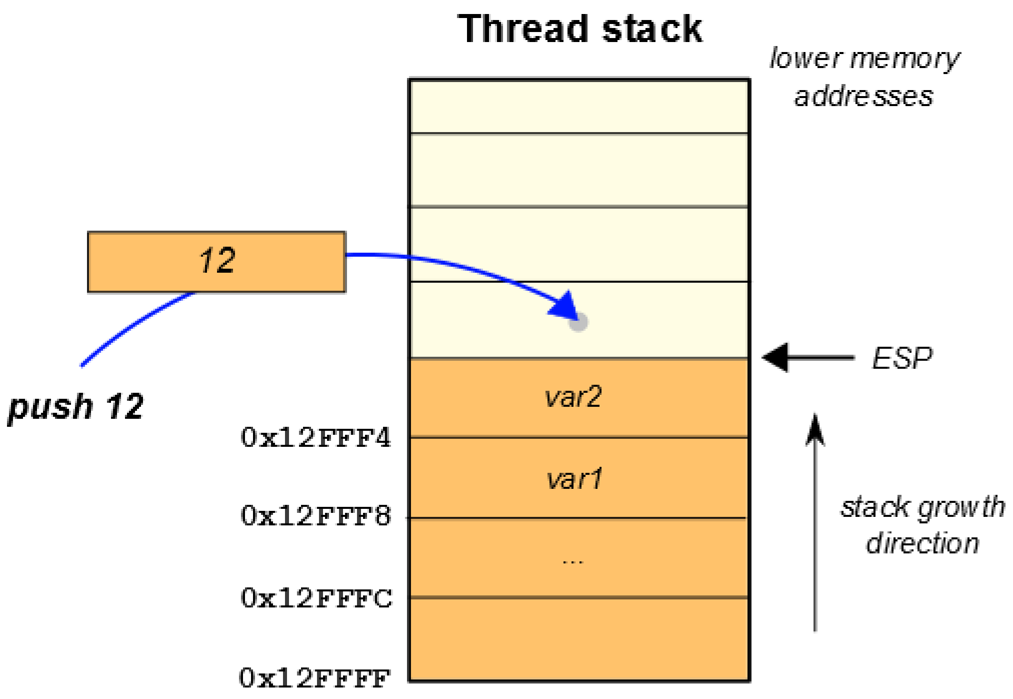


Address	Type	Size	Privat...	Protection	Details
00160000	Mapped File	412 K		Read	C:\Windows\System32\locale.nls
001D0000	Private Data	4 K	4 K	Read/Write	
001E0000	Heap (Private Data)	64 K	12 K	Read/Write	Heap ID: 3 [COMPATABILITY]
001F0000	Shareable	800 K		Read	
002C0000	Shareable	1,028 K		Read	
003D0000	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.

[30] Changing Environment Variables https://msdn.microsoft.com/en-us/library/windows/desktop/ms682009%28v=vs.85%29.aspx (last accessed 11.09.2015)
[31] 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.

3.5 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)

3.5.1 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 increases/decreases ESI and EDI according to DF flag.	None
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

3.5.2 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 EAX and remainder in EDX.	CF, OF, SF, ZF, AF, PF
mul <reg> **mul <mem>**	Unsigned multiply. Multiples value in EAX (destination) and operand. Result stored in EDX:EAX.	OF, CF
imul <reg> **imul <mem>**	Signed multiply. Multiples value in EAX (destination) and operand. Result stored 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

3.5.3 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.

3.5.4 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

3.5.5 Jump instructions

Instruction	Description	Affected flags
jmp <rel> **jmp <reg>** **jmp <mem>**	Always jumps to the address specified by the operand.	None
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

3.6 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.

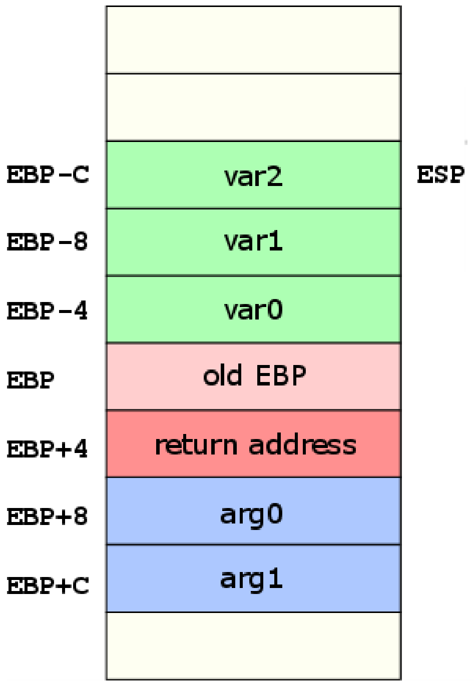
1. Passing parameters for the function (if any).
2. Calling function.
3. Reserving stack memory for local variables.
4. Function operations.
5. Restoring stack.
6. Function return and optionally cleaning function arguments.

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[32].

- 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[33].

- 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[34].

- 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[35]. 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.

[32] __cdecl https://msdn.microsoft.com/en-us/library/zkwh89ks.aspx (last accessed 11.09.2015)
[33] __stdcall https://msdn.microsoft.com/en-us/library/zxk0tw93.aspx (last accessed 11.09.2015)
[34] __fastcall https://msdn.microsoft.com/en-us/library/6xa169sk.aspx (last accessed 11.09.2015)
[35] __thiscall https://msdn.microsoft.com/en-us/library/ek8tkfbw.aspx (last accessed 11.09.2015)

4. Tools overview

In this training you will be using OllyDbg[36] as the primary debugger during the second part (*Advanced Dynamic Analysis*) and IDA Pro Free[37] as disassembler in the third part (*Advanced Static Analysis*). Even though both are very popular and powerful, they are not the only choices. This section presents some popular and freely available debuggers and disassemblers for *Portable Executable* files.

