
Module 1

Intro and environment setup

Foreword

New self-replicating malicious programs are built daily, and antiviruses daily release updates to protect systems against these attacks. So how can it be that when a virus infects your computer, the antivirus could fail to detect it even for months and months? This training aims to debunk the mistaken belief that good and up-to-date antivirus software can successfully shield you from all threats.

Virus and malware creators are every bit as hardworking as antivirus developers. The former want to discover the inner mechanisms of antiviruses, while the latter are on a constant lookout for how new viruses work. This training shows techniques you can use to write code that is undetectable even for the best antiviruses out there. Additionally, we will also show you how to cloak a program to make it stealthy. When you know the threat, you'll also be able to train yourself how to detect items that are seemingly impossible to spot.

What is a rootkit?

A rootkit is not a threat itself since none of its functions are harmful for the user. A rootkit is a program or auxiliary module of another program that is intended to hide files, processes, Windows Registry entries, network connections and other items from users. To do this, a rootkit changes program or system library code to make them return false data (e.g., a process list that lacks a particular element). This guide covers rootkits written for the user mode (Ring3). We'll leave out the kernel mode (Ring0) since newer Windows systems make it virtually impossible to modify the kernel and its structures.

Rootkit structure

Contrary to what you may think, the structure of a typical rootkit is very basic. A rootkit consists of an application that modifies the code of other processes and tracks whether new processes appear. It includes a set of system functions to change and a set of codes to replace the original function code excerpts.

Examples of rootkits

- ✓ Bluepill: implemented by Joanna Rutkowska. This highly advanced rootkit makes use of virtualization. Very difficult to detect as it runs the operating system as a virtual machine controlled by the rootkit (using the AMD Pacifica virtualization technology).
- ✓ FU Rootkit: a kernel mode rootkit based on the DKOM (Direct Kernel Object Manipulation) technique. Can cloak a variety of elements without hooking.
- ✓ Vanquish Rootkit: a user mode rootkit using DLL injection and API hooking.

What you need to know to start

All codes presented in the book are written mostly in C++, a language that is flexible and easy-to-use for low-level functions, and therefore a good match for the task. The compiler we'll use is Microsoft Visual C++ 2010 Express.

At least a passing knowledge of the assembly language will come in handy at the start as well. But if you don't know this language, don't worry: the next module features a short overview of what's needed to know to write and understand basic programs.

Also highly useful is a knowledge of Windows internals, especially with regard to API routines and system libraries. If you don't have it either, there's no need to panic – the book includes critical information where needed.

Compatibility and current code version

The training videos for modules 1 to 9 have been made in the Microsoft Windows 7 32-bit version. Module 10, a summary of the training, contains implementations of all the methods and has been developed for the 64-bit operating system versions.

The methods in this training are universal. The guide's sample programs have also been tested on the newest (at the time of writing) version of Windows, Microsoft Windows 8.x (32-bit for modules 1 to 9 and 64-bit for module 10). Note that almost all applications compiled for the 32-bit OS version should run in the 64-bit system without any need for adjustments. A few exceptions will require you to make changes to their codes: the how-to can be found in module 10.

A current archive containing sample codes and applications can be downloaded at: http://hackingschool.com/download/rtk_modules.zip. The archive will be brought up to date for all new OS versions released in the future.

Further reading

“Rootkits: Subverting the Windows Kernel”: Greg Hoggund, James Butler. This resourceful book is exhaustive for kernel mode rootkits, but focused mostly on Windows XP.

“Windows NT/2000 Native API Reference”: Gary Nebbett. Descriptions of many undocumented Windows API services that haven't changed a lot since the days of Windows NT.

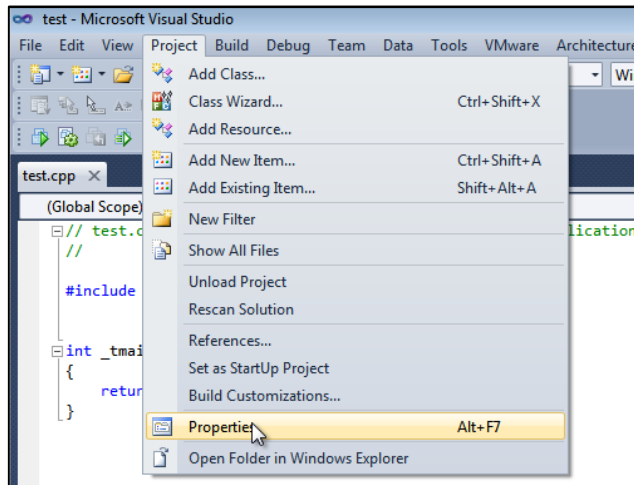


Practice: video module transcript

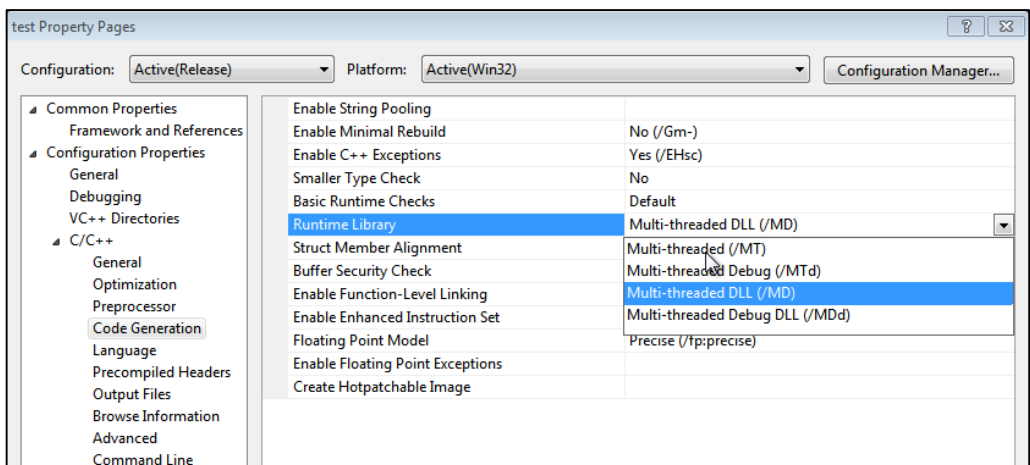
Welcome to the first module of the training. We'll learn how to use the applications used throughout this training. For instance, we'll be using Microsoft Visual Studio 2010 in its Express version. We'll also use the Olly Debugger in DeFixed version, which is a slightly modified version of the original Olly Debugger. Another tool which can come in handy is PEview. PEview is a program which shows us the headers of executable files and where we can see all the sections and imports specified in a binary file. We'll also use Hexplorer, a hexadecimal editor.

Now let's go on to discuss particular programs. The first application we'll get acquainted with is Microsoft Visual Studio 2010. The environment can be downloaded from the Microsoft website. It's one of the best C++ compilers available for the Microsoft Windows platform. The next application is Olly Debugger. Using it, we'll be able to see the results of the operation of our program as well as check whether everything is performed as planned. The next application, as I've already mentioned, is PEview. It will be used to learn about the structure of a PE executable file. Using it, we'll learn which imports and exports are used by a given application. Of course, it's only a small part of the features of this program, but it's precisely the one essential for us.

Another application is Hexplorer. It's the editor we'll use to edit binary files. We can use it to, for example, edit the character strings present in a binary file. We've briefly discussed the tools. Now let's create a sample project in Microsoft Visual Studio. We run the compiler. In order to create a new project, we click File, New, and next Project. We'll create a console application and place it in the Modules directory in subdirectory 1, because it's the first module of our training. The project will be named TEST. We click Next... and Finish. The project creation is in progress. In the screen we can see all the files which compose our project. We also have a template. We start from changing the mode of compilation of our project to Release. Now let's configure our project. We click Properties.