4.1 OllyDbg

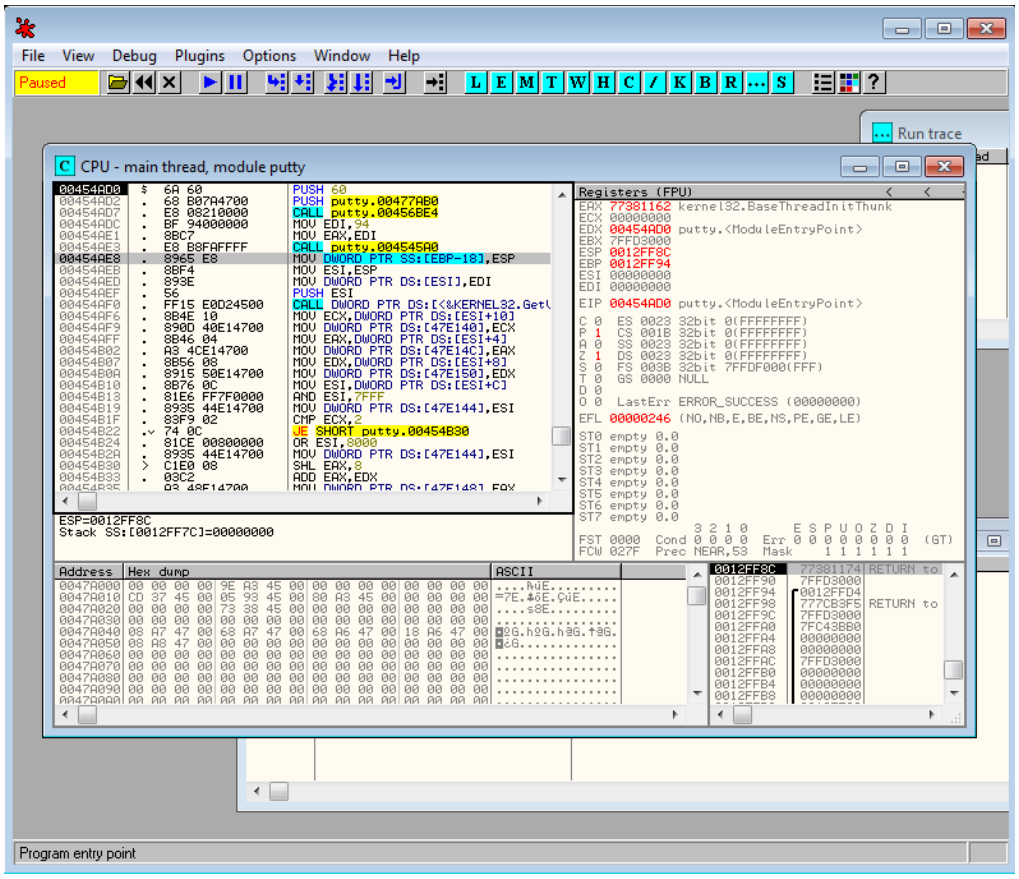
Very popular and powerful graphical debugger created by Oleh Yuschuk for 32-bit Portable Executable (PE) files (at the time of writing the work on a 64bit version started). A big advantage of OllyDbg is its community which over the years created dozens of plugins, scripts and tutorials freely available online for anyone interested.

OllyDbg is small, lightweight and has a very intuitive interface. Some of its features are:

- Multiple debugging modes (step over, step into, trace, animate, execute till return, etc.).
- Software, hardware and memory breakpoints.
- Conditional breakpoints.
- Properly recognizing most Windows API functions.
- Dynamic annotations in the code (what API function is called, arguments names).
- Various search modes (all referenced strings, searching whole memory).
- Customizability through the plugins mechanism.
- Multiple configuration options (debugging and appearance).
- Tracing of stack frames.
- Support for easy patching code and editing assembly.
- Can create a new process or attach to existing process.
- Can be set as just-in-time debugger.