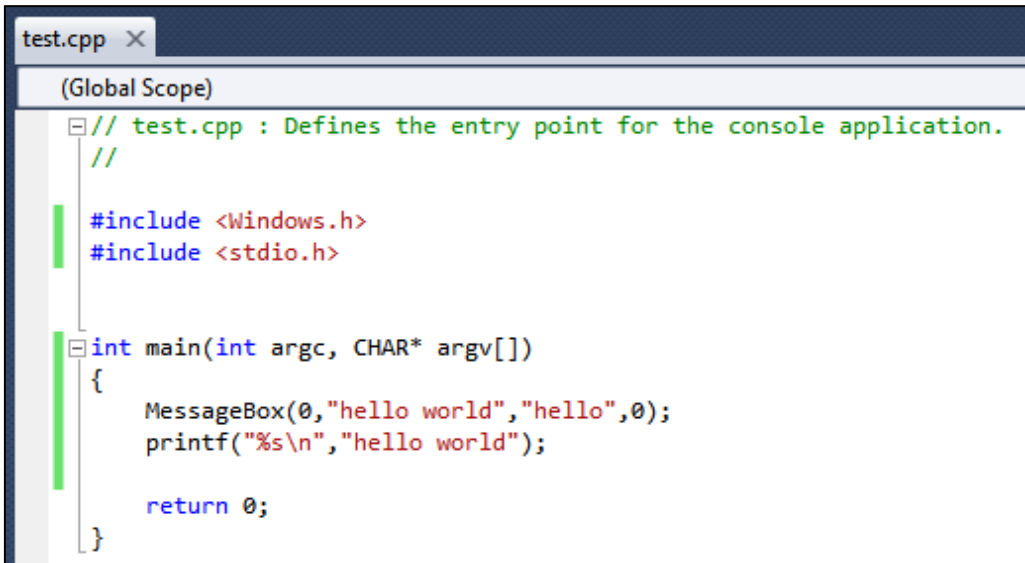


Next, we go to Configuration Properties, General and we choose Use Multi-Byte Character Set so that the strings we use are ANSI by default. Next, we click C/C++, choose Code Generation and click the Multi Threaded /MT option.



Then we hover over the Precompiled header and choose the option “Not using precompiled headers”. Finally, we click OK. After these changes, the project compiles without any problems, we just need to include the windows.h file which has all the WinApi functions declarations. In our program we'll use one of them. We'll need stdio headers in order to print the message in the console. We also have to remove `_t` prefixes. Let's call a simple MessageBox which will display the message Hello World. The Hello World message should appear in

an upcoming window. Let's also print Hello World in the window title and the console.

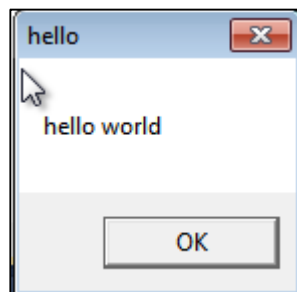


```
test.cpp x
(Global Scope)
// test.cpp : Defines the entry point for the console application.
//
#include <Windows.h>
#include <stdio.h>

int main(int argc, CHAR* argv[])
{
    MessageBox(0,"hello world","hello",0);
    printf("%s\n","hello world");

    return 0;
}
```

Let's now compile the project using the F7 key. At the bottom of the window we can see that the compilation is already in progress. As we can see, everything went OK. Now we can start our program. We can see the console and the MessageBox window. The Hello World text, which we defined in the program, appeared as well.



For a moment, we might have caught a glimpse of the Hello World text in the console, after which the application closes due to the fact that the return instruction was executed. Now let's have a look at our program in the debugger. It's located in the directory Modules\1\Test\Release. We simply drag our program to the debugger. In the screen we can see all the instructions used

by the application. We can see, for instance, the loaded modules. There are quite a lot of them, but we'll be mainly interested in libraries, such as kernel32, ntdll or kernelbase.