OllyDbg can be downloaded from: http://www.ollydbg.de/

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


File View Debug Plugins Options Window Help
Paused
L E M T W H C / K B R ... S
Run trace
CPU - main thread, module putty
Program entry point

```
00454AD0  $  6A 60            PUSH 60
00454AD2  .  68 B07A4700      PUSH putty.00477AB0
00454AD7  .  E8 08210000      CALL putty.00456BE4
00454ADC  .  BF 94000000      MOV EDI,94
00454AE1  .  8BC7             MOV EAX,EDI
00454AE3  .  E8 B8FAFFFF      CALL putty.004545A0
00454AE8  .  8965 E8          MOV DWORD PTR SS:[EBP-18],ESP
00454AEB  .  8BF4             MOV ESI,ESP
00454AED  .  893E             MOV DWORD PTR DS:[ESI],EDI
00454AEF  .  56               PUSH ESI
00454AF0  .  FF15 E0D24500    CALL DWORD PTR DS:[<&KERNEL32.Get
00454AF6  .  8B4E 10          MOV ECX,DWORD PTR DS:[ESI+10]
00454AF9  .  890D 40E14700    MOV DWORD PTR DS:[47E140],ECX
00454AFF  .  8B46 04          MOV EAX,DWORD PTR DS:[ESI+4]
00454B02  .  A3 4CE14700      MOV DWORD PTR DS:[47E14C],EAX
00454B07  .  8B56 08          MOV EDX,DWORD PTR DS:[ESI+8]
00454B0A  .  8915 50E14700    MOV DWORD PTR DS:[47E150],EDX
00454B10  .  8B76 0C          MOV ESI,DWORD PTR DS:[ESI+C]
00454B13  .  81E6 FF7F0000    AND ESI,7FFF
00454B19  .  8935 44E14700    MOV DWORD PTR DS:[47E144],ESI
00454B1F  .  83F9 02          CMP ECX,2
00454B22  .v 74 0C            JE SHORT putty.00454B30
00454B24  .  81CE 00800000    OR ESI,8000
00454B2A  .  8935 44E14700    MOV DWORD PTR DS:[47E144],ESI
00454B30  >  C1E0 08          SHL EAX,8
00454B33  .  03C2             ADD EAX,EDX
00454B35  .  A3 48E14700      MOV DWORD PTR DS:[47E148],EAX

ESP=0012FF8C
Stack SS:[0012FF7C]=00000000
```

```
Registers (FPU)
EAX 77381162 kernel32.BaseThreadInitThunk
ECX 00000000
EDX 00454AD0 putty.<ModuleEntryPoint>
EBX 7FFD3000
ESP 0012FF8C
EBP 0012FF94
ESI 00000000
EDI 00000000

EIP 00454AD0 putty.<ModuleEntryPoint>

C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)

EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)

ST0 empty 0.0
ST1 empty 0.0
ST2 empty 0.0
ST3 empty 0.0
ST4 empty 0.0
ST5 empty 0.0
ST6 empty 0.0
ST7 empty 0.0
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (GT)
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

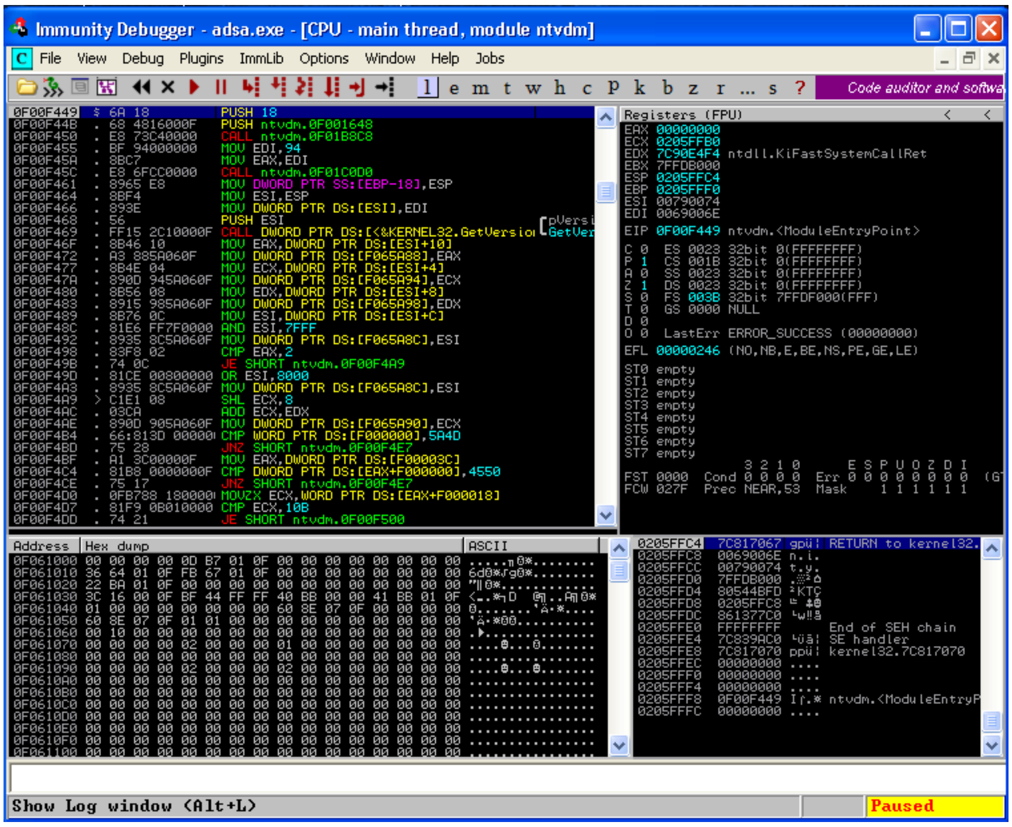
```
Address  Hex dump                                         ASCII
0047A000 00 00 00 00 9E A3 45 00 00 00 00 00 00 00 00 00 ....ÑúE.........
0047A010 CD 37 45 00 05 93 45 00 80 A3 45 00 00 00 00 00 =7E.‡ôE.ÇúE.....
0047A020 00 00 00 00 73 38 45 00 00 00 00 00 00 00 00 00 ....s8E.........
0047A030 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0047A040 08 A7 47 00 68 A7 47 00 68 A6 47 00 18 A6 47 00 ◘ºG.hºG.hªG.↑ªG.
0047A050 08 A8 47 00 00 00 00 00 00 00 00 00 00 00 00 00 ◘¿G.............
0047A060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0047A070 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0047A080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0047A090 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0047A0A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
0012FF8C  77381174  RETURN to
0012FF90  7FFD3000
0012FF94  0012FFD4
0012FF98  777CB3F5  RETURN to
0012FF9C  7FFD3000
0012FFA0  7FC43BB0
0012FFA4  00000000
0012FFA8  00000000
0012FFAC  7FFD3000
0012FFB0  00000000
0012FFB4  00000000
0012FFB8  00000000
```

4.2 Immunity Debugger

A popular graphical debugger that is based on the OllyDbg code, which makes it a very similartool. It gained popularity mostly among exploit developers but it's also widely used for other reverse engineering tasks. In comparison to OllyDbg its advantage is the integration with Python allowing to automate certain tasks using Python scripts. It also has the capability of making function graphs similar to the graphs in IDA disassembler. The Immunity Debugger can be downloaded from:

http://immunityinc.com/products/debugger/index.html

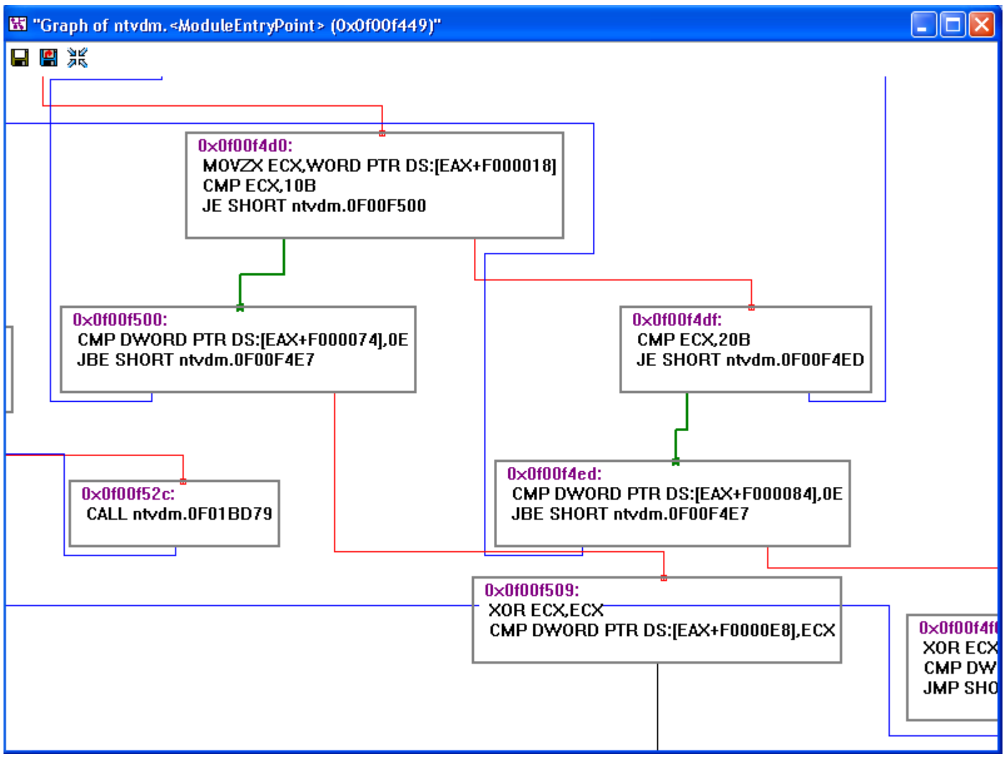

Immunity Debugger - adsa.exe - [CPU - main thread, module ntvdm]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c P k b z r ... s ?
Code auditor and softwa
Show Log window (Alt+L)
Paused