00850000	00003000	test		PE header	Priv	RW	RW
00800000	00001000	test		code	Imag	R	RWE
00B01000	00007000	test	.text	code	Imag	R	RWE
00B08000	00003000	test	.rdata	imports	Imag	R	RWE
00B0E000	00003000	test	.data	data	Imag	R	RWE
00B0F000	00001000	test	.rsrc	resources	Imag	R	RWE
00B10000	00001000	test	.reloc	relocations	Imag	R	RWE
00B10000	00147000			Map	R		R
6E050000	00001000	WINSPool		PE header	Imag	R	RWE
6E051000	000035000	WINSPool	.text	code, import	Imag	R	RWE
6E086000	00001000	WINSPool	.data		Imag	R	RWE
6E087000	00017000	WINSPool	.rsrc	resources	Imag	R	RWE
6E09E000	00003000	WINSPool	.reloc	relocations	Imag	R	RWE
717C0000	00001000	AcLayers		PE header	Imag	R	RWE
717C1000	00069000	AcLayers	.text	code, import	Imag	R	RWE
7182A000	0000A000	AcLayers	.data	data	Imag	R	RWE
71834000	00011000	AcLayers	.rsrc	resources	Imag	R	RWE
71845000	00008000	AcLayers	.reloc	relocations	Imag	R	RWE
71DB0000	00001000	MPR		PE header	Imag	R	RWE
71DB1000	0000E000	MPR	.text	code, import	Imag	R	RWE
71DBF000	00001000	MPR	.data	data	Imag	R	RWE
71DC0000	00001000	MPR	.rsrc	resources	Imag	R	RWE
71DC1000	00001000	MPR	.reloc	relocations	Imag	R	RWE
75060000	00001000	USERENU		PE header	Imag	R	RWE
75061000	00011000	USERENU	.text	code, import	Imag	R	RWE
75072000	00001000	USERENU	.orpc		Imag	R	RWE
75073000	00001000	USERENU	.data	data	Imag	R	RWE
75074000	00002000	USERENU	.rsrc	resources	Imag	R	RWE
75076000	00001000	USERENU	.reloc	relocations	Imag	R	RWE
75910000	00001000	SspiCli		PE header	Imag	R	RWE
75911000	00017000	SspiCli	.text	code, import	Imag	R	RWE
75928000	00001000	SspiCli	.data	data	Imag	R	RWE
75929000	00001000	SspiCli	.rsrc	resources	Imag	R	RWE
7592A000	00001000	SspiCli	.reloc	relocations	Imag	R	RWE
75930000	00001000	apphelp		PE header	Imag	R	RWE
75931000	0003C000	apphelp	.text	code, import	Imag	R	RWE
7596D000	00003000	apphelp	.data	data	Imag	R	RWE
75970000	00009000	apphelp	.rsrc	resources	Imag	R	RWE
75979000	00003000	apphelp	.reloc	relocations	Imag	R	RWE
75A00000	00001000	profapi		PE header	Imag	R	RWE
75A01000	00007000	profapi	.text	code, import	Imag	R	RWE
75A08000	00001000	profapi	.data	data	Imag	R	RWE
75A09000	00001000	profapi	.rsrc	resources	Imag	R	RWE
75A0A000	00001000	profapi	.reloc	relocations	Imag	R	RWE
75AA0000	00001000	KERNELBA		PE header	Imag	R	RWE

Let's return to the code. What we see at the beginning of the code is the program prologue added by the compiler. In case of the Visual Studio compiler, the prologue always looks the same. We can go to the next instruction without entering the call using the F8 key. The next instruction is a jump. We press F7. We're now in the place we've jumped to. What we can see here is a jump to the main function.

00B01207	. E8 F0170000	CALL _amsg_exit	
00B0120C	. 59	POP ECX	
00B0120D	> A1 14C1B000	MOV EAX, DWORD PTR DS:[_environ]	
00B01212	. A3 18C1B000	MOV DWORD PTR DS:[__inifenv], EAX	
00B01217	. 50	PUSH EAX	
00B01218	. FF35 0CC1B000	PUSH DWORD PTR DS:[__argv]	
00B0121E	. FF35 08C1B000	PUSH DWORD PTR DS:[__argc]	
00B01224	. E8 D7F0FFFF	CALL main	
00B01229	. 83C4 0C	ADD ESP, 0C	

We can go to it by pressing the F4 key. It works as follows: the debugger places a breakpoint, a kind of trap for this instruction, and subsequently starts the program. We press F4 again and we are at the place the instruction is called. We press F7 to step inside this call. Here we get the code we've just created. We can find here the call of functions MessageBox and printf. We can also see the parameters entered in the code, but they are in the reverse order to the one declared in C++.