```
0F00F449  $ 6A 18           PUSH 18
0F00F44B  . 68 4816000F     PUSH ntvdm.0F001648
0F00F450  . E8 73C40000     CALL ntvdm.0F01B8C8
0F00F455  . BF 94000000     MOV EDI,94
0F00F45A  . 8BC7            MOV EAX,EDI
0F00F45C  . E8 6FCC0000     CALL ntvdm.0F01C0D0
0F00F461  . 8965 E8         MOV DWORD PTR SS:[EBP-18],ESP
0F00F464  . 8BF4            MOV ESI,ESP
0F00F466  . 893E            MOV DWORD PTR DS:[ESI],EDI
0F00F468  . 56              PUSH ESI                                  [pVersi
0F00F469  . FF15 2C10000F   CALL DWORD PTR DS:[<&KERNEL32.GetVersio  [GetVer
0F00F46F  . 8B46 10         MOV EAX,DWORD PTR DS:[ESI+10]
0F00F472  . A3 885A060F     MOV DWORD PTR DS:[F065A88],EAX
0F00F477  . 8B4E 04         MOV ECX,DWORD PTR DS:[ESI+4]
0F00F47A  . 890D 945A060F   MOV DWORD PTR DS:[F065A94],ECX
0F00F480  . 8B56 08         MOV EDX,DWORD PTR DS:[ESI+8]
0F00F483  . 8915 985A060F   MOV DWORD PTR DS:[F065A98],EDX
0F00F489  . 8B76 0C         MOV ESI,DWORD PTR DS:[ESI+C]
0F00F48C  . 81E6 FF7F0000   AND ESI,7FFF
0F00F492  . 8935 8C5A060F   MOV DWORD PTR DS:[F065A8C],ESI
0F00F498  . 83F8 02         CMP EAX,2
0F00F49B  . 74 0C           JE SHORT ntvdm.0F00F4A9
0F00F49D  . 81CE 00800000   OR ESI,8000
0F00F4A3  . 8935 8C5A060F   MOV DWORD PTR DS:[F065A8C],ESI
0F00F4A9  > C1E1 08         SHL ECX,8
0F00F4AC  . 03CA            ADD ECX,EDX
0F00F4AE  . 890D 905A060F   MOV DWORD PTR DS:[F065A90],ECX
0F00F4B4  . 66:813D 000000  CMP WORD PTR DS:[F000000],5A4D
0F00F4BD  . 75 28           JNZ SHORT ntvdm.0F00F4E7
0F00F4BF  . A1 3C00000F     MOV EAX,DWORD PTR DS:[F00003C]
0F00F4C4  . 81B8 0000000F   CMP DWORD PTR DS:[EAX+F000000],4550
0F00F4CE  . 75 17           JNZ SHORT ntvdm.0F00F4E7
0F00F4D0  . 0FB788 180000   MOVZX ECX,WORD PTR DS:[EAX+F000018]
0F00F4D7  . 81F9 0B010000   CMP ECX,10B
0F00F4DD  . 74 21           JE SHORT ntvdm.0F00F500
```

```
Registers (FPU)
EAX 00000000
ECX 0205FFB0
EDX 7C90E4F4 ntdll.KiFastSystemCallRet
EBX 7FFDB000
ESP 0205FFC4
EBP 0205FFF0
ESI 00790074
EDI 0069006E
EIP 0F00F449 ntvdm.<ModuleEntryPoint>
C 0  ES 0023 32bit 0(FFFFFFFF)
P 1  CS 001B 32bit 0(FFFFFFFF)
A 0  SS 0023 32bit 0(FFFFFFFF)
Z 1  DS 0023 32bit 0(FFFFFFFF)
S 0  FS 003B 32bit 7FFDF000(FFF)
T 0  GS 0000 NULL
D 0
O 0  LastErr ERROR_SUCCESS (00000000)
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)
ST0 empty
ST1 empty
ST2 empty
ST3 empty
ST4 empty
ST5 empty
ST6 empty
ST7 empty
               3 2 1 0      E S P U O Z D I
FST 0000  Cond 0 0 0 0  Err 0 0 0 0 0 0 0 0  (G
FCW 027F  Prec NEAR,53  Mask    1 1 1 1 1 1
```

```
Address  Hex dump                                         ASCII
0F061000 00 00 00 00 0D B7 01 0F 00 00 00 00 00 00 00 00 .....╖0*........
0F061010 36 64 01 0F FB 67 01 0F 00 00 00 00 00 00 00 00 6d0*√g0*........
0F061020 22 BA 01 0F 00 00 00 00 00 00 00 00 00 00 00 00 "‖0*............
0F061030 3C 16 00 0F BF 44 FF FF 40 BB 00 00 41 BB 01 0F <▬.*┐D  @╗..A╗0*
0F061040 01 00 00 00 00 00 00 00 60 8E 07 0F 00 00 00 00 0.......`Ä•*....
0F061050 60 8E 07 0F 01 01 00 00 00 00 00 00 00 00 00 00 `Ä•*00..........
0F061060 00 10 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .►..............
0F061070 00 00 00 00 02 00 00 00 01 00 00 00 00 00 00 00 ....☻...0.......
0F061080 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F061090 00 00 00 00 02 00 00 00 02 00 00 00 00 00 00 00 ....☻...☻.......
0F0610A0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F0610B0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F0610C0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F0610D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F0610E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F0610F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0F061100 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
```