Address	Hex dump	Disassembly	Comment
00B01000	6A 00	PUSH 0	Style = MB_OK MB_APPLMODAL
00B01002	68 B099B000	PUSH OFFSET ??_CO_05CJBACGMB@hello?5AA0	Title = "hello"
00B01007	68 B899B000	PUSH OFFSET ??_CO_0MeLACCNMM@hello?5wo	Text = "hello world"
00B0100C	6A 00	PUSH 0	hOwner = NULL
00B0100E	FF15 0081B000	CALL DWORD PTR DS:[&USER32.MessageBoxA	MessageBoxA
00B01014	68 B899B000	PUSH OFFSET ??_CO_0MeLACCNMM@hello?5wo	<%s> = "hello world"
00B01019	68 C499B000	PUSH OFFSET ??_CO_030FAPEBGM@?5CFs?6?5A	format = "%s\n"
00B0101E	E8 15000000	CALL printf	printf
00B01023	83C4 08	ADD ESP, 8	
00B01026	33C0	XOR EAX, EAX	
00B01028	C3	RET	

We press F8 to go to the next instruction. We can see that a new value appeared on the stack. It's 0. The next instruction is PUSH, that is putting a number on the stack. This number is an address of the Hello string. We press F8 and see that the value 00B099B0 appeared on the stack. The Hello string is under this address and we see it in the preview. The next instruction is push Hello World. As we can see, it should be the address B899B000. If we look here, we'll see that the Hello World string begins exactly at that location.

Address	Hex dump	ASCII
00B099B0	68 65 6C 6C 6F 00 00 00 68 65 6C 6C 6F 20 77 6F	hello...hello wo
00B099C0	72 6C 64 00 25 73 0A 00 48 00 00 00 00 00 00 00	rld.%s..H.....
00B099D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B099E0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B099F0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B09A00	00 00 00 00 04 B0 B0 00 60 9A B0 00 03 00 00 00
00B09A10	52 53 44 53 0B 20 64 1D 72 DA C4 49 AF 55 41 09	RSDsø d#r—I>UA.
00B09A20	A4 BD 9E 9B 01 00 00 00 43 3A 5C 55 73 65 72 73	#Rc0...C:\Users
00B09A30	5C 47 72 7A 6F 6E 75 5C 44 65 73 68 74 6F 70 5C	\Grzonu\Desktop\
00B09A40	4D 6F 64 75 6C 65 73 5C 31 5C 74 65 73 74 5C 52	Modules\1\test\Re
00B09A50	65 6C 65 61 73 65 5C 74 65 73 74 2E 70 64 62 00	elease\test.pdb.
00B09A60	50 25 00 00 20 4E 00 00 A0 6E 00 00 00 00 00 00	P%.. N..án.....
00B09A70	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00B09A80	FE FF FF FF 00 00 00 00 04 FF FF FF 00 00 00 00
00B09A90	FE FF FF FF 00 00 00 00 CC 10 B0 00 00 00 00 00
00B09AA0	FE FF FF FF 00 00 00 00 CC FF FF FF 00 00 00 00

We press F8 and we see 00B099B8 on the stack. The values in the code and on the stack are different because encoding the message takes place in a Little endian bit order. It's an encoding method where the most significant bit is stored at the end, with less significant bits at the beginning. Here we can see that B8 is at the beginning, even though it's last on the stack.

0018FB40	00B099B8	ASCII "hello world"
0018FB44	00B099B0	ASCII "hello"
0018FB48	00000000	
0018FB4C	00B01229	RETURN to test.00B01229 from test.main
0018FB50	00000001	
0018FB54	005E1A90	
0018FB58	005E1AE8	
0018FB5C	3884A54A	
0018FB60	00000000	
0018FB64	00000000	
0018FB68	7FFDF000	
0018FB6C	0018FB7C	
0018FB70	00000000	
0018FB74	00000000	
0018FB78	0018FB5C	
0018FB7C	0BF31218	
0018FB80	0018FBCC	Pointer to next SEH record

When returning to our string, we can see that the 00B099B8 address indicates the Hello World string. We press F8 and we can see that the parameters on the stack, when looking from the top, are saved in the same order as in our C++ code. We press F8, so as not to step inside the function responsible for displaying the MessageBox. The MessageBox appeared. We press OK.

The next function we called was the printf function. As we can see, the Hello World message is placed on the stack, and right after it goes the format string. That means a string, and then Enter. We press F8. The difference between calling printf and MessageBox is that the MessageBox is a function of type STDCALL and what we've put on the stack as parameters was removed from the stack automatically by the MessageBox function. Printf is a function of Pascal type; it's characterised by the fact that if we put something on a stack, we have to remove it on our own. That's what the ADD ESP,8 instruction is for.

ESP is a register which indicates the top of the stack, that is the value which will be removed from it first. If we add 8 to it, ESP will indicate 0018FB4C, that's the address which was a stack top before we called the printf function. Now the return address is present on the stack. We could, for instance, change the return address, but we won't do that because that would cause an incorrect closing of the program. We will see how to do so, just in case.

We click Modify and enter any value, for instance 28, and at this moment the program would jump to the address present on the stack when a RET instruction is executed. We'll change it back to the previous value, so that the program doesn't crash. Another instruction which will be performed by the

program is XOR EAX, EAX. We want the EAX to be set to 0 because XOR-ing two identical values always gives 0 as a result. We can see that in accordance with our assumptions, a 0 appeared in the EAX register.

Registers (FPU)	
EAX	00000000
ECX	000010CB test.00B010CB
EDX	778B7094 ntdll.KiFastSystemCallRet
EBX	7FFDF000
ESP	0018FB90
EBP	0018FB90
ESI	00000000
EDI	00000000

However, before the RET instruction, we can change the value, for instance to 1. We press F7 to return and we can see that this value is removed from the stack. The value disappeared from the stack and we're in the place indicated by the last address. Next, the values passed earlier to the main function are removed from the stack, that is the earlier EAX, argv and argc, together 12 bytes. 12 in hexadecimal notation is 0C. Next we see EBP - 20. EBP indicates the 0018FB90 address. Olly marks these fragments with frames. They are stack frames which appear the moment a function is called.

0018FB6C	0018FB7C	
0018FB70	00000000	
0018FB74	00000000	
0018FB78	0018FB5C	
0018FB7C	0BF31218	
0018FB80	0018FBCC	Pointer to next SEH record
0018FB84	00B02550	SE handler
0018FB88	382CC47A	
0018FB8C	00000000	
0018FB90	0018FB9C	
0018FB94	76A8ED6C	
0018FB98	7FFDF000	
0018FB9C	0018FBDC	
0018FBA0	778D377B	
0018FBA4	7FFDF000	
0018FBA8	7738E309	
0018FBAC	00000000	

The EAX we've modified will be loaded to this address. 90 - 20, equals 70. Under this address we get 1. It's an exit code from the main function, because the value the function returns is present in the EAX register. Later we can see a comparison and a jump.

0018FB6C	0018FB7C	
0018FB70	00000001	
0018FB74	00000000	
0018FB78	0018FB5C	
0018FB7C	0BF31218	
0018FB80	0018FBCC	Pointer to next SEH record
0018FB84	00B02550	SE handler
0018FB88	382CC47A	
0018FB8C	00000000	
0018FB90	0018FB9C	
0018FB94	76A8ED6C	
0018FB98	7FFDF000	
0018FB9C	0018FBDC	
0018FBA0	778D377B	

We can see that the EAX register is put on the stack. It will be a parameter of the exit function, which will exit the program with code 1 because that's the value we set before exiting the main function.

Address	Hex dump	Disassembly
00B01200	> A1 14C1B000	MOV EAX, DWORD PTR DS:[_environ]
00B01212	. A3 18C1B000	MOV DWORD PTR DS:[__initenv], EAX
00B01217	. 50	PUSH EAX
00B01218	. FF35 0CC1B000	PUSH DWORD PTR DS:[__argv]
00B0121E	. FF35 08C1B000	PUSH DWORD PTR DS:[__argc]
00B01224	. E8 07FDFFFF	CALL main
00B01229	. 83C4 0C	ADD ESP, 0C
00B0122C	. 8945 E0	MOV DWORD PTR SS:[EBP-20], EAX
00B0122F	. 3975 E4	CMP DWORD PTR SS:[EBP-1C], ESI
00B01232	. 75 06	JNZ SHORT 00B0123A
00B01234	. 50	PUSH EAX
00B01235	. E8 78170000	CALL exit
00B0123A	> E8 9F170000	CALL _start
00B0123F	. 5B 2E	JMP SHORT 00B0126F
00B01241	. 8B45 EC	MOV EAX, DWORD PTR SS:[EBP-14]
00B01244	. 8B08	MOV ECX, DWORD PTR DS:[EAX]
00B01246	. 8B09	MOV ECX, DWORD PTR DS:[ECX]
00B01248	. 894D DC	MOV DWORD PTR SS:[EBP-24], ECX
00B0124B	. 50	PUSH EAX
00B0124C	. 51	PUSH ECX
00B0124D	. E8 D6190000	CALL _xoptFilter