```
0205FFC4  7C817067 gpü|  RETURN to kernel32.
0205FFC8  0069006E n.i.
0205FFCC  00790074 t.y.
0205FFD0  7FFDB000 .░²⌂
0205FFD4  80544BFD ²KTÇ
0205FFD8  0205FFC8 ╚ ♣☻
0205FFDC  861377C0 └w‼å
0205FFE0  FFFFFFFF       End of SEH chain
0205FFE4  7C839AC0 └üä|  SE handler
0205FFE8  7C817070 ppü|  kernel32.7C817070
0205FFEC  00000000 ....
0205FFF0  00000000 ....
0205FFF4  00000000 ....
0205FFF8  0F00F449 I♪.*  ntvdm.<ModuleEntryP
0205FFFC  00000000 ....
```

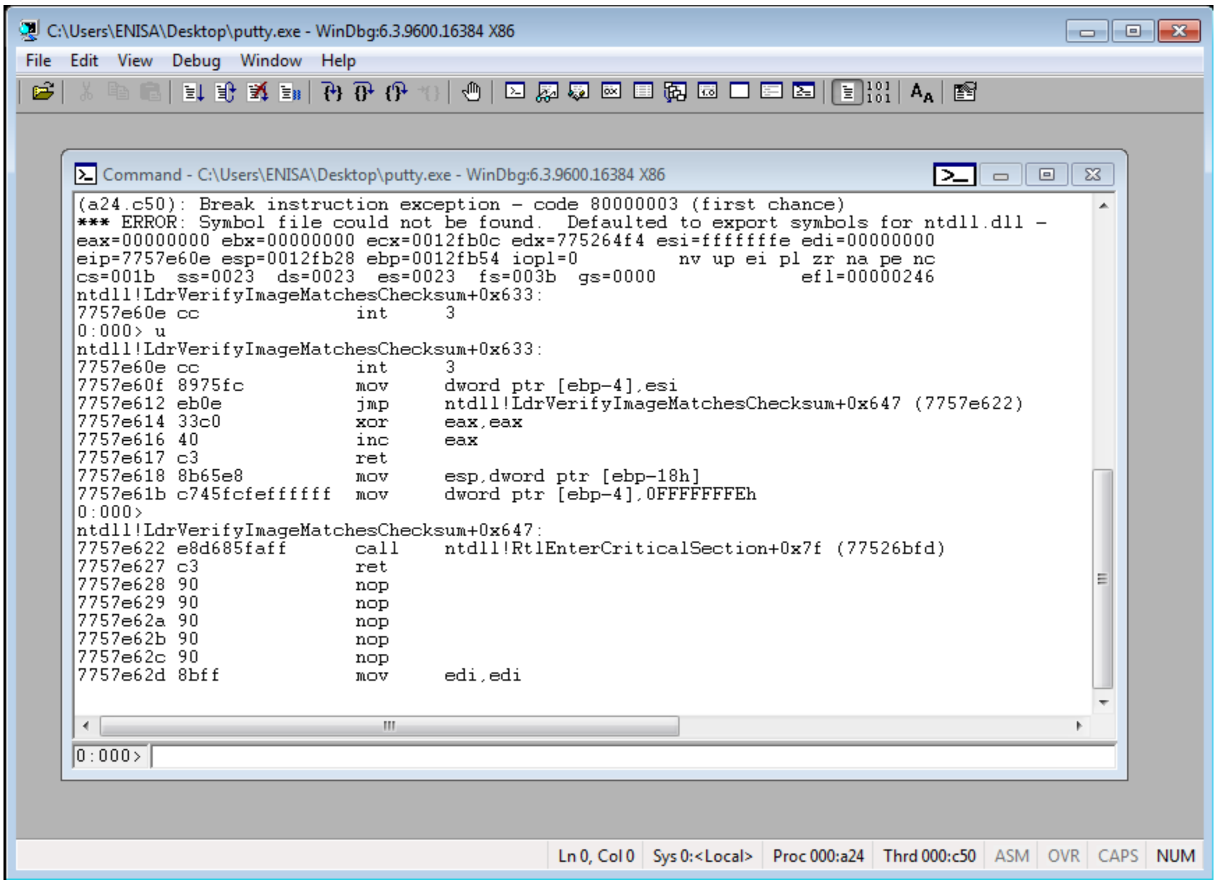


```
0x0f00f4d0:
 MOVZX ECX,WORD PTR DS:[EAX+F000018]
 CMP ECX,10B
 JE SHORT ntvdm.0F00F500

0x0f00f500:
 CMP DWORD PTR DS:[EAX+F000074],0E
 JBE SHORT ntvdm.0F00F4E7

0x0f00f4df:
 CMP ECX,20B
 JE SHORT ntvdm.0F00F4ED

0x0f00f52c:
 CALL ntvdm.0F01BD79

0x0f00f4ed:
 CMP DWORD PTR DS:[EAX+F000084],0E
 JBE SHORT ntvdm.0F00F4E7

0x0f00f509:
 XOR ECX,ECX
 CMP DWORD PTR DS:[EAX+F0000E8],ECX

0x0f00f4f
 XOR ECX
 CMP DW
 JMP SHO
```

4.3 WinDbg

WinDbg[38] is a powerful debugger distributed for free by Microsoft. In contrast with OllyDbg and Immunity Debugger, WinDbg is a text mode debugger without a nice visual presentation of disassembled code. Most of the things in WinDbg are done by executing commands at the command line. This makes it more difficult for the beginner to learn.

What makes WinDbg so powerful is its integration with symbol files provided by Microsoft which lets you see additional information about various system structures. Unlike OllyDbg and Immunity Debugger, WinDbg also allows to debug 64-bit executables and do kernel-level debugging or analysis of crash dump files. WinDbg has its own scripting language, but thanks to PyKd[39] project it's also possible to integrate WinDbg with Python.


C:\Users\ENISA\Desktop\putty.exe - WinDbg:6.3.9600.16384 X86
File Edit View Debug Window Help
Command - C:\Users\ENISA\Desktop\putty.exe - WinDbg:6.3.9600.16384 X86
Ln 0, Col 0
Sys 0:<Local>
Proc 000:a24
Thrd 000:c50
ASM
OVR
CAPS
NUM