Now the exit from the program takes place. Let's click the Play button. The program finished its execution and we get the message "Process terminated, exit code 1". As we can see, the program exit code equalled 1. If we hadn't modified anything in our code, the output code would equal 0. Now we'll look at our code in PView. As we've said before, this program can be used to preview application headers. The program consists of a standard DOS header, which tells us virtually nothing, but offers backward compatibility. From it, we only need an offset, which is a pointer to the next header. We can see that this offset value is E0 and the address indicates Image NT header.

	pFile		
test.exe			
IMAGE_DOS_HEADER	000000E0	50	45 00
MS-DOS Stub Program	000000F0	00	00 00
+ IMAGE_NT HEADERS	00000100	00	40 00
IMAGE_SECTION_HEAD	00000110	00	80 00
IMAGE_SECTION_HEAD	00000120	05	00 01
IMAGE_SECTION_HEAD	00000130	00	00 01
IMAGE_SECTION_HEAD	00000140	00	00 10
IMAGE_SECTION_HEAD	00000150	00	00 00
SECTION .text	00000160	34	9D 00
+ SECTION .rdata	00000170	00	00 00
SECTION .data	00000180	00	F0 00
+ SECTION .rsrc	00000190	00	00 00
+ SECTION .reloc	000001A0	00	00 00

Before it, there is also a DOS stub, a code fragment which would run if we ran our program under DOS. Similarly to DOS header, it has to ensure a backward compatibility with 16-bit systems. We can see that NT header is located here. We'll extend its structure. We see that it consists of a signature, based on which we can determine whether it's a correct PE file. This signature is simply a PE text.

	pFile	Data	Description	Value
test.exe				
IMAGE_DOS_HEADER	000000E0	00004550	Signature	IMAGE_NT_SIGNATURE PE
MS-DOS Stub Program				
+ IMAGE_NT HEADERS				
+ Signature				
IMAGE_FILE_HEADER				
IMAGE_OPTIONAL_H				

The next structure in the NT header is Image File header. Its fields include inter alia Machine, which determines the type of processor the application is for. Another value we're interested in is Number of sections. It specifies the number of sections in the file. Here we've got 5 of them. In a moment we'll discuss what a section is. The next field is Size of optional header. It's the size of a subsequent structure in NT header. Then we have a Characteristic field which informs us about the file type. In this case it's an executable file intended for 32-bit systems. It could just as well be an application for a 64-bit architecture. In such a situation, however, this field would have a slightly different value.

Another structure is Image optional header. From the fields we're particularly interested in, we should mention Address of entry point, that is the address at which our program starts its execution. It's a default base address which will be modified if a file is relocated. If a file has no relocation, the program will be automatically loaded to this address. Another important field is Size of image. Obviously, the field informs us about the file size. In this structure we'll also be interested in the Data Directories array, which includes information about, for instance, import array, export array, as well as their sizes.

After the Image NT header there are headers of subsequent sections, one after another. Section is a file part which has its access rights. E.g. the first section is usually the code section. It has executing and reading rights and is also marked as the one containing the code. As we can see, the data section has reading and writing rights. Thanks to that we are able to write or read something from our variables, but not overwrite a code section too easily. Obviously, it can be bypassed and we will do so many times throughout this training, but we'll talk about it later.

IMAGE_SECTION_HEAD	000001F0	00000000	Pointer to Relocations	
IMAGE_SECTION_HEAD	000001F4	00000000	Pointer to Line Numbers	
IMAGE_SECTION_HEAD	000001F8	0000	Number of Relocations	
IMAGE_SECTION_HEAD	000001FA	0000	Number of Line Numbers	
IMAGE_SECTION_HEAD	000001FC	60000020	Characteristics	
SECTION .text		00000020		IMAGE_SCN_CNT_CODE
SECTION .rdata		20000000		IMAGE_SCN_MEM_EXECUTE
SECTION .data		40000000		IMAGE_SCN_MEM_READ
SECTION .rsrc				
SECTION .reloc				