```
(a24.c50): Break instruction exception - code 80000003 (first chance)
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll.dll -
eax=00000000 ebx=00000000 ecx=0012fb0c edx=775264f4 esi=fffffffe edi=00000000
eip=7757e60e esp=0012fb28 ebp=0012fb54 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ntdll!LdrVerifyImageMatchesChecksum+0x633:
7757e60e cc              int     3
0:000> u
ntdll!LdrVerifyImageMatchesChecksum+0x633:
7757e60e cc              int     3
7757e60f 8975fc          mov     dword ptr [ebp-4],esi
7757e612 eb0e            jmp     ntdll!LdrVerifyImageMatchesChecksum+0x647 (7757e622)
7757e614 33c0            xor     eax,eax
7757e616 40              inc     eax
7757e617 c3              ret
7757e618 8b65e8          mov     esp,dword ptr [ebp-18h]
7757e61b c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
0:000>
ntdll!LdrVerifyImageMatchesChecksum+0x647:
7757e622 e8d685faff      call    ntdll!RtlEnterCriticalSection+0x7f (77526bfd)
7757e627 c3              ret
7757e628 90              nop
7757e629 90              nop
7757e62a 90              nop
7757e62b 90              nop
7757e62c 90              nop
7757e62d 8bff            mov     edi,edi
```

```
0:000>
```

4.4 WinAppDbg / PyDbg

WinAppDbg[40] and older PyDbg (part of PaiMei[41] project) are not debuggers in a classical sense intended for instruction tracing and code analysis. Instead they are Python modules allowing you to create your own fully scriptable debugger. You can use them to create debugging scripts automating certain debugging tasks.

There are many use cases for such scripts. They usually come in handy if you know that the malicious code is following some particular scheme. For example if you are repeatedly observing campaigns of the same malware but

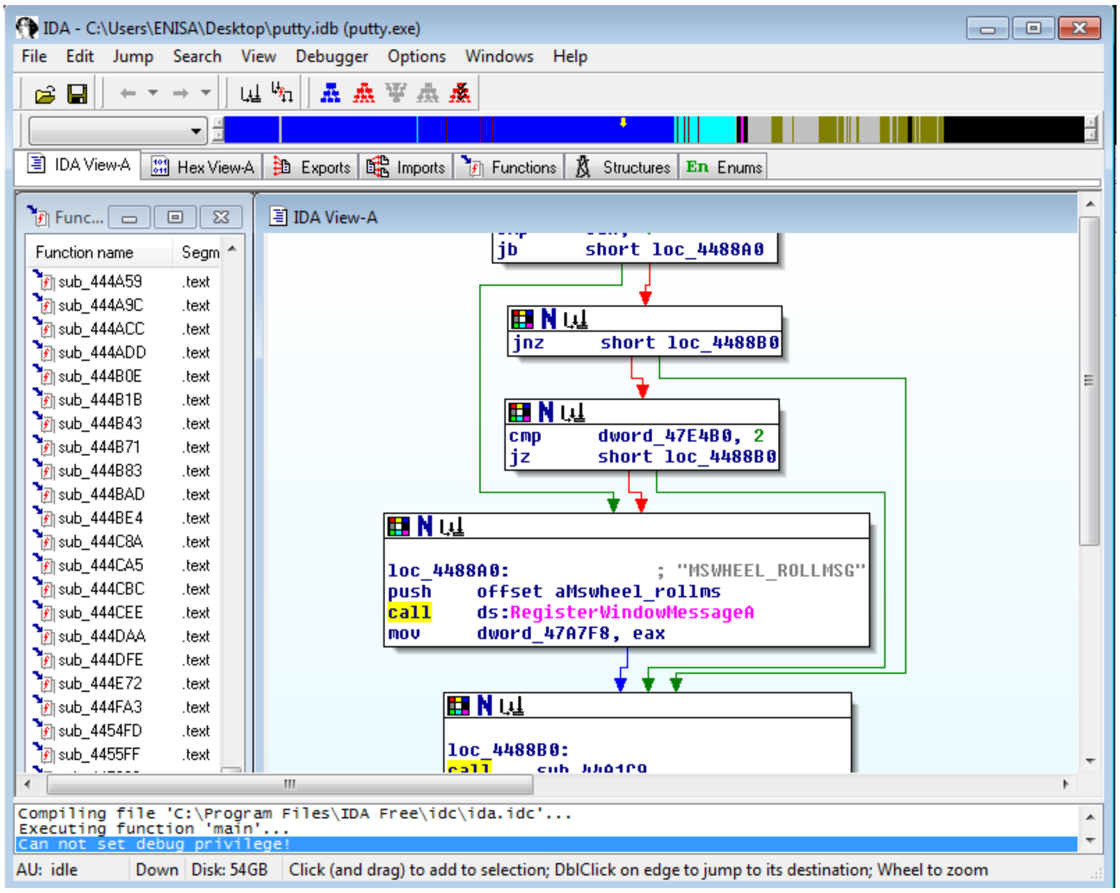
[38]Debugging Tools for Windows (WinDbg, KD, CDB, NTSD)
https://msdn.microsoft.com/library/windows/hardware/ff551063%28v=vs.85%29.aspx (last accessed 11.09.2015)
[39]Python extension for WinDbg https://pykd.codeplex.com/ (last accessed 11.09.2015)
[40]WinAppDbg http://winappdbg.sourceforge.net/ (last accessed 11.09.2015)
[41]PaiMei http://pedramamini.com/PaiMei/docs/ (last accessed 11.09.2015)

connecting to different botnet infrastructures, you might create a script automatically finding C&C addresses and dumping configuration files. Another use case would be to create unpacking scripts for repeatedly observed samples packed with the same packer. Finally, you can also create application fuzzers used in vulnerability research or various system monitoring tools which could be for example used in high interaction honeypots.

4.5 IDA Pro

Many malware researchers and reverse engineers nowadays use IDA Pro. Though it is a commercial and non-free product they also provide a freeware version – IDA Freeware[42]. IDA Freeware is an older version of IDA Pro that is limited in certain functions and intended for a non-commercial use. Despite those limitations IDA Freeware is still a very powerful disassembler which allows you to perform advanced analyses of malicious code.

IDA is an interactive disassembler supporting many executable file types and processor architectures. One of its most valuable features is graph view which offers a visualisation of the function structure. This helps to understand function execution flow and sometimes guess its purpose. IDA also provides many different types of data views helping to better understand analysed binary files. IDA is very customisable offering many configuration options. It's also possible to create custom scripts in its scripting language (IDC), and in newer versions use Python scripts. Additionally, thanks to the FLIRT engine, IDA not only disassembles the executable code but also tries to detect the code from statically linked libraries – which are usually not necessary to analyse.


IDA - C:\Users\ENISA\Desktop\putty.idb (putty.exe)
File Edit Jump Search View Debugger Options Windows Help
IDA View-A
Hex View-A
Exports
Imports
Functions
Structures
Enums
Func...
IDA View-A

Function name	Segm
sub_444A59	.text
sub_444A9C	.text
sub_444ACC	.text
sub_444ADD	.text
sub_444B0E	.text
sub_444B1B	.text
sub_444B43	.text
sub_444B71	.text
sub_444B83	.text
sub_444BAD	.text
sub_444BE4	.text
sub_444C8A	.text
sub_444CA5	.text
sub_444CBC	.text
sub_444CEE	.text
sub_444DAA	.text
sub_444DFE	.text
sub_444E72	.text
sub_444FA3	.text
sub_4454FD	.text
sub_4455FF	.text

```
jb      short loc_4488A0
```

```
jnz     short loc_4488B0
```

```
cmp     dword_47E4B0, 2
jz      short loc_4488B0
```

```
loc_4488A0:                ; "MSWHEEL_ROLLMSG"
push    offset aMswheel_rollms
call    ds:RegisterWindowMessageA
mov     dword_47A7F8, eax
```

```
loc_4488B0:
```

```
Compiling file 'C:\Program Files\IDA Free\idc\ida.idc'...
Executing function 'main'...
Can not set debug privilege!
```

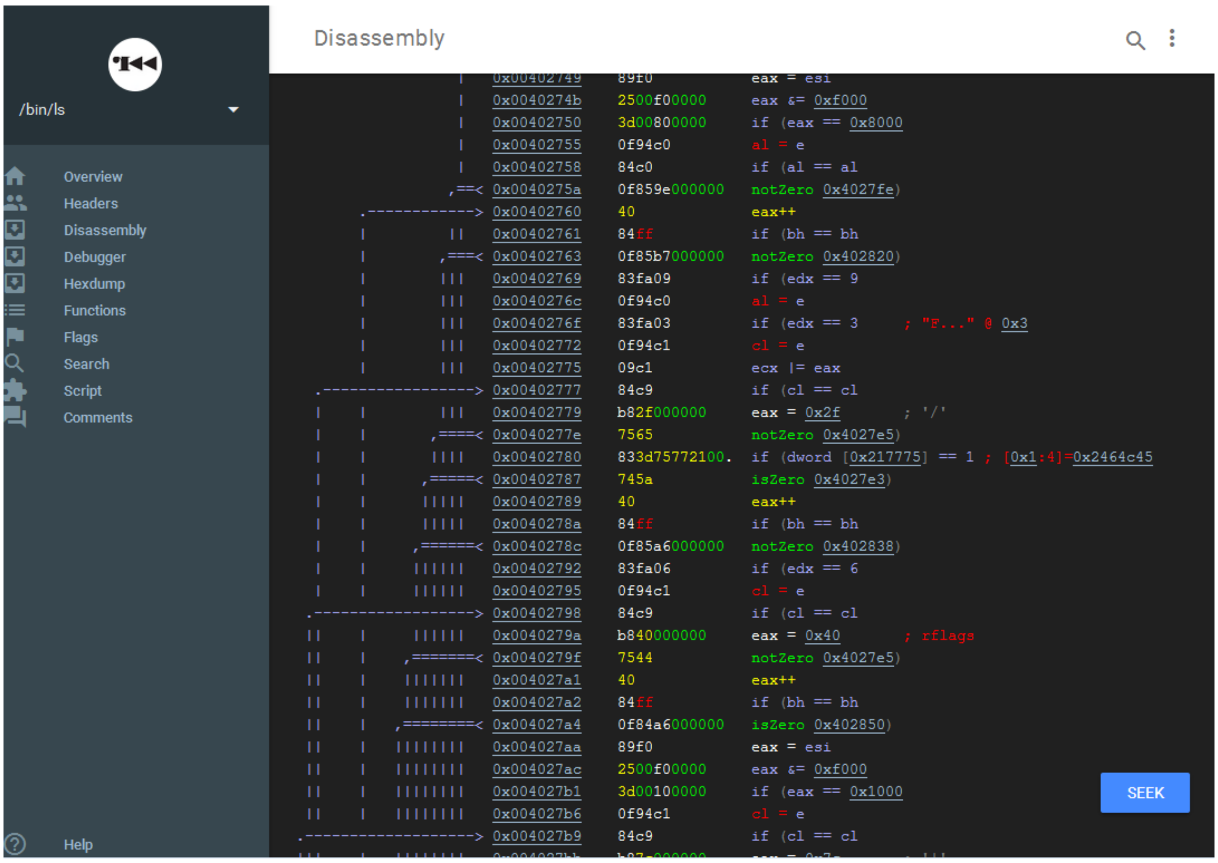
AU: idle | Down | Disk: 54GB | Click (and drag) to add to selection; DblClick on edge to jump to its destination; Wheel to zoom

[42] Freeware version of IDA v5.0 https://www.hex-rays.com/products/ida/support/download_freeware.shtml (last accessed 11.09.2015)

4.6 Radare2

Radare2[43] is a dynamically developing open source project, providing a complete reverse-engineering framework. One of its elements is a disassembler supporting multiple types of executable files and processor architectures. Besides disassembling executable files, Radare2 allows debugging samples with either a local debugger or remote debuggers. Radare2 also provides forensic capabilities and aids exploit development.

Radare2 is available for multiple platforms (Linux, Windows, OS X) and is also a part of REMnux[44] Linux distribution. By origin it is a command line (CLI) utility, but a web interface and PyGTK GUI are also available, as seen from the following picture acquired from http://cloud.radare.org/m/ .