In each section we're interested in the Virtual Size value, that is the size in the memory, RVA address, that is the location in the memory in relation to the base address, the size of the data in the file and the location within the file. The first section is the code section, the second section includes imports, the third one is the data section, the fourth one, in our case, is a section for storing resources, the fifth section includes relocations. As we can see, in this case the division is pretty logical and depends on the data we store in particular sections. In the first section we can just see the code. We can view it in a hexadecimal or a text form (also called ASCII).

pFile	Raw Data	Value
00001640	00 74 13 FF B5 B4 FD FF FF E8 91 17 00 00 83 A5	t.....
00001650	B4 FD FF FF 00 59 8B BD C4 FD FF FF 8A 07 88 85Y.....
00001660	EF FD FF FF 84 C0 74 15 8B 8D 94 FD FF FF 8B 9Dt.....
00001670	D8 FD FF FF 33 F6 8A D0 E9 C6 F5 FF FF 80 BD B03.....
00001680	FD FF FF 00 74 0A 8B 85 AC FD FF FF 83 60 70 FDt.....`p.
00001690	8B 85 DC FD FF FF 8B 4D FC 5F 5E 33 CD 5B E8 86M._^3.[..
000016A0	ED FF FF C9 C3 8B FF 92 1A 40 00 91 18 40 00 C1@...@..
000016B0	18 40 00 1F 19 40 00 6B 19 40 00 76 19 40 00 BC	@...@.k.@.v.@..
000016C0	19 40 00 ED 1A 40 00 8B FF 55 8B EC 8B 45 08 A3	@...@...U...E..
000016D0	00 C1 40 00 5D C3 8B FF 55 8B EC 81 EC 28 03 00	..@.]...U...(..
000016E0	00 A1 04 B0 40 00 33 C5 89 45 FC 53 8B 5D 08 57	...@.3..E.S.]..W

Let's look into the rdata section, which is an import section. It includes an import array, which in turn has addresses of all the functions used in the program. We can see that the functions are pretty numerous, even though in our code we've only used functions MessageBox and printf. All the remaining functions are used by the program prologue and are automatically added during the compilation. We'll talk more about the import section in one of the next modules, where we'll insert the so-called hooks directly into the import section.

In the data section we can see that there are many zeroes, because many variables in our program are set to 0 as a default. Currently, there is only one resource in the resource section – manifest – automatically added by the compiler. There is also the relocation section. It's used if the program has to be loaded under the base address other than the default, which is present in the Image Optional Header. Relocations are simply addresses of all the places in the code which must be appropriately modified.

That's basically everything the PView program offers. Now, we'll view our application using the Hexplorer program. We see the preview of our program. The editor enables us to edit each byte. If we changed a single letter in the PE string, to T for instance, the program wouldn't start due to an incorrect header of executable file. It would halt before executing the first code line. We can see it for ourselves. We click File, Save as and save the new program as test2. We have a modified application. We start it and we can see that the program simply closed because the header was incorrect. If we change the header back

to PE and save it, the program will start correctly. Even though it's just a short string which doesn't bring anything new to the program, we can't modify it.

Using Hexplorer we can also browse all strings or search the contents for specific character strings. We click Find. In the program we've used for instance a Hello string. Let's step into it. As we can see, the program localized it. We can modify it, for instance change it to Bye. For this purpose, we have to add 2 zeroes at the end. They have to be binary zeroes.

62 79 65 00 00	00 00 00	68 65 6C 6C 6F 20 77 6F	bye
72 6C 64 00 25 73 0A 00	48 00 00 00 00 00 00 00	00 00 00 00	rl d %s
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00	
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00	
00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00	
00 00 00 00 04 B0 40 00	60 9A 40 00 03 00 00 00		␣ @

We save the modified version in the test2 file and launch it. As we can see, the window title was changed successfully. We press OK. Another feature offered by Hexplorer is casting other binary data to headers. Let's choose Structures and PE header, because we know it's a PE header. Now we see all the fields in this structure.

This way, we've reached the end of the software presentation. We've managed to discuss the basics which will come in handy when working with our training. Of course, during the subsequent modules we'll get to know a range of other applications. I strongly invite you to the next module, where we will learn how to create our own shellcode. See you there.