```
                    |  0x00402749    89f0          eax = esi
                    |  0x0040274b    2500f00000    eax &= 0xf000
                    |  0x00402750    3d00800000    if (eax == 0x8000
                    |  0x00402755    0f94c0        al = e
                    |  0x00402758    84c0          if (al == al
                 ,==<  0x0040275a    0f859e000000  notZero 0x4027fe)
      .------------->  0x00402760    40            eax++
      |           ||   0x00402761    84ff          if (bh == bh
      |        ,===<   0x00402763    0f85b7000000  notZero 0x402820)
      |          |||   0x00402769    83fa09        if (edx == 9
      |          |||   0x0040276c    0f94c0        al = e
      |          |||   0x0040276f    83fa03        if (edx == 3     ; "F..." @ 0x3
      |          |||   0x00402772    0f94c1        cl = e
      |          |||   0x00402775    09c1          ecx |= eax
  .----------------->  0x00402777    84c9          if (cl == cl
  |   |          |||   0x00402779    b82f000000    eax = 0x2f       ; '/'
  |   |       ,====<   0x0040277e    7565          notZero 0x4027e5)
  |   |         ||||   0x00402780    833d75772100. if (dword [0x217775] == 1 ; [0x1:4]=0x2464c45
  |   |      ,=====<   0x00402787    745a          isZero 0x4027e3)
  |   |        |||||   0x00402789    40            eax++
  |   |        |||||   0x0040278a    84ff          if (bh == bh
  |   |     ,======<   0x0040278c    0f85a6000000  notZero 0x402838)
  |   |       ||||||   0x00402792    83fa06        if (edx == 6
  |   |       ||||||   0x00402795    0f94c1        cl = e
 .------------------>  0x00402798    84c9          if (cl == cl
 ||   |       ||||||   0x0040279a    b840000000    eax = 0x40       ; rflags
 ||   |    ,=======<   0x0040279f    7544          notZero 0x4027e5)
 ||   |      |||||||   0x004027a1    40            eax++
 ||   |      |||||||   0x004027a2    84ff          if (bh == bh
 ||   |   ,========<   0x004027a4    0f84a6000000  isZero 0x402850)
 ||   |     ||||||||   0x004027aa    89f0          eax = esi
 ||   |     ||||||||   0x004027ac    2500f00000    eax &= 0xf000
 ||   |     ||||||||   0x004027b1    3d00100000    if (eax == 0x1000
 ||   |     ||||||||   0x004027b6    0f94c1        cl = e
.------------------->  0x004027b9    84c9          if (cl == cl
```

---

[43] Radare2https://github.com/radare/radare2 (last accessed 11.09.2015)
[44] REMnux https://remnux.org/ (last accessed 11.09.2015)

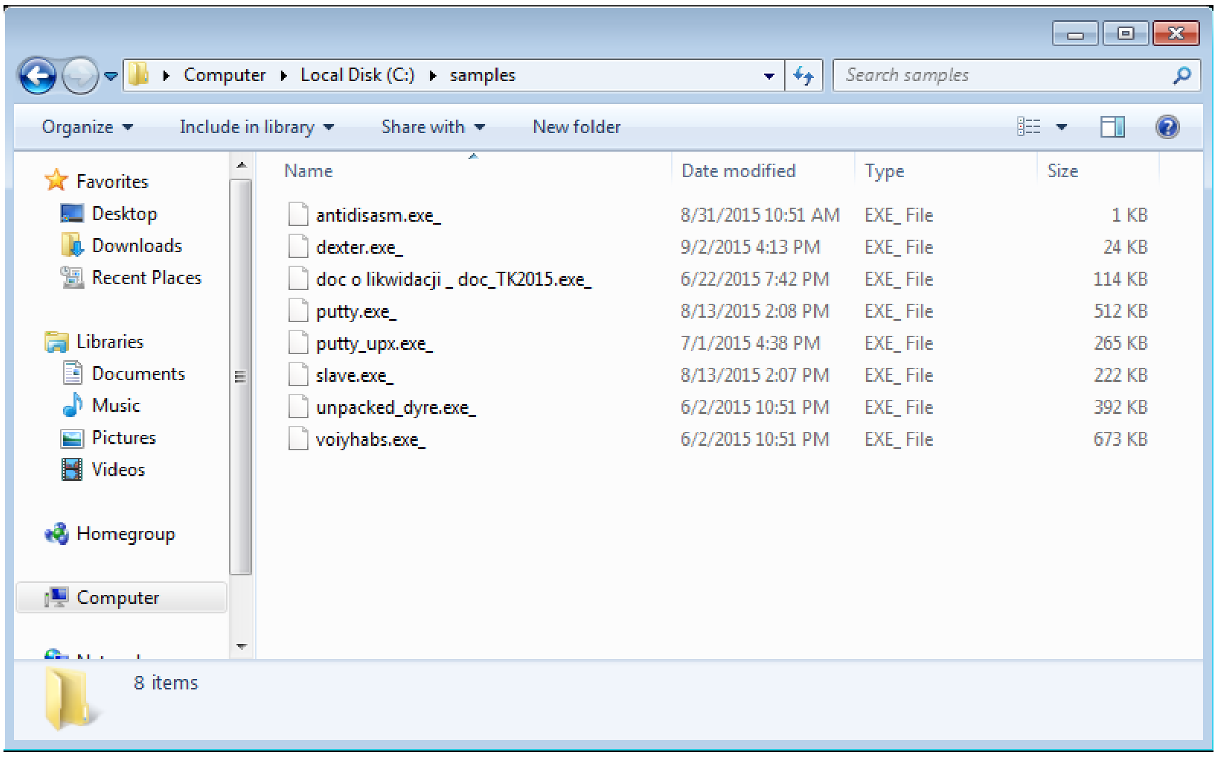
5. Environment preparation

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*[45].

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.



Name	Date modified	Type	Size
antidisasm.exe_	8/31/2015 10:51 AM	EXE_ File	1 KB
dexter.exe_	9/2/2015 4:13 PM	EXE_ File	24 KB
doc o likwidacji _ doc_TK2015.exe_	6/22/2015 7:42 PM	EXE_ File	114 KB
putty.exe_	8/13/2015 2:08 PM	EXE_ File	512 KB
putty_upx.exe_	7/1/2015 4:38 PM	EXE_ File	265 KB
slave.exe_	8/13/2015 2:07 PM	EXE_ File	222 KB
unpacked_dyre.exe_	6/2/2015 10:51 PM	EXE_ File	392 KB
voiyhabs.exe_	6/2/2015 10:51 PM	EXE_ File	673 KB

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:

[45] 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.

6. Training summary

In this training you have been introduced to the *advanced dynamic* and *static analysis* concept. You have learnt what the difference between both is and how they differ from the basic analysis presented in the ENISA *Artefact Analysis training*. Following you were given an introduction to the x86 assembly language, which is an essential skill when it comes to advanced analysis of the malicious code. This part explained fundamental concepts of x86 assembly language like instructions, opcodes, operands, registers, memory organisation and functions. Most common assembly instructions were also presented. Additionally, you were presented with most commonly used and freely available debuggers and disassemblers. Finally, this part of the training ended with a short summary describing how to prepare the environment for the exercises and what security precautions should be undertaken during the analysis of the malicious code.

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